UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

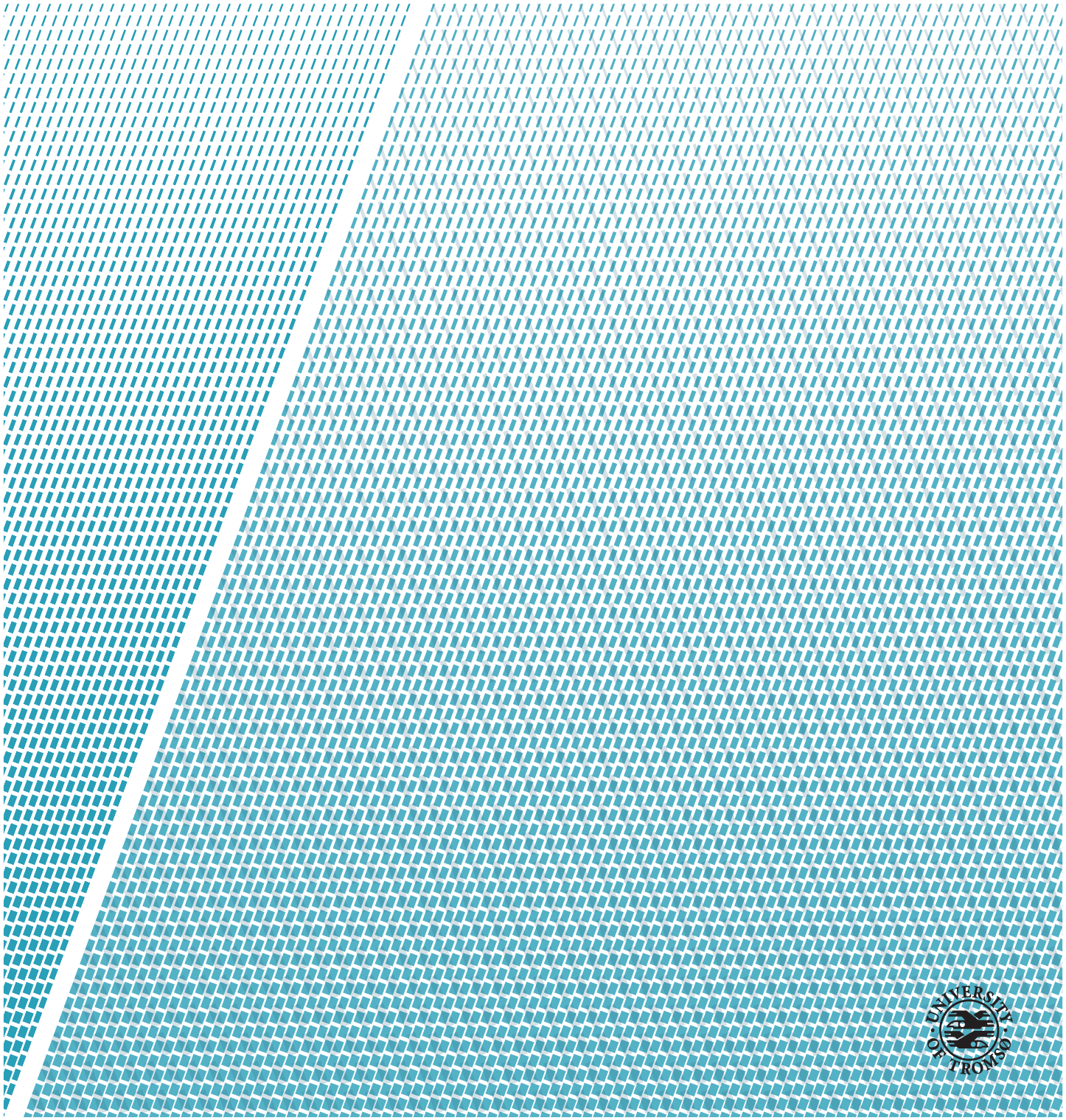
Faculty of Science and Technology / Department of Computer Science

Distributed Media Versioning

—

Michael J. Murphy

Master's Thesis in Computer Science [INF-3990] – May 2017



Abstract

It is still strangely difficult to backup and synchronize data. Cloud computing solves the problem by centralizing everything and letting someone else handle the backups. But what about situations with low connectivity or sensitive data?

For this, software developers have an interesting distributed, decentralized, and partition-tolerant data storage system right at their fingertips: distributed version control.

Inspired by distributed version control, we have researched and developed a prototype for a scalable high-availability system called *Distributed Media Versioning (DMV)*. DMV expands Git's data model to allow files to be broken into more digestible chunks via a rolling hash. DMV will also allow data to be sharded according to data locality needs, slicing the data set in space (subset of data with full history), time (subset of history for full data set), or both. DMV repositories will be able to read and to update any subset of the data that they have locally, and then synchronize with other repositories in an ad-hoc network.

We have performed experiments to probe the scalability limits of existing version control systems, specifically what happens as file sizes grow ever larger or as the number of files grow. We found that processing files whole limits maximum file size to what can fit in RAM, and that storing millions of objects loose as files with hash-based names incurs disk space overhead and write speed penalties. We have observed a system needing 24 s to store a 6.8 KiB file.

We conclude that the key to storing large files is the break them into many small chunks, and that the key to storing many chunks is to aggregate them into pack files. And though the current DMV prototype does only the former, we have a clear path forward as we continue our work.

Acknowledgements

I would like to thank my advisor, Otto J. Anshus, for his guidance, encouragement, and invaluable feedback, Ken Arne Jensen for finding free computers to run experiments on, and Jan Fuglesteg for guiding me through administrative matters, answering the quotidian questions of a thirty-something who had forgotten much of what he knew about how universities work.

I would also like to thank Rolf Ole Jenssen and Coleman McFarland for their feedback on early drafts, and Sergiusz Michalik and Lars Dalheim for their friendship and encouragement.

Thank you to my former supervisor, Pierce Hanley, who taught me most of what I know about programming and who was always fun to argue with.

Moving to the arctic was my dream, and computer science is my passion. Thank you to the University of Tromsø for giving me the opportunity to pursue both at the same time.

Finally, I would like to thank Katrine Mellem for interrupting me during a lunchtime discussion about US elections.

“I live in DC, so ——”
“No! You live in Tromsø!”

That quip stands as the single most welcoming thing a local has ever said to me, and it marks the moment I started feeling like I truly had a place here.

Takk.

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 CAP Theorem and the Importance of Availability 3
- 1.2 Version Control, Git, and the DAG 4
  - 1.2.1 How Data is Stored in Git 5
  - 1.2.2 The Power of the DAG 7

2 Idea: Distributed Media Versioning (DMV) 11
- 2.1 Characteristics 11
- 2.2 Security Model 14
- 2.3 What's in a Name? 14

3 DMV Architecture 17

4 DMV Design 21
- 4.1 DMV's DAG 21
- 4.2 Working with an Incomplete DAG 23

5 DMV Implementation 27
- 5.1 Command Line Control, Explicit operations 27
- 5.2 Rust 29
- 5.3 Working Directory and Object Store 29
- 5.4 Chunking Algorithm 29

6 VCS Scaling Experiments **31**
6.1 Version Control Systems Evaluated 31
6.2 Procedure 32
6.3 Automation and Measurement 33
6.4 Experiment Platform 35
6.5 Results: File Size 37
6.5.1 File Size Limits 37
6.5.2 Time for File-Size Initial Commit 41
6.5.3 Time for File-Size Update Commit 43
6.5.4 CPU Usage During File-Size Commits 45
6.5.5 Repository Size after File-Size Update Commit . . . 49
6.6 Results: Number of Files 51
6.6.1 File Quantity Limits 51
6.6.2 Time for Number-of-Files Initial Commit 53
6.6.3 Time for Number-of-Files Update Commit 55
6.6.4 CPU Usage During Number-of-Files Commits . . . 57
6.6.5 Time for Number-of-Files Status Check 59
6.6.6 Repository Size after Number-of-Files Update Commit 61

7 Performance Tuning Experiments **63**
7.1 Object Store Directory Layout 63
7.1.1 Procedure 65
7.1.2 Environment 66
7.1.3 Results 67
7.1.4 Object Directory Layouts in Action 70
7.2 Linux I/O Scheduler 72
7.3 Chunk Size 74
7.3.1 Procedure 74
7.3.2 Environment 74
7.3.3 Window Reset Bug 74
7.3.4 Results 75
7.3.5 Chunk Sizes in Action 78

8 Discussion **81**
8.1 Data Granularity and Storage Schemes 81
8.2 Subtleties of the Rolling Hash 83
8.3 DMV Prototype development 84
8.4 Aggregating Data about a Sharded DAG 84
8.5 Potential Applications of DMV 86
8.6 What DMV should not do 86

9 Related Works **89**
9.1 Distributed storage and synchronization systems 89
9.2 De-duplicating Storage and Backup 92

10 Conclusion and Summary of Contributions **95**

11 Future Work **97**

Glossary **99**

Bibliography **103**

List of Figures

1.1 A simple file hierarchy represented by Git tree and blob objects and their SHA-1 hash IDs 6
1.2 A simple Git DAG with three commits 8
1.3 Git DAG Object Types 8

3.1 Repositories, object stores, and working directories 17
3.2 Repositories in an ad-hoc network 18

4.1 A simple DMV DAG with three commits 22
4.2 DMV DAG Object Types 22
4.3 A DMV DAG, sliced in different dimensions 24
4.4 Speculative DMV ls output showing remote files 24

5.1 DMV help output, listing subcommands 28
5.2 Example object file name 29

6.1 Wall-clock time to commit one large file to a fresh repository 40
6.2 Wall-clock time to commit one updated large file 42
6.3 CPU utilization while committing one large file to a fresh repository 44
6.4 CPU utilization while committing changes to one large file 44
6.5 Total repository size after committing, updating, and committing again 48
6.6 Wall-clock time to commit many 1KiB files to a fresh repository 52
6.7 Wall-clock time to commit many updated files 54
6.8 CPU utilization while committing many 1KiB files to a fresh repository 56
6.9 CPU utilization while committing many 1KiB files after one of every 16 files has been updated 56

6.10 Real time required to check the status of many files after update 58
6.11 Total repository size after committing, updating, and committing again 60

7.1 DMV output showing varying object write times 64
7.2 Number of Files vs. number of directories filling a disk . . 66
7.3 Unusually high write times 69
7.4 Time to commit one large file, with different object directory schemes 70
7.5 Time for DMV prototype to commit an increasing number of 1KiB files to a fresh repository, by I/O scheduler . . 73
7.6 Mean chunk size 76
7.7 Mean chunk size, with reset bug 76
7.8 Chunk sizes in action 79

8.1 DMV branching and merging functionality 85

List of Tables

6.1 Version control systems evaluated and their versions . . . 34
6.2 Experiment computer specifications 34
6.3 Observations as file size increases 39
6.4 Effective size limits for VCSs evaluated 39
6.5 Selected CPU usage data for copy operation 47
6.6 Bup initial commit times with unusually high variance . . 53

7.1 Sample object store directory variations 64
7.2 Top-ten longest writes 69
7.3 Chunk sizes for a window size of 4096 75
7.4 DMV versions examined with different rolling hash configurations 79

Chapter 1

Introduction

It is still strangely difficult to keep backups and synchronize data. Many of us have several computers, perhaps a laptop, a phone, and a work computer, and we would like to synchronize data between them. We want to keep a Word document synchronized between home and work. We want to put new music on our phones, and pull photos off of camera SD cards. We have backups on removable drives, but we don't remember when it was that we last did a backup, or what is new since then. We have these sets of files that tend to fragment themselves across our devices, and we lose track of what is where.

Cloud computing promises to centralize and safeguard our data, keeping it all in one place and taking care of the backups for us. Google Drive gives us a shared document that many people can edit in real time. Spotify offers endless music streams. Instagram lets us save and share photos. Dropbox gives us a centralized cloud filesystem. But many of these solutions are specialized applications for specific media, which can limit their general usefulness. Most require constant network connectivity, making them ill-suited for intermittent or high-latency connections. And all require entrusting your data to a third-party service, which raises concerns about privacy and storage longevity.

Why can't we simply track the files we have across the devices we have?

Software developers have an excellent system for backup and sync right at their fingertips: *distributed version control systems (DVCSs)*, such as Git and Mercurial. Version control systems (VCSs) track all changes

to a collection of files, allowing collaborators to work independently and then synchronize and share their work. Additionally, in a DVCS, every collaborator has a full copy of the project's history. That redundancy not only allows collaborators to work offline, but it also functions as a backup. Linus Torvalds, the creator of Linux and Git, once famously joked that he doesn't keep backups, he simply publishes his work on the internet and lets others copy it [41].

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 compound.

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" [44], even as the modern software development practice of continuous delivery was commanding teams to "keep absolutely everything in version control." [17, p.33]

> Every single artifact related to the creation of your software should be under version control. Developers should use it for source code, of course, but also for tests, database scripts, build and deployment scripts, documentation, libraries and configuration files for your application, your compiler and collection of tools, and so on — so that a new member of your team can start working from scratch. [17, p.33]

What if we could generalize the distributed version control concept to store a wider variety of file sizes, from kilobyte text files to multi-gigabyte videos? In addition, what if we relaxed the assumption that every replica contain the complete history, and allowed each replica to choose what subsets of the files and the history to store, according to the replica's capacity and need? The answer could be a new abstraction for tracking a data set and its history as a cohesive whole, even when it is physically spread over many different nodes.

1.1 CAP Theorem and the Importance of Availability

Traditional databases that operated on one powerful server could focus on data integrity and the *ACID* guarantees: *atomicity, consistency, durability, and isolation* [35, Chapter 1]. As demand increased, the server would be scaled up, beefing up the server hardware with more disk space, more RAM, and more and faster CPUs [35, Chapter 4]. But there is a limit to scaling up the hardware of a single machine, and as data kept growing, a new wave of systems appeared that scaled *out* to many commodity servers, distributed systems [35, Chapter 4].

Spreading data across different computers creates new problems of synchronization. How does one ensure that replicated data is updated correctly on all replicas? How can separate machines agree on the order of updates?

Distributed systems are ruled by the *CAP-theorem* [14], which states that a system cannot be completely consistent (C), available (A), and tolerant of network partitions (P) all at the same time. When communication between replicas 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 aimed at improving consistency. Vector clocks [20] and consensus algorithms such as Paxos [21, 43] make sure the same updates are applied in the same order on all replicas even, if a minority of nodes cannot respond. 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 [37]. But in general, when systems cannot communicate, the CAP theorem cannot be avoided [15], and the system is still faced with the partition decision.

Amazon's Dynamo [11] was a pioneer in relaxing consistency guarantees to ensure availability. Dynamo is a key-value store that can accept updates to a value even if not all replicas respond. This can lead to inconsistencies, but for a global e-commerce website like Amazon, any outage represents lost revenue and so availability is paramount. Dynamo's answer to the partition decision is to always proceed.

When multiple Dynamo replicas receive updates to the same value and the order of those updates cannot be determined, Dynamo keeps the different versions of the value and presents them together during a read. That way, the higher-level application that is using Dynamo as a data store can resolve the conflict and write a new, reconciled value. Dynamo recognizes the *end-to-end argument* [36] that conflicting updates cannot be resolved generally by a storage platform or network protocol. Resolution is dependent on the structure of the data and on the needs of the application using it.

Though maybe not designed with the CAP theorem explicitly in mind, a DVCS is in fact a small-scale distributed system that takes the availability-first approach to the extreme. 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.

The distributed version control concept may have something to teach larger-scale systems about availability.

1.2 Version Control, Git, and the DAG

The first version control systems were *source code managers (SCMs)* created as a way to efficiently store different versions of a source code files by encoding them as a series of deltas [34]. CVS introduced a collaborative client-server model [34, 6]. Subversion kept the client-server model but began focusing on the versions of the whole collection of files together, rather than on individual files [34, 30]. Doing so made branching easier, and branching quickly became a key feature for collaborative work.

BitKeeper, a commercial SCM, pioneered the distributed version control concept by giving each developer a local copy of the whole repository, allowing local writes, and then making it easy to push the local changes to a central server [34, 10]. This feature was so important

that Linus Torvalds, the creator of Linux, adopted BitKeeper as the main source code manager for the Linux kernel, despite the licensing concerns of using a proprietary tool to develop an open-source project. When the licensing trouble came to a head and BitKeeper was no longer an option, Torvalds, unsatisfied with all other SCMs available at the time, wrote his own: Git. [34, 10]

The ability to always write locally separates concerns such as handling security from the underlying problem of storing the data — repositories are private by default unless they are specifically hosted on a public server; any user can write to their own repositories; and any ad-hoc group of users can exchange updates. Each developer can decide what updates they want to incorporate into their particular repository, and the group of developers can decide which repositories and which versions are official and what is to be included into official releases. These are human questions that groups of collaborators can solve by arranging their networks and policies how they see fit, enabled by the tool rather than constrained by it. As Torvalds put it:

> The big thing about distributed source control is that it makes one of the main issues with SCMs go away — the politics around "who can make changes." BK [BitKeeper] showed that you can avoid that by just giving everybody their own source repository. [10]

Since Git's release, it has quickly become the dominant VCS in use [1], making distributed version control the dominant paradigm for source code management.

1.2.1 How Data is Stored in Git

One of the key aspects of Git is that all data — files, directories, and history — is stored according to its content; it is *content addressable storage*.

When Git stores a file, it creates a *blob (binary large object)* by copying the file's content and prepending a short header. Git then calculates the SHA-1 hash of the blob, and stores the blob in an *object store*, using the SHA-1 hash as the blob's ID. To store a directory, Git creates a *tree* object that maps filenames to SHA-1 blob IDs for each file in the directory. This tree object is also stored in the object store with its SHA-1

hash as its ID. Tree objects can refer not only to blobs, but to other trees, representing subdirectories.

