ENABLING EFFICIENT STORAGE OF GIT REPOSITORIES IN PACLAB

Rebecca Brunner

A Thesis
Submitted to the Graduate College of Bowling Green State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2020

Committee:
Robert Dyer, Advisor
Michael Decker
Robert C. Green II
ABSTRACT

Robert Dyer, Advisor

A current trend in the field of static analysis is to use open source platforms such as GitHub to develop benchmark program suites. These suites are used for the evaluation of static analysis tools. Programs collected from GitHub often must be transformed by hand to work with these tools. PAClab is an online platform that automatically performs code transformations to solve this problem. PAClab stores a base selection of Git projects that expand over time, creating a need to maintain multiple versions of each Git project effectively. In this work, we propose a solution to store multiple versions of a Git project efficiently in the PAClab system. We visualize Git projects as a graph, and by focusing on reducing the graph size and removing node duplication, we found filtering projects reduced the storage size of 2,955 Git projects by 96%. Compared with a naive approach, we also found our approach utilized around 6x less space when storing a week’s worth of snapshots.
ACKNOWLEDGMENTS

This research was supported in part by the U.S. National Science Foundation (NSF) under grant 18-23294. Without their support, this research would not be possible.

I also want to thank my advisor, Dr. Robert Dyer. Dr. Dyer offered me a position as his research assistant before I even began my journey through graduate school. This assistantship has been the highlight of my current academic career, offering an incredible learning experience that has cemented my interest in computer science and software engineering. Dr. Dyer generously extended his grant to support me through my two-year degree, with this financial support, I have been able to pay for my master’s education without incurring debt. Working on completing my thesis under his direction, I have received a taste of what it is like to conduct a research project. I hope to take this experience with me into my future, and I have started to develop a budding interest in performing research.

I also thank my committee members, Dr. Robert Green and Dr. Michael Decker. Both Dr. Green and Dr. Decker have served as my professors over the last two years, and have provided me with invaluable knowledge for completing my thesis. I also thank Dr. Hassan Rajaei, Dr. Decker, and Jadwiga Carlson for providing me with recommendations for entrance into graduate school. They all taught me as part of my undergraduate career and trusted in my abilities. They vouched for me and played a significant role in my acceptance into the program. Jadwiga Carlson has supported me as a computer scientist during both my undergraduate and graduate careers.

Finally, I thank my friends and family. My parents Patricia Brunner and George Brunner, have been incredibly supportive of my choice to further my education and have supported me both financially and emotionally. These last few months have been challenging for the whole world, and I am blessed to have close family members and friends who have watched out for me in these difficult times.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CHAPTER 1</strong></td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td><strong>CHAPTER 2</strong></td>
<td>BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>PAClab - the Program Analysis Collaboratory</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Git</td>
<td>4</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Git Graphs</td>
<td>5</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Other Useful Git Commands</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Zettabyte File System (ZFS)</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Snapshots and Cloning</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Deduplication</td>
<td>11</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Compression</td>
<td>12</td>
</tr>
<tr>
<td><strong>CHAPTER 3</strong></td>
<td>PROPOSED SOLUTION</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>ZFS</td>
<td>14</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Compression</td>
<td>14</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Deduplication</td>
<td>15</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Snapshots and Cloning</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Git</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Sparse Checkouts and Sparse Clones</td>
<td>16</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Blobs</td>
<td>18</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Filter-Branch</td>
<td>19</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Shallow Clones, Pulls, and Fetches</td>
<td>20</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Multi-layered Git Graphs</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Proposed Solution</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Custom Scripts</td>
<td>26</td>
</tr>
</tbody>
</table>
CHAPTER 4 EVALUATION

4.1 Obtaining the Datasets
4.2 Evaluating Storage Savings
  4.2.1 General Statistics
  4.2.2 Comparing Distributions
  4.2.3 Overall Size Reduction
  4.2.4 Relationship Between Repository Size and Savings
4.3 Evaluating our Approach
  4.3.1 Strategies for Storing Changes
  4.3.2 Cost of Unpacking
  4.3.3 Deduplication with ZFS
  4.3.4 Delta Compression ZFS
4.4 Result Summary
4.5 Threats to Validity

CHAPTER 5 RELATED WORKS

5.1 Mining Software Repository Frameworks
5.2 Graph Optimizations
5.3 Large Git Repositories
5.4 Filesystems

CHAPTER 6 FUTURE WORK AND CONCLUSION

BIBLIOGRAPHY
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Blob representation in Git</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Tree representation in Git</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Graph structure of trees and blobs</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Commit representation in Git</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Graph structure of commits, trees and blobs</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>Using git cat-file to display the contents of a tag object</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Shallow clone of the repository shown in Figure 2.5</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>The repository shown in Figure 2.5 after running filter-repo to keep only Java source files</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>The repository shown in Figure 2.5 after performing a shallow clone then running filter-repo to keep only Java source files</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Git graph evolution over time</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>The create script, employing several key storage saving strategies</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Script for updating a repository</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Failsafe script</td>
<td>28</td>
</tr>
<tr>
<td>3.8</td>
<td>Representation of Transformed Project Table in PAClab’s backend</td>
<td>29</td>
</tr>
<tr>
<td>3.9</td>
<td>Visualization of storage infrastructure</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Size distributions for control, shallow, filtered, and shallow filtered projects</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Median values for each group</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Mean values for each group</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Control group size vs. savings from shallow cloning</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>Control group size vs. savings from filtering</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Control group size vs. savings from shallow cloning and filtering</td>
<td>39</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Results from testing deleting blobs</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Results for using filter-branch</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Testing filter-repo against force pushes</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Repository size statistics for 2,955 Java-based Git projects from GitHub</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Compression ratio statistics for 2,955 Java-based Git projects from GitHub</td>
<td>32</td>
</tr>
<tr>
<td>4.3 Difference between boxplot distributions</td>
<td>33</td>
</tr>
<tr>
<td>4.4 Statistics for unpacked filtered projects vs. packed filtered projects</td>
<td>37</td>
</tr>
<tr>
<td>4.5 Total sizes for naive approach vs. our efficient approach</td>
<td>41</td>
</tr>
</tbody>
</table>
CHAPTER 1 INTRODUCTION

Static analysis tools are used in the fields of programming languages and software engineering in order to evaluate programs before execution. Practical examples of static analysis include usage in modern IDEs, compilers, and bug detecting tools. Static analysis researchers evaluate their tools by running them against benchmark programs. These benchmark programs are often out of date and do not give an accurate representation of real-world applications. In order to obtain up to date benchmark programs, researchers have turned to open source platforms such as GitHub to mine candidate programs. However for static analysis tools that infer semantics at a high level, many programs found on these platforms do not have the necessary characteristics for usage as a benchmark program. As a result researchers must perform manual transformations of a project’s source code in order to get the selection to have the necessary criteria. This process is time consuming, and typically results in a small selection of benchmark programs limiting development of such tools.

PAClab is an online collaboratory system in which researchers in the field of static analysis can create and share datasets by automatically filtering data sources and applying transformations to found projects [10]. PAClab consists of a service for filtering projects from a data source in order to select potential benchmark programs, a repository management system that stores and caches Git projects, and a transformation service that automatically performs user-defined transformations on candidate selections. PAClab attempts to address the issues mentioned in the field of static analysis by providing a way to develop benchmark programs from real-world data as well as eliminating the need for manual transformations, allowing for bigger sample sizes for testing more heavyweight static analysis tools. PAClab also facilitates collaboration by promoting the sharing of datasets.

PAClab’s repository management system consists of only Git projects. Each Git project can be visualized as a graph, consisting of four different types of objects, tags, commits, trees, and blobs. As these graphs evolve and change, PAClab needs to maintain access to all old versions of the graph so that users can access old datasets for the foreseen future, but also have access to
newer data if desired. The main objective of this study is to create a system that stores Git graphs efficiently so that we have the storage capacity to maintain older Git graphs while including new changes.

In order to efficiently store Git graphs, we focused on two ideas. The first idea was to reduce the overall size of each Git graph, with the general principle being the larger the graph the more space it consumes. In order to accomplish this we focus on pruning unnecessary branches and removing unwanted objects. The second idea is based on the premise that as a graph evolves, it will still contain many identical components to its ancestor. We wanted to have a way to distinctly store different versions of a graph, however only maintain one instance of repeated vertices and edges.

We selected 2,955 projects on GitHub containing Java code to evaluate our methodology. We ran two tests. In the first test, we evaluated the individual components of our proposed solution. We initially cloned each of the 2,955 projects and collected their initial file size. We then shallow cloned each project and recorded its file size. Finally, we cloned the project and filtered out all files that were not Java and recorded the file size. We found that shallow clones did not affect file size as initially anticipated, however filtering projects reduced the size of our testing dataset by 96%.

In our second test, we ran a naive approach and our approach over a week-long time frame. We used the same 2,955 projects and recorded file size for each project after receiving a week’s worth of daily pulls. We also examined different strategies for storing changes. We collected data on whether force pushes had occurred or not and file size and after collecting the data, we analyzed file sizes from the naive approach and our approach. We found our approach utilized around 6x less space than the naive approach.

In the next chapter, we provide background information about some of the technologies used in our approach. In Chapter 3 we outline our proposed solution. We then evaluate that solution in Chapter 4. Chapter 5 provides details on closely related prior works. We then discuss some future work in and conclude in Chapter 6.
CHAPTER 2  BACKGROUND

In this chapter, we outline some of the technologies used by our approach. The first is PAClab, an online system designed to promote collaboration and facilitate static program analysis research. The second is ZFS, a filesystem technology. Finally, we used Git, a distributed version control system used to manage stored files. Here we provide an overview of PAClab’s design philosophies and architecture, note some of ZFS’s main features, and examine some of Git’s internal architecture and lower-level commands.

