Scalability of Distributed Version Control Systems

Michael J. Murphy John Markus Bjørndalen Otto J. Anshus

November 2017

Abstract

A distributed version control system allows for replicas of a repository with the full history of updates. Updates and new commits can be done to each replica independent of each other. Updates, including conflicting updates, are reconciled later in a peer-to-peer merge operation. This will bring replicas to a consistent state with each other.

These systems are popular, but their intended use is typically limited to the small text files of source code. However, distributed version control can also be useful for larger files, including binary files of images, audio and video.

We report on the results from a set of experiments designed to characterise the behaviour of some widely used distributed version control systems. The experiments measured commit times and repository sizes when storing single files of increasing size, and when storing increasing numbers of single-kilobyte files.

The goal is to better understand the behaviour and approaches used in those systems with the purpose of building a scalable, highly-available, distributed storage system for many large text and binary files residing on computers ranging from smart phones to servers.

An early prototype of such a system, Distributed Media Versioning (DMV), is briefly described and compared with Git and Mercurial, and the Git-based backup tool Bup.

We find that processing large files without splitting them into smaller parts will limit maximum file size to what can fit in RAM. Storing millions of small files will result in inefficient use of disk space. And storing files with hash-based file and directory names will result in high-latency write operations, due to having to switch between directories rather than performing a sequential write. We conclude that large files must be split into smaller chunks to avoid the physical memory limitation, and that many small chunks should be aggregated into larger files to achieve low latency write operations and efficient use of disk space.

We intend to incorporate these insights into future versions of DMV.

This paper was presented at the NIK-2017 conference; see http://www.nik.no/.

1 Introduction

Written: Distributed version control is an interesting form of distributed system because it takes eventual consistency to the extreme.

The CAP-theorem [10] states that a distributed system cannot be completely consistent (C), available (A), and tolerant of network partitions (P) all at the same time. When communication between nodes breaks down and they cannot all acknowledge an operation, the system is faced with ”the partition decision: block the operation and thus decrease availability, or proceed and thus risk inconsistency.” [3]

Much research is focused on consistency. However distributed version control systems focus on availability.

Written: Every replica of a repository contains the full history in an append-only data structure, any replica may add new commits, and conflicting updates are reconciled later in a merge operation.

Though maybe not designed with the CAP theorem explicitly in mind, a distributed version control system (DVCS) is in fact a small-scale distributed system, where nodes are completely autonomous. Rather than a set of connected nodes that may occasionally lose contact in a network partition, a DVCS's repositories are self-contained and offline by default. They allow writes to local data at any time, and only connect to other repositories intermittently by user command to exchange updates. Concurrent updates are not only allowed but embraced as different branches of development. A DVCS can track many different branches at the same time, and conflicting branches can be combined and resolved by the user in a merge operation.

Written: These systems are popular, but their use is generally limited to the small text files of source code.

DVCSs are designed primarily to store program source code: plain text files in the range of tens of kilobytes. Checking in larger binary files such as images, sound, or video affects performance. Actions that require copying data in and out of the system slow from hundredths of a second to full seconds or minutes. And since a DVCS keeps every version of every file in every repository, forever, the disk space needs only increase.

This has lead to a conventional wisdom that binary files should never be stored in version control, inspiring blog posts with titles such as ”Don't ever commit binary files to Git! Or what to do if you do” [27], even as the modern software development practice of continuous delivery was commanding teams to ”keep absolutely everything in version control.” [12, p.33]

Written: This paper explores the challenges of using version control to store larger binary files, with the goal of building a scalable, highly-available, distributed storage system for media files such as images, audio, and video.

This paper evaluates version control when storing larger binary files, with the goal of building a scalable, highly-available, distributed storage system with versioning for media files such as images, audio, and video.

2 Distributed Media Versioning

Repositories are ”full” replicas, but they are not necessarily up to date

Git stores data in a directed acyclic graph (DAG) data structure [25]. Each version of each file is hashed with the cryptographic SHA-1 digest, becoming a blob object, which is stored in an object store with the SHA-1 hash as its ID. Directory states are stored by creating a list of hash IDs for each file in the directory, a tree object, and also storing it by SHA-1 hash ID. Tree objects can also refer to other trees, representing subdirectories. Commit objects contain the hash ID of the tree object representing the directory state at the time of commit, plus metadata such as the author and commit message. The resulting graph is directed because the links between objects are directional. It is acyclic because objects are content-addressed. An object can only refer to another object by hash, so it must refer to an existing object whose hash is known. And objects cannot be updated without changing their hash. Therefore, it is impossible to create a circular reference.