Thus, a file hierarchy in a given state is represented by a hash tree, with tree objects as nodes and blobs as leaves, and the entire state can be referred to by a single hash ID, that of the top-level tree object. A simple example is shown in Figure 1.1.

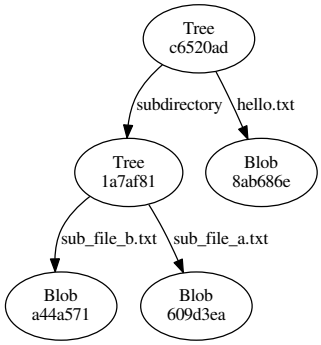


Figure 1.1: A simple file hierarchy represented by Git tree and blob objects and their SHA-1 hash IDs

Git then links different file hierarchy states with *commit* objects. A commit object includes a hash ID for a tree, representing the state of the file hierarchy, and one or more hash IDs of parent commits, representing the previous states that this one was built from. Like with blobs and trees, the commit object is also hashed and stored with the hash as its ID.

The resulting data structure is a directed graph: new commits point to previous commits, and each commit points to a tree that represents the state of the file set at the time. This graph is also append-only: objects are immutable and are referenced by cryptographic hash of their content, which includes the hashes of all objects that they depend on. So new objects can only refer to existing objects, which makes the graph acyclic. Storing history in this way will naturally de-duplicate unchanged files and directories, because the resulting blobs and trees will be identical in content and thus have the same hash ID. This *directed, acyclic graph* structure is referred to as a *DAG*.

A DAG can be uniquely identified by the hash IDs of those commits which do not yet have child commits to refer to them. Such commits

are the *heads* of each current branch of development. A simple example with one head and three commits is shown in Figure 1.2.

A Git repository, then, is a collection of the blob, tree, and commit objects that make up the file set's history, stored by hash ID in the object store, with *references (refs)* to the commits at the head of each branch [40]. These object types and their relationships are shown in Figure 1.3.

1.2.2 The Power of the DAG

Such a DAG has many properties that make it useful for distributed collaborative work and for long-term data storage.

De-duplication As noted above, unchanged and duplicate files are naturally de-duplicated by the DAG's content-addressing: if two files are identical, they will have the same hash and thus be the same blob.

Distributability and Availability Because the DAG is immutable and append-only, it can be replicated simply by copying all of its objects. Any replica can make its own updates by appending to the DAG. Rather than have a centralized notion of a "current" version, development in Git naturally diverges into different branches, as different users with their own replica of the DAG make changes. Branches created on one replica can be synchronized to another simply by comparing sets of objects and transferring new objects that that the other does not have. Branches are reconciled with a merge operation, creating a new commit that incorporates changes from both branches.

Atomicity The DAG's append-only nature makes commits atomic. Objects are added to the object store frst, and then once all necessary objects are stored, the ref can be updated to point to the new commit object. The ref is the size of the hash digest (in Git's case, 160 bytes for an SHA-1 hash), so it can be updated atomically. If the ref is updated, the commit was successful. The objects themselves do not have to be added to the object store atomically because their presence does not change the existing DAG. An interrupted object transfer may leave orphaned objects in the object store, but it cannot corrupt previously-written data, nor can

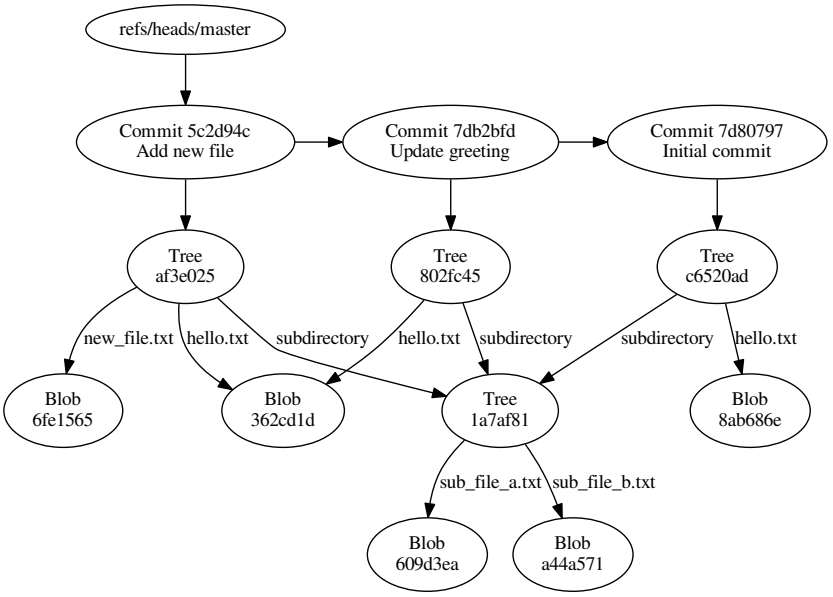


Figure 1.2: A simple Git DAG with three commits

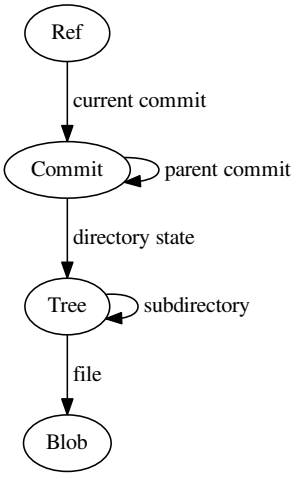


Figure 1.3: Git DAG Object Types

it leave the repository in an inconsistent state. Orphaned objects can be swept up during a garbage-collection phase, walking the DAG and marking all objects that are reachable from the current refs.

Verifiability Perhaps most importantly, since objects are identified by a cryptographic hash, data integrity can be verified at any time by re-computing an object's hash and comparing it to its ID. Corrupt objects can be replaced with an intact copy from another replica.

The main weakness of Git's DAG is that blobs and files are one and the same. This makes the file the unit of de-duplication, which can lead to inefficient storage of larger files. Git gets around this by *packing* objects during its garbage-collection phase, storing similar objects as bases and deltas behind the scenes. But this is an optimization.

Calculating deltas during this packing phase requires loading the objects into memory, and so it can cause an out-of-memory error if an object is too large to fit into available RAM. Because Git stores files whole in blobs, it cannot pack objects that are larger than available RAM.

If the DAG operated at a granularity smaller than the file, it could become even more powerful. It could naturally de-duplicate chunks of files the way that Git already de-duplicates whole files, and it could ensure that all objects fit into RAM for packing or other operations.

This sub-file granularity and de-duplication is the core idea behind our new data storage system, *Distributed Media Versioning*.

Chapter 2

Idea: Distributed Media Versioning (DMV)

Distributed Media Versioning (DMV) is our new distributed data storage platform. 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 ultimate goal is to create a new abstraction, of *many devices, one data item* in varying states of synchronization.

2.1 Characteristics

General storage

DMV is a generalized storage platform that places no restriction on file types. Its data model is a classic hierarchical filesystem, but with history. Applications on each node can read and write to the files via the

filesystem as normal. DMV is dogmatic about the end-to-end argument [36], realizing that it cannot anticipate all the needs of end users and applications. So it aims to be as general and neutral as possible, focusing on the core task of storage and tracking, and providing a platform for other applications to build on.

Inspired by version control

DMV is inspired by distributed version control systems. Its core data structure is a DAG, based on Git's but modified to store a wider range of file sizes. The key difference is that large files are broken into smaller chunks (around 19 KiB), which is what ensures that no single object is too large for memory or disk. Breaking files into chunks also allows the data structure to naturally de-duplicate parts of files that do not change. For example, if a large media file has its metadata block updated, only the chunk containing the updated metadata is new. The other chunks will simply be reused.

Scalability

DMV can store a wide variety of file sizes and file quantities. Current VCSs load entire files into memory to calculate deltas, which limits the maximum size of files they can store to what can fit into RAM. Additionally, they create separate files for each input file, leading to disk-space overhead and write-speed slowdowns as the number of files on one filesystem increases. DMV successfully avoids the file-size limitation, but as currently described and implemented, it does succumb to the number-of-files limitations. However, we know of an effective method to avoid the number-of-files limitations in the future. These limitations are explored experimentally in chapter 6.

Additionally, the stored data set can also be sharded, so that the data set as a whole can scale to sizes too large to fit on any one computer in the network.

Versatility

DMV will be able to run on a wide range of hardware. The current prototype runs on Linux PCs, but it is designed with an eye towards running on low-powered and mobile devices such as mobile phones or sensor networks.

Configurable sharding

The DAG structure tracks the data set in three dimensions:

1. The set of files themselves
2. The history of the files
3. The parallel branches of development in the history

Traditional DVCSs tend to assume that each repository has the full history of all files, though not every branch of development. In contrast, each DMV node will store a subset of the data, sharded along any of those dimensions, configurable by the user. A node could keep the full history of only a small subset of files, or only the most recent few versions of the full set of files, or only a few branches, or any combination.

Availability

Like in a DVCS, the DAG structure records all history of the data set and allows many different branches of development to exist in parallel. This allows high availability. Any node can always make updates autonomously, without coordinating with other nodes. Reconciliation of conflicting writes happens later — via explicit, user-driven merging. DMV will only require a connection to another node during explicit synchronize operations, and so it should be well-suited for applications with intermittent or high-latency connectivity.

Data integrity

Because the DAG is append-only, and DAG objects are addressed by a cryptographic hash of their contents, this allows DMV to verify stored

objects and detect corrupted data. Corrupted objects can be replaced by retrieving an intact copy from another replica.

DMV should never lose data accidentally. However, because DMV tolerates an incomplete DAG, objects can be deliberately deleted from all nodes to save space or to redact sensitive information.

2.2 Security Model

The DAG's append-only nature and cryptographic content addressing provide protection from tampering. As long as the complete DAG is available, its integrity can be verified. Allowing an incomplete DAG does introduce an opening for tampering, because not all objects are available to verify, but we ignore such a possibility for now. Because DMV is primarily meant to allow individuals or organizations to manage their own data on their own hardware, we assume that all nodes will be under the user's control, and that users will only accept additions to their DAG from trusted collaborators. This defers questions of security to the systems and networks that are running the DMV nodes. Data can be kept private by keeping all DMV nodes on a private network. Though DMV's checksums can be checked to detect tampering, DMV itself has no way to detect unauthorized reads. We also do not consider byzantine failures or guard against malicious nodes at this time.

2.3 What's in a Name?

We chose the name *Distributed Media Versioning* because it is a concise way to describe the system, emphasizing its distributed nature, its roots in version control, and its goal of storing a wide range of media rather than just source code. The acronym DMV makes for a short and easy-to-type base command for command line control, in the grand tradition of `cvs`, `svn`, `git`, and `hg`. And though in many places the acronym is associated with a Department of Motor Vehicles, it does not seem to have any prior conflicting uses in the computing domain.[1] It is also a

[1] Possibly because of negative associations with the Department of Motor Vehicles

nod to the author's home town of Washington DC, where the Washington metropolitan area is sometimes referred to as "The DMV" as it spills out of the District of Columbia and into Maryland and Virgina.

Chapter 3

DMV Architecture

Each DMV node may have one or more *repositories*, each consisting of a content-addressable object store for immutable DAG objects and a *working directory* for active file editing (Figure 3.1). Repositories that are used only for storage may omit the working directory, similar to a Git bare repository.

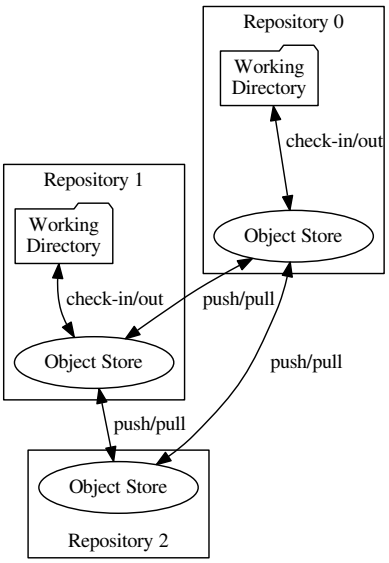


Figure 3.1: Repositories, object stores, and working directories

Each repository is autonomous, storing a portion of the DAG, and able to add to it at any time. However, it can transfer DAG objects to and from other repositories, and it can cache data about what DAG objects are available at a remote repository. Thus, DMV forms an ad-hoc, unstructured network of repositories, and each repository can inform the user about what data is available where. Repositories may exist on separate computers or coexist on the same computer. Some repositories may be on removable storage and accessed only when that storage is connected.

Together, the repositories hold the entire data set — or the portion of the data set that has not been intentionally deleted — in varying states of sharding and replication. An example ad-hoc network of repositories is shown in Figure 3.2.

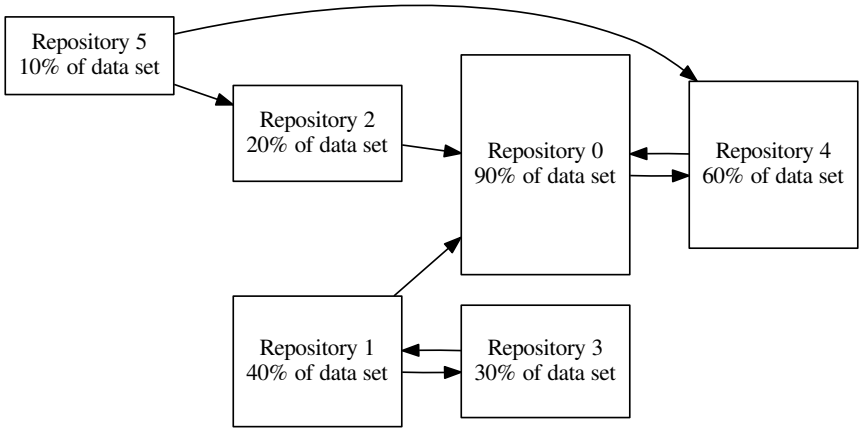


Figure 3.2: Repositories in an ad-hoc network

DMV assumes that each repository will connect to a human-scale number of other repositories, maybe tens or hundreds. DMV does not dictate network structure. The user or a higher-level application may determine the network topology and workflow according to their needs.

DMV operates as a command-line utility, performing disk operations and acting as a client to remote repositories according to explicit user commands. The command-line utility is built as a thin shell around a library, so that applications can use it as well. An optional server

component will listen for incoming connections from the command-line utility.

Chapter 4

DMV Design

4.1 DMV's DAG

DMV's data set and its history are represented as a DAG (directed acyclic graph, see subsection 1.2.1), and the rest of the design flows from that. 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 4.1.

The object types that make up the DMV DAG are as follows:

Blob As with Git, a blob simply holds binary data.

Chunked blob Unlike with Git, larger blobs in DMV are broken into chunks. Each chunked blob is an index of the blobs (or other chunked blobs) that make up the larger blob. A file that is stored in the system may be represented by a blob (if it is only one chunk) or a chunked blob.

Tree As with Git, a directory that is stored in the system is represented by a tree object. The tree refers to the blob, chunked blob, or tree that represents each file or subdirectory, along with metadata such as the filename.

Commit As with Git, a commit represents a given state of the data set. It refers to the tree that represents the top-level directory of the data set at that state, along with metadata such as author, date, and description. It also refers to the previous commit (or commits) that represented the previous state of the data set.

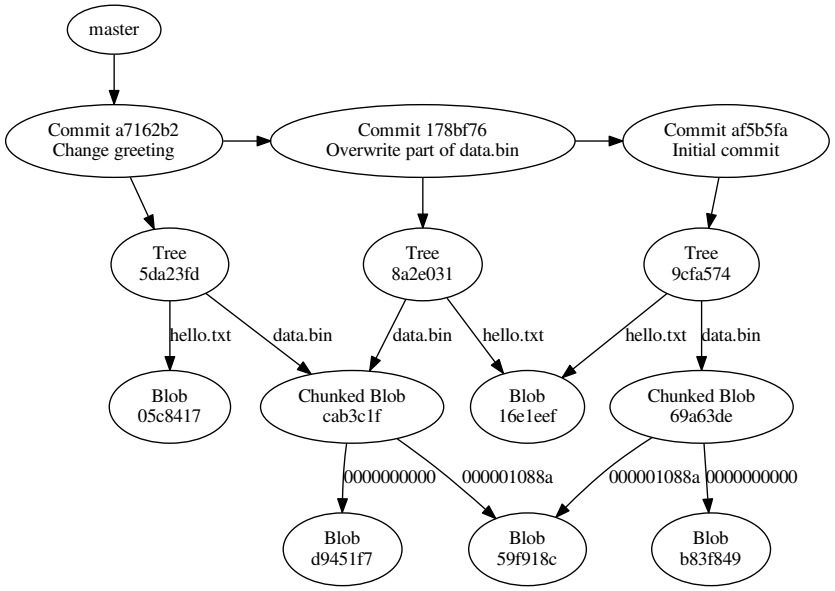


Figure 4.1: A simple DMV DAG with three commits

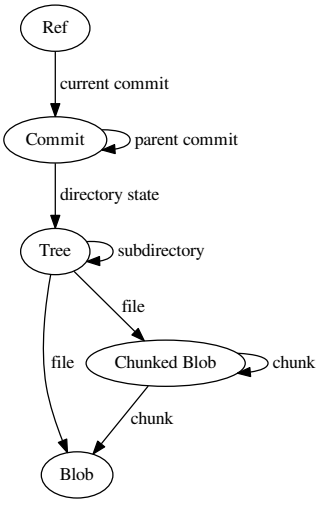


Figure 4.2: DMV DAG Object Types

Ref As with Git, a ref is a reference to a particular commit. A ref might represent the current state of a branch of development, or a particular state to save for later (a tag).

The DAG begins with refs. Each repository has a list of refs that lead into the DAG. Each ref refers to a commit. Each commit refers to a tree that represents the state at the time of committing, plus one or more parent commits. A tree refers to the blob, chunked blob, or tree that makes up each entry in the directory. A chunked blob can refer to other chunked blobs or to blobs. And finally, the blobs are the leaves of the graph. These relationships are illustrated in Figure 4.2.