2.1 PAClab - the Program Analysis Collaboratory

Static analysis tools are validated by running them against a collection of benchmark programs. These programs are typically sourced from benchmark datasets, however because these datasets are not well maintained, researchers turned to open-source software platforms for the collection of benchmark programs.

For static analysis tools that need to interpret semantics of programs at the highest level, many open-source programs are not initially viable candidates to be selected as a benchmark program. Many programs require transformations in order to be run through a static analysis tool. These transformations are done manually, increasing time to develop these datasets while also decreasing the total number of benchmark programs used for testing.

PAClab attempts to solve the above issues by increasing collaboration between researchers and sharing of datasets, as well as providing automatic dataset generation [10]. PAClab can be split up into four stages. The first stage is to collect potential open-source projects. The second stage is to filter this data to fit selected criteria. This criterion is defined by users through PAClab’s online interface. The next stage is to perform user-defined code transformations on the generated dataset, which further filters the dataset. Finally, the resulting transformed dataset is gathered into a zip and provided for download to the user. Every part of this process is designed to be chained and shared with others to promote collaboration [10].
Currently, the only data source supported by PAClab is GitHub. The users begin the process by accessing PAClab’s interface which is provided through a website. The user creates a project selection, which is a model containing user-defined criteria to filter out GitHub projects. Example filters include filtering by number of commits or source files [10].

These filters are then generated into an initial discovery query. Currently, PAClab supports the query language Boa for initial GitHub project discovery [10, 11]. PAClab’s architecture is designed to be modular, so Boa could be replaced or used alongside additional backends in the future. Boa runs the generated query and returns a list of potential GitHub projects. PAClab proceeds to clone all projects in the generated list [10] and then performs final filtering based on the most recent Git repository data.

After the list of URLs has been cloned, the projects are then transformed. The user begins this process by beginning with either a selection of Git projects or an existing set of transformed projects, then defines what transformations are to be performed. Before transformation, the transformer further filters out projects based on transformation parameters. The transformer then automatically transforms the code and stores all successful transformations in our cluster [10].

The final stage of the process is to make these transformations available for download. Projects are stored across the cluster. The location of a project stored on this cluster is contained in a table as part of a relational database used by PAClab. The table contains the host and path of the project, which is used for forming the download.

2.2 Git

All PAClab projects are stored in Git repositories taken from GitHub. Git is a popular distributed version control system, and GitHub is a popular website for hosting remote Git repositories [8, 10, 30]. All files stored in a Git repository can be represented as a graph, and reducing the size of the graph results in a more compact project for long term storage.
Figure 2.1 Blob representation in Git. *git hash-object* is a lower level command to store content in the Git objects folder, without creating the typical commit → tree → blob structure [14, Ch.10.2]

```
$ echo "test" > test.txt && git hash-object -w test.txt
9daeafb9864cf43055ae93beb0afd6c7d144bfa4
$ find .git/objects -type f
  .git/objects/9d/aeafb9864cf43055ae93beb0afd6c7d144bfa4
$ git cat-file -t 9daea
blob
$ git cat-file -p 9daea
test
```

2.2.1 Git Graphs

Git stores every tracked file in a graph [6, 14, 36]. This graph consists of Git’s basic data structures which are blobs, trees, commits, and tags. For explaining the different Git data structures, we walk through a small example by creating each object manually instead of using high-level operations such as *git commit*. This example uses the same structure as the demo in the Git SCM documentation [14].

Figure 2.1 details the process of creating the first type of Git object: a blob. The blob data structure contains all the file content for a committed file. The name of the blob file is a hashed 40 character checksum used as an identifier by Git and for data integrity verification. The blob object does not contain filename information, author, commit message, or other information typically saved when committing objects [14].

Figure 2.2 shows Git’s tree data structure. The tree data structure is in charge of holding all file names for stored blobs as well as maintaining the folder structure of committed files. Subdirectories are stored as trees which point to more files and more trees. Figure 2.3 gives a visual representation of this relationship between blobs and trees.

Figure 2.4 shows the creation of commit objects based on the trees created in the last step. Commit objects store commit messages, information about the author, and ultimately build a history of changes. Figure 2.5 provides a visual representation of what this structure looks like. The graph can be created using the *cat-file* command [14, Ch.10.2]. At this point, *git log* can
Figure 2.2 Tree representation in Git. `git update-index` creates a staging area, necessary for Git to create a tree. Notice `src/` is another tree [14].

```
1 $ git update-index --add --cacheinfo 100644 9
daeafb9864cf43055ae93beb0afd6c7d144bfa4 test.txt
2 $ git write-tree
3 2b297e643c551e76cfa1f93810c50811382f9117
4 $ echo "data/" > .gitignore
5 $ echo "Hello World" > README.md
6 $ git update-index --add .gitignore README.md
7 $ git write-tree
8 c316ce13aade220153e060d94ada4200c3c24ba
9 $ git read-tree --prefix=src 2b297
10 $ git write-tree
11 bccbb552a10ea99041668fb9f9653b48d5c8303d
12 $ git cat-file -t bccbb
tree
13 $ git cat-file -p bccbb
14 100644 blob 8fce603003c1e5857013afec915ace9fc8bcdb8d .gitignore
15 100644 blob 557db03de997c86a4a028e1ebd3a1ceb225be238 README.md
16 040000 tree 2b297e643c551e76cfa1f93810c50811382f9117 src
```

Figure 2.3 Graph structure of trees and blobs. Based on graphs from the Git object documentation [14] Ch.10.2].
The final type of Git object is a tag. A tag points to another Git object, usually a commit. A tag includes the SHA-1 object key, the type of object, tag name, potential messages, and other metadata such as author and date. Tags can be created using the `git tag` command. See Figure 2.6 for an example.

As seen in both Figure 2.3 and Figure 2.5, over time a graph is built up of a Git repository. We shrink graphs of downloaded projects by pruning leaves and branches that we do not need to store. In general, it can be assumed that a smaller internal Git graph means a smaller total file size. Ways to shrink this graph include creating shallow clones and filtering unnecessary blobs.

### 2.2.2 Other Useful Git Commands

Git contains a working directory and the Git repository itself. The repository is stored in the `.git` directory in the root of a project file. The `.git` folder contains an object directory that contains all packed or unpacked objects. In order to remove clutter, Git will store objects together in one compressed file called a *pack* file. Otherwise, objects are stored in files named after all but the first two identifiers of the object’s unique SHA-1 hash value [6][14][36].

The working directory is what is visible to the user, and contains all files and folders that are available to be edited. A clone or pull essentially duplicates what is being stored. One copy is
Figure 2.5 Graph structure of commits, trees and blobs. Based on graphs from Git object documentation [14, Ch.10.2].

Figure 2.6 Using git cat-file to display the contents of a tag object

```
1 $ git tag --a v1.0 2f949553c4a4937fc4bc1bedf077e7112ea5310e --m "release 1"
2 $ git cat-file tag v1.0
3 object 2f949553c4a4937fc4bc1bedf077e7112ea5310e
4 type commit
5 tag v1.0
6 tagger Rebecca Brunner <brunner@bgsu.edu> 1582852533 -0500
7 release 1
```
uncompressed, fully editable in the working directory, the other copy is compressed stored in the
.git directory either as loose object files or packed together in one packfile [6, 14, 36].

One way to minimize space is to not checkout a copy of the working directory. A checkout
copies over the files from the .git repo into the visible working space – in other words it du-
plicates them. By not having a working directory, you still have access to all data but in a much
smaller compressed format within the Git directory [6, 14, 36].

There are multiple ways to achieve this effect. A Git user can specify the --no-checkout
option when initially cloning the project. This will only copy the .git folder with no visible files [6, 14, 36]. Another option is to use the --bare option during cloning. The bare option
forgoes the working directory altogether, only giving access to the .git folder. In order to create
a working directory of a bare repository, you can re-clone the folder locally to create a working
directory or disable the bare flag in the project’s settings [6, 14, 36]. Another non-git solution
is to remove all files and folders from the project folder except the .git directory - manually
performing the operation of --no-checkout [6, 14, 36]. The important aspect is that Git is still
fully operational in every aspect including access to pull and fetch commands.

Another way to reduce the size of a repository is to only clone the latest commit. Since we
have no intention of committing to the project and only want the current version of data, we do
not need the project’s history. Normally when you perform a clone you receive the entire history
of a project. By adding the --depth=1 flag to a clone, you only copy the current version of the
project [6, 14, 36]. On files with large histories, this can have a significant effect on minimizing
storage size, and also reduces time to clone the repository by cutting out commit and tree objects
that are not needed. Shallow cloning actually results in a structural change to the git repository
itself. It removes all but one commit object and also removes trees that are no longer referenced.
This results in shrinking the overall size of the Git graph [14].

Another command of Git is the sparse checkout option. The sparse checkout option allows the
checkout of a subsection of the repository [6, 14, 36]. For example, PAClab is only concerned with
checking out source files. By enabling sparse checkouts and adding the ‘*/.*.java’ specifica-
tion to the .git/info directory, when the operation git pull is run on an empty working directory, only Java files will be checked out. This results in only duplicating data that will be used by PAClab.

Similarly, as of Git 2.25, initial support for sparse cloning/partial checkouts is starting to be included \[6, 14, 36\]. This is done through the –filter flag. Options for the –filter flag include blob:none, blob:limit, sparse:oid, and tree:0 as well as a combine option to mix filters. Unfortunately, there is no current way to specify specific files that a Git user would like to clone \[6, 14, 36\]. Also unfortunate is that testing shows GitHub does not recognize the filter option \[8, 30\].