This DAG data structure has several interesting properties for distributed data storage. The content-addressing naturally de-duplicates identical objects, since identical objects will have the same hash ID. This results in a natural data compression. The append-only nature of the DAG allows replicas to make independent updates without disturbing the existing history. Then, when transferring updates from one replica to another, only new objects need to be transferred. Concurrent updates will result in multiple branches of history, but references from child commit to parent commit establish a happens-before relationship and give a chain of causality. Data integrity can also be verified by re-hashing each object and comparing to its ID, protecting against tampering and bit rot. Updates can also be made atomic by waiting to update branch head references until after all new objects are written to the object store.

The efficiency of de-duplication depends on how well identical pieces of data map to identical objects. In Git, the redundant objects are the files and directories that do not change between commits. However, small changes to a file will result in a new object that is very similar to the previous one, and the two could be compressed further. Git compresses this redundant data between files by aggregating objects into archives called pack files. Inside pack files, Git stores similar objects as a series of deltas against a base revision [5, Section 10.4]. This secondary compression requires reading objects again, comparing them, and calculating deltas. Also, if the algorithm is implemented with an assumption that objects are small enough to fit into RAM, attempting to process large files could result in an out-of-memory error. This extra effort could be avoided by more fine-grained mapping of data to objects, so that repeated sections within files become their own objects that can be reused.

With better mapping, the DAG would de-duplicate redundant chunks of files the way that it already de-duplicates whole files. It could also ensure that all objects are a reasonable size that can fit into RAM for processing. This sub-file granularity and de-duplication is one of the core ideas behind our new data storage system, **Distributed Media Versioning (DMV)**.

The core idea is relatively simple — store data in a Git-like directed acyclic graph (DAG), but make the following changes:

1. Store data at a finer granularity than the file
2. Allow nodes to store only a portion of the DAG as a whole

Doing so allows a data set to be replicated or sharded across many nodes according to the capacity of nodes and the needs of local users. The focus is on data locality: tracking what data is where, presenting that information to the user, and making it easy to transfer

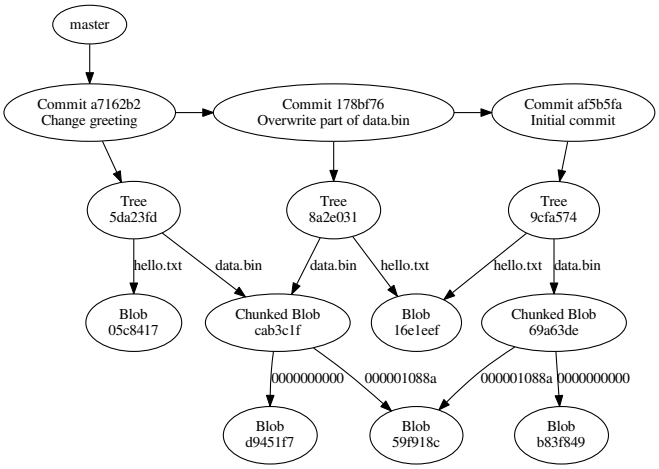


Figure 1: A simple DMV DAG with three commits

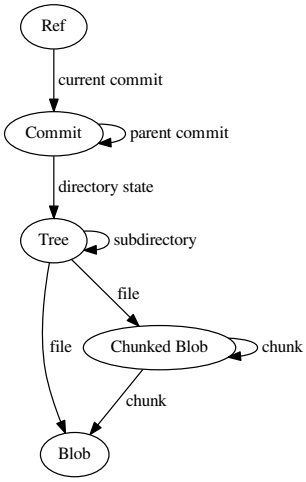


Figure 2: DMV DAG Object Types

data to other nodes as desired. The goal is to create a new abstraction, of *many devices, one data item* in varying states of synchronization.

Distributed Media Versioning (DMV)'s DAG is based on Git's, but it adds a new object type, the chunked blob, which represents a blob (binary large object) that has been broken into several smaller chunks. An example DMV DAG is shown in Figure 1, and the relationships between object types are shown in Figure 2.

Files are split into chunks using the rsync rolling hash algorithm algorithm. This splits the files into chunks by content rather than position, so that identical chunks within files (and especially different versions of the same file) will be found and stored as identical objects, regardless of their position within the file. This way, identical chunks will be naturally de-duplicated by the DAG, and only the changed portions of files need to be stored as new objects.

Written: DMV is different because it starts with the assumption that the repo is incomplete. The repository itself can be distributed.

DMV will also distribute the repository itself. Repositories will have the option of only storing a portion of the data set or a portion of its history, in order to save space. A DMV repository will start with the assumption that it does not hold all objects in the data set. The goal is to allow DMV to run on devices with widely varying available resources, from servers to mobile devices.

