Proceedings of the
Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Towards a Better SCM: Revlog and Mercurial

Matt Mackall
Selenic Consulting
mpm@selenic.com

Abstract

Large projects need scalable, performant, and robust software configuration management systems. If common revision control operations are not cheap, they present a large barrier to proper software engineering practice. This paper will investigate the theoretical limits on SCM performance, and examines how existing systems fall short of those ideals.

I then describe the Revlog data storage scheme created for the Mercurial SCM. The Revlog scheme allows all common SCM operations to be performed in near-optimal time, while providing excellent compression and robustness.

Finally, I look at how a full distributed SCM (Mercurial) is built on top of the Revlog scheme, some of the pitfalls we've surmounted in on-disk layout and I/O performance and the protocols used to efficiently communicate between repositories.

1 Introduction: the need for SCM scalability

As software projects grow larger and open development practices become more widely adopted, the demands on source control management systems are greatly increased. Large projects demand scalability in multiple dimensions, including efficient handling of large numbers of files, large numbers of revisions, and large numbers of developers.

As an example of a moderately large project, we can look at the Linux kernel, a project now in its second decade. The Linux kernel source tree has tens of thousands of files and has collected on the order of a hundred thousand changesets since adopting version control only a few years ago. It also has on the order of a thousand contributors scattered around the globe. It also continues to grow rapidly. So it's not hard to imagine projects growing to manage millions of files, millions of changesets, and many thousands of people developing in parallel over a timescale of decades.

At the same time, certain SCM features become increasingly important. Decentralization is crucial: thousands of users with varying levels of network access can't hope to efficiently cooperate if they're all struggling to commit changesets to a central repository due to locking and bandwidth concerns. So it becomes critical that a system be able to painlessly handle per-user development branches and repeated merging with the branches of other developers. Grouping of interdependent changes in multiple files into a single "atomic" changeset becomes a necessity for understanding and working with the huge number of changes that are introduced. And robust, compact storage of the revision history is essential for large number of developers to work with their own decentralized repositories.

2 Overview of Scalability Limits and Existing Systems

To intelligently evaluate scalability issues over a timescale of a decade or more, we need to look at the likely trends in both project growth and computer performance. Many facets of computer performance have historically followed an exponential curve, but at different rates. If we order the more important of these facets by rate, we might have a list like the following:

- CPU speed
- disk capacity
- memory capacity
- memory bandwidth
- LAN bandwidth
- disk bandwidth
- WAN bandwidth
- disk seek rate

So while CPU speed has changed by many orders of magnitude, disk seek rate has only changed slightly. In fact, seek rate is now dominated by disk rotational latency and thus its rate of improvement has already run up against a wall imposed by physics. Similarly, WAN bandwidth runs up against limits of existing communications infrastructure.

So as technology progresses, it makes more and more sense to trade off CPU power to save disk seeks and network bandwidth. We have in fact already long since reached a point where disk seeks heavily dominate many workloads, including ours. So we'll examine the important aspects of source control from the perspective of the facet it's most likely to eventually be constrained by.

For simplicity, we'll make a couple simplifying assumptions. First, we'll assume files of a constant size, or rather that performance should generally be linearly related to file size. Second, we'll assume for now that a filesystem's block allocation scheme is reasonably good and that fragmentation of single files is fairly small. Third, we'll assume that file lookup is roughly constant time for moderately-sized directories.

With that in mind, let's look at some of the theoretical limits for the most important SCM operations as well as scalability of existing systems, starting with operations on individual files:

Storage compression: For compressing single file revisions, the best schemes known include SCCS-style "weaves" [8] or RCS-style deltas [5], together with a more generic compression algorithm like gzip. For files in a typical project, this results in average compression on the order of 10:1 to 20:1 with relatively constant CPU overhead (see more about calculating deltas below).

Retrieving arbitrary file revisions: It's easy to see that we can easily achieve constant time (O(1) seeks) retrieval of individual file revisions, simply by storing each revision in a separate file in the history repository. In terms of big-O notation, we can do no better. This ignores some details of filesystem scalability, but it's a good approximation. This is perhaps the most fundamental operation in an SCM, so the scalability of this operation is crucial.