2.3 Zettabyte File System (ZFS)

ZFS stands for the Zettabyte file system \[4\]. ZFS includes several features for minimizing the on-disk storage space of files. These features include snapshots and cloning, deduplication, and compression. We decided to use ZFS over other file system technologies because of these features and its simple interface \[7, 18, 22, 35\].

ZFS works by first creating a pool, then partitioning different file systems under these pools. Repositories can be directly cloned into these file systems and the entire file system can have a snapshot taken of it for backup. All filesystems are stored under the pool which serves as a root directory. The pool and every file system come with a .zfs folder which contains individual snapshots that have been taken \[12, 29\].

2.3.1 Snapshots and Cloning

Both snapshots and clones serve as ways to create near instantaneous backups of a filesystem. A key difference between them is that snapshots are read-only while clones are writable. Both snapshots and clones use copy-on-write, providing very fast initial backups.

The snapshot command provides a copy-on-write backup of an existing file system. Upon creation, the snapshots are stored in an individual filesystem’s .zfs/snapshot folder. Since snapshots are copy-on-write, they originally take up almost no additional space when stored mak-
ing them extremely cost effective [29]. After making a change to the file system, ZFS creates a delta of changes between the revision and the snapshot, and this is what is stored within the snapshot file [29]. Since only the changes are stored, the backup is much smaller in size than the original repository, conserving space. Snapshots are read-only and can only be rolled back to the previous one.

Clones can only be created from snapshots. Clones are also copy-on-write making them near instantaneous in creation. Instead of being placed under the original file system, when a snapshot is cloned it creates a new file system that is considered the child of the original [29]. When a file system has been cloned, the original file system cannot be deleted. In order to delete the original file system, the clone will either have to be destroyed or promoted [29]. After a promotion, a file system switches places with its parent and the original file system becomes the child instead. Clones, unlike snapshots, have write permissions and can be edited while preserving the original file system [29].

2.3.2 Deduplication

Deduplication is a process in which identical blocks are not copied and instead are flagged as being in multiple locations [42]. Deduplication reduces overall disk consumption but comes at the cost of extra processing and RAM [42, 43]. Since PAClab emphasizes the need to preserve storage space, the additional cost in processing power has been initially determined as acceptable.

Deduplication works by scanning each block of written data to identify if it is identical to any already stored blocks. Each block is hashed into a unique identifier that is used to compare new blocks with old blocks. If the new block has an identical hash when computed, it is assumed the incoming block is identical. If the two blocks are identical, instead of creating a new instance of the repeated block, the system will link to the original block from the destination of the new block.

In ZFS, the dedup flag can be used to turn deduplication on or off [29, 43]. The scope of deduplication is on the zpool level and cannot be observed on an individual filesystem residing within the zpool. Deduplication is not recorded with regular ZFS list tools and is not reflected with other tools such as du or df. In order to evaluate deduplication, the zdb tool can be run.
on the entire zpool to determine storage savings as well as the overall deduplication ratio. The 
zdb tool returns a histogram which separates storage into two groups, allocated and referenced. 
The referenced group displays the original block sizes for data while the allocated shows what has 
actually been stored on disk. When deduplication is working correctly, results from the allocated 
group will display less than the results from the referenced group. The refcount columns display 
the number of blocks that share the same space. Using the results of zdb it is possible to see which 
blocks saved the most amount of space. The tool also calculates the overall deduplication ratio to 
help determine effectiveness.

Combining the usage of deduplication with snapshots results in a system that allows incoming 
changes to be stored without adding additional storage space. Identical blocks are referenced, 
not copied, and changes are stored without copying information that is already provided. This is 
critical functionality for providing a system that can access all old versions of a constantly evolving 
Git graph without duplicating existing data or losing track of older versions.

2.3.3 Compression

The final feature of interest in ZFS is compression on write. ZFS uses a combination of regular 
compression and delta compression. ZFS uses regular compression as a default when writing new 
files to a file system, and uses delta compression when creating snapshot files [29].

Default compression can be enabled on a ZFS filesystem with zfs set 
compression={algorithm} {poolname}/{filesystem}. ZFS supports the lz4, lzjb, 
gzip, and zle compression algorithms [29]. Since lz4 is considered the most optimal algorithm at 
this time, we decided to use this compression algorithm over the others. When default compression 
is enabled on a file system, any file that is written into the file system will be compressed using the 
chosen algorithm, reducing size on disk for stored projects.

When creating a snapshot, ZFS creates a delta of changes. A delta is a calculation of all changes 
between an original file and a revision. Instead of storing an entire secondary copy of a file, storing 
a revision is usually expected to be much smaller in size reducing disk space [19] [26] [39]. Changes 
are calculated when data is written.
CHAPTER 3  PROPOSED SOLUTION

Our primary research mission was to find an efficient way to store multiple copies of a Git repository. PAClab contains hundreds of thousands of Git repositories. Not only that, but it is expected that over time these graphs will evolve. We want to make sure original graphs are preserved. While it is considered bad practice normally to manipulate the history of a Git project, the reality is that it is done. Literature even suggests this happens a surprising amount of the time [13]. We need to guarantee the original graphs are available to users while also preventing duplication of identical nodes for space reasons.

The repository data of PAClab is stored across 15 nodes in a cluster. They are stored using their original Git repository, as maintaining access to Git functionality is important. We need to be able to use commands such as Git pull to be able to update the data after a certain amount of time has transpired.

PAClab wants to guarantee that the benchmark selection generated today will be the same three years from now. Due to the fact software evolves very quickly however, we also need the ability to update projects after a certain period of time in order to provide relevant data. Due to these philosophies, we cannot store just one copy of a Git graph.

A simple solution that might originally come to mind is to simply store Git hashes. As mentioned above, Git history is subject to manipulation, and this occurs quite frequently [13]. Due to this fact, it is necessary to store all Git graphs locally within PAClab’s cluster. Due to how many graphs will be stored, it is also necessary to find a way to store the graphs efficiently.

In order to create a solution to our problem, we have combined two general tactics: minimize the size of the Git graph and storing changes without duplicating nodes. We focus on two underlying technologies, Git and ZFS to accomplish this feat. By using ZFS, we can utilize default compression, deduplication, and snapshots to track multiple versions of Git projects, and by using under the hood utility features and knowledge of Git, we reduce the overall size of individual Git graphs.
3.1 ZFS

The first technology we tested was ZFS. With ZFS, we were looking to confirm that the features we wanted were present within the technology, while simultaneously looking for potential issues that could prevent the usage of ZFS as a part of our solution. We looked at three filesystem features - compression, deduplication, and snapshots. At first, we were unsure of whether we wanted to use snapshots or clones so we decided to test both commands. We tested each feature using the Boa compiler repository from GitHub [11].

3.1.1 Compression

We installed ZFS on a server for testing purposes and created a pool. This was the test machine used to experiment with ZFS’s operation. For our test dataset we used the Boa compiler [11]. The Boa compiler was chosen because of its relatively large size (457MB) and mixture of Java source files with binary data.

When testing ZFS, we were mainly concerned with testing for three different properties: compression, deduplication, and snapshots/clones. We started by creating a filesystem dubbed 'compiler'. We disabled compression on this filesystem, then cloned the full Boa compiler project into the filesystem. We observed the size as 457MB with a total usage of 5% of our zpool’s total capacity. N.B. that the 457MB includes the full history of the Boa compiler alongside the Git directory which results in additional space to the actual size of a project. After observing the file size, we deleted the filesystem. We then recreated the filesystem, this time leaving compression on. We once again cloned the full compiler into the filesystem. This time we recorded 368MB as the total size and usage remained at 5%.

On our test dataset, compression made a difference of over 100MB, which was approximately a 20% decrease in space consumption. Git already uses delta compression to minimize storage space, so it makes sense the files were not compressed down even further. It is likely the greatest impacts of compression would be on the files that reside in the working tree directory. While we plan to store our files in a bare repository format, when forming a download package we will have
to create a working tree to do so, which is what will most likely be positively impacted by having compression turned on by default.

3.1.2 Deduplication

As mentioned in the background section, deduplication is a method used to save space in which copied blocks are only referenced, not completely duplicated. With deduplication, we expect that when copying files from one location to another, to not see a double in storage space. Without deduplication, we expect to see a double in storage space. Our process for this was to first create a filesystem and turn off compression to prevent possible interference. We enabled deduplication, then cloned down the Boa compiler repository to use as our test dataset. We then used the `zdb -DD` command to observe disk size. At the time this test was performed, the Boa compiler was listed at 543MB. We then duplicate the project via simple copy and paste and observe the results. The new listed size with `zdb -DD` is 542MB and the deduplication ratio is listed at 2.00x. Without deduplication this would have been 1.06G.

There is a 50% reduction with and without deduplication. Combined with compression this could approximate to a 82% reduction in size. Of course, results will vary between different projects holding different amounts of data. The Boa project was chosen because it represents a similar composition of data we would like to store for PAClab.

3.1.3 Snapshots and Cloning

Our goal for testing snapshots and clones on ZFS was to simply observe the mechanics of ZFS and determine how to use each tool as well as confirm limitations of each functionality. For the snapshot command, we took a snapshot of an existing filesystem. We then edited that filesystem and took another snapshot. We tested the rollback command afterwards. We confirmed the rollback command only works for the latest snapshot, however you can still read the contents of a snapshot via the `cat` command, which works for retrieving source code, the data type targeted by PAClab. We found the snapshot folder inside the filesystem and listed out its contents. We observed the most recent snapshot taken took up 0MB of space. We observed the first snapshot however took
Both snapshots and clones are confirmed as copy-on-write as indicated by the fact they start with 0MB consumption and only increase with edits. The edits themselves are smaller than the total filesystem confirming the usage of delta differencing. We determined you can only rollback to the latest snapshot, however this was determined not to be problematic because we could still read source code from snapshots via cat. Clones create a completely separate filesystem from the original. When a filesystem is cloned, you cannot delete its parent filesystem. Ultimately, we determined that snapshots were the tool we would like to use with our project instead of clones. The usage of both copy-on-write and delta differencing means we can create low cost backups of cloned projects and preserve a project’s history guaranteeing results remain unchanged even if the history of the repository ends up being altered.