We have written a DMV prototype. The current early prototype can store and retrieve data locally, but the distributed features are not yet implemented.

The DMV prototype was developed with Rust stable versions 1.15 and 1.16 on Debian Linux 8.6 ("Jessie"). The current DMV prototype stands at 7592 lines of Rust code (6565 excluding comments). Source code is available at `http://dmv.sleepymurph.com/`.

Ask Otto: Move Related Works section here?

3 Experiments

Written: We perform experiments with the popular version control systems Git and Mercurial, the Git-based backup tool Bup, and our DMV prototype.

Written: We measured commit times and repository sizes when storing single files of increasing size, and when storing increasing numbers of single-kilobyte files.

Methodology

We conducted two major experiments. In order to measure the effect of file size, we committed a single file of increasing size to a each target version control system (VCS). And to measure the effect of numbers of files, we committed increasing number of small (1 KiB) files to each target VCS.

Otto: Effect on what?

We ran each experiment with four different VCSs: Git, Mercurial, Bup, the DMV prototype. As a control, we also ran the experiments with a dummy VCS that simply copied the files to a hidden directory.

Otto: Effect on what?

For each experiment, the procedure for a single trial was as follows:

1. Create an empty repository of the target VCS in a temporary directory
2. Generate target data to store, either a single file of the target size, or the target number of 1 KiB files
3. Commit the target data to the repository, measuring wall-clock time to commit
4. Verify that the first commit exists in the repository, and if there was any kind of error, run the repository's integrity check operation
5. Measure the total repository size
6. Overwrite a fraction (1/1024) of each target file
7. (Number-of-files experiment only) Run the VCS's status command that lists what files have changed, and measure the wall-clock time that it takes to complete
8. Commit again, measuring wall-clock time to commit
9. Verify that the second commit exists in the repository, and if there was any kind of error, run the repository's integrity check operation
10. Measure the total repository size again
11. (File-size experiment only, Git only) Run Git's garbage collector (`git fsck`) to pack objects, then measure total repository size again
12. Delete temporary directory and all trial files

We increased file sizes exponentially by powers of two from 1 B up to 128 GiB, adding an additional step at 1.5 times the base size at each order of magnitude. For example, on the megabyte scale, the file sizes are 1 MiB, 1.5 MiB, 2 MiB, 3 MiB, 4 MiB, 6 MiB, 8 MiB, 12 MiB, and so on.

We increased numbers of files exponentially by powers of ten from one file to ten million files, adding additional steps at 2.5, 5, and 7.5 times the base number at each order of magnitude. For example, at the hundreds and thousands scales, the file quantities are 100, 250, 500, 750, 1000, 2500, 5000, 7500, 10 000, and so on.

Input data files consisted of pseudorandom bytes taken from the operating system's pseudorandom number generator (`/dev/urandom` on Linux).

[0] Git can read a repository written by Bup, but it will see the large file as a directory full of smaller chunk files.

Experiment Platform

We ran the trials on four dedicated computers with no other load. Each was a typical office desktop with a 3.16 GHz 64-bit dual-core processor and 8 GiB of RAM, running Debian version 8.6 (”Jessie”). Each computer had one normal SATA hard disk (spinning platter, not solid-state), and trials were conducted on a dedicated 197 GiB LVM partition formatted with the ext4 filesystem. All came from the same manufacturer with the same specifications and were, for practical purposes, identical.

Include platform table?

We ran every trial four times, once on each of the experiment computers, and took the mean and standard deviation of each time and disk space measurement. However, because the experiment computers are practically identical, there was little variation.

4 Results

File Size

File Size Limits: RAM, Time, Disk Space

Written: We find that processing files whole will limit maximum file size to what can fit in RAM.

In the experiments, both Git and Mercurial had file size limits that were related to the size of RAM. Mercurial refused to commit a file 2 GiB or larger. It exited with an error code and print an error message saying ”up to 6442 MB of RAM may be required to manage this file.” This is because Mercurial stores file revisions as deltas against a base revision, so it has to do its delta calculation up front. It loads each revision of the file into memory to do the calculations, plus it allocates memory to write the output. As a result, Mercurial needs to be able to fit the file into memory three times over in order to commit it. We saw that in each case, the commit was not stored, and the repository was left unchanged. Mercurial commits are atomic.

Git’s commit operation appeared to fail with files 12 GiB and larger. It exited with an error code and print an error message saying ”fatal: Out of memory, malloc failed (tried to allocate 12884901889 bytes).” However, the commit was be written to the repository, and git’s `fsck` operation reported no errors. So the commit operation completes successfully, even though an error is reported.

