Scalability of Distributed Version Control Systems

Michael J. Murphy John Markus Bjørndalen Otto J. Anshus

November 2017

Abstract

Distributed version control systems are popular for storing source code, but they are notoriously ill suited for storing large binary files.

We report on the results from a set of experiments designed to characterise the behaviour of some widely used distributed version control systems with respect to scaling. 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 build a distributed storage system with characteristics similar to version control but for much larger data sets. An early prototype of such a system, Distributed Media Versioning (DMV), is briefly described and compared with Git, 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.

The next-phase strategy for DMV will be to break files into chunks by content for de-duplication, then re-aggregating the chunks into append-only log files for low-latency write operations and efficient use of disk space.

1 Introduction

The CAP-theorem [9] 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.

A distributed version control system (DVCS) is a small-scale distributed system, where nodes are autonomous. Rather than a set of nodes that is connected by default, a DVCS's repositories are self-contained and offline by default. A DVCS allows writes to local data at any time, and only connects to other repositories intermittently to exchange updates at the user's command. Concurrent updates are not only allowed but embraced

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

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.

Version control systems are historically designed primarily to store program source code [18], 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” [22], even as the modern software development practice of continuous delivery was commanding teams to ”keep absolutely everything in version control.” [11, p.33]

This paper evaluates the behavior of current version control systems 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 Related Works

The primary related works are distributed version control systems such as Git and Mercurial, which are the inspiration for DMV. DMV hopes to expand on the distributed version control concept with better handling of large binary files, and the ability to distribute the repository itself.

There are existing projects that extend Git with special handling for larger files, such as Git-annex [10] and Git-media [4]. Both store information about the larger files in the normal Git repository and then store the files themselves in a separate location. Git-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 for DMV However, Git-annex is not quite as flexible as we aim for with Distributed Media Versioning (DMV). It considers the large files atomic units, and it does not break them into smaller chunks for de-duplication.

Boar [7] and Bup [15] are open-source backup systems based on version control. Bup even uses Git’s data format. They are also an inspiration for DMV. Both use a rolling hash algorithm to break files into chunks by content. However, both are focused on a backup workflow. Both also assume that the whole repository will fit on one filesystem, and both have limited distributed capabilities. Apple’s Time Machine [6] is another backup system that de-duplicates unchanged files, but it uses filesystem hardlinks rather than content-addressing. Time Machine’s functionality can be mimicked by using Rsync with the `--link-dest` option [12].

Dat [13] is an open-source project for publishing and sharing scientific data sets for research. IPFS [2] is an open-source project to create a global content-addressed filesystem. Both use content addressing to keep versions of content. Where Dat and IPFS focus on publishing on the open internet, DMV will focus on ad-hoc networks and data that may be private.

Camlistore [8] and Eyo [19] are systems for storing personal media collections. Both eschew traditional filesystems for databases to store various media types and their metadata. Eyo in particular leans on the insight that in media files, the metadata is more likely to change than the image/audio/video data. And so it separates metadata from data. This allows efficient storage and syncing of metadata. However, it requires that the software be able to parse many different media formats, and it requires client software to be rewritten to open the separate metadata and data streams. A rolling hash will not be likely to find these exact seams between changing metadata and static data. So it will not compress data as well. However, it will avoid additional developer effort to support new file formats.

Rsync [21] is the origin of the rolling hash algorithm that DMV and Bup use.

3 Distributed Media Versioning

Git stores data in a directed acyclic graph (DAG) data structure [20]. Each version of each file is hashed with the cryptographic SHA-1 digest, becoming a blob (binary large 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. This list is called a tree object, and it is also hashed and stored by its SHA-1 hash ID. It is called a tree because tree objects can also refer to other trees, representing subdirectories. Commit objects contain the hash ID of the tree object that represents the directory state at the time of commit, plus metadata such as the author and commit message. It too is hashed and stored by ID. These objects form a graph with directed links from one object to another and no circular references (cycles), a directed acyclic graph. It is the content-addressing that makes cycles impossible. 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