3.2 Git

The next step in our approach was to try and shrink the overall space consumption of a Git graph. We tried a variety of combinations and different approaches including the testing of sparse-checkout and sparse-clones as well as to try and remove blob files from the git repository. Our primary target was to shrink Git’s internal object tree by pruning branches with blobs that PAClab does not need access to.

3.2.1 Sparse Checkouts and Sparse Clones

Sparse-checkout is a feature in Git that allows a Git user to only copy over a certain selection of files from the Git repository into their working tree. This can be used to ignore checking out large files you do not need to work with and improve storage capacity when working with both the Git repository and working tree. Unfortunately, the sparse checkout feature still requires a complete git repository, meaning it has no bearing on Git fetch and still results in downloading large files and ultimately has no effect on the Git graph. Despite this, sparse checkout still remains useful for use in PAClab in order to shrink the size of a project’s working directory. The exploration of the sparse checkout feature led to the testing of Git’s still-in-development sparse clone feature. The sparse
clone feature allows filtering during cloning, preventing the downloading of larger blob files.

To test the functionality of the sparse checkout feature, we initially cloned a repository with the –no-checkout flag enabled. In order to use sparse checkouts, we had to first enable them by editing Git’s config files for the project. After that, we created a configuration file defining what we would like to checkout. We used the path "**/*.java" to specify we wanted to checkout all Java files under any directory. We then pulled and observed the results. All changes occurred in the working directory, with only java files existing in the tree. Unfortunately, any ignored file still technically exists under the Git repository. However there is still potential to use sparse-checkouts when creating download folders.

Next, we attempted to perform a sparse clone in order to eliminate the fetching issue. We started locally by creating a fork of the Boa compiler and then cloning the full repository locally. We modified the local clone’s settings to enable –allowfilter and –allowanysha1inwant. We then cloned this local copy using the filter flag –filter=combine:blob:none+tree:0. The filter feature only works in Git 2.26+. Combine tells Git to combine two different filters. Blob:none tells git to ignore fetching blobs and tree:0 omits all trees and blobs whose depth exceeds 0. After cloning, the result was a folder of 87KB. This is a drastic reduction from 457MB but includes reductions from a shallow clone as well as a filter from blobs. Next, we wanted to see if GitHub recognizes the developing filter flag for Git. Using the fork, we attempted to clone the Boa compiler. Unfortunately, GitHub returned a failure mentioning it did not recognize the filter command. Since PAClab relies on GitHub for collecting projects, this means using sparse clones is not practical at this time.

Next, we looked into using Git’s still-in-development feature sparse-cloning. Sparse-clone does not have a user friendly way to specify blobs to omit and is not supported by GitHub, which is the primary data source for PAClab. This means that while the results were impressive from running sparse-checkout locally during initial testing of the feature, it is unusable with PAClab in its current state. This is ignoring the fact that allowing sparse cloning requires project specific configuration. We would also like to emphasize that since GitHub did not support the feature, we did not push further testings which should be done to isolate and confirm the results in terms of
Table 3.1 Results from testing deleting blobs. A S denotes test was successful, F denotes test failed.

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updating deleted file</td>
<td>S</td>
</tr>
<tr>
<td>Reverting a change</td>
<td>S</td>
</tr>
<tr>
<td>Updating an existing change</td>
<td>S</td>
</tr>
<tr>
<td>Revert an existing file</td>
<td>S</td>
</tr>
<tr>
<td>Updating a deleted file and an existing file</td>
<td>S</td>
</tr>
<tr>
<td>Update a deleted file, revert update, update existing file</td>
<td>F</td>
</tr>
<tr>
<td>Update an existing file, revert update, update deleted file</td>
<td>S</td>
</tr>
</tbody>
</table>

total storage capacity improvements.

3.2.2 Blobs

To review, Git has four different types of objects. These are blobs, commits, flags, and trees. Blobs contain the actual contents for files in a Git repository and as such, have the most potential for savings in storage costs. With Git, one of our focuses was to eliminate the storage of binary trees and blobs that are not representative of source code. PAClab only focuses on source code, so we knew we could remove them safely in order to save space. As a practical example of the impact of removing all non-source code blobs from a repository, the Boa compiler shrinks by over 200MB when these types of blobs are removed from the Git repository. This is a savings of at least 40%. The potential for space savings by removing binaries from projects varies greatly on a project per project basis but was worth investigating. Our first approach to this problem was to attempt to delete the blobs directly.

We started by creating shell scripts to set up our test environment. The script creates a full clone of a test project, of which we used Boa. We dubbed the clone origin, as it was intended to simulate the remote. The script then creates a shallow clone of the origin locally eliminating prior history. The script then unpacks all of the Git objects, identifies the blob containing the readme and removes it. A readme file is an example of a non-source code file that would be removed. This setup serves as the starting point for our series of tests.

There were 8 tests in total. The results can be seen in Table 3.1. The tests consisted of variations between updating existing files and reverting changes. The tests did not include what would happen
if history had been altered via force pushes or rebases. The tests were performed by making alterations to the files in the origin repository, committing the changes, then pulling inside the shallow clone. Most tests were successful, however when reverting a change on a removed file (in this case the readme), this would result in corrupting the repository. No further changes could be pulled from our origin at this point.

Due to the fact that Git still has references to removed blobs, this results in the chance that the repository may become corrupted by simply removing the blob file. This meant that our first approach was too unstable and simple to use for PAClab. In order to remove a blob safely, we would need a clean way to remove references to deleted blobs which would entail the alteration of history. This led to the testing of Git’s filter-branch commands.

3.2.3 Filter-Branch

Git’s filter-branch command is Git’s in-built feature for altering history. The filter-branch can be used to completely strip references to a file over the entire course of a repository’s history. By removing the references to a blob, we can then run Git’s garbage collection command to remove the blobs without risking corruption of the repository. Git’s documentation advises against the usage of the filter-branch command because it is very easy to corrupt a repository while using it [14]. The filter-branch command also runs slow and will not scale well. There are alternatives to filter-branch in the forms of filter-repo and BFG, but we started testing by using Git’s in-built tools before moving on to alternatives.

We used a methodology that was practically the same as before with a few alterations. The script was updated to run filter-branch instead of unpacking and directly removing blobs. A call to Git’s garbage collection was added with aggressive and prune options. We used the same tests and executed them in the same manner - by editing the mock origin repository and fetching in the copy.

This time, all tests succeeded. Results are shown in 3.2. Alteration of history results in divergent histories, however since we are not concerned about maintaining history we can pull in the remote by adding –allow-unrelated-histories when pulling. This force updates the repository. We
Table 3.2 Results for using filter-branch. A S denotes success and a F denotes failure.

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test updating deleted file</td>
<td>S</td>
</tr>
<tr>
<td>Test updating existing file</td>
<td>S</td>
</tr>
<tr>
<td>Test updating and reverting a deleted file</td>
<td>S</td>
</tr>
<tr>
<td>Test updating and reverting an existing file</td>
<td>S</td>
</tr>
<tr>
<td>Test updating and reverting a deleted file then updating an existing file</td>
<td>S</td>
</tr>
<tr>
<td>Test updating and reverting an existing file then updating a deleted file</td>
<td>S</td>
</tr>
<tr>
<td>Test updating and reverting a deleted file together</td>
<td>S</td>
</tr>
<tr>
<td>Test updating and reverting an existing file together</td>
<td>S</td>
</tr>
<tr>
<td>Test updating and reverting a deleted file together then updating and reverting an existing file together</td>
<td>S</td>
</tr>
<tr>
<td>Test updating and reverting all</td>
<td>S</td>
</tr>
</tbody>
</table>

can then rerun the commands to clean the updated copy. The biggest drawback to this approach is the repository must be cleaned after every update. Filter-branch is also slow to the point of impracticality. After determining this approach was viable, we looked into the alternatives.

Filter-repo is the script recommended by the official Git documentation. It is a python script that is more user friendly, reduces risk, and is at speeds that are usable [27]. BFG is another popular alternative, however requires the installation of Java [40]. Our servers will already have python installed on them, making the usage of filter-repo the simplest. The only drawback is that filter-repo needs to be installed to path on each machine in our cluster. By developing a script that utilizes filter-repo, we gain the ability to strip out binary files from a repository without great risk of repository corruption. Depending on the project, this could result in massive improvements in storage space consumption.

3.2.4 Shallow Clones, Pulls, and Fetches

All of the data on PAClab was pulled using the –depth=1 flag. This is what is known as a shallow clone. The depth flag specifies how much history to clone, and with depth set to 1 we only pull down the latest commit and disregard previous history. In PAClab’s situation, we do not need a complete history of projects we are storing and the benefit of using a shallow clone is that it saves
Table 3.3 Testing filter-repo against force pushes. A S denotes success and a F denotes a failure.

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altering history</td>
<td>S</td>
</tr>
<tr>
<td>Resetting history</td>
<td>S</td>
</tr>
<tr>
<td>Removing commits via cherry-picking</td>
<td>S</td>
</tr>
</tbody>
</table>

space as we are essentially pruning off parts of the internal Git graph by removing commit objects. Similarly, git also has a depth option for fetch and pull commands.