Otto: Is this A, D, or I

With files 24 GiB and larger, Git’s `fsck` operation itself failed. In each case, the `fsck` command exited with an error code and give a similar ”fatal ... malloc” error. However, the 24 GiB file could still be checked out from the repository without error. So we continued the trials assuming that these were also false alarms.

Git’s delta compression takes place in a separate garbage collection step. For Git, we ran the garbage collector at the end of each trial and measured repository size before and after garbage collection. With file sizes up to and including 1 GiB, the garbage collection resulted in a reduction in repository size from approximately three times the input data size (the input file, and two separately stored revisions) to approximately twice the input data size (the input file, and compressed stored revisions). At 1.5 GiB and above, the repository size remained approximately three times the input data size after garbage collection. This indicates that Git’s delta compression also requires that the file be able to fit into disk space three times over.

The DMV prototype was able to store a file up to 64 GiB in size, but time became a limiting factor as file size increased. We set an arbitrary five and a half hour timeout for commits in our experiment script. At 96 GiB, the DMV commit operation hit this limit and was terminated.

Table 1: Observations as file size increases

Size	Observation
1.5 GiB	Largest successful commit with Mercurial
1.5 GiB	Git commit successful, but garbage collection fails to compress
2 GiB	Mercurial commit rejected
8 GiB	Largest successful commit with Git
12 GiB	Git false-alarm errors begin, but commit still intact
16 GiB	Largest successful Git fsck command
24 GiB	Git false-alarm errors begin during fsck, but commit still intact
64 GiB	Largest successful DMV commit
96 GiB	DMV timeout after 5.5 h
96 GiB	Last successful commit with Bup (and Git, ignoring false-alarm errors)
128 GiB	All fail due to size of test partition

The largest file size committed in the trials was 96 GiB. This was a limitation of the experiment environment, not a limit of the systems under test. The experiments were performed on a 197 GiB partition. The next trial size 128 GiB is too large to fit two copies on the partition. So no version control system was able to commit it, since they all save a copy of the file.

Bup was able to store a 96 GiB file with no errors in just under two hours. Git could also store such a large file, but one must ignore the false-alarm "fatal" errors being reported by the user interface.

These findings are summarized in Table 1.

Commit Times for Increasing File Sizes

Written: Give file size result timing

Figure 3 shows the wall-clock time required for the initial commit, adding a single file of the given size to a fresh repository. Over all, the trend is clear and unsurprising: commit time increases with file size. It increases linearly for Git, Mercurial, and Bup. DMV's commit times increase in a more parabolic fashion, similar to how it and Git respond to increasing numbers of files. This is because DMV breaks the large files into many smaller objects, trading the large file problem for the many file problem.

Number of Files

File Quantity Limits: inodes

Written: And we find that storing millions of objects loose as files with hash-based names will result in inefficient write speeds and use of disk space.

Git, Mercurial, DMV, and the copy operation all failed when trying to store 7.5 million files or more, reporting that the disk was full. However, the disk was not actually out of space. It was out of inodes.

Unix filesystems, ext4 included, store file and directory metadata in a data structure called an inode, which reside in a fixed-length table [20]. When all of the inodes in the table are allocated, the filesystem cannot store any more files or directories. The number of inodes is tunable at filesystem creation by passing a bytes-per-inode parameter (`-i`) to

Figure 3: Wall-clock time to commit one large file to a fresh repository

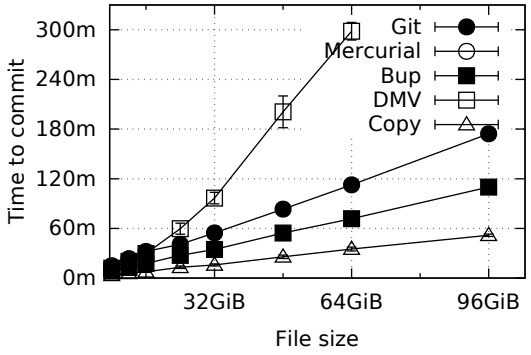
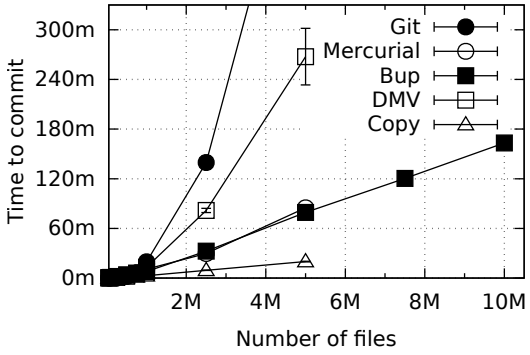


Figure 4: Wall-clock time to commit many 1KiB files to a fresh repository



`mke2fs`, but our experiment partitions used the default setting. The 197 GiB partitions had approximately 13 million inodes.