Most SCMs, including CVS, and Bitkeeper use delta or weave-based schemes that store all the revisions for a given file in a single back-end file. Reconstructing arbitrary versions requires reading some or all of the history file, thus making performance O(revisions) in disk bandwidth and computation. As a special case, CVS stores the most recent revision of a given file uncompressed, but still must read and parse the entire history file to retrieve it.

SVN uses a skip-delta [7] scheme that requires reading O(log revisions) deltas (and seeks) to reconstruct a revision.

Unpacked git stores a back-end file for each file revision, giving O(1) performance, but packed git requires searching a collection of indices that grow as the project history grows. Also worth noting is that git does not store an index information at the file level. Thus operations like finding the previous version of a file to calculate a delta or the finding the common ancestor of two file revisions can require searching the entirety of the project's history.

Adding file revisions: Similarly, we can see that adding file revisions by the same scheme is also O(1) seeks. Most systems use schemes that require rewriting the entire history file, thus making their performance decrease linearly as the number of revisions increase.

Annotate file history: We could, in principle, incrementally calculate and store an annotated version of each version of a file we store and thus achieve file history annotation in O(1) seeks by adding more up-front CPU and storage overhead. Almost all systems instead take O(revision) disk bandwidth to construct annotations, if not significantly more. While annotation is traditionally not performance critical, some newer merge algorithms rely on it [8].

Next, we can look at the performance limits of working with revisions at the project level:

Checking out a project revision: Assuming O(1) seeks to check out file revisions, we might expect O(files) seeks to check out all the files in the project. But if we arrange so that file revisions are nearly consecutive, we can avoid most of those seeks, and instead our limit becomes O(files) as measured by disk bandwidth. A comparable operation is untarring an archive of that revision's files.

As most systems must read the entirety of a file's history to reconstruct a revision, they'll require O(total file revisions) disk bandwidth and CPU to check out an entire project tree. SCMs that don't visit the repository history in the filesystem's natural layout can easily see this performance degrade into O(total files) random disk seeks, which can happen with SCMs backed by database engines or with systems like git which store objects by hash (unpacked) or creation time (packed).

Committing changes: For a change to a small set of the files in a project, we can expect to be bound by O(changed files) seeks, or, as the number of changes approaches the number of files, O(files) disk bandwidth.

Systems like git can meet this target for commit, as they simply create a new back-end file for each file committed. Systems like CVS require rewriting file histories and thus take increasingly more time as histories grow deeper. Non-distributed systems also need to deal with lock contention which grows quite rapidly with the number of concurrent users, lock hold time, and network bandwidth and latency.

Finding differences in the working directory: There are two different approaches to this problem. One is to have the user explicitly acquire write permissions on a set of files, which nicely limits the set of files that need to be examined so that we only need O(writable files) comparisons. Another is to allow all files to be edited and to detect changes to all managed files.

As most systems require O(file revisions) to retrieve a file version for comparison, this operation can be expensive. It can be greatly accelerated by keeping a cache of file timestamps and sizes as checkout time.

3 Revlog: A Solid Foundation

The core of the Mercurial system [4] is a storage scheme known as a "revlog," which seeks to address the basic scalability problems of other systems. First and foremost, it aims to provide O(1) seek performances for both reading and writing revisions, but still retain effective compression and integrity.

The fundamental scheme is to store a separate index and data file for each file managed. The index contains a fixed-sized record for each revision managed while the data file contains a linear series of compressed hunks, one per revision.

Each hunk is either a full revision or a delta against the immediately preceding hunk or hunks. This somewhat resembles the format of an MPEG video stream, which contains a series of delta frames with occassional full frames for synchronization.

Looking up a given revision is easy—simply calculate the position of the relevant record in the index and read it. It will contain a pointer to the first full revision of the delta chain and the entire chain can then be read in a single contiguous chunk. Thus we need only O(1) seeks to locate and retrieve a revision.