Our script works by altering Git’s history to remove references to blobs we want to prune from Git’s internal graph. We hypothesized that performing shallow fetches and pulls would boost the performance of our scripts because less commits means a smaller graph and a smaller tree traversal that would have to be performed looking for references to unwanted blobs. In order to test shallow fetches and pulls, we created a repository locally and then proceeded to create a shallow clone of this repo. We then updated the original repository to include several commits altering one file. We performed a normal pull operation in the clone, then ran Git filter-branch to remove the altered file. There were a total of three commits that altered the removed file, and it took about 2 seconds for filter-branch to scrub the file from the clone’s history. We then repeated this procedure, this time adding the –depth=1 flag to our pull operation. After running filter-branch, only one commit had to be altered, and the procedure took less than 1 second.

For this experiment, we determined that the inclusion of shallow fetches and pulls into our solution would most likely result in positive performance gains. The exact numbers will vary depending on how often deleted blobs are updated in commits. A problem that was encountered during testing was divergent histories. Git includes an option during pull operations to allow unrelated histories, however a problem that stems from this is the possibility of merge conflicts. Since we are only intending to obtain data, not alter and push, we can apply a policy to accept all incoming changes in order to bypass this issue.

We wanted to make sure shallow fetches and clones were resilient towards anything that could go wrong in the remote repository, ensuring our clones will not go into a corrupted state, so we performed additional testing using our local origin repository. We performed four more additional
tests to ensure shallow pulls had no complications when shrinking the graph. The tests included altering history, resetting history, removing commits, and combining tests. History alteration was done by using filter-repo itself to remove several files from the history of our test repository. We reset history using the Git reset command and chose an earlier commit hash in git log. We used cherry-picking to completely remove commits from the history of the master branch, and we did all operations together for our final test. Tests resulted in divergent histories and merge conflicts. However, as previously stated we can accept all incoming changes and allow unrelated histories, so none of these tests resulted in our repositories entering a corrupted state. Our results are shown in 3.3.

Figure 3.1 Shallow clone of the repository shown in Figure 2.5

Both the process of removing history from a Git repository as well as filtering out blobs results in changes to the internal Git graph. Removing commits from the history of a Git graph results in removing not only commit objects but also unused trees, leaving only the most recent commit object and only trees that reference currently existing files in the most recent copy. Taking our initial representation of a Git graph, Figure 3.1 shows what the graph looks like after removing
previous commits. Filtering out blobs also results in graph alteration. Filtered blobs are removed from the graph, as well as the edges that originally pointed to them. Trees that originally pointed to the blobs are left intact, even if they do not point to any other object. Figure 3.2 shows the altered graph. By combining both graphs together, the intersection that remains becomes the final representation of our altered Git graph, which can be seen in Figure 3.3. The original graph had a total of 9 nodes and 11 edges. After removing old commits and trees and filtering out blobs, the resulting graph contains 4 nodes and 3 edges.

3.2.5 Multi-layered Git Graphs

Bringing everything together, our ideal solution is to store multiple Git graph’s together in what could be seen as a larger overall graph. We have defined this larger graph as a Multi-layered Git Graph (MGG). A Multi-layered Git Graph can be considered a join between Git graphs from the graph’s initial version downloaded up to its current version. A MGG consists of nodes and edges, with nodes representing tags, commits, trees, and blobs and edges representing the references...
between each object. Every branch in a MGG represents a singular Git graph. MGG’s have three primary purposes: to provide easy access to all stored Git graphs for any cached project, to provide resilience to force pushes, and to minimize duplication of data.
Figure 3.4 Git graph evolution over time

(a) Git commands showing several changes and a force push

1. `echo "..." > file.java && git add . && git commit && git push`
2. `# T1 - Snapshot 1 happens here`
3. `echo "..." > README.md && git add . && git commit && git push`
4. `# T2 - Snapshot 2 happens here`
5. `mkdir src && git mv file.java src/ && git add . && git commit && git push`
6. `# T3 - Snapshot 3 happens here`
7. `git update-ref -d HEAD`
8. `git add . && git commit && git push --force`
9. `# T4 - Force Push history alteration`
10. `git rm README.md && git add . && git commit && git push`
11. `# T5 - Snapshot 4 happens here`

(b) Resulting graph at four points in time
Figure 3.4b shows what a MGG might look like over time. The figure shows four snapshots taken for four different commits. In the first commit, a single file is added to the repository. In the second commit, another file is added to root. In the third commit, the first file is moved into a subdirectory. In the final commit the second file added is removed. Boxes with dotted lines represent nodes already present in the MGG. Trees contain hashes and filenames to blobs and other trees, so when dd0acc is moved into a subdirectory src/ the hash for the tree still remains the same. In between the third and fourth snapshots, a force push removes the current history. The MGG still preserves this history in snapshot 3. In this case, the MGG is immune to history alterations which preserves the integrity of previously generated datasets. Figure 3.4a shows the git commands used to create the figure displayed in Figure 3.4b.

3.3 Proposed Solution

Our proposed solution consists of a storage architecture that serves as PAClab’s cache of Git projects, and a set up scripts that automatically implements the proposed changes to a Git project. We have created a formal definition of these changes dubbed a Multi-layered Git Graph. By using a combination of Git commands and ZFS we create a solution to implement a MGG.

3.3.1 Custom Scripts

In order to implement our changes that would alter Git repositories, we created several scripts that would help filter out Git repositories during the creation, updating, and maintaining of each directory. The creation script is in charge of creating shallow clones, enabling sparse checkouts, and filtering out unwanted files. It also creates a snapshot after cloning a new repository. The update script is in charge of performing a shallow pull in order to take in new changes to a repository. The update script is in charge of resolving conflicts with merging the .git repository into the working tree as well as allowing unrelated histories to be accepted. The failsafe script is a variation of the creation script that is in charge of recovering a Git repository if it should become corrupted. Instead of cloning, the failsafe script recovers from the existing .git folder.

The create script, as show in Figure 3.5, is in charge of several setup processes for our solution.
Figure 3.5 The create script, employing several key storage saving strategies

```bash
# Enable sparse checkout
git init
git config --local core.sparseCheckout true
echo "**/*.java" > .git/info/sparse-checkout

# Get objects
git remote add origin "$url"
git fetch

# Filter out unwanted objects
remote=$(git remote) && url=$(git remote get-url --push $remote)
git filter-repo --path-regex '^.*/*.java$'
git remote add origin "$url"

# Unpack objects
cd .git
mv objects/pack pack
pack=$(ls pack | grep '.pack')
git unpack-objects < pack/"$pack"
rm -rf pack
cd ../

# Checkout working tree
git checkout master
```
Figure 3.6 Script for updating a repository

```bash
# Update repo, accept all incoming changes, allow divergent histories, and create pull
git pull -X theirs --allow-unrelated-histories

# Filter out unwanted files
remote=$(git remote) && url=$(git remote get-url --push $remote)
git filter-repo --path-regex '.*/*.java$'
git remote add origin "$url"

# Unpack objects
cd .git
mv objects/pack pack
pack=$(ls pack | grep '.pack')
git unpack-objects < pack/$pack

cd ../
```

Figure 3.7 The failsafe script. It resets the directory then calls the create script.

```bash
remote=$(git remote) && url=$(git remote get-url --push $remote)
rm -rf *
rm -rf .git
source ../create.sh "$url"
```

The first step before anything else happens is to enable sparse checkouts inside the repository. After this, objects are fetched into the Git repository without setting up a working tree. We first filter out all unwanted objects before we create the working tree by checking out master.

The update script can be seen in Figure 3.6. It performs a pull and assumes in the case of merge conflicts to simply accept all incoming changes. We also allow unrelated histories in case of divergent histories. Filter-repo is used to filter out unwanted files. The lines above and below the filter-repo call work to conserve the remote which is stripped from the repository’s references during cleaning. Snapshots are handled by PAClab’s backend, and snapshots are done only when a project selection using a given Git URL is created. This way we are only caching used parts of the Git graph.

The failsafe script in Figure 3.7 resets the remote from the .git project, removes the contents of the folder, then recalls the create script using the grabbed remote URL. We attempted to make our solution as resilient as possible using filter-repo, so the hope is that the failsafe script never has
Figure 3.8 Representation of Transformed Project Table in PAClab’s backend. Host and path used for accessing stored projects.

<table>
<thead>
<tr>
<th>TransformedProject</th>
</tr>
</thead>
<tbody>
<tr>
<td>id AutoField</td>
</tr>
<tr>
<td>parent ForeignKey</td>
</tr>
<tr>
<td>project ForeignKey</td>
</tr>
<tr>
<td>datetime processed DateTimeField</td>
</tr>
<tr>
<td>host CharField</td>
</tr>
<tr>
<td>path CharField</td>
</tr>
</tbody>
</table>

to be called. However, we wanted to have such a script in place in order to save a repository if it should become corrupted during a bad fetch or merge process.

3.3.2 Data Storage Structure

PAClab uses a relational database to store information in regards to projects and their current processing status along with metadata. Figure 3.8 shows a simple diagram of a transformed project table. The TransformedProject table and Project table include information in regards to where stored Git projects exist on PAClab’s cluster. They contain two character fields, a host and path, that point to the machine where each project is stored along with the associated path. When creating the storage structure for holding each project, it was important to separate out each project into a certain file hierarchy so that we could easily access each file programmatically and keep track of each project’s processing status. Necessary divisions include the separation of a project into its raw Git repository, in which we run our filtering scripts and create snapshots, and a transformed directory in which we store the output from each individual transformation. In the transform directory, we also must take into account that each transform can be run with different parameters, and have created a subdirectory for each set of parameters.

The path for a stored transformed GitHub project ends up looking like:

```
data/transformed/transform.pk/transform_option.pk/data_source/us/ername/project
```
A filled in example might look like:

/data/transformed/32/3/github.com/re/bbrunner/paclab-www.git

Figure 3.9 shows a visualization for this folder hierarchy.