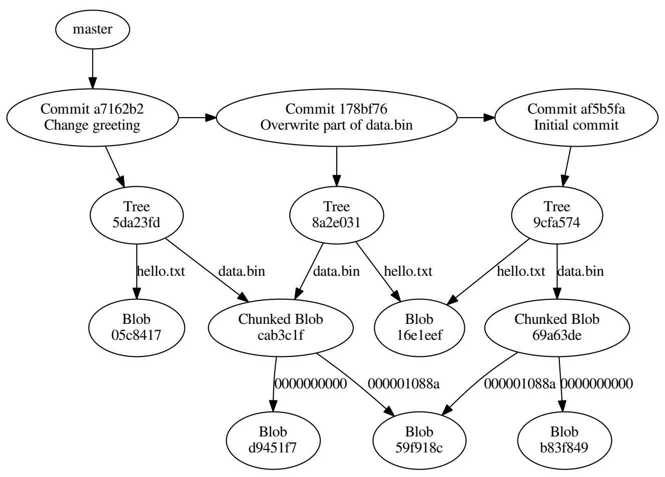


Figure 1: A simple DMV DAG with three commits

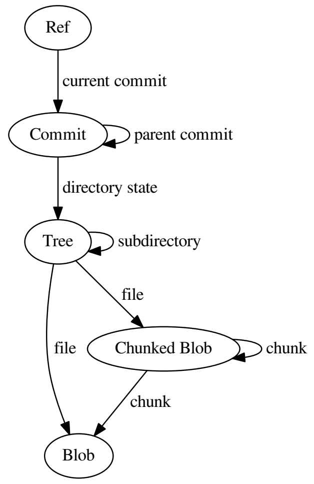


Figure 2: DMV DAG Object Types

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 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 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.

DMV's DAG is based on Git's, but it adds a new object type, the chunked blob, which represents a blob 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 [21]. 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.

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/` .

4 Experiments

Methodology

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

We ran each experiment with four different VCSs: Git, Mercurial, Bup, and the DMV prototype. We chose Git because it is the most popular DVCS in use today [1] and the main inspiration for DMV. 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 [14, Chapter 4], much like older systems such as RCS, CVS, and Subversion [18]. Bup uses Git's exact data model and pack file format. However, 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 [16]. As a control, we also ran the experiments with a dummy VCS that simply copied the files to a hidden directory.

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,

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

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).

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.

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.

Software versions used where Git 2.1.4, Mercurial 3.1.2, Bup debian/0.25-1. The DMV prototype was compiled from version c9baf3a in the DMV source Git repository. The dummy copy VCS simply used the `cp` utility bundled with Debian, GNU `cp` version 8.23.

5 Results

File Size Limits: RAM, Time, Disk Space

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 printed 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 printed an error message saying "fatal: Out of memory, malloc failed (tried to allocate 12 884 901 889 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.

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. We measured total size, which included input data. With

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

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, the base revision, and a negligible delta). At 1.5 GiB and above, the repository size remained approximately three times the input data size after garbage collection. So Git's garbage collection was silently failing with larger files. 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.

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. And so every system tested ran out of disk space while trying to commit the 128 GiB file, because each system saves a copy of the file during commit.

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

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. Files up to 8 MiB were committed in under 1 s for all systems. So one should be able to keep a small number images or short audio files in version control and still have reasonable interactive response times.

File Quantity Limits: Inodes, 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 [17]. 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 `mke2fs`. However, our experiment partitions used the default setting, giving the 197 GiB partitions 13 107 200 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, almost 2 million more than the 13.1 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 while attempting to generate 25 million input files.

Bup also made more efficient use of disk space. The input files were 1 KiB, while the filesystem's block size was the default 4 KiB. Therefore, the input data set used four times the amount of disk space as it needed. Because Git, Mercurial, DMV, and the copy control all made one new file for each input file, the commit used another 4 KiB for each file, for a total of approximately eight times the disk space used. The total size measured in the Bup experiments, including input data, was 5.374 times the size of the input data at 5 million files. And since the input files themselves account for just over 4 times the theoretical size, we can see that Bup is storing the data in a form that is much closer to its theoretical size, taking just under 1.374 times the space.

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. This is due to the way that each system stores objects. Git and DMV store objects with directory and files names taken from the object's SHA-1 hash ID, so they are effectively random. These randomized writes jump from directory to directory. So rather than updating a single directory with multiple new files, the write operate must jump to another directory and open it for appending. This slows both Git and DMV down significantly as the number of objects increases. Mercurial stores data in files named after the original input files, so they can be written in the order that they are read, without all the jumping back and forth between directories. Bup also breaks large files into chunks, but it aggregates objects into pack files. So in addition to conserving inodes, Bup's writes are sequential appends to a single file. Disks and filesystems are optimized for this kind of sequential write, and so Bup had the fastest commit times of all systems tested.