By limiting the total length of the hunks needed to reconstruct a given version to a small multiple of that version's uncompressed size, we guarantee that we'll never need to read more than O(1) file equivalents from the data file and our CPU time for reconstruction is similarly bounded.

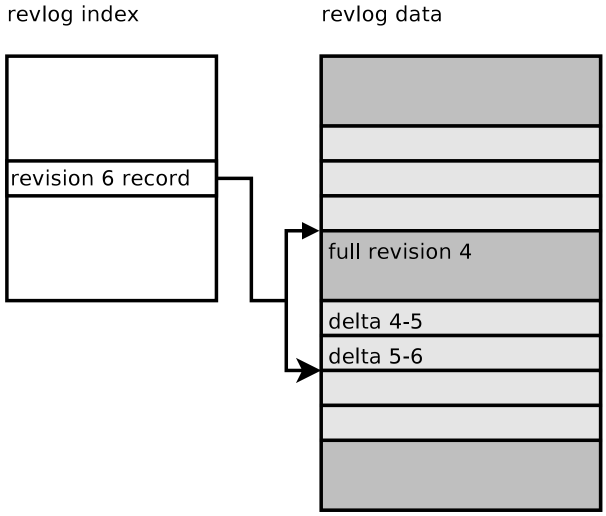

Figure 1: Revlog Layout

To add a new version, we simply append a new index record and data hunk to our revlog, again an O(1) operation. This append-only approach provides several advantages. In addition to being fast, we also avoid the risk of corrupting earlier data because we don't rewrite it. If we are careful to only write index entries after data hunks, we also need no special provisions for locking out readers while writes are in progress. And finally, it makes possible a trivial journalling scheme, described later.

Because our system needs to allow more than simple linear development with repeated branching and merging, our revision identifiers cannot be just simple integers. And because our scheme is intended to be decentralized, we'd like to have identifiers that are global. Thus, Mercurial generates identifiers from a strong hash of its contents (currently using SHA1). This hash also provides a robust integrity check when revisions are reconstructed.

As our project can contain arbitrary branching and merging through time, each revlog can have an arbitrary directed acyclic graph (DAG) describing its history. But if we revert a change, we will have the same hash value appearing in two places on the graph, which results in a cycle!

Revlogs address this by incorporating the hash of the parent revisions into a given revision's hash identifier, thus making the hash dependent on both the revision's contents and its position in the tree. Not only does this avoid our cycle problem, it makes merging DAGs from multiple revlogs trivial: simply throw out any revisions with duplicate hashes and append the remainder.

Each 64 byte revlog index record contains the following information:

```
 2 bytes: flags
 6 bytes: hunk offset
 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
```

Data hunks are composed of a flag byte indicating compression type followed by a full revision or a delta.

4 Delta Encoding Considerations

Revlog deltas themselves are quite simple. They're simply a collection of chunks specifying a start point, an end point, and a string of replacement bytes. But calculating deltas and applying deltas both turn out to have some interesting issues.

First, let's consider delta calculation. We carefully tested three algorithms before selecting the one used by revlogs.

The classic algorithm used by GNU diff and most other textual tools is the Myers algorithm, which generates optimal output for the so-called longest common substring (LCS) problem. This algorithm is fairly complex, and the variant used by GNU diff has heuristics to avoid a couple forms of quadratic run-time behavior.

While the algorithm is optimal for LCS, it does not in fact generate the shortest deltas for our purposes. This is because it weighs insertions, changes, and deletions equally (with a weight of one). For us, deleting long strings of characters is cheaper than inserting or changing them.

Another algorithm we tried was xdelta [3], which is aimed at calculating diffs for large binary files efficiently. While much simpler and somewhat faster than the Myers algorithm, it also generated slightly larger deltas on average.

Finally, we tried a C reimplementation ("bdiff") of the algorithm found in Python's difflib [1]. In short, this algorithm finds the longest contiguous match in a file, then recursively matches on either side. While also having worst-case quadratic performance like the Myer's algorithm, it more often approximates linear performance.