Files are split into chunks using a *rolling hash algorithm* such as Rabin-Karp fingerprinting [19]. 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.

4.2 Working with an Incomplete DAG

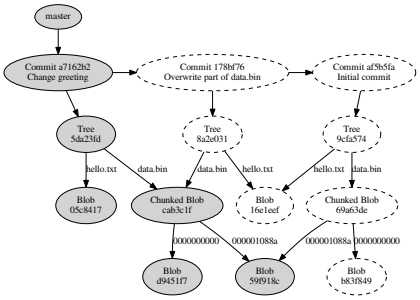
The DAG stores the full history of a data set, and it can be sliced to partition the data in space and time.

Partial history of full data set Keep a subset of commits, plus all trees, blobs, and chunked blobs reachable from them (Figure 4.3a).

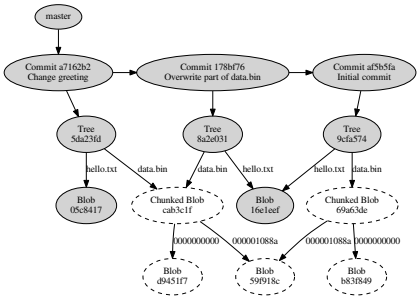
Full history of part of data set Keep all commits, but keep only the trees, blobs, and chunked blobs for certain paths (Figure 4.3b).

Full history of metadata Keep all commits and trees, but omit blobs and chunked blobs (Figure 4.3c).

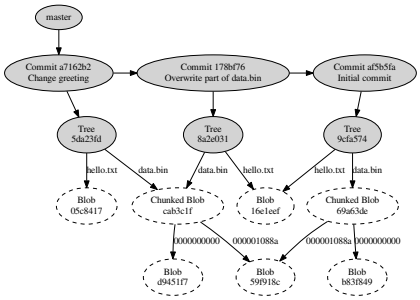
A DMV repository may use any combination of these slicing techniques to keep only those objects needed at that location, and new commits can still be added to a partial DAG. A repository that stores only part of the data set can create new commits that represent changes to that portion of the data set by assuming that everything else remains unchanged. Even a metadata-only repository could create new commits that represent renames and reorganization of the same files.



(a) Partial history of full data set



(b) Full history of part of data set



(c) Full history of metadata

Figure 4.3: A DMV DAG, sliced in different dimensions

Figure 4.4: Speculative DMV ls output showing remote files

```
-rw-r--r-- 1 user user  121306 Oct 21 18:28 local   filex
-rw-r--r-- 1 user user   25475 Oct 21 17:52 100ms   filey
-rw-r--r-- 1 user user   32031 Oct 21 17:52 20min   filez
-rw-r--r-- 1 user user   74968 Oct 18 17:12 missing filexx
-rw-r--r-- 1 user user   83977 Sep 22 21:23 unknown fileyy
```

So the DAG can be spread across many repositories, sliced according to what is needed at each location. Repositories will keep a record of what objects exist at their neighbors, along with statistics about latency when connecting to those repositories. This will allow them to estimate how long it would take to transfer a file that is not currently present on the system.

Though we did not have time to implement this feature, we are envisioning an enhanced `ls` command output that shows these estimates, as illustrated in Figure 4.4.

Chapter 5

DMV Implementation

We have written a DMV prototype as a proof-of-concept. The DMV prototype is written in the Rust programming language and it runs from the Unix command line, with the executable built as a thin wrapper around a library, so that it can be used by other applications.

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 alongside this dissertation in Munin, the University of Tromsø's open research archive (`http://munin.uit.no`), and continued work can be found via the author's website (`http://dmv.sleepymurph.com/`).

5.1 Command Line Control, Explicit operations

Like with familiar DVCSs, file changes in DMV are explicitly committed by user command. Synchronization is an explicit command as well. Applications built on DMV may add daemons to automatically commit and sync, but DMV focuses only on providing the commands.

DMV is used from the command line and includes familiar subcommands such as branch, commit, and checkout (Figure 5.1).

Figure 5.1: DMV help output, listing subcommands

```
dmv 0.1.0 (df7b9bd) (debug)
Mike Murphy <dmv@sleepymurph.com>
A distributed version-control system for media files

USAGE:
    dmv [SUBCOMMAND]

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

SUBCOMMANDS:
    branch           show/update branch information
    cache-status     show cache status of a file
    checkout         check out another revision
    commit           commit current files to the repository
    extract-object   extract a file or tree
    fsck             verify repository integrity
    hash-object      store a file or directory in the object store
    help             Prints this message or the help of the given subcommand(s)
    init             initialize repository
    log              show commit history
    ls-files         list files
    merge            combine revisions
    merge-base       find common ancestor
    parents          show current parent commits
    show-object      print information about an object
    show-ref         show refs
    status           show status of files
```

5.2 Rust

The implementation language, Rust, is a new C-like systems language that uses a sophisticated type system to guarantee memory safety [24]. Rust's memory safety checking prevents many common bugs, including data races in concurrent code, buffer overflows, access to uninitialized or freed memory.

Rust is also a compiled language with an ability to create libraries with C-compatible ABIs. Compiling to machine code should make it easier to port DMV to low-powered devices, and a C-compatible ABI will allow client applications that use the DMV library to be written in nearly any language.

5.3 Working Directory and Object Store

The DMV prototype stores its objects as regular files on the file system, using the same structure that Git does. The object store is in a hidden directory inside of the working directory (`.dmv/objects`). Objects are stored using their SHA-1 hash as their filename, with the first two hex digits removed to create a directory name (Figure 5.2). This leads to 256 subdirectories, each holding roughly $1/256$th of all the objects stored. We experimented with other schemes to divide the objects (section 7.1), but we found that other schemes tended to create too many subdirectories, exhausting the filesystem's available inodes before using the available disk space.

Figure 5.2: Example object file name

Object SHA-1 hash	`c6e2f43ddee3c00041cdae8fedc3bd6961e61f69`
Object file name	`.dmv/objects/c6/e2f43ddee3c00041cdae8fedc3bd6961e61f69`

5.4 Chunking Algorithm

Files are broken into chunks using the same rolling hash algorithm used by Gzip and Rsyncrypto [38] to respectively compress and encrypt files by chunks so that the result is "rsyncable" — a remote copy of

the compressed or encrypted file can be updated by transferring only those chunks that have changed. This algorithm uses Rsync's rolling checksum [42], creating a chunk boundary when the Rsync rolling hash is equal to 0.

The algorithm keeps a sum of the previous 8192 bytes of input data, and creates a chunk boundary when that sum is evenly divisible by 4096. These parameters are arbitrary and can be adjusted. Let w denote the *window size*, the number of previous bytes summed, and let d denote the *divisor* of the modulus operation. Also, let P_n denote the rolling hash sum for position n of the input stream. Then a chunk boundary is where

$$P_n = \left(\sum_{i=n-w}^{n} P_i \right) \bmod d = 0 \tag{5.1}$$

In experiments with our implementation of this rolling hash algorithm (section 7.3), Rsyncrypto's parameters with a window size of 8192 B and a divisor of 4096 yielded a mean chunk size of 4.1 KiB with a standard deviation of 4.6 KiB. We adjusted the parameters for DMV to a window size of 32 KiB and a divisor of 16 Ki, which yielded a mean chunk size of 18.7 KiB with a standard deviation of 22.0 KiB. These larger chunk sizes resulted in faster commit times for large files in our experiments (section 7.3).

Chapter 6

VCS Scaling Experiments

We performed experiments to probe the limits of existing version control systems, to see how they would cope with file sizes and numbers of files in ranges beyond what would be expected in a source code tree. We wanted to see how long it would take for each VCS to store that amount of data, how much disk space it used, and what CPU utilization was like during storage. And since the purpose of a VCS is to track changes, we also wanted to measure those same metrics when a small subset of that data was modified and then committed again.

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

6.1 Version Control Systems Evaluated

We ran each experiment using five different VCSs:

Git The most popular DVCSs, and one of the main inspirations of DMV. See subsection 1.2.1 for details about Git and how it stores data.

Mercurial A DVCSs that models data and history in the same manner as Git, but stores it differently. Rather than storing by hash object ID in an object store, Mercurial creates a *filelog* for each input file, storing its different versions as a base version followed by a series of deltas [27, Chapter 4].

Bup A backup system that uses Git's data model and pack file format. Like DMV, Bup breaks files into chunks using a rolling hash, reusing Git's tree object as a chunked blob index[1]. Unlike Git, it writes to the pack file format directly, without Git's separate commit and pack steps, and without bothering to calculate deltas [29]. See section 8.1 and section 9.2 for more discussion about Bup and its similarities and differences to DMV.

DMV Our prototype system.

Copy A control for the experiment, a dummy VCS that simply copied the input files into another directory using the standard Unix `cp` command.

The exact version of each VCS used in the experiments is listed in Table 6.1. DMV versions are referenced by commit ID in the DMV source's Git repository.

6.2 Procedure

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 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

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

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) 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, starting at 1 MiB, we would run trails with 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, starting at 100 files we would run trials with 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).

When updating data files for the second commit, we would overwrite a single contiguous section of each file with new pseudorandom bytes. We would start one-quarter of the way into the file, and overwrite $1/1024$th of the file's size (or 1 byte if the file was smaller than 1024 KiB). So a 1 MiB file would have 1 KiB overwritten, a 1 GiB file would have 1 MiB overwritten, and so on.

6.3 Automation and Measurement

The trials were run via a Python script that would set up, run, and clean up each trial in a loop, covering the full range of sizes or numbers for a given VCS. The script would measure the wall-clock time duration taken by each commit command and collect CPU utilization metrics. It would

Table 6.1: Version control systems evaluated and their versions

VCS	Version
Git	2.1.4
Mercurial	3.1.2
Bup	debian/0.25-1
DMV	c9baf3a
Copy (GNU `cp`)	8.23

Table 6.2: Experiment computer specifications

Hardware	
Vendor	Hewlett Packard
CPU	Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz
RAM	8 GiB
Hard disk	ATA model ST3250318AS
Operating system	
Operating system	Debian 8.6 (“Jessie”)
Kernel	Linux 3.16.0
Filesystem	
Test partition	197 GiB LVM partition
Filesystem	ext4
Block size	4 KiB
I/O scheduler	cfq (unless otherwise noted)
DMV compilation	
Rust compiler version	1.15 stable or 1.16 stable
Rust compiler flags	--release

also terminate any individual VCS operation that ran longer than five and a half hours. After commit and verification, the script would also measure repository size.

The script measured the wall-clock time duration for each commit by checking the system time (`time.time()` in Python) just before and just after using Python's `subprocess` module to execute the necessary VCS command. CPU utilization was measured by sampling the CPU status lines provided in Linux's `/proc/stat` information. The status lines show a cumulative count of CPU ticks ($1/100$th of a second) that the CPU has spent in user mode, system mode, idle, and waiting for I/O [22]. Like with the time measurements, the script samples CPU utilization before and after executing a VCS command, and then subtracts to get the number of time slices spent in each state during execution. We then compare the relative number of time slices in each state to get an idea of whether the operation is CPU-bound or I/O-bound.

The script measures repository size using the standard Unix disk usage command (`du`) and measures the size of the trial's entire temporary directory, which includes the generated input data itself along with the VCS's storage.

Finally, the script would delete the test directories by reformatting the partition. It ensured that the filesystem would be completely reset for each trial. Also, at the larger numbers of files, deleting recursively could take hours, much longer than it took to create the files and longer than it took to run the trial. Reformatting was much faster.

Experiment scripts used and raw result data collected are available alongside this dissertation in Munin (`http://munin.uit.no`) and can also be found via the author's website (`http://dmv.sleepymurph.com/`).

6.4 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. Additional details can be found in Table 6.2.

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 real variation.

6.5 Results: File Size

6.5.1 File Size Limits

Both Git and Mercurial had limits to the size of file they could store successfully.

With a 2 GiB file, Mercurial's commit operation would exit with an error code and message, saying "up to 6442 MB of RAM may be required to manage this file," and the commit would not be stored. However, the repository would be left in a consistent empty state. The atomicity of the commit operation held. All commits of files 2 GiB and larger would be rejected in a similar manner.

Git's behavior was more erratic. Starting with a file 12 GiB and larger, Git's commit operation would exit with an error code, reporting a fatal out-of-memory error saying that `malloc` failed to allocate 12 GiB. However, the commit would be successfully stored — no consistency errors in the repository were reported by `git fsck`. Starting at 24 GiB, the commit operation would report the same error and the commit would still be written, but then the `git fsck` integrity check itself would also exit with an error code.

However, the error that `git fsck` reported in its output was a `malloc` error very similar to the "fatal" error from the commit operation, and it did not report any actual integrity errors in the repository.

So to check the commit, we extracted the 24 GiB file from the repository and compared it. It was the same as the original. So the commit was intact. We also deliberately corrupted the Git pack file that stored the 24 GiB file by overwriting one 1 MiB block at an offset of 22 GiB with new pseudorandom data. When we ran the `fsck` command again with the corrupted repository, it reported the integrity error, but it did not report the `malloc` error that it did before.

The `git fsck` command found the integrity error surprisingly quickly, reporting the error and exiting instantaneously (from the user's perspective), whereas it had taken 7 min to complete before. So it does not seem to have hashed the whole file again to find the error. We took care to preserve the modification time for the pack file when we corrupted it, so we are not sure how Git detected the error so quickly. Perhaps our corruption overwrote some internal indexing structure of

the pack file along with the file data. More investigation would be necessary to know for sure.

We continued the experiment under the assumption that Git's errors were a false alarm, and allowed the trials to continue at even larger sizes.

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. At 96 GiB, our experiment script timed out and terminated the commit after five and a half hours. 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 (subsection 6.6.2), and later performed more experiments to examine in detail (chapter 7).

Our experiment environment itself limited the largest file stored by any VCS to 96 GiB. Any larger and it was simply impossible to store a second copy of the file on our 197 GiB test partition. 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 6.3 and Table 6.4.

Table 6.3: Observations as file size increases

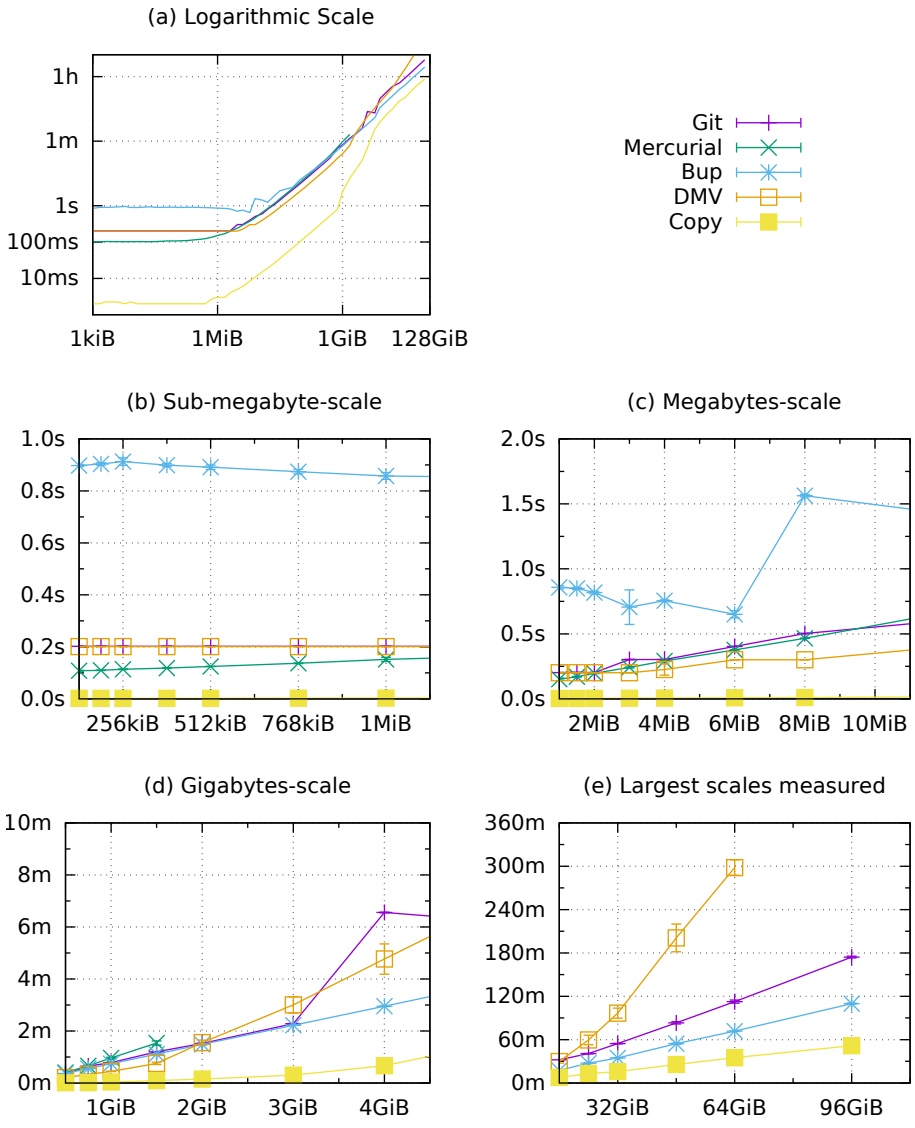
Size	Observation
1.5 GiB	Largest successful commit with Mercurial
2 GiB	Mercurial commit rejected
8 GiB	Largest successful commit with Git
12 GiB	Git false-alarm errors begin, but commit still intact
16 GiB	Largest successful Git fsck command
24 GiB	Git false-alarm errors begin during fsck, but commit 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

Table 6.4: Effective size limits for VCSs evaluated

VCS	Effective limit
Git	Commit intact at all sizes, UI reports errors at 12 GiB and larger
Mercurial	Commit rejected at 2 GiB and larger
Bup	Successful commits at all sizes tried, up to 96 GiB
DMV	Successful commits up to 64 GiB, timeout at 5.5 h during 96 GiB trial

Figure 6.1: Wall-clock time to commit one large file to a fresh repository

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars.