6 Discussion

Reading Whole Files into RAM Limits File Size to RAM

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

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

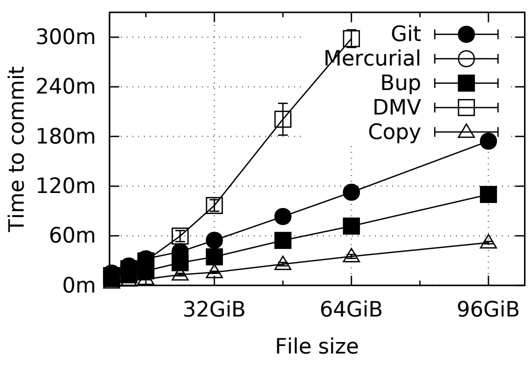
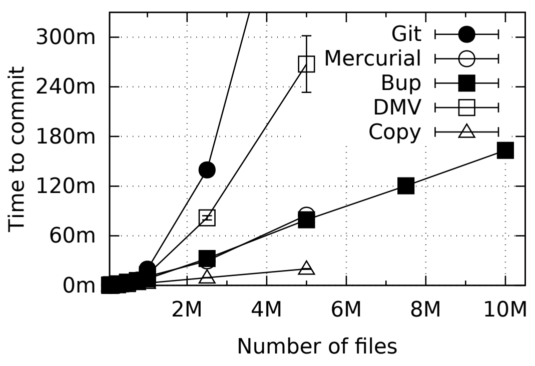


Figure 4: Wall-clock time to commit many 1KiB files to a fresh repository



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.

These limits could be circumvented by increasing RAM, or by implementing the diff algorithms in a streaming fashion that does not load the whole file at once.

Both DMV and Bup avoid these pitfalls by operating with a finer granularity than the file, 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 practically unlimited. Dividing into chunks by rolling hash also makes delta compression unnecessary, because identical chunks in different files or file revisions will naturally deduplicate. From there, it is the method of object storage that becomes the bottleneck.

Naming Files by Hash Leads to Inefficient Writes

The way DMV's commit time increased super-linearly with file size is due to the way it breaks the large file into chunks and stores objects as individual files on the filesystem. It turns the problem of storing one large file into the problem of storing many small files. And so its performance characteristic is closer to that of storing many files. And both Git's and DMV's commit times increased super-linearly with the number of files increases, due to their randomized directory and file names. Mercurial's many-files commit times were faster, with many files written sequentially, without jumping between directories. Bup's appends to pack files where faster still, though the speed gains over Mercurial were marginal.

Storing Many Small Files Leads to Inefficient Use of Disk Space

Git, Mercurial, DMV, and the copy control all create one file in their object stores for each input file. So they all used up the filesystem's available inodes when storing millions of files. Bup, with its aggregated storage, did not. Storing many small files also makes less efficient use of disk space when the file sizes are smaller than the filesystem's block size. With immutable stored objects and an append-only history, the usage pattern of version control does not require room for objects to grow. Therefore it makes sense to aggregate objects together into larger files.

7 Conclusion

We have performed experiments to probe the scalability limits of Git, Mercurial, Bup, and our DMV prototype. 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 their implementations load the entire file into memory in order to do so. Better diff algorithm implementations could work around this problem. However, a rolling hash algorithm can also work around this problem by finding duplicate content at a finer granularity than the file. Using those chunks as objects would make use of the DAG's natural de-duplication of identical objects, and make the secondary compression less necessary.

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 a VCS that stores objects as individual files on the filesystem will encounter these 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 archives as Bup does.

8 Future Work

The most vital future step for DMV is to implement the planned network features, to move DMV from the local case to the distributed case. We would also incorporate the performance insights from these experiments and switch DMV to an aggregating storage format. An important disk-space optimization would be to create a virtual copy-on-write filesystem that is a view into a tree in the DMV repository. A virtual filesystem could be used as the working directory, eliminating the wasted disk space of having a second writable copy of all files, and it would eliminate the copying of those files back into the immutable data store on commit. It would also be interesting to perform these experiments on other filesystems and disk hardware to see how they behave. It would be particularly interesting to see if solid state drives experience the same slowdown with randomized writes. We would also like to measure the compression of files when broken up by rolling hash and stored in a DAG, using a sampling of real-world data in common image, audio, and video formats.

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] Ekberg, M. et al. *Boar*. URL: `http://www.boarvcs.org/`.

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

[9] 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`.

[10] Hess, J. et al. *git-annex*. 2015. URL: `http://git-annex.branchable.com/`.

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

[12] 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).

[13] Ogden, M., Buus, M., McKelvey, K., et al. *Dat Data*. URL: `http://dat-data.com/`.

[14] O’Sullivan, B. *Mercurial: The Definitive Guide*. O’Reilly Media, Inc., 2009. ISBN: 0596800673, 9780596800673. URL: `http://hgbook.red-bean.com/`.

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

[16] 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).

[17] 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`.

[18] 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.

[19] 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.

[20] 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).

[21] 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).

[22] 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).