The bdiff algorithm has several advantages. On average, it produced slightly smaller output than either the Myers or xdelta algorithms, and was as fast or faster. It also had the shortest and simplest code of the three. And lastly, when used for textual diffs, it often generates more "intuitive" output than GNU diff.

We may eventually include the xdelta algorithm as a fallback for exceptionally large files, but the bdiff algorithm has proven satisfactory so far.

```
  rev    offset  length   base linkrev nodeid       p1           p2
    0         0     453      0       0 ee9b82ca6948 000000000000 000000000000
    1       453     107      0       1 98f3df0f2f4f ee9b82ca6948 000000000000
    2       560      73      0       2 8553cbcb6563 98f3df0f2f4f 000000000000
    3       633      63      0       3 09f628a628a8 8553cbcb6563 000000000000
    4       696      69      0       4 3413f6c67a5a 09f628a628a8 000000000000
...
   15      1435      69      0      15 69d47ab5fc42 9e64605e7ab0 000000000000
   16      1504     111      0      16 81322d98ee1f 69d47ab5fc42 000000000000
   17      1615     525     17      17 20f563caf71e 81322d98ee1f 000000000000
   18      2140      56     17      18 1d47c3ef857a 20f563caf71e 000000000000
...
```

Table 1: Part of a typical revlog index

Next is the issue of applying deltas. In the worst case, we may need to apply many thousands of small deltas to large files. To sequentially apply 1K deltas to a 1M, we'll effectively have to do 1G of memory copies—not terribly efficient. Mercurial addresses this problem by recursively merging neighboring patches into a single delta. Not only does this reduce us to only applying a single delta, it eliminates or joins the hunks that overlap, further reducing the work. Thus, patch application is reduced in CPU time from O(file size * deltas) to approximately O(file size + delta size). As we've already constrained the total data to be proportional to our original file size, this is again O(1) in terms of file units.

5 Mercurial: A Simple Hierarchy

With revlogs providing a solid basis for storing file revisions, we can now describe Mercurial. Naturally, Mercurial stores a revlog for each managed file. Above that, it stores a "manifest" which contains a list of all file:revision hash pairs in a project level revision. And finally, it stores a "changeset" that describes the change that corresponds to each manifest. And conveniently, changesets and manifests are also stored in revlogs!

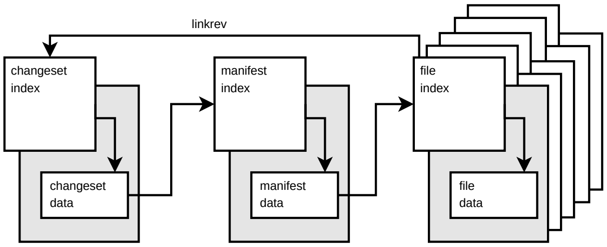

Figure 2: The Mercurial Hierarchy

This schema of file/manifest/changeset hashing was directly inspired by Monotone [2] (which also inspired the scheme used by git [6]). Mercurial adds a couple important schema improvements beyond using revlogs for efficient storage. As already described, revlog hashes avoid issues with graph cycles and make merging graphs extremely easy. Also, revlogs at each level contain a "linkrev" for each revision that points to the associated changeset, allowing one to quickly walk both up and down the hierarchy. It also has file-level DAGs which allow for more efficient log and annotate, and more accurate merge in some situations.

Checking out Project Revisions: Checkout is a simple process. Retrieve a changeset, retrieve the associated manifest, and retrieve all file revisions associated with that manifest.

Mercurial takes the approach of keeping a cache of managed file sizes and timestamps for rapid detection of file changes for future commits. This also lets us know which files need updating when we checkout a changeset. Naturally, Mercurial also tracks which changeset the current working directory is based on, and in the case of merge operations, we'll have two such parents.

The manifest is carefully maintained in sorted order and all operations on files are done in sorted order as well. An early version stored revlogs by hash (as git still does) and that scheme was found to rapidly degrade over time. Simply copying a repo would often reorder it on disk by hash, giving worst-case performance.