6.5.2 Time for File-Size Initial Commit

Figure 6.1 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, which is most apparent in Figure 6.1e.

At file sizes below around 2 MiB (Figure 6.1a and b), commit times are dominated by overhead — around 5 ms for Git, 100 ms for Mercurial, 180 ms for DMV, and 900 ms for Bup, vs only 2 ms for the copy.

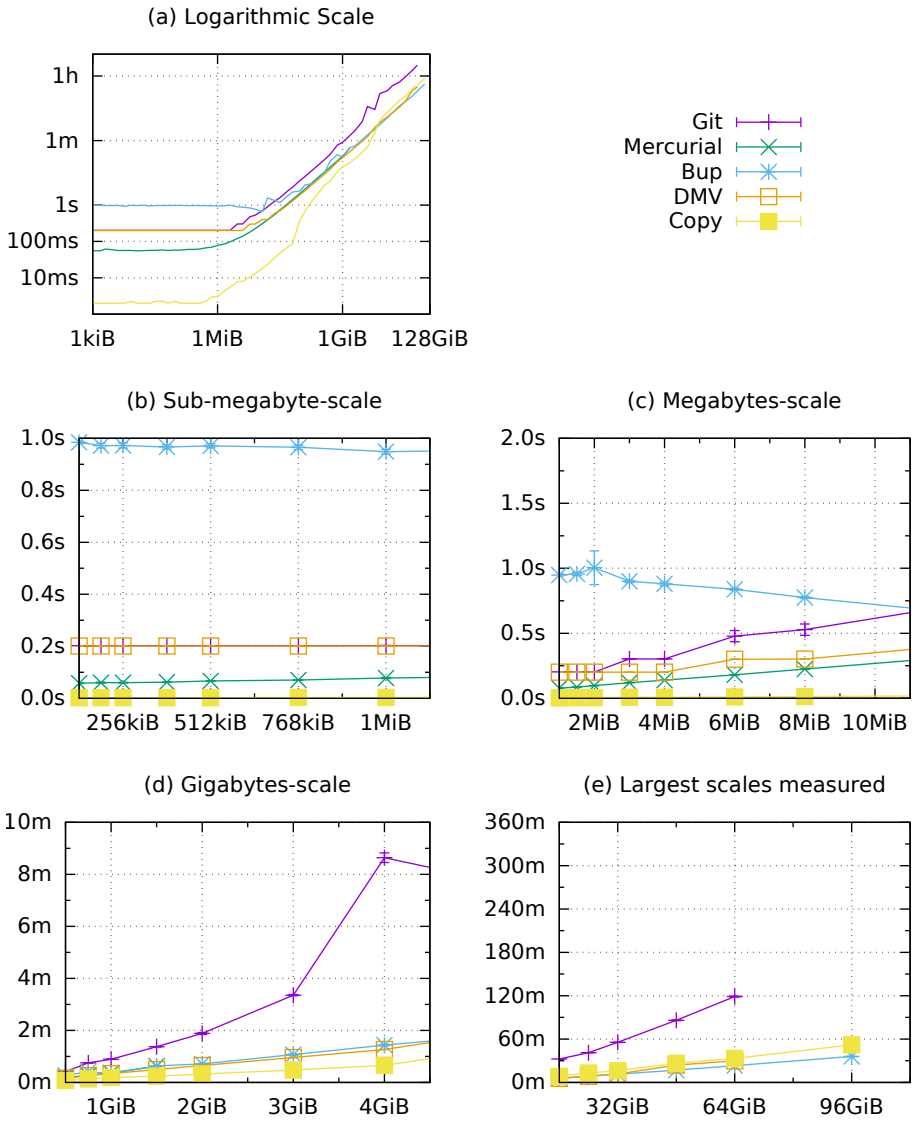
Bup, after starting with the highest overhead, goes on to have the fastest initial commit of all the systems evaluated for large files. It takes the lead at 2 GiB, where Mercurial drops out (Figure 6.1d). To commit the 2 GiB file, Git's average time is 91.1 s, Bup's is 89.1 s, and DMV's is 90.8 s. All of these are a factor of around ten times slower than the direct copy at 9.1 s. The differences get more pronounced as the file sizes continue to increase. At 64 GiB, Git's average time is 110 min, Bup's is 72 min, DMV's is 298 min. The average 64 GiB copy takes 35 min.

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 (section 6.6). 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 chapter 7.

Bup's commit times behave strangely in that there are places where Bup is actually faster than it was with a smaller file. This is most apparent in the slow downward slope of Figure 6.1b and the zig-zag of Figure 6.1c. Git also has a point where the time decreases, taking 393 s (SD 0.76 s) to commit a 4 GiB file and only 361 s (SD 3.90 s) to commit an 8 GiB file. Even more interestingly, these decreases are consistent across the four trials on separate hardware. We do not know what might be causing this.

Figure 6.2: Wall-clock time to commit one updated large file

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars. X axis shows the total size of the file. The updated portion was $1/1024\,\mathrm{th}$ of the total file size.



6.5.3 Time for File-Size Update Commit

Figure 6.2 shows the wall-clock time required for the second commit, after updating $1/1024$th of the file. Ideally this operation should be faster than the first commit, because the system should only be storing the changed portion of the file. Indeed this is the case for Mercurial, Bup, and DMV, which do store only the changed portion. Git, however, copies the entire updated file into its repository as a new object, and so its commit time is virtually identical. The same is true of the copy control, though for sizes smaller than 8 GiB it is still faster than all the other systems.

As with the initial commit, Bup gets faster as file size increases at certain points, with the same gradual downward slope in the sub-megabyte and low megabyte-ranges, leading to a prominent jump up then fall back down that occurs just outside the range of Figure 6.2c. The jump comes at a slightly larger size with this update commit, at 8, 12, 16, and 24 MiB, as opposed to 4, 6, 8, and 12 MiB in the initial commit. It can still be seen in miniature in Figure 6.2a.

The fact that there is still a drop — but at a larger size — suggests that the decrease is related to the amount of data written to the disk, since Bup breaks the file into chunks and only writes the updated chunks.

Git also shows a commit time decrease between 4 GiB (518 s with SD 11.1 s) and 8 GiB (429 s with SD 4.0 s) just as it did with the initial commit. Unlike Bup, its decrease is not shifted to higher file sizes, which is another hint that the decrease has something to do with the amount of data written, since Git writes the whole file again, rather than just the updated portion.

Figure 6.3: CPU utilization while committing one large file to a fresh repository

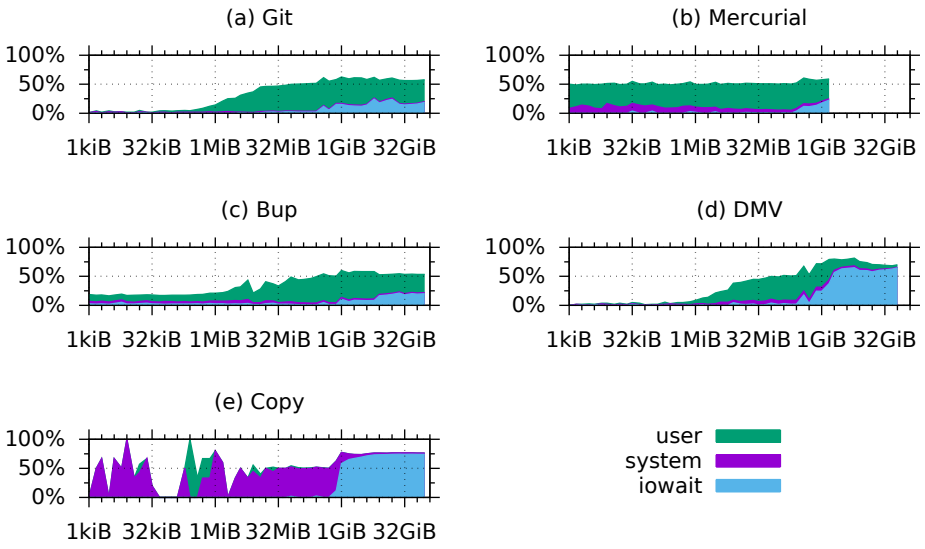
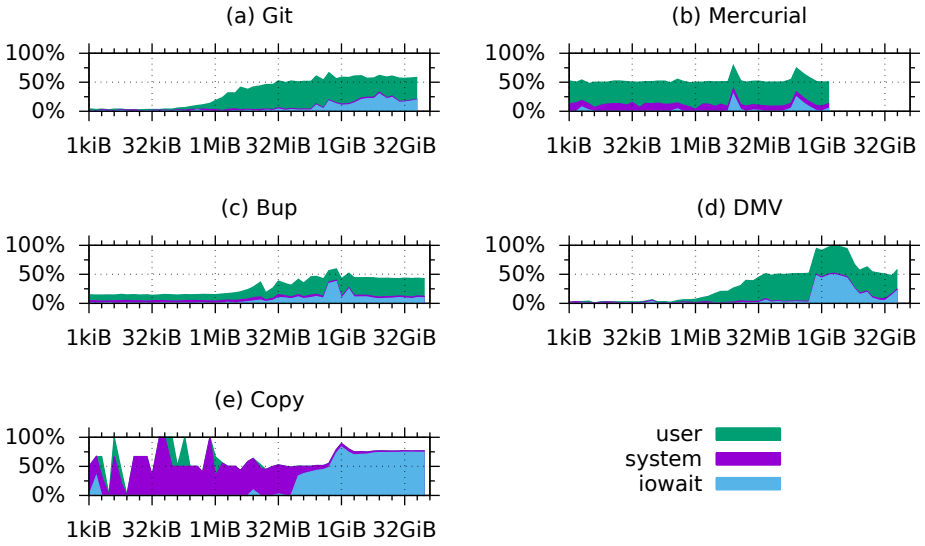


Figure 6.4: CPU utilization while committing changes to one large file



6.5.4 CPU Usage During File-Size Commits

Figure 6.3 shows CPU usage during the initial commit, and Figure 6.4 shows CPU usage during the update commit. "User" indicates user-space computation, "system" indicates kernel-space computation, and "iowait" indicates that the CPU was waiting on an I/O operation.

We expected the commit operations to be I/O bound, and that seems to be the case. It certainly is with DMV and with the copy, especially at file sizes 1 GiB and larger. However, there is also a significant amount of user-space work going on in all of the version-control systems, such as hashing the data and, in Mercurial's case, calculating deltas. The low I/O wait activity in Git, Mercurial, and Bup is surprising. It is possible that the commit is actually CPU bound in those systems, but seems more likely that the implementations are multi-threaded and use non-blocking I/O calls, allowing the CPU to do useful work instead of sitting in the I/O wait state.

DMV's high I/O wait is probably primarily due to the filesystem overhead that is slowing down its commit times, but the current single-threaded implementation might also be contributing, and the prototype might benefit from having separate threads to load data, hash it, and write it to dish.

There is a small peak in the DMV I/O wait percentage for the first commit (Figure 6.3d), peaking at 386 MiB with 337 ticks (SD 13.8 ticks), then falling at 512 MiB to 89 ticks (SD 49.2 ticks), and then rising again at 768 MiB to 972 ticks (SD 54.3 ticks). We are not sure what is causing this. We also do not know why the DMV I/O wait during the update commit actually decreases between 1 GiB and 32 GiB (Figure 6.4d).

The copy operation shows an erratic amount of system activity at low sizes, which is surprising since the copy operation is almost pure I/O. This is a result of how the data was collected rather than any unexpected behavior in the copy operation. At sizes below about 10 MiB, the copy operation is faster than the `/proc/stat` ticks used to CPU measure activity ($1/100$th of a second [22]). At such small intervals, most measurements will yield zero ticks, though one of the states will occasionally measure one tick (see Table 6.5). Then, when the graph normalizes the total usage to a percentage, 1 system tick out of 1 total ticks makes 100 %, so the graph will jump from 0 % 100 %, or somewhere in between depending on how many of the four trials measured one tick.

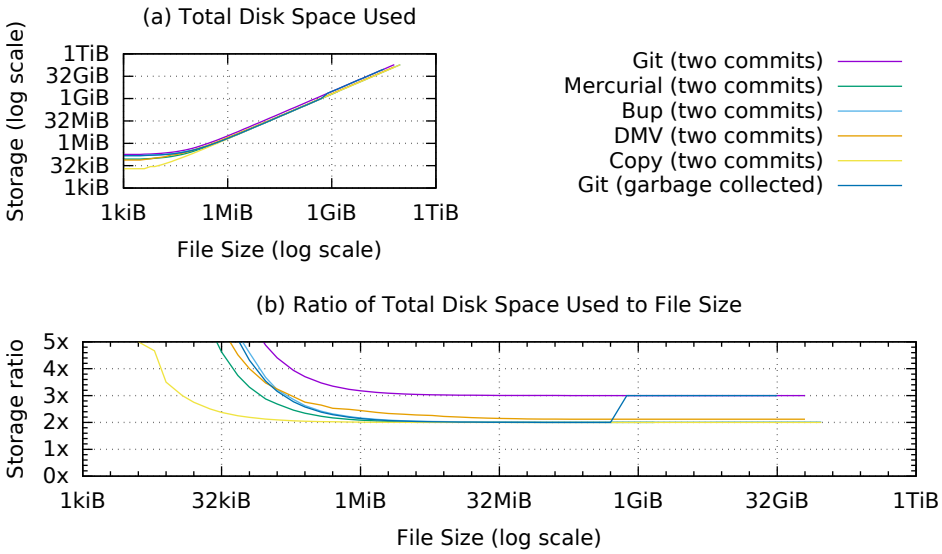
It isn't until around 48 MiB that the measurement includes enough CPU ticks to yield useful percentages, which is where we see the graph start to stabilize.

Table 6.5: Selected CPU usage data for copy operation

Size	User	System	Idle	Iowait
4.0MiB	0	0	0	0
6.0MiB	0	0	1	0
8.0MiB	0	1	1	0
12.0MiB	1	1	2	0
16.0MiB	0	2	2	0
24.0MiB	1	3	3	0
32.0MiB	0	3	4	0
48.0MiB	0	5	5	0
64.0MiB	0	7	6	0
96.0MiB	0	10	10	0

Figure 6.5: Total repository size after committing, updating, and committing again

Subfigure (a) shows repository size on a logarithmic scale, while subfigure (b) shows the ratio of total repository size to input data size.



6.5.5 Repository Size after File-Size Update Commit

Figure 6.5 shows the total repository size after the update commit, including the original file. This is after committing, updating $1/1024$th of the file, and committing again.

The stored data overtakes the initial repository overhead after a file size of around 1 MiB, and the repository size for all systems converges to about twice the size of the file. This is to be expected, since each measurement includes the original file, the first copy of the file, and the updated $1/1024$th. The exception is Git, which stores the entire updated file during the update commit, leading to a total disk space usage of three times the file size. However, Git has a separate garbage collection stage where it cleans up the repository and aggregates similar objects together in pack files. The post-garbage collection size for Git is shown as a separate line on the graph. This post-GC size converges to double the original file size, but then jumps to three times at a file size of 1.5 GiB. This suggests that the pack step is failing silently at 1.5 GiB and larger. This is probably related to the way Mercurial's commits begin failing at 2 GiB and larger. Both operations are trying to load multiple versions of the file into memory to calculate deltas for packing.

6.6 Results: Number of Files

6.6.1 File Quantity Limits

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*.

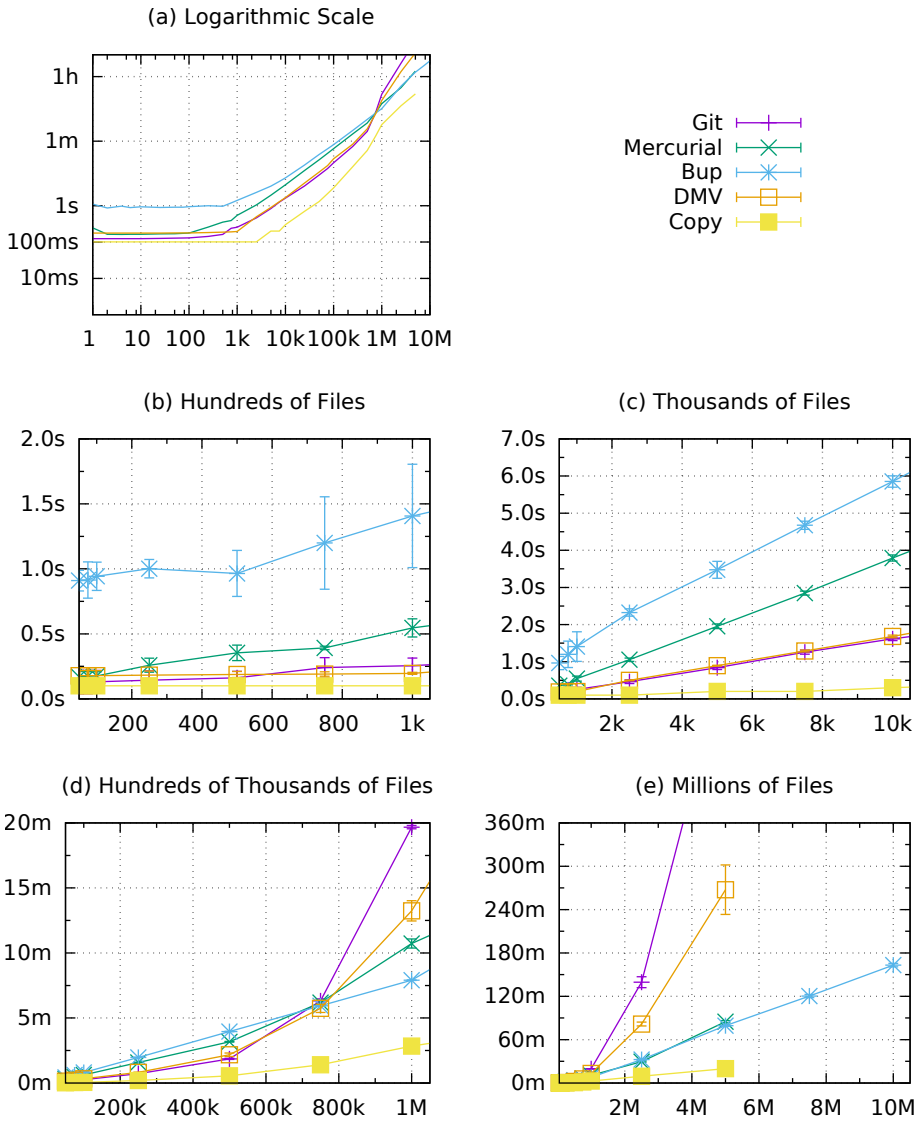
Unix filesystems, ext4 included, store file and directory metadata in a data structure called an *inode*, which reside in a fixed-length table [33]. When all of the inodes in the table are allocated, the filesystem cannot store any more files or directories.