PAClab’s cluster contains 15 nodes each containing 2x1TB hard drives. The first drive is reserved for the OS while the second drive has ZFS installed on it and is used for storing PAClab’s repository data. Snapshots are only reserved for raw Git repositories. We have an aspect of lineage in our system, denoted by the parent field found in entries in the TransformedProject table. We keep track of how each TransformedProject is calculated, and instead of snapshotting transformed projects, we can instead use the base raw Git repository along with the stored history in our relational database to recalculate the new transformed project. Snapshots are taken as part of the cloner subprocess of PAClab, which is in charge of cloning projects from GitHub that match an end user’s defined set of search parameters when submitting queries to PAClab [10].
CHAPTER 4 EVALUATION

We have proposed shallow cloning and filtering unwanted objects as techniques for reducing the overall footprint of a Git repository. Both shallow cloning and filtering objects shrink the size of a Git graph and we have asserted usage of these techniques results in shrinking the storage space requirements for a Git project. By using such techniques we can store more Git projects in less space, allowing for storage space to update Git projects in PAClab’s cache without having to timebox existing data. In order to evaluate our assertion that these techniques shrink repositories, as well as to determine the overall effectiveness of each technique, the first part of our evaluation involves running each technique on a series of Git projects and measuring the size of each project afterwards.

We have also proposed that de-duplication is important to efficiently store multiple versions of a Git repository. Usage of de-duplication, in our case through utilization of the filesystem technology ZFS, allows for multiple versions of a Git repository to be stored without data duplication. We propose that as the number of commits grows, the savings from de-duplication should go up significantly. In order to test the storage savings from using de-duplication, for the second part of our evaluation we run a naive approach without de-duplication against our approach over a week. We store repositories unpacked in order to take full advantage of deduplication. We wanted to ensure that the cost in storage savings from unpacking Git projects is eventually passed by the savings from deduplication, so we examine the median ratio of unpacked projects versus packed projects as well as performing a sensitivity analysis.

4.1 Obtaining the Datasets

Using Boa, we searched for GitHub URLs to use as a testing dataset for evaluating our techniques. We started with an initial size of 5000 URLs for Java projects and started creating our dataset by filtering out projects that were removed and gave 404 errors. During testing, we discovered one URL that failed to clone and one URL that failed to shallow clone. We were unable to
unpack an additional two URLs, leaving our final dataset at 2,955 projects. We use several sets of scripts to clone, filter, and snapshot projects for each part of the evaluation.

4.2 Evaluating Storage Savings

The first set of scripts collected storage savings for the components in our solution. We first recorded the URL of the project and the base project size. We then performed a shallow clone of each URL and recorded the URL and size. Then we filtered out non-Java files and also recorded the URL and storage size. We then combined both shallow clones and filtering. Finally, we recorded the compression ratio for each project after moving each project to a ZFS filesystem with compression enabled. The sizes for the control dataset, shallow dataset, filtered dataset, and shallow filtered datasets were recorded using the du command on Linux while the compression ratios were recorded on ZFS.

4.2.1 General Statistics

For our dataset, the unaltered control group had an average size of approximately 22.55MB. The median for the control group was 8.54MB and the standard deviation was around 112.91MB. The general statistics for each group are in Figure 4.1. The maximum value for the control group was around 3.83GB and the minimum was 96.13KB. For the shallow, filtered, and shallow filtered groups, the values decrease from those in the control group.

The datasets for compression ratios, shown in Figure 4.2, have less variation than the first four
groups. Unlike the first group however, the values do not consistently trend downward between groups. For example, the mean value is higher for the shallow group’s compression ratio than the control group’s compression.

4.2.2 Comparing Distributions

Figure 4.1 shows the distribution for the control group, shallow clone group, and shallow+filtered group. The control group and shallow group have similar distributions visually. The filtered group

<table>
<thead>
<tr>
<th>test</th>
<th>not shallow vs. shallow</th>
<th>unfiltered vs. filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>unfiltered</td>
<td>filtered</td>
</tr>
<tr>
<td>difference</td>
<td>0.54%</td>
<td>3.21%</td>
</tr>
</tbody>
</table>
and shallow+filtered group also have distributions that visually appear similar to each other. There is a visual split between the four distributions separating the four datasets into two overarching groups split on filtered vs. not filtered.

In order to determine the difference in distribution, we calculated the overall visible spread by comparing each group to the control group. We subtracted the lowest Q1 value from the highest Q3 for each set to get the overall visual spread (OVS). We then calculated the difference between medians (DBM) by subtracting the lowest median value from the larger median value in each set. We then took the difference between medians over the overall visible spread and multiplied by 100 in order to create a percentage difference between each distribution. We did this for each pair of groups.

For Figure 4.3, we tested to see if filtering or shallow cloning had a significant impact on the differences of each group’s distribution. For testing whether filtering had an impact on distribution, we compared all groups that had filtering against groups that did not while keeping shallow cloning consistent in each comparison. We followed an identical process with testing filtering, keeping the usage of shallow cloning consistent while changing between filtered and unfiltered groups. All groups that had filtering had differences above 3%. The only group that had a difference below 3% was the group that did not have filtering. Our threshold is 10%, so no variable had a significant impact on overall distribution.

4.2.3 Overall Size Reduction

Both Figure 4.2 and Figure 4.3 give an overview of the overall reduction in storage size between each group. The filtered group and shallow filtered groups display the greatest reduction in size. Taking the median value from the shallow group over the control group produces a ratio of 1.20x while using the same calculation using the median value from the filtered group and shallow group results in 6.39x and 6.89x respectively. The reduction from both filtered groups is roughly 5x greater than the reduction from the shallow clone group and control group. This ratio is even more pronounced with the mean value where the ratio for shallow clones is 1.22x while the ratio for shallow filtered projects is 32.72x. The improvement for the filtered group using the mean value is
approximately 25.74x. Mean values will be skewed because of outliers. Combined with the visual results from the boxplots, shallow cloning appears to result in very little gain when compared to filtering.

4.2.4 Relationship Between Repository Size and Savings

One question is whether or not there was a relationship between the original size of a repository and the savings produced by performing both shallow clones and filtering on a project. We compared both groups against each other to see if there were any trends. Figure 4.4 shows savings from shallow cloning as the project size increases. The savings from using shallow cloning appear to be constant based on the visual results.

Figure 4.5 and Figure 4.6 show the same comparison but for filtering projects. The trend does not appear to be a constant improvement with control, however the trend does not appear to follow a strict regression line. In order to capture better metrics on this relationship, a dataset that has a
better balance between large projects and small projects would be ideal.

4.3 Evaluating our Approach

After evaluating the effect of each significant component of our approach, we wanted to provide an overall evaluation of our system over a small time-frame. For this approach, we ran two approaches over a week using our list of 2,955 URLs. The first approach is considered a “naive” approach. For the naive approach, we assume a simple solution to store all versions of a Git project. This naive method is to clone each project every day as a different version. The naive approach includes filtering.

The second approach uses our methodology to create MGGs for each URL. Using the clones from the naive approach, we copy each daily update into a zfs filesystem and use snapshots to create a MGG.
Table 4.4 Statistics for unpacked filtered projects vs. packed filtered projects. Sizes are in bytes.

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std. dev.</th>
<th>min value</th>
<th>max value</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpacked size</td>
<td>1,965,821.08</td>
<td>14,089,030.02</td>
<td>91,292.00</td>
<td>533,010,198.00</td>
<td>223,003.00</td>
<td>419,052.00</td>
<td>1,059,860.00</td>
</tr>
<tr>
<td>packed size</td>
<td>918,725.38</td>
<td>6,410,595.08</td>
<td>92,418.00</td>
<td>205,817,883.00</td>
<td>137,311.50</td>
<td>187,960.00</td>
<td>353,776.50</td>
</tr>
<tr>
<td>ratio</td>
<td>2.19</td>
<td>0.99</td>
<td>0.99</td>
<td>8.21</td>
<td>1.42</td>
<td>1.81</td>
<td>2.75</td>
</tr>
</tbody>
</table>

4.3.1 Strategies for Storing Changes

Implementing a MGG involves ensuring that every updated Git graph is accessible, that Git graphs already stored in the MGG will not be modified by force pushes, and that identical nodes are not duplicated across different versions. Ensuring nodes are not duplicated across versions can be done using one of two strategies outlined in the approach and background sections. These strategies are to either use deduplication or delta compression, both of which are features provided by ZFS. Our original approach was to use deduplication as our primary means to implement a MGG, however we decided ultimately to switch to delta compression after attempting to utilize deduplication on ZFS as well as considering the costs to ensure deduplication is occurring.
4.3.2 Cost of Unpacking

In the case of deduplication, when two identical blocks are copied into a filesystem, only one copy is stored while subsequent copies share the original data via references. Git is capable of storing objects as either packed into one file or unpacked as separate files. When objects are packed into a single file they are delta compressed by Git taking up less space than when stored unpacked, however, deduplication algorithms will not be able to determine if the objects are duplicated or not since they are all stored in one file. In order to use deduplication to its full advantage, Git objects will need to be unpacked resulting in losing Git’s delta compression. The first step in evaluating this approach to creating a MGG was to ensure the cost of losing Git’s delta compression from packing files will be overcome quickly with new version additions to the MGG.

In order to estimate the number of versions that must be added to the MGG in order to overcome the cost of losing Git’s compression, we took the filtered results from the first part in our evaluation and unpacked them. We then compared the size of the original packed project to the size of the same project unpacked and calculated a ratio between the two sizes. The results for this can be seen in Figure 4.4. With a mean and median hovering at approximately 2x, we determined that the
cost in unpacking should be overcome in most projects in as little as three snapshots and at most 8 snapshots.