With revlogs instead stored in a directory tree mirroring the project and all reads done in sorted order, filesystems and utilities like cp and rsync are given every opportunity to optimize on-disk layout so that we minimize seeks between revlogs in normal usage. This gets us very close to O(files) disk bandwidth with typical filesystems.

Performance for full tree checkouts for a large project tend to be very comparable to time needed to uncompress and untar an equivalent set of files.

Committing Changes: Commits are atomic and are carefully ordered so as to not need any locking for readers. First all relevant file revlogs are extended, followed by the manifest and finally the changelog. Adding an entry to the changelog index makes it visible to readers.

Commit operations are also journalled. For each revlog file appended to during a commit, we simply record the original length of the file. If a commit is interrupted, we simply truncate all the files back to the earlier lengths, in reverse order.

Note that because locking is only needed on writes and each user commits to primarily to their own private repository, lock contention is effectively nil.

Commit performance is high enough that Mercurial can apply and import a large series of patches into its repository faster than tools like quilt can simply apply them to the working directory.

Cloning: While Mercurial repositories can contain numerous development branches, branching is typically done by "cloning" a repository wholesale. By using hardlinks and copy-on-write techniques, Mercurial can create independent lightweight copies of entire repositories in seconds. This is an important part of Mercurial's distributed model—branches are cheap and discardable.

Pushing and Pulling: A fundamental operation of a distributed SCM is synchronization between the private repositories of different users. In Mercurial, this operation is called "pulling" (importing changes from a remote repository) or "pushing" (sending local changes to a remote repository).

Pulling typically involves finding all changesets present in the remote tree but not in the local tree and asking them to be sent. Because this may involve many thousands of changesets, we can't simply ask for a list of all changesets and compare. Instead, Mercurial asks the remote end for a list of heads and walks backwards until it finds the root nodes of all remote changesets. It then requests all changesets starting at these root nodes.

To minimize the number of round trips required for this search, we make three optimizations. First, we allow many requests in a single query so that we can search different branches of the graph in parallel. Second, we combine chains of linear changes into single "twigs" so that we can quickly step across large regions of the graph. And finally, we use a special binary search approach to quickly find narrow our search if new changes appear in the middle of a twig. This approach lets us find our new changesets in approximately O(log(new changesets)) bandwidth and round-trips.

Once the outstanding changes are found, the remote end begins to stream the changes across in Mercurial's "bundle" format. Rather than being ordered changeset by changeset, a bundle is instead ordered revlog by revlog. First, all new changelog entries are sent as deltas. As changesets are visited, a list of changed files is constructed.

Armed with a list of new changesets, Mercurial can quickly scan the manifest for changesets with a matching linkrev and send all new manifest deltas. We can also quickly visit our list of changed files and find their relevant deltas (again in sorted order). This minimizes seeking on both ends and avoids visiting unchanged parts of the repository.

Note that because this scheme makes deltas for the same revlog adjacent, the stream can also be more effectively compressed, preserving valuable WAN bandwidth.

6 Conclusions

By careful attention to scalability and performance issues for all common operations from the ground up, Mercurial performs well even as projects become very large in terms of history, files, or users.

As most of the performance improvements made by Mercurial are in the back-end data representation, they should be applicable to many other systems.

Mercurial is still a young and rapidly improving system. Contributions are encouraged!

References

[1] difflib. `http://docs.python.org/lib/module-difflib.html`.

[2] Graydon Hoare. http://www.venge.net/monotone/. `http://www.venge.net/monotone/`.

[3] Davide Libenzi. Libxdiff. `http://www.xmailserver.org/xdiff-lib.html`.

[4] Matt Mackall. The mercurial scm. `http://selenic.com/mercurial`.

[5] Walter F. Tichy. Rcs–a system for version control. `http://docs.freebsd.org/44doc/psd/13.rcs/paper.html`.

[6] Linus Torvalds. Git - tree history storage tool. `http://git.or.cz/`.

[7] Unknown. Skip-deltas in subversion. `http://svn.collab.net/repos/svn/trunk/notes/skip-deltas`.

[8] Various. Weave - revctrl wiki. `http://revctrl.org/Weave`.