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.

Figure 6.6: Wall-clock time to commit many 1KiB files to a fresh repository

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars.



6.6.2 Time for Number-of-Files Initial Commit

Figure 6.6 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.

We saw in the file-size commit times (subsection 6.5.2) 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.

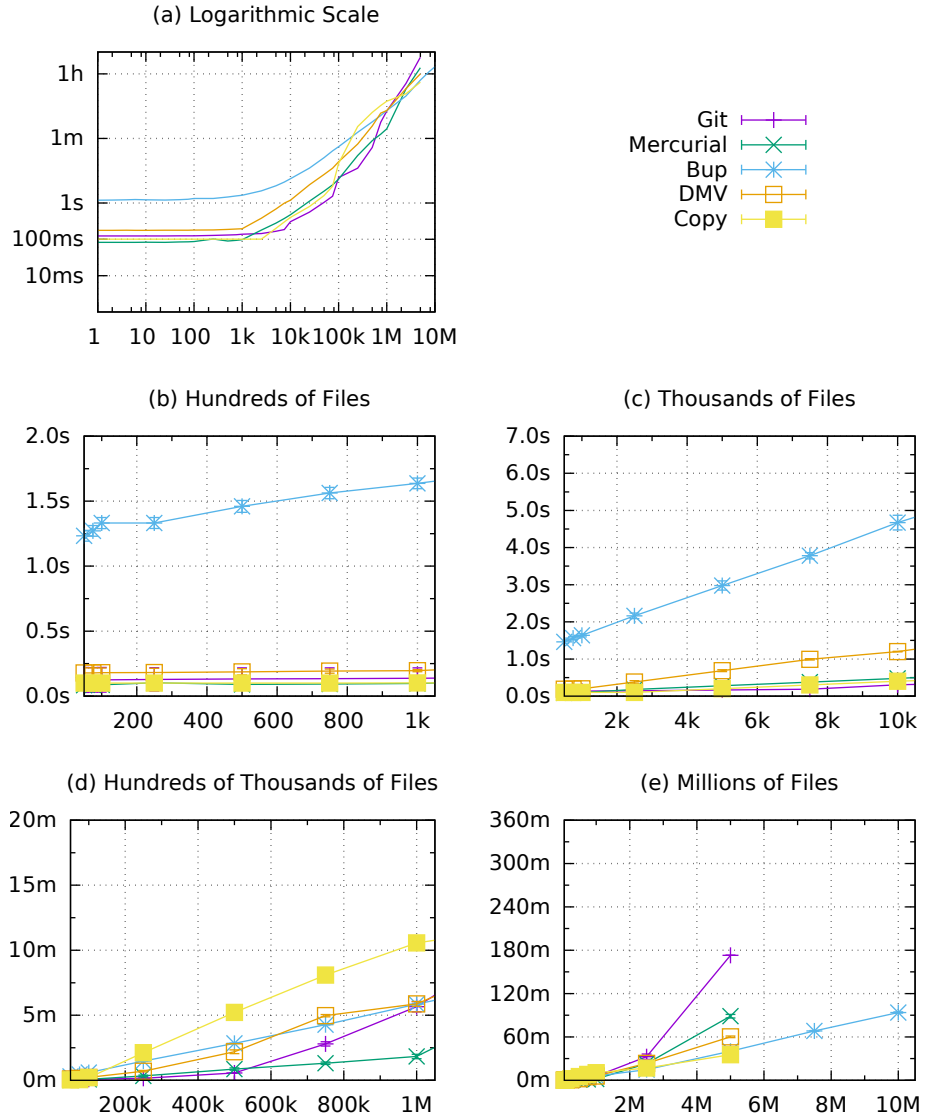
At 500, 750, and 1000 files (Figure 6.6b), Bup's commit times have an unusually high variance compared to the other systems and the other experiments. At those file counts, the standard deviations for Bup's commit times are 18.3 %, 29.6 %, 28.3 % of the means, respectively (see Table 6.6), whereas the mean of the other such standard deviation percentages is only 6.1 %. We are not sure what is causing this.

Table 6.6: Bup initial commit times with unusually high variance

Num. files	Mean time (s)	SD (s)	SD as % of mean
500	0.965	0.177	18.3
750	1.199	0.355	29.6
1000	1.407	0.398	28.3

Figure 6.7: Wall-clock time to commit many updated files

X axis shows the total number of files. 1 out of every 16 files was updated.
Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars.



6.6.3 Time for Number-of-Files Update Commit

Figure 6.7 shows the wall-clock time required for the second commit, after updating 1 out of every 16 files. As with the file-size experiment (subsection 6.5.3), storing only the updated files is faster, and in this case the difference is more pronounced. This is because every system understands the file as a unit of data, and can naturally separate the changed and files from the unchanged files.

The exception is the copy operation, which blindly copies all files again. This is why it is actually slower than the other systems at some points. It would be interesting to run this experiment with Rsync as well (section 9.1), to get a baseline for comparing and copying only the files that have changed.

Here all commit times appear to increase linearly with respect to number of files, except for Git, which shows some quadratic growth as the number of files gets into the millions.

Bup does not exhibit the interesting variance here that it did in the initial commit. All standard deviations are small enough that the error bars are barely discernible on the graph.

Figure 6.8: CPU utilization while committing many 1KiB files to a fresh repository

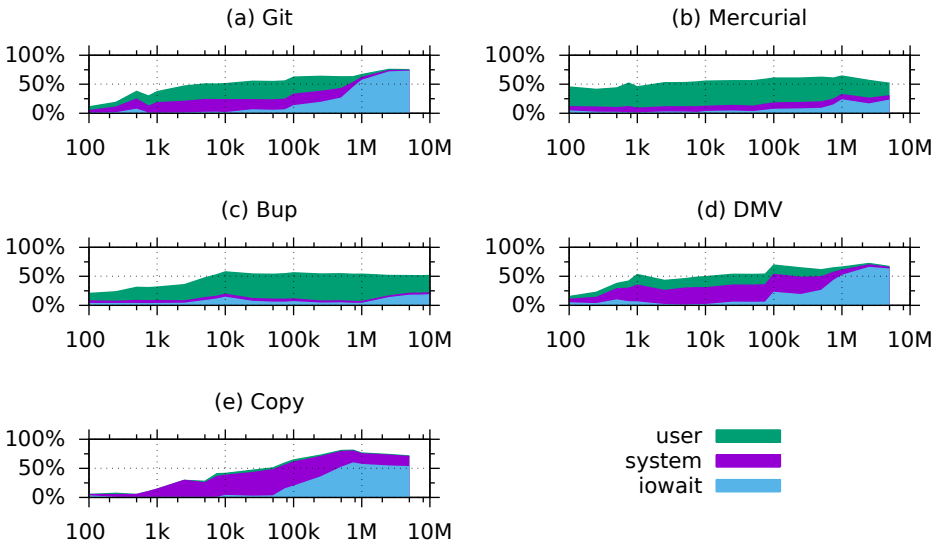
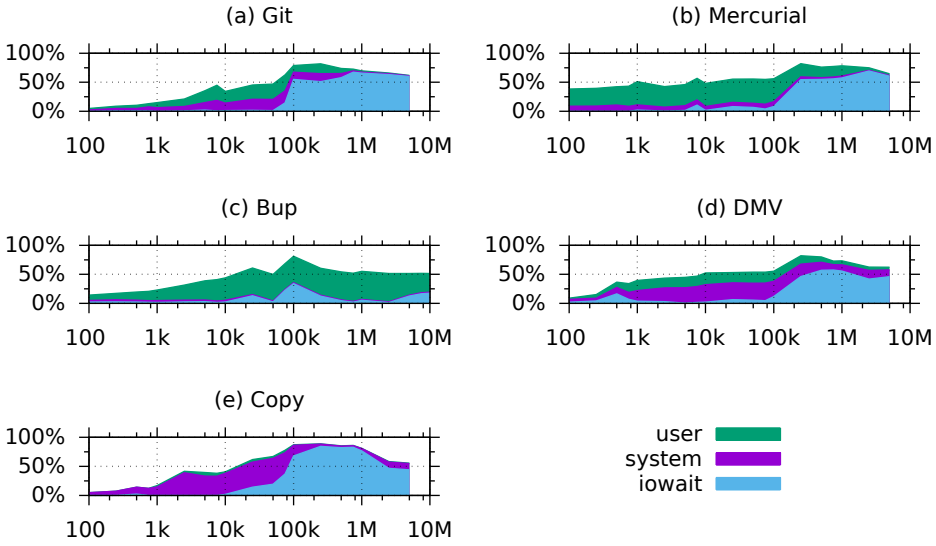


Figure 6.9: CPU utilization while committing many 1KiB files after one of every 16 files has been updated



6.6.4 CPU Usage During Number-of-Files Commits

Figure 6.8 shows CPU utilization during the initial commit and Figure 6.8 shows CPU utilization during the update commit.

With the initial commit, Git, DMV, and the copy spend more time waiting on I/O than Bup or Mercurial, and they also spend more time in system mode. In the case of the copy, this is probably because the operation is almost pure I/O. In the case of Git and DMV, this is more evidence to suggest that writing files with effectively random names incurs more filesystem overhead than writing sequentially, as Mercurial does, or appending to large files, as Bup does. We can see that Bup and Mercurial both spend more time processing in user mode than waiting for I/O.

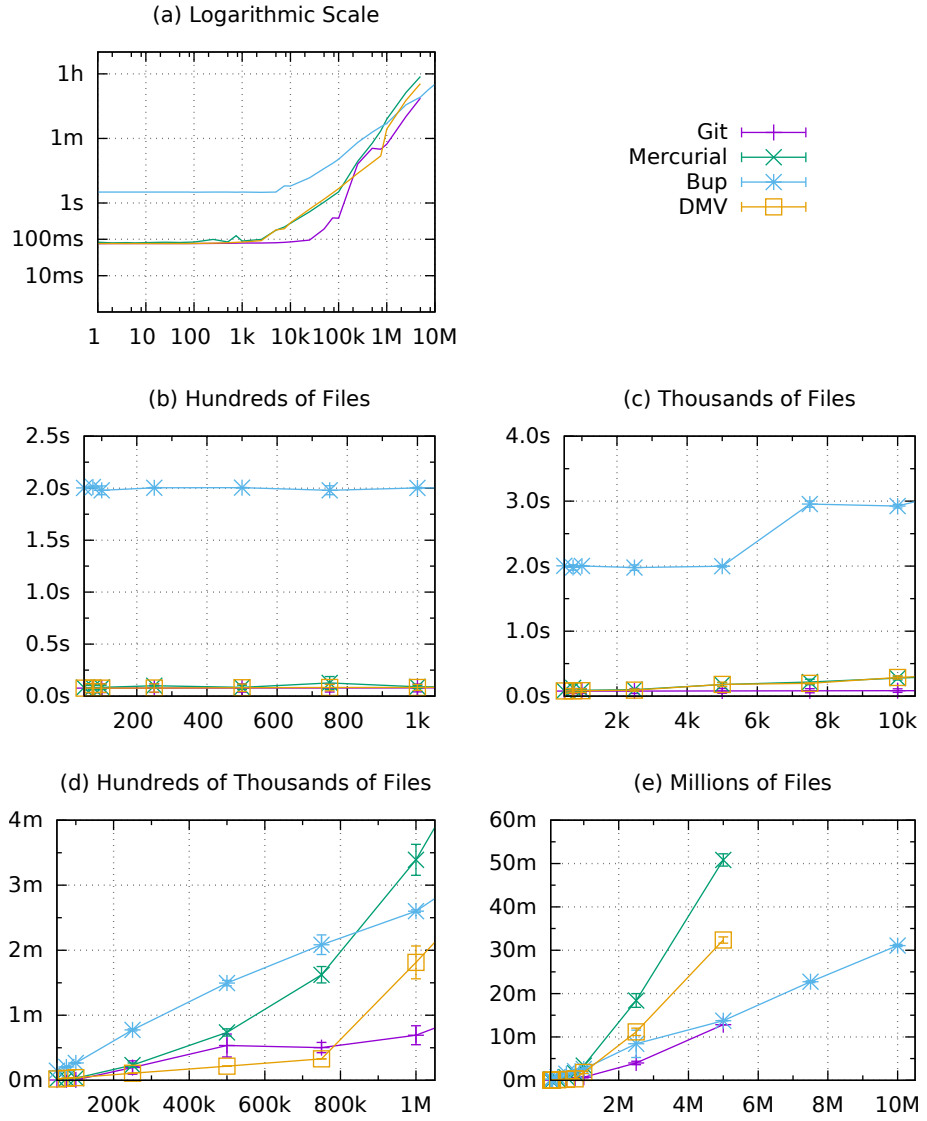
With the update commit, Mercurial loses its sequential write advantage, since it has to seek to the filelog that corresponds to the current input file and append to it. And so we see much more I/O wait with Mercurial. Bup continues to simply append objects to its pack files as always, and so it retains a low I/O wait profile.

Interestingly, Mercurial's and DMV's I/O wait in the update commit has a gradual rise starting around 100 thousand files, while Git's is a sudden rise starting just before 100 thousand files. The copy is somewhere in between. We are not sure why that is, since DMV has much more in common with Git than with Mercurial in terms of disk usage patterns.

Git, DMV, and the copy all decrease in I/O wait from 1 million to 5 million files. And Mercurial also shows a slight decrease from 2.5 million to 5 million files. Again, we are not sure what would be causing that.

Figure 6.10: Real time required to check the status of many files after update

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars. Copy not shown because it has no status-check operation.



6.6.5 Time for Number-of-Files Status Check

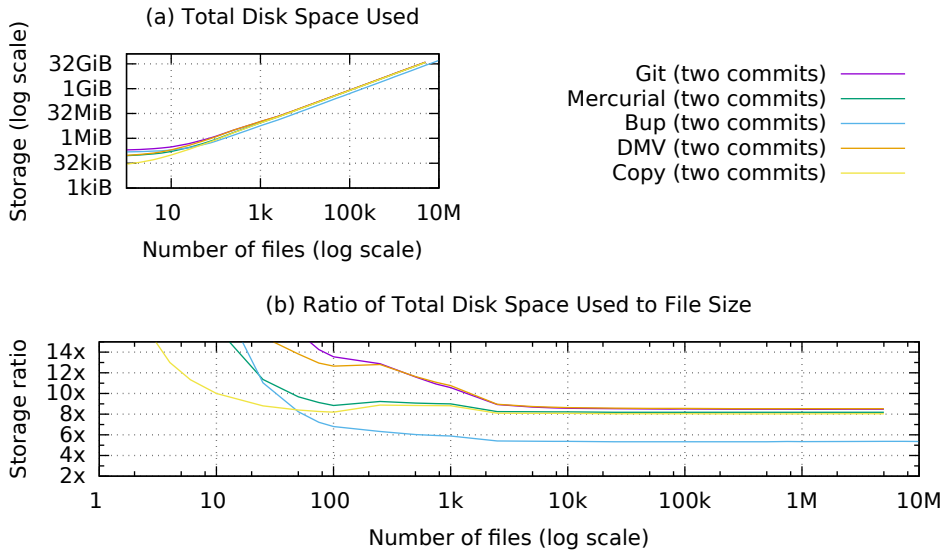
With the number-of-files experiment, we also timed how long it would take each VCS to run its status command and check which files had changed. Figure 6.10 shows the time required to check the status of all files just after updating them.

DMV and Mercurial seem to slow quadratically with number of files. Bup seems to have a general overhead of 2 s, jumping to 3 s at 7500 files, but after that increasing linearly.

Git has an interesting drop where it actually gets faster from 500 000 to 750 000 files, dropping from a mean of 32.145 s to 30.210 s. However, the measurement at 500 000 files has a high standard deviation, 10.623 s, compared to only 4.705 s at 750 000 files. The four measurements at 500 000 files are 20.330 s, 23.638 s, 38.251 s and 46.362 s. So it is the two unusually high measurements that are pulling the average up, but we do not know why those measurements are especially high.

Figure 6.11: Total repository size after committing, updating, and committing again

Subfigure (a) shows repository size on a logarithmic scale, while subfigure (b) shows the ratio of total repository size to input data size.



6.6.6 Repository Size after Number-of-Files Update Commit

Figure 6.11 shows the total repository size after the update commit, including the original files. This is after committing once, changing a single byte in every sixteenth file, and committing again.