While these results were promising for deduplication, there are other implications for unpacking. Unpacking impacts performance negatively, and while we are not primarily concerned with overall performance, this becomes an issue with PAClab’s frontend. Since unpacked files are generally twice as large as their packed counterparts, creating zip files for download on PAClab takes twice as long. In order to provide zip files to users as efficiently as possible, an additional caching system would be necessary to determine which files should be stored packed for downloading and which files should be stored unpacked for longer-term storage. An additional problem with unpacking is that it risks repository corruption. As we were in the process of implementing our storage system into PAClab, we had instances where repositories would corrupt. The exact cause of these corruptions is still undetermined, but we speculate that this might be because the unpacking operation either crashed or ran at the same time as a pull. Pulling does not clean out already existing objects, and when unpacked again, altered history from the result of filtering becomes at risk of entangling with unaltered, unfiltered history.
4.3.3 Deduplication with ZFS

Due to the fact deduplication on ZFS is performed on the zpool level, not for individual filesystems, it is more challenging to evaluate. To test deduplication with the second part of our evaluation, we ensured deduplication was enabled on our zpool then took a random project from our naive approach with 8 snapshots. We copied the naive approach into a ZFS filesystem, unpacked the .git folder, and created a snapshot. We repeated this process with each one of the naive snapshots. When we listed out the sizes of all snapshots they were listed to consume the same amount of space. We then took a python script called zfs-find-deduped found on GitHub to print out all files that were deduplicated in each snapshot [34]. To our surprise the script returned empty saying that no deduplication had occurred. When we ran zdb to get our deduplication ratio, it returned with a value of only 1.05x. Due to the fact the deduplication ratio of our zpool was displaying very small gains, coupled with the complications from unpacking, we moved on to testing the second way to store identical files without copying information, delta compression.

4.3.4 Delta Compression ZFS

Delta compression and deduplication allow achievement of the same end goal by allowing access to multiple versions of a file without copying identical information. For deduplication on ZFS, we store every snapshot full size and allow deduplication to handle identical files. For delta compression every snapshot is stored as only a compressed file of changes between each instance. While ZFS snapshots use delta compression by default, we found that our approach of copying in the naive approach’s daily updates were treated as writes, and when we checked each snapshot using zfs diff we found snapshots were marking every git file as changed. To fix this, we used rsync to ensure snapshots were only storing changes between each copy. The use of delta compression over deduplication meant we could simplify our approach and we could avoid unpacking Git objects. Since we no longer need to store objects unpacked, Git’s packing also provides additional deltas for each specific Git graph.

We started by first using the zdb tool to observe the disk size already consumed by projects
on PAClab, then proceeded to copy over all the naive projects, storing them as separate snapshots. The size before copying was 166GB. We again recorded disk size with the `zdb` tool and this time the size was 169GB. We subtracted the original size from the size after copying to get the total size of our dataset moved into ZFS with delta differencing between snapshots. We concluded by recording the naive approach’s size by using the `du` command in the root of our folder containing all projects as part of the naive approach. All of these results are in Figure 4.5. The naive size was 19GB in size. The reduction from 19GB to 3GB is approximately 88% in savings.

4.4 Result Summary

From our results, both shallow cloning and filtering lower the overall size of the repository, however, we were surprised to find that shallow cloning does not reduce the size of the repository nearly as much as we would have initially guessed. Filtering, on the other hand, reduces the overall storage size of our selected projects by approximately 96%. This is calculated by summing both the control group’s sizes (67GB) and the filtered group’s sizes (2GB) and dividing the sum from the filtered group over the sum from the control group. Based on the fact that shallow cloning does not reduce storage size nearly as much as we originally expected, we have decided to remove shallow cloning and shallow pulls from our original proposed solution. A benefit to removing these two parts of our solution is we now maintain the full history for source files that remain in the repository, which allows for further exploration and analyses.

We originally thought that since filtering relies on the presence of files that are not Java source files, the effect of filtering would range widely. We suspected that some projects would greatly benefit from filtering while other projects would not benefit as much. This seems to be backed by Figure 4.5. If filtering affected the majority of projects to an equal degree we would have expected a linear or constant trend, however the upwards trend does not appear to be consistent. Further
exploration could be done into the composition of Git projects and the exact percentage of source code to other types of files found within the different projects.

We originally wanted to use deduplication in order to ensure identical files across multiple versions were not resulting in data duplication. However we found deduplication difficult to measure in ZFS. Unpacking projects to utilize deduplication interferes with download times, loses access to Git’s compression provided through packfiles, and leaves repositories open to potential corruption. We ultimately switched to using a delta based approach using ZFS snapshots to store only the differences between versions instead, allowing us to keep objects packed together. We had 8 snapshots as part of our time-based assessment. Taking into consideration the original 67GB of data, without filtering this would have probably been 536GB of data. By filtering our dataset for only Java source files and using snapshots between versions our total size was approximately 3GB for 99% improvement. This all with guaranteeing force pushes will not alter data, and providing PAClab users with an immediate cache of projects.

4.5 Threats to Validity

Our method of determining force pushes may be inaccurate. To the best of our knowledge, we have determined the only way to have dangling objects is through history alterations, and by extension, force pushes. We had four cases that failed to clone or be unpacked. For consistency all four projects were removed from all our testing datasets.

There was one case where a project could not be filtered. The failure was due to an automated commit by cvs2svn that contained no author information or commit change. Git does not typically allow pushing empty commits or pushing commits without author information, so we consider this to be an edge case. The unfiltered project was approximately 900MB.

There have been cases of under-filtering where filtering did not remove all non-Java projects. This was from an implementation error when defining filter criteria.

During our week-long assessment, most projects did not produce any new commits. This means that the naive approach was doubling size for no new updates. However in PAClab, we will be updating based on a set time schedule regardless of whether or not a project has been updated.
As such, the naive approach is still considered by us to be valid to compare with the efficient approach.

Since we decided to not use deduplication, we cannot guarantee identical nodes are not stored across each MGG. However deduplication still occurs in each individual Git graph and similar nodes are now stored via delta compression. If no changes occur between snapshots information is not duplicated.
CHAPTER 5 RELATED WORKS

Related work is broken into the categories of mining software repositories, graph optimizations, very large Git repository support, and filesystems. Mining software repositories consist of technologies that aid with software repository mining as well as ongoing mining projects. Graph optimizations include projects attempting to efficiently manipulate graphs, which includes Git graphs as well as non-Git graphs. Large Git repositories include repositories by Microsoft and Google and techniques used for managing these large storage systems. Finally, filesystems include filesystem technology that includes feature sets similar to those found in ZFS.

5.1 Mining Software Repository Frameworks

Boa is a domain-specific language used for mining software repositories. Boa is one of the backend options provided by PAClab for filtering datasets [10] [11]. The architecture for Boa is similar to PAClab; users also submit entries to PAClab via an online interface that then returns results from our data stores. However, Boa’s primary usage is for analytics while PAClab is for generating datasets and collaboration between researchers [11]. PAClab’s architecture consists of the frontend interface, a transformer that runs on a filtered set of projects, and our own data storage system. Boa’s architecture consists of the frontend, a compiler, and their data system. Boa faces similar challenges with data storage, so their solution to this challenge is of interest [11].

Boa has similar storage problems to PAClab and addresses this storage problem by using a distributed file system via Hadoop while also snapshotting project folders. PAClab’s storage system also takes advantage of snapshots and also has data distributed, but we are not using Hadoop on our system and are attempting to solve fundamentally different problems [11].

World of Code is a large collection of open source software. The collection of data is aimed at researchers studying the open source software development ecosystem. The current implementation of WoC contains 12B Git objects and is updated on a monthly basis [23]. World of Code focuses on providing tools to enable analytics. World of Code stores Git objects in a single database
split into four parts. Each part corresponds to a different type of Git object. Deduplication is used to remove repeated code libraries in order to store the data collection. Unlike PAClab, World of Code tries to maintain history for all Git objects (they don’t filter), which results in greater demand for storage. In total, World of Code’s data collection exceeds 80TB after removing redundancies [23].

GHTorrent is a tool for collecting event data from GitHub. It uses peer-to-peer protocols for data retrieval and downloading, and uses a NoSql database for storing data. GHTorrent boasts datasets that contain hundreds of thousands of events and GHTorrent faces data storage challenges like PAClab [15, 16].

GHTorrent is a mining service capable of listening to GitHub event streams. Users of GHTorrent can either use GHTorrent to listen to current GitHub streams, or they can use one of GHTorrent’s existing data snapshots. GHTorrent does not store entire repositories on its data servers. Instead, GHTorrent stores events that occur, reducing storage space requirements [15].

SourcererDB is a combined repository containing the code for Java projects hosted on SourceForge [21]. SourcererDB performs static analysis on these stored projects to extract information about code, then stores these results in a relational database [21]. SourcererDB shares the objective to allow researchers to share datasets along with PAClab, however does not provide a system for providing code transformations. Instead, SourcererDB attempts to provide an analysis of source code itself [21].

Software Heritage is a project for the preservation and the archiving of source code. Di Cosmo and Zacchiroli focus on how to collect software from multiple sources (GitHub, SourceForge, etc), protecting against cases where code is deleted/destroyed from sources, and the collection of metadata pertaining to the code’s history. Software Heritage uses deduplication to preserve storage space [9]. Software Heritage attempts to capture all details of a software repository including the entire history of a project [9]. We create snapshots of a project at the time it is requested by a user of PAClab, using filtering to prune off unneeded history. This results in a smaller graph to store and quicker traversal times. Software Heritage collects all source code without filtering projects [9]. PAClab provides users the ability to filter projects to get specific results.
propose using the *Software Heritage* archive in order to download Git projects to analyze their structures [9, 31]. The projects are illustrated as a series of Merkle DAGs all connected together in a larger graph structure similar to our MGG. The storage model is based on *Software Heritage’s* storage model. As such each Git graph stored is not filtered to only include source code [31]. One of the research questions posed by Pietri et al. is to determine the shortest path length to all source code nodes [31]. As the work is exploratory, the authors have no estimation of how long this process will take. If they pre-filter their data model to only include source code as in our solution this could speed up this process. Instead, they suggest using graph compression with the WebGraph framework and scaling work to multiple nodes with parallelization to overcome these obstacles [3].