All systems tested except for Bup store a new copy of the input data, with one stored file per input file. So committing 7.5 million input files would create an additional 7.5 million stored files, for a total of 15 million inodes, 2 million more than the 13 million on the filesystem.

Bup avoided the inode limit because it writes directly into Git's pack file format. It aggregates objects and conserves inodes. Bup trials could continue until the input data itself exhausted the system's inodes attempting to generate 25 million input files.

Commit Times for Increasing Numbers of Files

Figure 4 shows the time required for the initial commit, storing all files into a fresh empty repository. Here we see the commit times for Git and DMV increasing quadratically with the number of files, while Mercurial, Bup, and the copy increase linearly.

5 Discussion

File Size Limits

Written: Discuss implementation details that lead to size limits

By using files as the basic unit of storage, and storing files as deltas against a base revision, both Git and Mercurial will at some point load an entire file into memory in order to compare it to another version. This limits the maximum file size that the system can work with to what can fit into RAM. In Mercurial's case, the error message that appears when attempting to commit a 2 GiB file warns that 6 GiB will be required to manage it. And because it has to calculate deltas in order to store a file at all, Mercurial simply cannot work with any file that it can't fit into memory three times over. This is why Mercurial could not store files larger than 1.5 GiB in the file-size experiments (**??**).

Because Git's delta calculation happens behind-the-scenes in a secondary phase, it can still manage to commit files larger than available RAM, but it prints errors as the other operations fail. The two-phase approach also requires extra disk space and processing power. If a large file is changed, then both revisions will be written in full, taking twice the disk space. Then a separate operation will have to reread both blobs in full to calculate deltas and pack the objects.

Both DMV and Bup avoid these pitfalls by operating with a finer granularity, using a rolling hash to divide files into chunks by their content. It is the chunks and their indexes that must fit into memory, not the entire file. And then since chunks are only a few kilobytes and chunk indexes are hierarchical, file size becomes theoretically unlimited. Dividing into chunks by rolling hash also makes delta compression unnecessary, because identical chunks in different files or file revisions will naturally de-duplicate. At this point, it is the method of object storage that becomes the bottleneck.

DMV Sluggishness

DMV's parabolic increase is due to the way it breaks the large file into chunks and stores objects as individual files on the filesystem. While it is reading one large file, it is writing many small files, which incurs filesystem overhead. So its performance characteristic for storing a large file is closer to that of storing many files (**??**). Bup also breaks the file into many chunks, but it avoids the filesystem overhead by recombining the chunks into pack files. We investigate the filesystem overhead further in **??**.

This sluggishness is due to the way DMV stores chunks of the file as individual files on the filesystem, turning the problem of storing one large file into the problem of storing many small files. Storing many small files in this way incurs filesystem overhead, as we discovered in the results of the number-of-files experiment (**??**), and later performed more experiments to examine in detail (**??**).

File Quantity Limits

Git, Mercurial, DMV, and the copy all create one file in their object stores for each input file. So to store 7.5 million files, they will create 7.5 million more, resulting in 15 million files on the filesystem, plus directories. However, the 197 GiB experiment partition has 13 107 200 total inodes, so storing 15 million files is impossible.

Bup is able to store more files because it does not write a separate object file for each input file. Bup aggregates its DAG objects into pack files, writing several large files instead many small files. As such, it does not exhaust the disk's inodes, and can continue until the experiment itself exhausts the system's inodes when it tries to go up from 10 million files to the next step and run a trial with 25 million files.

Hash-Based Directory Names Cause Disk Seeking

Written: Discuss Git and DMV slowing down due to random writes

We saw in the file-size commit times (**??**) that DMV's time increased quadratically, and we suspected that was because it was creating many small files and incurring filesystem overhead. This effect would explain why both Git and DMV do so poorly here while Bup would fare much better. But why then would Mercurial and the copy also have a linear increase instead of an quadratic one?

The difference is the naming schemes of stored files. Git and DMV name each object file according to the SHA-1 hash of the object's contents, while Mercurial, like the copy, uses the original input file's name. This means that Git and DMV write files in a random order with respect to their names, jumping between different object store subdirectories, while Mercurial and the copy can write files in the order they read them, one subdirectory at a time. The filesystem is most likely optimized for that kind of sequential write.

6 Related Works

Ask Otto: Move Related Works section to earlier in paper?