Git, Mercurial, DMV, and the copy all converge to using just over 8 times the theoretical size of the data set, while Bup is closer to 5 times. This has to do with the block size of the filesystem. The underlying filesystem uses a 4 KiB block size, so each 1 KiB file uses one 4 KiB block. So the input data itself takes 4 times its theoretical size. So each will end up with 4 times for the input files, plus 4 times for the copied files, plus some overhead. So a storage ratio of just over 8 is to be expected for those systems that store objects as individual files.

Naturally the copy would have the least overhead, with a ratio of 8.043 at 5 million files. This shows that the directory hierarchy of the input files itself probably adds around 0.021 times, or 2.1 %. Mercurial has the second lowest overhead, with a storage ratio of 8.186. This makes sense because, while Mercurial creates a filelog for each input files, it reuses filelogs to store the updates versions, so it does not create any new files during the second commit.

Git and DMV have similar, higher ratios because they do create new object files for each new version of each input file. At 5 million files Git's ratio is 8.478 and DMV's is 8.538. It would be interesting to see Git's ratio after garbage collection and packing, but unfortunately we did not run Git's garbage collection as part of this experiment as we did in the file-size experiment. We assume the results would be similar to Bup's.

Bup uses significantly less disk space, with a ratio of 5.374 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.

Chapter 7

Performance Tuning Experiments

After noticing DMV's long commit times, we tried tuning certain DMV parameters in an attempt to speed it up. We re-ran the file-size and many-files experiments on DMV several times, varying first the object store directory layout, then Linux I/O scheduler, and finally chunk size. We also ran new, more targeted experiments to investigate the effects of directory layout and chunk size.

7.1 Object Store Directory Layout

During initial runs of the many-files experiment (chapter 6), we would often notice the disk being reported as full even though the total bytes used was less than the capacity of the disk partition. This had to do with how each system stores its objects as files on the filesystem and how it organizes them into directories. Each file and directory on a Unix filesystem requires one inode, of which the filesystem has only a finite number. A storage scheme that allocates too many files or directories will exhaust the filesystem's available inodes before it uses all the available disk space.

We also noticed that average write speed would slow down as the operation progressed. The progress meter we added to DMV's commit operation would show a rate of 30 MiB s to 40 MiB s at the beginning of an operation but slow to less than 300 KiB by the end of a long one. We

Figure 7.1: DMV output showing varying object write times

Not shown: many objects written in under 10 ms, which are logged at `TRACE` level.

```
DEBUG: prototype::object_store: store blob 23a74606 —
 28.3 KiB stored in 0.062s ( 454.3 KiB/s)
WARN: prototype::object_store: store blob ff2942d6 —
 6.8 KiB stored in 24.735s ( 281 bytes/s)
DEBUG: prototype::object_store: store blob 415dba44 —
 16.9 KiB stored in 0.223s (  75.5 KiB/s)
WARN: prototype::object_store: store blob 9e810318 —
 15.2 KiB stored in 24.132s ( 644 bytes/s)
   Hashing:     1.5 GiB/    1.5 GiB 100.0% 199.6s     7.7 MiB/s done
```

Table 7.1: Sample object store directory variations

Hex digits	Depth	Example
0	0	`03d37679d1fab86e5286decd6cd2a94efcfe083f`
1	1	`7/9332ca7ce9163f78e3c115a2173bd8fd853d286`
1	3	`6/8/c/40e64f3e74e6ebefdcf2f5f30fb8da004792c`
2	1	`9f/4ec22c3e0289b29eefefe4728dece14e67e6ac`
2	2	`dd/52/bcccff156a179cdac0793ef049039372d8a1`
3	1	`cc5/199084d70f7c5ba325a240e1927579ee24bb1`
3	4	`472/e98/e88/0b1/c5905065c70cbe806361d32f6429`
4	3	`1ed2/bd51/01fe/5b23763e8c76852739f59201280f`

added log output to print the write times for individual objects, and we discovered that while most objects would be written in milliseconds, occasionally a single object write would take multiple seconds or tens of seconds, even though there was no appreciable difference in size between the objects (Figure 7.1).

DMV stores its objects as individual files in an object store directory, in the same manner as Git. The object's SHA-1 hash is used as its file name, except that the first two hex digits are removed and used as a subdirectory (also described in section 5.3). Our prototype originally took the first four hex digits to create two levels of subdirectories, under the assumption that we would store more objects than Git and need to spread them out with more subdirectories. That original prototype was showing this odd behavior, and it stored files much more slowly than Git. We suspected that the number of subdirectories could be at fault, so we experimented with different subdirectory schemes to see their effects.

7.1.1 Procedure

To measure the effects of different object storage schemes, we performed a new experiment wherein we created a new 100 MiB partition on one of the dedicated experiment computers, and then generated a series of pseudorandom files of 4 KiB each until the disk was reported full.

For each file, we would give it a pseudorandom name that resembled an SHA-1 hash, and store it according to the object storage scheme under test. We increased a counter each time we created a file, and another each time we created a new directory. If storing the file required creating several subdirectories to get to the proper depth, we would count each subdirectory. We also checked the number of files already in the target directory before writing, timed the write, and used the Unix `df` utility to measure free disk space in bytes and the number of free inodes.

The directory schemes we evaluated were all variations of the basic scheme of taking leading hex digits of the SHA-1 hash to form directories. We varied the number of directories taken (depth) and the number of hex digits per directory (see Table 7.1 for examples). We tried depths from 0 to 6 and digits per directory from 0 to 16, discarding

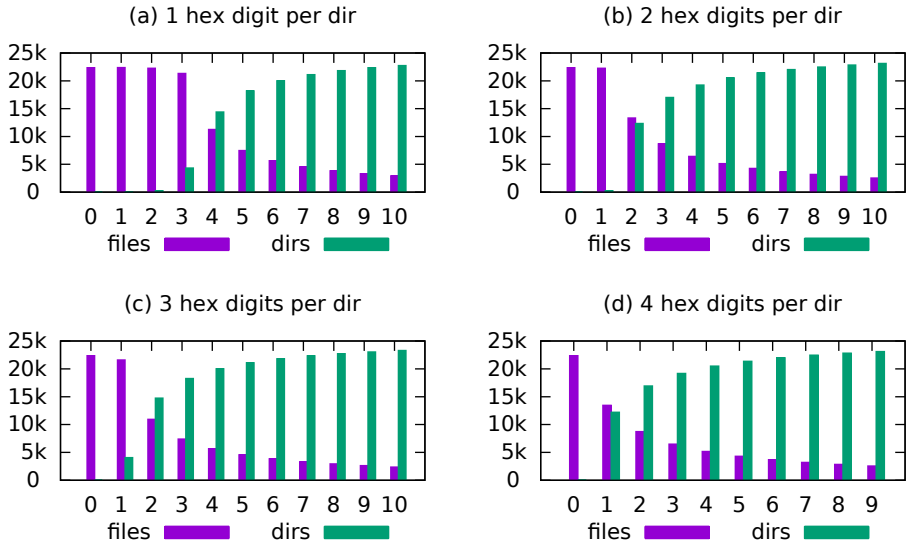
combinations that did not make sense, such as combinations involving 0 and another number (which would all simply be undivided), or those that required more than the 40 hex digits of a 160-byte SHA-1 hash.

7.1.2 Environment

Like the many-files experiment, this was automated as a Python script and run on one of the dedicated computers used for that experiment (specs shown in Table 6.2). However, rather than spending hours to fill the 197 GiB partition used for the other experiments, this experiment used a new 100 MiB LVM partition.

Figure 7.2: Number of Files vs. number of directories filling a disk

The number of files and directories present when the disk reported that it was full under the given directory scheme, shown by number of hex digits per directory (the different plots) and levels of depth (x axis)



7.1.3 Results

Out of Inodes

Figure 7.2 shows how quickly directories overtake files as subdirectory nesting goes deeper. Presented visually, the connection between files and directories becomes obvious. The maximum number of files plus directories is constant and limited by the number of inodes on the filesystem, which on the 100 MiB test partition is 25 688. However, the number of directories created increases exponentially with both the number of hex digits per directory and then again by directory depth. This can be expressed mathematically.

Let h denote the number of hex digits per subdirectory and let n denote the subdirectory depth. Then the total number of directories created by the scheme, d is given by

$$d = \sum_{i=1}^{n} \left(16^h\right)^i \quad . \tag{7.1}$$

The directories are not created all at once, only when a file that should be placed in that directory is stored. But because files are named according to a uniformly distributed hash function, no particular directory will be favored and the number of directories will trend towards d.

Let o denote the number of inodes available on the filesystem, and let f denote the number of files that can be stored on the filesystem when the directory scheme creates d directories. Then,

$$f = o - d \quad , \tag{7.2}$$

And therefore,

$$f = o - \sum_{i=1}^{n} \left(16^h\right)^i \quad . \tag{7.3}$$

So we can see that DMV's original scheme, with two hex digits per directory and a depth of two, would yield 65 792 subdirectories, which by itself is more than 2.5 times the total number of inodes available on

the 100 MiB test partition. So of course it ran out of inodes long before running out of disk space (in terms of blocks).

Long Write Times

From there, we turn our attention to the mysterious, intermittent long write times. In the experiment, across all directory schemes, there were 315 601 total writes. Of those, 312 813 (99.1 %) completed in 1 ms or less. The others are plotted in Figure 7.3, and data about the top ten longest writes is listed in Table 7.2. The spikes in the graph appear as curves radiating out from zero files and zero directories. Each curve represents a cluster of directory schemes that filled up the disk in the same pattern, corresponding more to subdirectory depth than to number of hex digits per subdirectory.

No single directory scheme stands out as worse than the others, though longer writes seem correlated with having more directories and having more inodes already used (shown by distance from origin). The scheme with the fewest and shortest long writes is the one that has no subdirectories at all (shown by the short green spikes along the files axis). So we conclude that there is no penalty for storing many thousands of files in one directory.

The two longest writes are clustered together near the center of the plot, near 11 000 files and 14 000 directories. Both occur in the directory scheme with 1 hex digit and a depth of 4. The longest was 2.306 s at 10 699 files and 13 997 directories (96.2 % of inodes used), and the second was 2.180 s at 11 025 files and 14 289 directories (98.6 % of inodes used).

Write times seem to follow a power law distribution. The two peaks are the only two writes out of 315 601 total that took longer than 2 s. Five took longer than 1.5 s (including the two over 2 s). Sixteen are longer than 1 s, 155 are longer than 0.5 s, 340 are longer than 0.1 s, and 1381 are longer than 0.01 s. The vast majority (312 813, 99.1 %) completed in 1 ms or less.

The long writes appear to be spaced apart somewhat regularly. This suggests that they are caused by upkeep that the filesystem has to do periodically, and that there is no obvious way to avoid them, at least not while storing many small files. Aggregating objects into pack files

Figure 7.3: Unusually high write times

4 KiB files that took 1 ms or longer to write, plotted according to the number of files and directories on the disk already, and colored by subdirectory depth.
Not shown: The 99.1 % of writes that were faster than 1 ms.

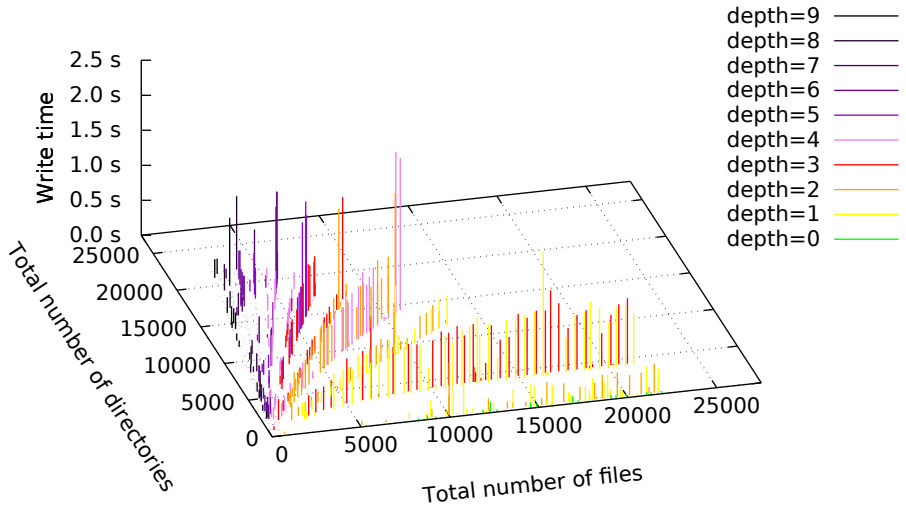


Table 7.2: Top-ten longest writes

Time (s)	Digits	Depth	Files	Dirs	Files in dir	% inodes used
2.306	1	4	10699	13997	1	96.2
2.180	1	4	11025	14289	1	98.6
1.775	3	1	16321	4008	5	79.2
1.654	1	5	5834	14755	1	80.2
1.646	3	2	10831	14635	1	99.2
1.466	1	5	5389	13790	1	74.7
1.456	2	3	8393	16550	1	97.1
1.443	4	2	7823	15225	1	89.8
1.434	1	5	5922	14949	1	81.3
1.379	1	6	5302	18885	1	94.2

would be a better strategy, as we saw when Bup was consistently the fastest VCS in the file-size and number-of-files experiments.

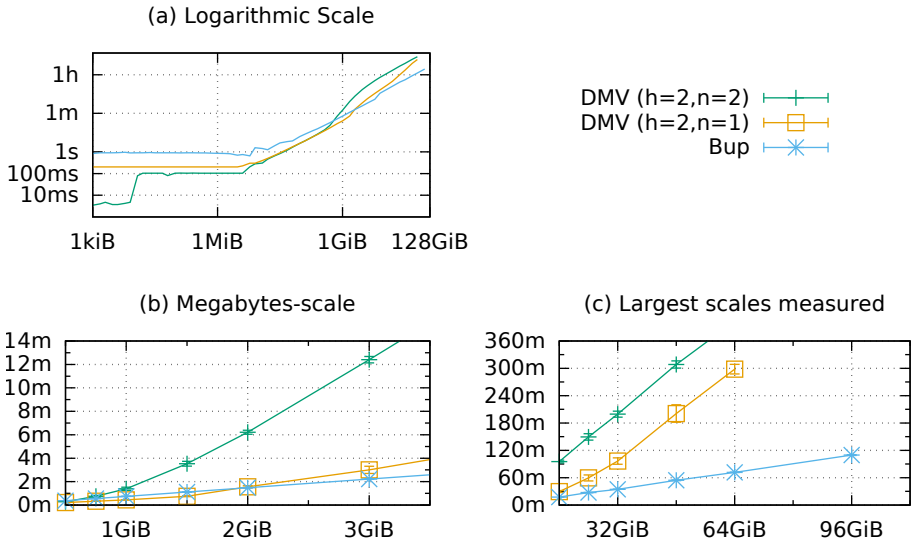
7.1.4 Object Directory Layouts in Action

We tuned the DMV prototype and re-ran the full file-size experiments (chapter 6) with two different directory schemes. First, with an early DMV version (fb2f43d) that used 2 hex digits per directory and a depth of 2, and then also with the reference DMV prototype (c9baf3a, as noted in Table 6.1) that used 2 hex digits per directory and a depth of 1. Figure 7.4 shows the initial commit times for both prototype versions, plus Bup for comparison.

As with the other runs of this experiment, the commit time for files under about 6 MiB is dominated by overhead. The depth-2 version of DMV has less overhead than the depth-1 version, and especially so at sizes up to 8 KiB, where the depth-2 version completes its commit in under 5 ms, rivaling the sub-millisecond write times for the 4 KiB files in the targeted experiment.

Figure 7.4: Time to commit one large file, with different object directory schemes

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars.



At up to 6 MiB, the depth-2 version has a consistent commit time of 103 ms to 104 ms while the depth-1 version has a consistent commit time of 201 ms to 202 ms. This difference might be caused by additional work that we did on DMV between running the experiments on the depth-2 version and switching to depth-1, including refactoring and adding some statistics collecting code to the verify (`fsck`) procedure. None of this should have impacted commit times directly, but it may have caused changes to the DMV executable's size or layout that made it take longer to load from disk and start up.

From 8 MiB to 384 MiB there is no noticeable difference between the two versions of DMV, but at 512 MiB and above, the commit results in enough chunk files that the directory layout starts to make a difference. The slight trend we noticed in Figure 7.3 for more directories to result in more long write times seems to have a more pronounced effect, and the depth-2 version starts to lag behind depth-1. At 768 MiB, the depth-1 version of DMV finally starts to lag behind Bup.

At 768 MiB and above, the commit times for both versions of DMV increase linearly with file size. They appear to have the same slope, with the depth-1 version shifted down. Bup, though also increasing linearly, does so with a flatter slope. This is further evidence to suggest that aggregating objects into pack files is not only less wasteful of disk space but also faster as the number of objects grows into the millions.

7.2 Linux I/O Scheduler

Since the anticipatory I/O scheduler was removed in version 2.6.33 [5], the Linux kernel has included three different I/O schedulers to choose from [32]:

Completely Fair Queueing The `cfq` scheduler is the default I/O scheduler as of Linux 2.6.18 [4]. It creates a separate queue for each process and handles requests in a round, preventing any one process from dominating I/O.

Deadline The `deadline` scheduler tries to set hard limits on wait time for scheduled I/O operations.

No-op The `noop` scheduler does as little as possible, passing requests directly to the device for it to manage. So this is the null benchmark for this experiment.