5.2 Graph Optimizations

Graph optimizations include the efficient manipulation of graphs. These graphs may relate to Git or they may relate to other types of graphs found in software engineering. Graph optimizations relevant include the use of deduplication to store multiple graph versions, pruning unwanted objects, and storing an ever expanding graph while maintaining old versions, or techniques for storing graph structures more efficiently in general such as graph compression.

Markovtsev and Long have created a large collection of open source Git projects. This collection includes 182,014 highly rated projects pulled from GitHub [24]. Markovtsev and Long use GHTorrent in order to determine what projects to download [15], focusing on projects with high star counts [24]. The Public Git Archive stores all forks of a repository in a single branch of a bare repository, coined by Markovtsev and Long as a rooted repository [24]. The Public Git Archive uses a snapshot system, however snapshots are timeboxed and eventually removed. Unlike the Public Git Archive, PAClab allows users to define selection criteria when downloading projects [10]. We also do not set lifespans on stored projects, instead focusing on efficiently storing Git projects by minimizing the size of their graphs. The Public Git Archive uses a specialized storage format for storing Git files called Siva while we store projects compressed using ZFS and Git [24].
Le and Pattison use graph optimizations in their *Multiversion Interprocedural Control Flow Graphs (MVICFG)* in order to perform patch verification across multiple releases of a software system [20]. An MVICFG is formed by forming control flow graphs for different versions of code. These control flow graphs are used to update a work-in-progress MVICFG. A node is considered a statement in a program. If a new version of the program contains an identical node already in the MVICFG the old node is kept in the MVICFG instead of adding a second copy. Instead, the edges are updated with the number of the new version. This concept of only keeping one copy of a node, while present in multiple versions of source code, correlates with our storage strategy. Our projects differ in the problems we are trying to solve. The graphs in the work of Le and Pattison are used for patch verification and each node represents a code statement. We are attempting to store multiple versions of a Git repository efficiently. Nodes in our system represent different objects in a Git repository.

Biazzini et al. analyze development patterns by examining the structure of DVCS graphs. They define a data structure called a *Metagraph* that captures information about a Git graph [2]. Nodes in a Metagraph are commits in a DVCS graph [2]. Biazzini et al. classify Git graph commits into three categories: terminal, sequential, and structural. Terminal commits represent the root and branch HEADs of a Git graph [2]. Structural commits represent commits that branch or merge. The authors claim sequential commits have no impact on the overall structure of a Git graph [2]. As such a Metagraph consists of only structural and terminal commits and excludes sequential commits. A Metagraph considers all branches of a Git graph in order to discern the graph’s overall structure and look for common development patterns [2]. Skipping sequential commits results in an optimized version of a Git graph with overall fewer nodes. This is similar to our usage of shallow clones and pulls to combine multiple commits into one, however a key difference is we only focus on pulling changes from a remote repository’s master branch.

WebGraph is a general compression framework for compressing web-based graphs [3]. WebGraph is part of the proposed study by Pietri et al. as a way to efficiently store and analyze data retrieved from the *Software Heritage* dataset [9 31]. The proposed structure for WebGraph uses
URLs for nodes and is based on web pages, not Git [3]. WebGraph exploits common similarities in the navigational structure of URLs in order to compress graphs. Utilization of WebGraph for usage with non-URL based graphs would require custom code as WebGraph is built for analyzing the structures of websites and their links [3]. WebGraph relies on the realization that websites have very similar navigational structures that allows simplification of graphs for compressing them [3]. In the case of Git graphs, similarities between each graph in the project’s MGG can be deduplicated or deltas can be stored to also result in reducing the size of each Git graph based on their similarities.

5.3 Large Git Repositories

Large Git repositories typically address storage problems using a distributed storage system. Clients that wish to interact with the large Git repository only interact with a section of the repository at a time. Operations such as fetch and checkout take long amounts of time to complete, and companies dealing with large Git repositories focus on the development of technologies to speed up these operations.

VFS for Git or GVFS is a virtual filesystem that runs on top of Git, developed by Microsoft for maintaining their massive mono-repo. VFS uses a combination of a virtualization system and architecture built by Microsoft alongside Git commands such as sparse-checkout in order to improve performance when dealing with extremely large repositories [25, 28]. We are also interested in the usage of sparse-checkout for the purposes of reducing overall storage consumption as opposed to performance. VFS primarily focuses on improving the speed of operations such as Git fetch and preventing merge races [28].

Scalar is another custom tool built by Microsoft for managing large scale repositories [37]. Scalar was built to address the needs of developers using GVFS and takes advantage of a custom version of Git modified by Microsoft. Scalar speeds up normal Git operations using methodologies such as utilizing sparse-checkout, reducing object transfer, and asynchronous operations [37]. Scalar utilizes sparse-checkouts and focuses on improving the performance of Git.

Google uses a trunk based development methodology and stores all of their source code in
a single mono-repo. Unlike the windows repository, Google does not use Git as their version control system. Instead they use their own in-built version control system. Developers wishing to interact with this version control system typically access it through cloud based services. Only modified files are stored on a developer’s local workspace, conserving storage space. Google’s version control system itself spans across multiple data centers across the world in order to contain the massive amount of data. The cloud system used to access the system itself uses snapshots to preserve changes [32]. Similarities to our solution include the usage of distributed servers to store data as well as the usage of snapshots to preserve changes. We are not storing nearly as much code as Google. However the amount we can scale our resources in terms of hardware is limited, unlike Google. This means we have increased pressure to implement strategies on managing the code size itself.

5.4 Filesystems

Next to Git, the other technology in the development of PAClab’s storage system was the utilization of ZFS. ZFS stands for the Zettabyte File System and was originally developed by Sun Microsystems as open software. After ZFS was purchased by Oracle, the last open release of ZFS was forked into what is now known as OpenZFS. We are using OpenZFS as our ZFS implementation, however refer to it as just ZFS. ZFS uses default compression in order to compress files as they are added to a file system to minimize space with access to compression algorithms such as gzip or, in our case, we used lz4. ZFS uses deduplication to minimize consumed blocks when copying files. ZFS also uses snapshots allowing us to preserve our own history of Git repositories, preventing loss of data [29]. ZFS is used as a technology in our solution however ZFS does not cover all of our solution. ZFS does not prune unwanted Git objects, however it does provide means to preserve multiple Git graphs without duplicating nodes.

Btrfs also has compression, deduplication, and snapshot tools [35]. Btrfs uses b-trees as its primary data structure and emphasizes the usage of copy-on-write to create low cost snapshots and clones [35]. Btrfs is a slightly newer file system and is considered less stable then ZFS. Some of its advanced features are considered buggy [5]. Btrfs and ZFS are competing file system
technologies. Ultimately we decided upon ZFS because it is a more mature filesystem with clear
documentation and a simple interface.

WAFL is a proprietary filesystem technology created by NetApp [17, 41]. WAFL utilizes all
three primary features of interest to us. However, since WAFL is proprietary software, it was
off-limit for our consideration. Bcachefs is meant to be a further expansion on BtrFS and ZFS
with the speed of XFS and ext4 [1]. Bcachefs is early in development and not ready for usage
in production [1]. Venti is a network based storage system meant for archival storage [33]. Venti
emphasizes the need to store data indefinitely and places extra emphasis on reducing duplication.
Venti employs write-only policies alongside standard deduplication. Venti also highlights its ability
to store snapshot files indefinitely [33].
CHAPTER 6  FUTURE WORK AND CONCLUSION

In the future, we would like to remove the need for ZFS. We are not using the entirety of
ZFS’ feature-set which results in unnecessary bloat. A potential future solution includes taking all
unique Git object hashes and combining them into one Git repository or directory while continuing
to create different snapshots or versions for PAClab’s project selection process.

While sparse-checkout does not change the .git folder, sparse-cloning would [38]. Sparse-
cloning is still a developmental feature of Git that is not currently supported by GitHub. When
sparse-cloning is fully implemented into Git and GitHub, we could decrease bandwidth utilization
and speed up our process by only downloading the blobs we want to keep (source code) instead of
downloading entire repositories.

Static analysis researchers develop tools and evaluate their effectiveness using benchmark pro-
grams. Static analysis researchers have turned to open source platforms to collect up-to-date
benchmark programs that reflect real-world codebases. These programs however typically need
to be manually transformed in order to work with researcher’s tools.

PAClab is an online system that provides these transformations automatically through user-
defined project selection and automatic code transformations. PAClab stores a cache of Git projects.
Each individual Git project can be visualized as a graph. PAClab needed a way to efficiently store
all past and future versions of these Git graphs. We created an approach that solved this efficiency
problem by reducing the overall size of each Git graph as well as removing node duplication in
each graph across multiple versions in what we are calling a MGG.

To evaluate this solution, we selected 2,955 Java projects from GitHub and ran two tests against
them. The first test evaluated the individual components of our solution including shallow clones
and filtering Git graphs. We found filtering to be an effective way to reduce a Git graph, reducing
the size of the original 2,955 projects by 96%. For our second test we ran our approach against a
naive approach and found our approach utilized around 6x less space than the naive approach.
BIBLIOGRAPHY