We chose Git because it is the most popular DVCS in use today [1]. We chose the Mercurial and Bup because they are both related to Git but each store data differently. Git and DMV both store objects in an object store directory as a file named for its hash ID. Git has a separate garbage collection step that takes object files and aggregates them into pack files [5, Section 10.7]. Mercurial stores revisions of each file as a base revision followed by a series of deltas [16, Chapter 4], much like previous VCSs such as RCS, CVS, and Subversion [21]. Bup uses Git's exact data model and pack file format, but Bup breaks files into chunks using a rolling hash, reusing Git's tree object as a chunked blob index[1]. Unlike Git, Bup writes to the pack file format directly, without Git's separate commit and pack steps, and without bothering to calculate deltas [18].

Ask Otto: You write "Refs ref? [1?]" here. I'm not sure what you mean. Can you clarify?

Trim and compress the related work section

Better work this into rest of section

Write: Mention COW filesystems

Distributed storage and synchronization systems

Camlistore Camlistore [9] is an open-source project to create a private long-term data storage system for personal users. It allows storage of diverse types of data and it synchronizes between multiple replicas of the data store. However, it eschews normal filesystems and creates its own schemas to store various media.

Dat Data Dat [15] is an open-source project for publishing and sharing scientific data sets for research. This project has a lot of overlap with ours, and several of the core ideas are similar, including breaking files into smaller chunks, and tracking changes via a Git-like DAG. However, their focus is different. The Dat team is concentrating on publishing research data, and making that specific task as simple as possible for non-technical researchers who might not be familiar with version control. By contrast, our project operates at a lower level of abstraction, offering the full power of version control in a very general way, exposing and illuminating the complexities rather than trying to hide them or automate them away.

Where Dat focuses on publishing on the open internet, we focus on ad-hoc networks and data that may be private. Where Dat has components for automating peer discovery and consensus, we work at a lower level, trying to perfect and generalize the storage aspect first. Dat seems to assume that data sets will be small enough to fit on a typical disk on a workstation, while we want to scale even larger.

We hope that our system could be used as a base to build something like Dat, but we intend for DMV to be more general than the Dat core.

Eyo Eyo [24] is system for storing personal media and synchronizing it between devices. It utilizes a Git-like content-addressed object store behind the scenes, but it works more like a networked filesystem than version control. It focuses on organizing media by metadata, which requires agreement on metadata formats, and it requires applications to be rewritten to access files via Eyo rather than the filesystem, both of which are thorny and ambitious problems. We focus purely on storage and synchronization.

Git-Annex and Git-Media Git-annex [11] and Git-media [4] are open-source projects that extend Git with special handling for larger files. Both store information about the larger files in the normal Git repository and then store the files themselves in a separate location. Git-media stores all the larger files in a separate data store which may be remote. Git-annex is more flexible. Annex files may be spread across several different remote repository clones or data stores, and Git-annex has features for tracking the locations of annex files in different remote repositories and moving them from one repository to another. These tracking and distribution features are very similar to our goals. However, Git-annex is not quite as flexible as we aim for with DMV. It considers the large files atomic units, and it does not break them into smaller chunks for de-duplication. Also, because metadata is processed by Git, it has the same limitations that Git does. All repositories must have all metadata, and performance suffers when metadata is too large to fit into RAM.

IPFS: The Interplanetary Filesystem IPFS [2] is an open-source project to create a global content-addressed filesystem. By its global nature, all files are stored publicly, in a global network of nodes with global addressing. IPFS should be an excellent resource for storing published information, but we want DMV to work with private data sets. We want individuals and organizations to be able manage their own data stores privately on their own hardware.

It should be noted that IPFS does have support for storing private objects by way of object-level encryption. However, this seems wasteful of disk space, since small changes in the plain text of a file would completely change the ciphertext, leaving no way to compress the redundancy.

Kademlia Kademlia [14] is an advanced distributed hash table system that updates its network topology information as part of normal lookups. Its focus is on efficient routing between nodes, while we are focusing first on storage.

Rsync Rsync [26] is a utility that synchronizes files across a low-bandwidth network link by using a rolling hash to find similar parts of the file, and then transfer only those portions of the file that have changed. Its rolling hash algorithm is the inspiration for the chunk-splitting algorithms used by Rsyncrypto [23], Bup [18], and DMV (**??**). As a utility, Rsync focuses exclusively on synchronizing files. It does not track history or the relation between files.

Rsync coauthor Andrew Tridgell is also indirectly responsible for the development of Git. His work to reverse-engineer the BitKeeper protocol is what caused BitMover to revoke the special free BitKeeper license for Linux kernel developers, which is what led Linus Torvalds to develop his own source code manager (SCM) [7].