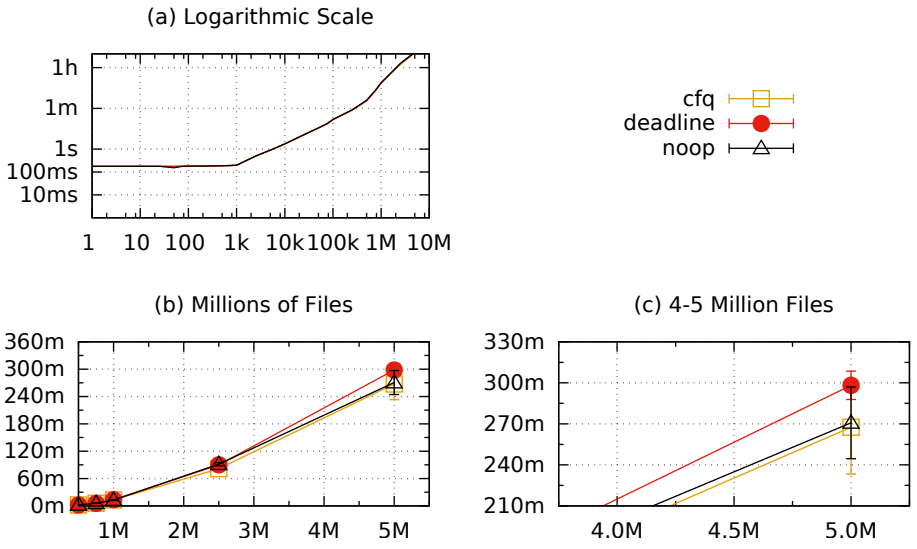
We were curious if the choice of scheduler would have any effect on performance. In particular, we aimed to document if it might reduce the long write times we were seeing. So we ran extra trials of the VCS scaling experiments using the DMV prototype (reference version c9baf3a) and different I/O schedulers.

The results of running the many-files experiment with different schedulers are shown in Figure 7.5. The I/O scheduler used made little difference. At 100 000 files, the average initial commit times were 19.666 s for CFQ, 19.708 s for deadline, and 19.598 s for no-op. The difference between each pair is less than any of the standard deviations at that number of files: 0.674 s, 0.3153 s, and 0.447 s, respectively.

In retrospect, these results are not surprising. The I/O scheduler deals mainly with juggling I/O access between different processes on the system, but the current DMV prototype is a single process. A multi-threaded or multi-process version of the prototype could give the scheduler something to work with.

Figure 7.5: Time for DMV prototype to commit an increasing number of 1KiB files to a fresh repository, by I/O scheduler

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars.



7.3 Chunk Size

The algorithm used to divide files into chunks (described in section 5.4) involves moving a window across the data, and setting a chunk boundary where the sum of the bytes in that window is evenly divisible by a given number. We ran a new experiment to determine the effects of these two parameters on chunk size.

7.3.1 Procedure

For each combination of window size and divisor, we would run the rolling hash algorithm on a stream of pseudorandom bytes until it had identified 100 chunks. Then we would compute the mean and standard deviation of the chunk sizes.

We used window sizes in powers of two from 128 B (2^7) to 128 KiB (2^{17}), and divisors in powers of two from 256 (2^8) to 128 Ki (2^{17}).

The pseudorandom number generator used was an xorshift RNG [23]. The experiment itself was automated as a unit test in the DMV prototype's Rust code.

7.3.2 Environment

Because this experiment measures only the output of calculations, the environment in which it is run should make no difference in the outcome. In fact, if the xorshift RNG is given the same initial seed value, the resulting random byte stream will be identical, which will lead to an identical sequence of chunks, which will lead to an identical average chunk size. This experiment is deterministic.

7.3.3 Window Reset Bug

During development, we noticed an error in the DMV rolling hash implementation: it would reset the window after every chunk. DMV's rolling hash implementation waits until the window is full before marking any chunks, so resetting the window after each chunk would force

Table 7.3: Chunk sizes for a window size of 4096

	Mean			Std.		
Divisor	As des.	w/reset	Diff	As des.	w/reset	Diff
256	300.8	4363.7	-4062.9	476.3	218.1	258.2
512	583.9	4622.9	-4039.0	718.9	512.6	206.3
1024	1069.1	5185.6	-4116.5	1176.7	1003.7	173.0
2048	1932.0	6242.9	-4310.9	2072.1	2179.4	-107.3
4096	4255.8	8629.8	-4374.0	5499.7	5449.4	50.3
8192	8324.8	13735.5	-5410.7	9506.4	9793.4	-287.0
16384	13304.2	20639.0	-7334.8	13263.2	16862.1	-3598.9
32768	13304.2	20639.0	-7334.8	13263.2	16862.1	-3598.9

it to fill again before marking the next chunk. This effectively created a minimum chunk size, the window size. We ran the experiment both with and without the bug to see how it would affect chunk size.

7.3.4 Results

Figure 7.6 shows the mean chunk sizes with the algorithm as designed, and Figure 7.7 shows them with the window reset bug. Table 7.3 shows specific values for a window size of 4096, both as designed (“as des.”) and with the reset bug (“w/reset”).

Increasing with divisor The overall trend is for chunk size to increase with both window size and divisor. As designed, though, the effect of the window size is much smaller, and the chunk size varies much more with the divisor. We did not perform a rigorous mathematical analysis of the algorithm, but this makes intuitive sense. The larger the divisor, the fewer numbers divide by it evenly, and so there is a lower probability that a given byte will push the sum to an even multiple. With a larger window, each bytes contribution to the sum is smaller, and that does have some effect on the probability, but the effect is less obvious.

Also, the first chunk must always be at least the size of the window, because the window has to fill before a chunk boundary can be triggered. Each value is the mean of 100 chunk sizes, and the first must be larger than the window. After the window is filled, the probability

Figure 7.6: Mean chunk size

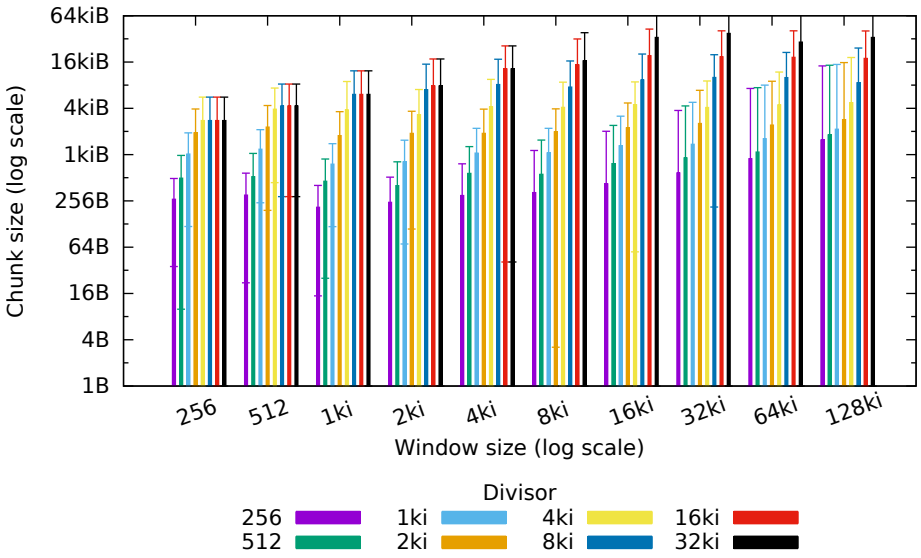
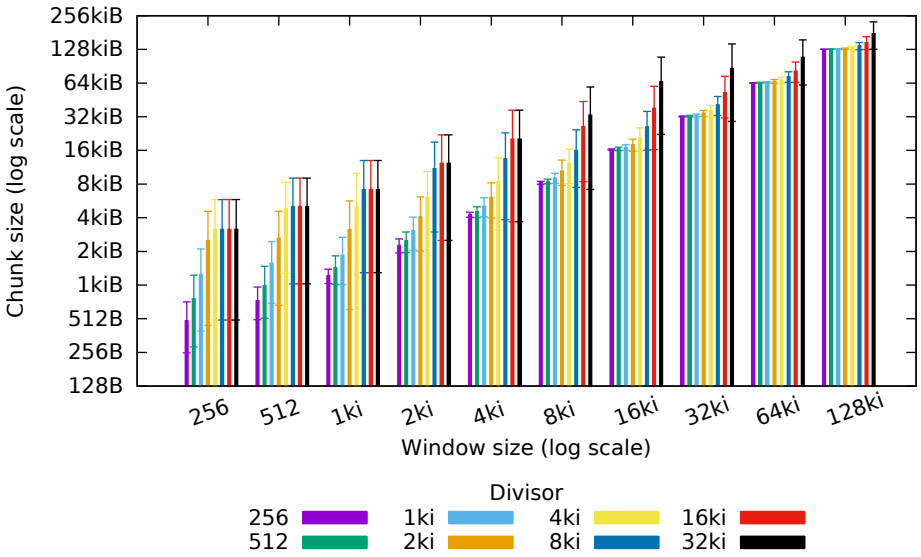


Figure 7.7: Mean chunk size, with reset bug



of triggering a chunk is the same, so the first chunk is w bytes larger than it would otherwise be. This would in turn increase the mean by $\frac{w}{100}$.

The standard deviation similarly tracks the divisor, with less effect from the window size. Interestingly, the standard deviation is often larger than the mean. So the chunk sizes are varying wildly. This also makes sense. Once the window is full, any byte could potentially trigger a chunk.

The greater effect of window size is that it sets the limit for the smallest pieces of identical content that can be identified. A common 1 KiB string will be lost inside a 4 KiB window. Chunks may be smaller than the window, but that indicates that the common data string started inside of the previous chunk. So for more effective de-duplication, it is probably better to set a smaller window size and then modify the divisor to tune the desired chunk size.

Reset bug makes every chunk a first chunk With the reset bug, standard deviations are similar to those without it, but mean chunk size is much larger. Interestingly, if one compares the mean chunk size with the reset to the mean chunk size without it for the same window size and divisor, it is often larger by an amount close to the window size itself. This also makes sense. The reset makes every chunk a first chunk, which is w bytes larger than it would otherwise be. So the window-reset bug's effect was to shift the mean chunk size up by the window size.

Flat tops There is another interesting effect in cases with small window sizes and large divisors. At some point, increasing the divisor has no more effect. A doubled divisor will yield exactly the same chunk size. This phenomenon can be seen in Table 7.3 where the values are identical for divisors of 16 384 and 32 768, and in Figure 7.6 and Figure 7.7, where the left-most histogram clusters flatten out on their right sides. This has to do with the fact that the divisors are powers of two, and that the same pseudorandom stream is used for every trial. When the divisor doubles, half of the values that would trigger chunk boundaries disappear, but half of them are exactly the same. For those trials, it just so happens that the first 100 chunk boundary trigger values that the al-

gorithm encounters in the byte stream are the ones that the divisors have in common.

7.3.5 Chunk Sizes in Action

We also ran the file-size experiments (chapter 6) with DMV versions that used four different rolling hash configurations. The reference DMV version used in other experiments (c9baf3a) had a window size of 4 KiB and a divisor of 16 Ki, chosen arbitrarily. It also has the window-reset bug. Later, in order to get a larger chunk size, we increased the window size to 32 KiB but left the divisor the same (version b134cca). It was after that that we discovered the window-reset bug and fixed it. We then re-ran the file-size experiments with both window sizes with the bug fixed (4 KiB in version a660730, 32 KiB in version 3e599e3). These versions and their mean chunk sizes are listed in Table 7.4, and the results are shown in Figure 7.8.

The file-size experiments are sensitive to chunk size because chunk size determines the number of chunk objects, and the number of objects on the disk affects write speed. Larger chunks leads to fewer objects which leads to less files and less filesystem overhead, as we found in the object-directory-layout experiment (section 7.1).

The version with a 4 KiB window and no reset bug has the smallest mean chunk size at 13.0 KiB and it is clearly the slowest. Next, the 4 KiB window with reset and the 32 KiB window without reset have similar mean chunk sizes at 20.1 KiB and 18.7 KiB respectively, and they have similar times. Finally, the 32 KiB window with reset has the largest mean chunk size at 52.5 KiB and it is clearly the fastest. However, we assume that as file sizes got even larger that it would also succumb to the many-files problem.

This is further evidence of the importance of aggregating objects into pack files. If files are packed, the number of objects becomes less of a concern, and then so does chunk size. The effects of chunk size then would be more subtle, smaller chunk sizes would let data be deduplicated with a finer granularity, better compressing the data. However, the smaller chunks would incur some overhead by way of larger chunked blob objects — because there would be more chunks to index. We would have to perform additional experiments that used real-world data to examine that trade-off.

Figure 7.8: Chunk sizes in action

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars.

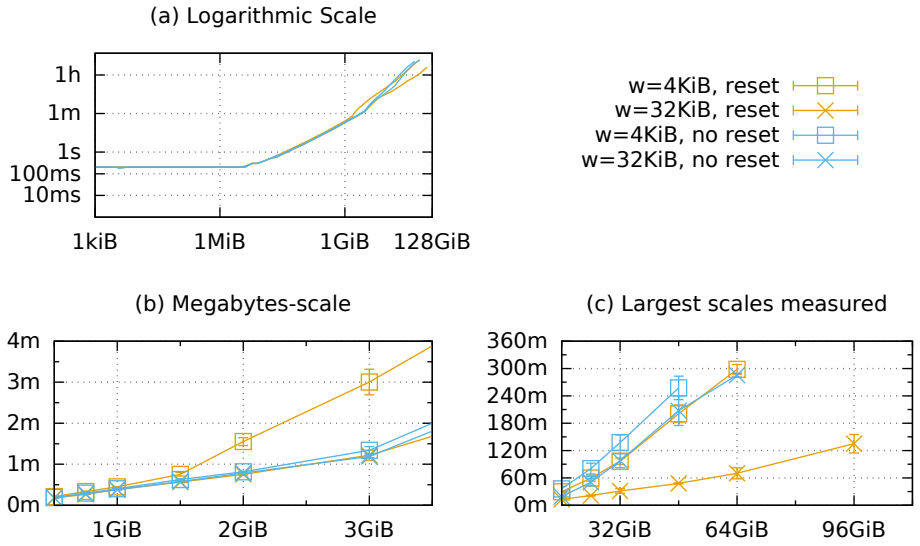


Table 7.4: DMV versions examined with different rolling hash configurations

Version	Window size	Divisor	Window reset	Mean	Std.
c9baf3a	4 KiB	16 Ki	Yes	20.1 KiB	16.5 KiB
b134cca	32 KiB	16 Ki	Yes	52.5 KiB	21.2 KiB
a660730	4 KiB	16 Ki	No	13.0 KiB	13.0 KiB
3e599e3	32 KiB	16 Ki	No	18.7 KiB	22.0 KiB

The “Mean” and “Std.” columns show the mean chunk size and its standard deviation for that configuration, as reported by the rolling-hash experiment (Figure 7.6).

Chapter 8

Discussion

8.1 Data Granularity and Storage Schemes

All four of the systems we examined in detail — Git, Mercurial, Bup, and DMV — model data and its history with a similar directed acyclic graph. The major difference is the granularity at which they work with data, and how they store it.

Both Git and Mercurial take the file as the basic unit of data granularity, though they approach storage differently. Git stores files whole as blobs during commit, storing them and other objects as files in object directories (as we experimented with in section 7.1). Later, an optional packing phase will compact objects together into pack files, where similar objects are stored as deltas against a base revision [8, Section 10.4]. Mercurial stores each file's different revisions as deltas against a base revision in a filelog structure [27, Chapter 4]. This is Mercurial's primary storage format, and it is constructed during commit.

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 (subsection 6.5.1).

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.

The current DMV prototype stores objects loose as files on the filesystem. This proves to be wasteful of disk space, taking up whole filesystem blocks with tiny objects and taking up inodes with subdirectories. Also, naming the object files by their SHA-1 hash effectively makes the filenames random, which causes dramatic write slowdowns compared to writing sequentially, as we saw in the number-of-files and object-directory-layout experiments (sections 6.6.2 and 7.1.3). Dividing into chunks solves the problem of storing large files by turning it into the problem of storing many files.

Bup's storage strategy is the best of both worlds. It first divides files into chunks, but then re-packs objects together into pack files. In fact, Bup uses Git's pack file format[1], but it writes it directly without the separate compacting phase, and without bothering to calculate deltas [29]. This makes efficient use of disk space, and allows the pack files to be written sequentially, minimizing disk seeks. This is why Bup was clearly the fastest of the systems evaluated in both the file-size and number-of-files experiments (sections 6.5.2 and 6.6.2).

So we see that the key to handling large files is to break them into

[1] Git has no notion of chunks, but Bup reuses Git's tree objects as chunk indices. Git can read a repository written by Bup, but it will see a large file as a directory full of small chunk files.

many smaller files, and the key to storing many small files is to combine them into larger files. The magic is in the combination, where files and revisions of files are broken into chunks by content, so that identical chunks are naturally de-duplicated in storage. That is what gives significant disk space savings over simply zipping up snapshots of the data.

So the next step for DMV will be to start aggregating objects.

8.2 Subtleties of the Rolling Hash

The rolling hash is the key to providing smaller granularity, because it is what identifies common byte strings within files. We saw in our chunk-size experiment (subsection 7.3.4), that smaller chunks lead to slower writes, but the effect was due to the number of files. Aggregating objects into pack files will alleviate that concern, so we can examine the subtler effects of chunk size.

First is the overhead of indexing the chunks. In DMV, we have the chunked blob objects, which keep a 168-byte record for every chunk: 160 bytes for the chunk's SHA-1 object ID, 4 for the chunk's offset within the file, and 4 for the chunk's size[2]. With an average chunk size of 4 KiB, a 1 GiB file would be broken into 262 144 chunks, which would require 42 MiB of chunk records. The blob header for each chunk would contribute some overhead as well. DMV's object header is currently only 8 bytes, but that still makes 2 MiB of additional overhead. Then there is metadata overhead in whatever scheme we use to store or pack the objects. Some overhead is inevitable, and 44 MiB per GiB of data is probably perfectly acceptable, but it is something to consider.

The less direct, but ultimately greater effect of chunk size is that it is the granularity of de-duplication. Smaller chunk sizes should lead to more de-duplication: small changes will lead to smaller changed chunks and

[2] Chunk size is actually redundant in the chunk index record, since it can be computed by comparing the chunk offset to the offset of the following chunk, or, in the case of the last chunk in the file, the total file size. We could eliminate it and save 1 MiB of overhead per GiB of data.

less new data to store. However, the utility of this de-duplication depends on how much redundancy is in the data, including:

- The degree of similarity between files in the data set
- How often the files are updated
- How much of the file is changed with each update

We hypothesize that most common media formats used by individual users (images, music, video, etc.) would have very little redundancy, and rarely be edited. Though some may update the internal metadata of their music files. The project files of popular professional software (Photoshop, ProTools, Avid, etc.) might be edited more frequently and benefit more from the de-duplication. We will need to perform additional experiments using real-world data to measure actual space savings.

8.3 DMV Prototype development

Unfortunately, the current DMV prototype does not yet include any of the planned networking features. However, basic local version control functionality is working, and the system can be used to store data, keep history, and retrieve old versions. Branching and rudimentary merging functionality are implemented as well (Figure 8.1).

Also, as a proof-of-concept for sharding the DAG, the DMV prototype is able to check out a subdirectory of the dataset to the working directory and commit changes to it. This demonstrates the ability to keep, and add to, a full history of part of the dataset, as described in section 4.2 and illustrated in Figure 4.3b.

8.4 Aggregating Data about a Sharded DAG

Though not implemented in the DMV prototype, we would like a DMV node to be able to aggregate data about what DAG objects are available at its neighbors and throughout the network. The data could be analyzed to give metrics about the data set in several dimensions:

Figure 8.1: DMV branching and merging functionality

(a) Actual output of DMV status command during merge

```
On branch master
P0: a76e9d0f Add new file
P1: a129ae17 Rename image

a  another_new_file.txt
a  melon—cat.jpg
 a new_file.txt
```

(b) Actual output of DMV log command after merge

```
* 04178597 (HEAD, master) Merge branch unstable
|\
* | a129ae17 (unstable) Rename image
* | 7bd4d7e8 Add something
| * a76e9d0f Add new file
| * f20aacd6 Remove image
|/
* cd214c6d Update greeting
* 7ef1b670 Initial commit
```

Coverage of data set How much of the data set is available locally or in neighboring nodes?

Coverage of data history How much of the data set's history is available locally or in neighboring nodes?

Divergence of versions How many different branches has this data been forked into, and how different are they?

Number of replicas How many times is the data replicated across neighboring nodes? Is any data in danger of being permanently lost?

Availability of or distance to replicas Of the replicas available, how available are they? What is the bandwidth of the connection to the neighboring nodes? What is the latency?

Such data could be useful for monitoring the health of the data set, alerting the user to shards of data that risk being lost without further replication.

8.5 Potential Applications of DMV

As a general distributed storage platform, DMV could have a wide range of potential applications:

Private data storage Individual users might use DMV to maintain a collection of important documents, photos, and media on their own devices, making it easier to keep up-to-date backups and to synchronize between computers, mobile devices, and removable drives without giving their data to a third-party cloud service.

Large-file version control Professional users that work with files too large for traditional version control, such as graphic designers, audio engineers, or video editors, might finally be able to adopt a version-control workflow.

Long-term data archiving Corporate or government users might use it to maintain large archives of data with full revision history.

Low-connectivity networks Far-flung networks with high-latency or rare connectivity, such as remote wildlife sensors or Mars rovers, could use it to manage and synchronize data.

8.6 What DMV should not do

We want to focus on the problem of storing file history and synchronizing files between replicas. We should be careful not to expand across the wrong abstraction boundaries or to try to do too much. In particular:

We do not want to reinvent the filesystem DMV should place and update files on the filesystem for applications to use normally, or offer a virtual filesystem as a view into a specific revision. Applications such as editors should not have to be rewritten to use our system.

We want to keep it simple We hope that DMV could eventually be used as a piece of infrastructure on which to build useful applications. It should not incorporate functionality that would better be left to an application.

We do not want to deal with media metadata and categorization

Metadata and categorization is best left to the applications that produce and consume those media formats. We will merely provide the storage.

However, knowledge of media formats might be used for behind-the-scenes optimization, such as setting chunk boundaries at natural boundaries in the file.

Chapter 9

Related Works

9.1 Distributed storage and synchronization systems

Camlistore Camlistore [13] 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 [26] 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 [39] 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 [16] and Git-media [7] 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 [25] 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 [42] 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 [38], Bup [29], and DMV (section 5.4). 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 SCM [10].

Rsyncrypto Rsyncrypto [38] 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 [38, 42], setting chunk boundaries where the checksum is zero. DMV uses this same chunking algorithm (see section 5.4).

9.2 De-duplicating Storage and Backup

Boar Boar [12] 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 [28] 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 (chapter 6), and we determined that Bup's manner of storing chunks is ideal for the next step for DMV (section 8.1). 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 [9] 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 [31]. 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 [18].

Chapter 10

Conclusion and Summary of Contributions

In this dissertation, we have examined the cryptographic directed acyclic graph (DAG) as a data structure for data storage, and the ways that it can be sliced to shard data across nodes in a distributed system, according to what data is needed locally at each location.

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.

We have performed experiments on the rolling hash algorithm used for chunking, and we have determined that adjusting the divisor has the most direct effect on chunk size. Larger divisors result in smaller chunks. And we have shown that adjusting window size has a lesser effect on chunk size, but we reason that smaller window sizes will be able to find smaller common chunks in the code.

And finally, we have described the idea, architecture, design, and implementation of a distributed data storage system we call *Distributed Media Versioning (DMV)* that expands on the distributed version control concept to store larger and more diverse data sets, with a high degree of control over data locality, and an availability to write updates for any data held locally. Though time constraints prevented us from implementing the network features we had planned, the DMV prototype has enough functionality to be experimented on against existing distributed version control systems and to demonstrate the addition of new commits to a partial DAG.

Chapter 11

Future Work

The primary goal of DMV is to be a general low-level storage platform for storing and tracking a data set across many nodes. The next step (after packing objects) would be to complete the networking features of the prototype so that it can actually be used in real-life applications.

Applications could potentially layer additional technologies on top of DMV to create interesting systems. For example, a gossip protocol could spread information about object availability on remote data stores that are not directly connected, allowing data to be spread and transfered across far-flung networks. To focus on usability, daemons could automatically commit changes and sync with other nodes, shielding users from the complexity of branching where possible and trying to present a coherent current state of the data set. An important optimization would be to create a virtual 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. Such a virtual filesystem would also make it much easier to write an auto-commit daemon, since writes would have to go through the filesystem.

We look forward to continuing to work on and expand the system. The DMV project is open source, and development continues online at `http://dmv.sleepymurph.com/` .

Glossary

ABI Application binary interface, the public interface between a system library and a client application. 29

ACID Atomicity, consistency, durability, and isolation, the guarantees of a traditional database commit. 3

blob Binary large object, a sequence of unstructured binary data. In Git and DMV, a DAG object holding file data. 5–7, 9, 21, 23, 81, 82, *see* DAG

branch In a version control system, separate concurrent lines of update history. 4, 7, *see* head & merge

CAP-theorem The fundamental theorem of distributed systems, that no system can simultaneously be consistent (C), available (A), and tolerant of network partitions (P). 3, *see* partition decision

chunked blob In DMV, a DAG object that is an index of blobs that make up a larger blob. 21, 23, 32, 78, 83, *see* DAG & blob

commit In version control, the operation for storing a particular version of the data. Also, the resulting DAG object that represents that version in the history. 6, 7, 21, 23, 27, 31–33, 35, 37, 38, 41, 43, 45, 49, 53, 55, 57, 63, 72, 81, 82, 96, 97, *see* DAG

content addressable storage Storage that stores immutable objects named by a hash of their content, which naturally de-duplicates identical objects. 5, 7, 17

DAG Directed acyclic graph, the type of graph data structure used to represent history in many distributed version control systems. Directed meaning all the edges have a direction, from one node to

another, and acyclic meaning that there are no cycles, no paths that revisit any node. 6, 7, 9, 11–14, 17, 18, 21, 23, 25, 84, 89, 95, 96, *see* blob, chunked blob, tree, commit & ref

divisor In a rolling hash algorithm, the divisor in the modulus operation. A chunk boundary is created when the sum of the bytes in the window, modulo this divisor, is equal to zero. 30, 96, *see* rolling hash & window size

DMV Distributed Media Versioning, the new distributed data storage platform described and introduced in this dissertation. iii, 9, 11–14, 17, 18, 21, 23, 27, 29, 31, 32, 38, 41, 43, 57, 59, 61, 63, 65, 67, 72, 74, 81–84, 86, 90–92, 96, 97, 108

DVCS Distributed version control system, such as Git, where individual repositories can operate independently without having to connect to a central repository. iii, 1, 2, 4, 5, 12, 13, 27, 31, 95, 96, *see* VCS

end-to-end argument When designing a communications system, the idea that there is certain functionality that can only be implemented correctly by the higher-level application at the endpoints of the communication, and so it is futile for the communication system to try to provide that functionality itself. 4, 12

filelog Mercurial's file format that stores different versions of the same file as a base version followed by a series of delta. 31, 57, 61, 81

head In a version control system, the most recent revision in a branch. 7, *see* branch

inode A data structure in a Unix filesystem that stores file metadata. Each filesystem has a fixed number of inodes, which limits the total number of files and directories the filesystem can hold. 29, 51, 63, 65, 67, 68, 82, 95

merge In a version control system, an operation that combines two branches and reconciles conflicting changes. 4, 7, *see* branch

object store Content-addressable storage for DAG objects. 5, 7, 17, 29, 31, 51, 53, 65, 90, *see* DAG

pack file An object store file format that aggregates many objects in one file. 9, 32, 37, 41, 49, 51, 57, 61, 68, 71, 78, 81–83, 95, 97, *see* object store

partition decision The dilemma faced by a distributed system during a network partition: to decrease availability or risk inconsistency. 3, *see* CAP-theorem

ref A reference to a commit object in the DAG. 7, 9, 23, *see* DAG

repository A location where data is stored in a version control system. Early systems would have a central repository that clients would check out from. In distributed version control, every client is a separate repository. iii, 2, 4, 5, 7, 9, 17, 18, 23, 25, 32, 33, 35, 37, 41, 43, 49, 53, 61, 90, 92

RNG random number generator. 74

rolling hash A hash checksum that operates over a moving window of data in a byte stream that can be used to find repeating patterns. iii, 23, 29, 30, 32, 74, 78, 82, 83, 91, 92, *see* window size & divisor

SCM Source code manager, a version control system that is designed primarily to store source code. 4, 5, 91, *see* VCS

tree In Git and DMV, a DAG object representing a particular state of a file hierarchy. 5–7, 21, 23, 32, 82, 97, *see* DAG

VCS Version control system, a program that stores many versions of a file or set of files, commonly used to track changes to source code. 1, 4, 5, 12, 31–33, 35, 38, 59, 70, 92, 95, *see* SCM

window size In a rolling hash algorithm, the number of previous bytes used in the rolling sum. 30, 77, 96, *see* rolling hash & divisor

working directory A directory where files that are tracked by a version control system are actively worked on and edited. 17, 29

Bibliography

[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 (cit. on p. 5).

[2] BENET, J. et al. *IPFS: The Interplanetary Filesystem*. GitHub. 2014. URL: https://github.com/ipfs/ipfs (cit. on p. 90).

[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 (cit. on p. 3).

[4] CALLEJA, D. et al. *Linux 2.6.18 Release Notes*. Kernel Newbies Wiki. URL: https://kernelnewbies.org/Linux_2_6_18 (visited on Apr. 26, 2017) (cit. on p. 72).

[5] CALLEJA, D. et al. *Linux 2.6.33 Release Notes*. Kernel Newbies Wiki. URL: https://kernelnewbies.org/Linux_2_6_33 (visited on Apr. 26, 2017) (cit. on p. 72).

[6] CEDERQVIST, P., PESCH, R., et al. *Version management with CVS*. Network Theory Ltd., 2002. URL: https://ftp.gnu.org/non-gnu/cvs/source/stable/1.11.22/cederqvist-1.11.22.pdf (visited on May 11, 2017) (cit. on p. 4).

[7] CHACON, S., LEBEDEV, A., et al. *git-media*. URL: https://github.com/alebedev/git-media (cit. on p. 90).

[8] 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) (cit. on p. 81).

[9] CISLER, P. et al. *System for electronic backup*. US Patent App. 11/499,848. 2008. URL: https://www.google.com/patents/US20080034004 (cit. on p. 92).

[10] 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) (cit. on pp. 4, 5, 91).

[11] DeCandia, G. et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281 (cit. on p. 3).

[12] Ekberg, M. et al. *Boar*. URL: http://www.boarvcs.org/ (cit. on p. 92).

[13] Fitzpatrick, B. et al. *Camlistore is your personal storage system for life*. URL: https://camlistore.org/ (cit. on p. 89).

[14] 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 (cit. on p. 3).

[15] Gilbert, S. and Lynch, N. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601 (cit. on p. 3).

[16] Hess, J. et al. *git-annex*. 2015. URL: http://git-annex.branchable.com/ (cit. on p. 90).

[17] Humble, J. and Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010 (cit. on p. 2).

[18] 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) (cit. on p. 93).

[19] Karp, R. M. and Rabin, M. O. "Efficient randomized pattern-matching algorithms". In: *IBM Journal of Research and Development* 31.2 (1987), pp. 249–260. ISSN: 0018-8646. DOI: 10.1147/rd.312.0249 (cit. on p. 23).

[20] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563 (cit. on p. 3).

[21] Lamport, L. et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25. URL: http://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/past/03F/notes/paxos-simple.pdf (visited on May 1, 2017) (cit. on p. 3).

[22] *Manual page for proc*. Linux man-pages project. `man proc`. URL: `http://man7.org/linux/man-pages/man5/proc.5.html` (visited on May 11, 2017) (cit. on pp. 35, 45).

[23] MARSAGLIA, G. et al. "Xorshift RNGs". In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6 (cit. on p. 74).

[24] MATSAKIS, N. D. and KLOCK II, F. S. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT '14. Portland, Oregon, USA: ACM, 2014, pp. 103–104. ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188. URL: http://doi.acm.org/10.1145/2663171.2663188 (cit. on p. 29).

[25] MAYMOUNKOV, P. and MAZIÈRES, D. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Peer-to-Peer Systems: First InternationalWorkshop, 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 (cit. on p. 91).

[26] OGDEN, M., BUUS, M., MCKELVEY, K., et al. *Dat Data*. URL: http://dat-data.com/ (cit. on p. 89).

[27] O'SULLIVAN, B. *Mercurial: The Definitive Guide*. O'Reilly Media, Inc., 2009. ISBN: 0596800673, 9780596800673. URL: http://hgbook.red-bean.com/ (cit. on pp. 31, 81).

[28] PENNARUN, A., BROWNING, R., et al. *Bup, it backs things up*. URL: https://bup.github.io/ (visited on Apr. 26, 2017) (cit. on p. 92).

[29] 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) (cit. on pp. 32, 82, 91).

[30] PILATO, C. M., COLLINS-SUSSMAN, B., and FITZPATRICK, B. *Version control with subversion*. " O'Reilly Media, Inc.", 2008. URL: http://svnbook.red-bean.com/ (visited on May 11, 2017) (cit. on p. 4).

[31] POND, J. *How Time Machine Works its Magic*. Website. Aug. 31, 2013. URL: http://pondini.org/TM/Works.html (visited on May 12, 2017) (cit. on p. 93).

[32] PRATT, S. and HEGER, D. A. "Workload dependent performance evaluation of the Linux 2.6 I/O schedulers". In: *2004 Linux Symposium*. 2004. URL: https://www.kernel.org/doc/ols/2004/ols2004v2-pages-139-162.pdf (visited on Apr. 26, 2017) (cit. on p. 72).

[33] 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 (cit. on p. 51).

[34] 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 (cit. on pp. 4, 5).

[35] SADALAGE, P. J. and FOWLER, M. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012 (cit. on p. 3).

[36] SALTZER, J. H., REED, D. P., and CLARK, D. D. "End-to-end Arguments in System Design". In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 277–288. ISSN: 0734-2071. DOI: 10.1145/357401.357402. URL: http://doi.acm.org/10.1145/357401.357402 (cit. on pp. 4, 12).

[37] SHAPIRO, M. and PREGUIÇA, N. *Designing a commutative replicated data type*. Research Report RR-6320. INRIA, 2007. URL: https://hal.inria.fr/inria-00177693 (cit. on p. 3).

[38] SHEMESH, S. *Rsyncrypto algorithm*. Rsyncrypto home page. URL: https://rsyncrypto.lingnu.com/index.php/Algorithm (visited on Apr. 17, 2017) (cit. on pp. 29, 91, 92).

[39] 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 (cit. on p. 90).

[40] 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) (cit. on p. 7).

[41] TORVALDS, L. *V2.0.8*. Linux Kernel Mailing List. July 1996. URL: http://www.webcitation.org/6P8EBZqQX (visited on Apr. 7, 2017) (cit. on p. 2).

[42] 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) (cit. on pp. 30, 91, 92).

[43] VAN RENESSE, R. and ALTINBUKEN, D. "Paxos Made Moderately Complex". In: *ACM Comput. Surv.* 47.3 (Feb. 2015), 42:1–42:36. ISSN: 0360-0300. DOI: 10.1145/2673577. URL: http://doi.acm.org/10.1145/2673577 (cit. on p. 3).

[44] 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) (cit. on p. 2).

Typeset by LaTeX.

This document, DMV source code, and other related materials are archived in Munin, the University of Tromsø's open research archive:
http://munin.uit.no

Active source repositories for the DMV project, this document, and other related related materials can be found via the author's website:
http://dmv.sleepymurph.com/

This document generated 2017-05-15 from revision 7a2ca58 of the dmv-publications repository.