Rsyncrypto Rsyncrypto [23] is an encryption system that uses a rolling hash to encrypt files by chunks, so that the encrypted files are still ”rsync-able.” Normally, a goal of an encryption algorithm is to have small changes in the input lead to a completely different ciphertext. This is good for security, but it negates the advantages of rsync — if the entire file is different, there are no advantages to be had in transferring only the changed parts when synchronizing files. Rsync uses a rolling hash to break the file into chunks and encrypt the chunks separately, so that a small change in the input will only change the

ciphertext for the chunk that contains it, and so rsync can transfer only the chunk that changes.

Rsyncrypto's uses the same rolling hash algorithm as Rsync [23, 26], setting chunk boundaries where the checksum is zero. DMV uses this same chunking algorithm (see **??**).

De-duplicating Storage and Backup

Boar Boar [8] is an open-source project to create a version control system for large binary files. It is one of the main inspirations for our project. It stores file versions in a content-addressed way, and provides de-duplication for large files that only change in small pieces, and it can truncate history to reclaim disk space. However, Boar retreats to the centralized version control paradigm, with a central repository that working directories must connect to in order to check files in or out. We want to provide the advantages of Boar in a flexible distributed version control model. Boar also has practical limitations on repository size and number of files. repositories are assumed to fit on one disk volume, and file metadata is assumed to fit into Ram. DMV tries to avoid both of those limitations.

Bup Bup [17] is an open-source file backup system that is based on Git's repository format. It does many of the things that DMV does: it breaks files into chunks by rolling hash, and it has considerations for metadata that is larger than RAM. Bup is one of the systems we evaluated along with Git and Mercurial (**??**), and we determined that Bup's manner of storing chunks is ideal for the next step for DMV (**??**). However, though Bup is built on version control, it is locked into a backup-based workflow. History is linear and based on clock time of backup. And it assumes that the whole data set and the whole repository can fit onto one filesystem. DMV is built to be more general and versatile.

Time Machine Time Machine [6] is a backup utility from Apple, included in Mac OS X Leopard, that makes frequent automated file hierarchy backups, using filesystem hard links to de-duplicate unchanged files [19]. This de-duplication allows it to store many different backup versions. Time Machine's functionality can be mimicked by using Rsync with the `--link-dest` option, which also hard-links unchanged files during a recursive sync [13].

7 Conclusion

Written: We conclude that the key to storing large files is to break them into smaller chunks, and that the key to storing many small chunks is to aggregate them into larger files.

We have performed experiments to probe the scalability limits of existing DAG-based distributed version control systems. We have shown that the maximum size of file that Git and Mercurial can store is limited by the amount of available memory in the system. We conclude that this is because those systems calculate deltas of files to de-duplicate data, and they load the entire file into memory in order to do so.

We have also rediscovered the limits of the Unix filesystem for storing many small files. We saw that writing files smaller than the filesystem block size incurs storage overhead, that splitting files among too many subdirectories takes inodes that are needed

to store files, and that jumping between directories when writing files incurs write-speed penalties.

We have shown that any VCS that stores objects as individual files on the filesystem will encounter these filesystem limitations as they try to scale in terms of number of files. A VCS that also breaks files into chunks will turn the problem of storing large files into the problem of storing many files, again encountering these limitations. However, the limitations can be avoided by aggregating objects into pack files as Bup does.

Write: We intend to incorporate these insights into future versions of DMV.

The most vital future work would be to go to the distributed case

References

[1] BARUA, A., THOMAS, S. W., and HASSAN, A. E. "What are developers talking about? An analysis of topics and trends in Stack Overflow". In: *Empirical Software Engineering* 19.3 (2014), pp. 619–654. ISSN: 1573-7616. DOI: 10.1007/s10664-012-9231-y. URL: http://dx.doi.org/10.1007/s10664-012-9231-y.

[2] BENET, J. et al. *IPFS: The Interplanetary Filesystem*. GitHub. 2014. URL: https://github.com/ipfs/ipfs.

[3] BREWER, E. "CAP twelve years later: How the 'rules' have changed". In: *Computer* 45.2 (Feb. 2012), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/MC.2012.37.

[4] CHACON, S., LEBEDEV, A., et al. *git-media*. URL: https://github.com/alebedev/git-media.

[5] CHACON, S. and STRAUB, B. *Pro Git*. 2nd. Berkely, CA, USA: Apress, 2014. ISBN: 1484200772, 9781484200773. URL: https://git-scm.com/book/en/v2 (visited on Apr. 27, 2017).

[6] CISLER, P. et al. *System for electronic backup*. US Patent App. 11/499,848. 2008. URL: https://www.google.com/patents/US20080034004.

[7] CLOER, J. *10 Years of Git: An Interview with Git Createor Linus Torvalds*. Linux.com. Apr. 6, 2015. URL: https://www.linux.com/blog/10-years-git-interview-git-creator-linus-torvalds (visited on May 2, 2017).

[8] EKBERG, M. et al. *Boar*. URL: http://www.boarvcs.org/.

[9] FITZPATRICK, B. et al. *Camlistore is your personal storage system for life*. URL: https://camlistore.org/.

[10] FOX, A. and BREWER, E. A. "Harvest, yield, and scalable tolerant systems". In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. 1999, pp. 174–178. DOI: 10.1109/HOTOS.1999.798396.

[11] HESS, J. et al. *git-annex*. 2015. URL: http://git-annex.branchable.com/.

[12] HUMBLE, J. and FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[13] JAKL, M. *Time Machine for every Unix out there*. Blog. Nov. 2007. URL: https://blog.interlinked.org/tutorials/rsync_time_machine.html (visited on May 12, 2017).

[14] MAYMOUNKOV, P. and MAZIÈRES, D. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Peer-to-Peer Systems: First International-Workshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers*. Ed. by DRUSCHEL, P., KAASHOEK, F., and ROWSTRON, A. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-45748-0. DOI: 10.1007/3-540-45748-8_5. URL: http://dx.doi.org/10.1007/3-540-45748-8_5.

[15] OGDEN, M., BUUS, M., MCKELVEY, K., et al. *Dat Data*. URL: http://dat-data.com/.

[16] O'SULLIVAN, B. *Mercurial: The Definitive Guide*. O'Reilly Media, Inc., 2009. ISBN: 0596800673, 9780596800673. URL: http://hgbook.red-bean.com/.

[17] PENNARUN, A., BROWNING, R., et al. *Bup, it backs things up*. URL: https://bup.github.io/ (visited on Apr. 26, 2017).

[18] PENNARUN, A., BROWNING, R., et al. *The Crazy Hacker's Crazy Guide to Bup Craziness*. "DESIGN" document in Bup source code. URL: https://github.com/bup/bup/blob/master/DESIGN (visited on Apr. 26, 2017).

[19] POND, J. *How Time Machine Works its Magic*. Website. Aug. 31, 2013. URL: http://pondini.org/TM/Works.html (visited on May 12, 2017).

[20] RITCHIE, O. M. and THOMPSON, K. "The UNIX time-sharing system". In: *The Bell System Technical Journal* 57.6 (1978), pp. 1905–1929. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1978.tb02136.x.

[21] RUPARELIA, N. B. "The History of Version Control". In: *SIGSOFT Softw. Eng. Notes* 35.1 (Jan. 2010), pp. 5–9. ISSN: 0163-5948. DOI: 10.1145/1668862.1668876. URL: http://doi.acm.org/10.1145/1668862.1668876.

[22] SHAPIRO, M. and PREGUIÇA, N. *Designing a commutative replicated data type*. Research Report RR-6320. INRIA, 2007. URL: https://hal.inria.fr/inria-00177693.

[23] SHEMESH, S. *Rsyncrypto algorithm*. Rsyncrypto home page. URL: https://rsyncrypto.lingnu.com/index.php/Algorithm (visited on Apr. 17, 2017).

[24] STRAUSS, J. et al. "Eyo: Device-transparent Personal Storage". In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'11. Portland, OR: USENIX Association, 2011, pp. 35–35. URL: http://dl.acm.org/citation.cfm?id=2002181.2002216.

[25] TORVALDS, L. *Git - the stupid content tracker*. Git source code README file. From the initial commit of Git's source code into Git itself (revision e83c516). Apr. 8, 2005. URL: https://github.com/git/git/blob/e83c5163316f89bfbde7d9ab23ca2e25604af290/README (visited on Apr. 25, 2017).

[26] TRIDGELL, A. and MACKERRAS, P. *The rsync algorithm*. Tech. rep. Australian National University, 1996. URL: http://hdl.handle.net/1885/40765 (visited on May 12, 2017).

[27] WINSLOW, R. *Don't ever commit binary files to Git! Or what to do if you do.* The Blog of Robin. June 11, 2013. URL: https://robinwinslow.uk/2013/06/11/dont-ever-commit-binary-files-to-git/ (visited on May 11, 2017).

Find places where more citations can be added

Fix all broken links in text

Todo list

Scratch Pad

This section contains text that I have written, but decided to set aside for now. It should disappear if the `final` option is applied to the document.

CRDTS There are also data types are cleverly designed to be commutative, so that the resulting data will be the same regardless of the order in which updates are applied [22].