Olli Raula

**Git Blame Scalability**

Master’s thesis
Espoo, April 18, 2018

Supervisor: Senior University Lecturer Vesa Hirvisalo
Advisor: Magnus Sandberg M.Sc. (Tech.)
Version control systems are a critical part of software projects today. Version control systems keep track of changes as well as allow simultaneous developing and browsing the history. Blame is a view in Git, which is the most used version control system, allowing the user to see who has changed which line and when. Blame is also used in literature to learn more from open-source repositories.

The problem in Git Blame is that it is very slow with large files that have undergone many changes. The Blame for 80000 line file with 8000 commits takes over one minute to generate. Git blame is used quite often, so the slow speed makes it unusable.

The main goal of this thesis was to develop a program that reduces blame generation time by using previous blames to generate the requested blame quickly. Particularly, this thesis sought answers for the following research questions: Is it possible to generate blames properly from the intermediate result? How does the amount of caching speed up the process? How much data needs be stored and how long it takes? Which caching level is best for production?

The developed program functioned well. It was found that the blame generation time can be dropped to one-tenth by the caching doubling the used disc-space from the plain Git repository. The disc space usage can be tuned in favor of the Blame speed. With maximum caching, the blame speed with large files was below one percent of the standard Git Blame. The disc space usage and initialization time are not too high and the maximum level of the caching is the best for the production. The limitations of the developed program are needed initialization time and used disk space. Further studies could aim to make incremental blame generation even faster.

Keywords: Git, Blame, Annotate, version control, cache, database, performance

Language: English
Tekijä: Olli Raula
Työn nimi: Git Blamen nopeus
Päiväys: 18.4.2018
Sivumäärä: 9-42
Pääaine: Tietotekniikka
Koodi: SCI3042
Valvoja: Vanhemi yliopistolehtori Vesa Hirvisalo
Ohjaaja: Diplomi-insinööri Magnus Sandberg


Git Blamen ongelma on sen hitaus suurilla ja paljon muokatuilla tiedostoilla. Blamen generointi tiedostosta jossa on 8000 kriiavä 8000 muutosta kestää yli minuutin. Git Blamea käytetään usein, joten hitaus aiheuttaa oikeita ongelmia.

Tämän diplomityön tavoitteena oli kehittää ohjelma joka lyhentää Blamen generoimiseen tarvittavaa aikaa käyttämällä aiemmin tallemnettuja Blameja generoidakseen tarvittavan Blamen nopeasti. Tärkeimmät tutkimuskysymykset ovat: Onko tämä generointi aiemmin tallemnetuista Blameista toimiva tapa? Kuinka paljon välimuistitus nopeuttaa prosessia? Kuinka paljon dataa pitää tallentaa ja paljonko se vie aikaa? Mikä on sopiva välimuistitustaso tuotantokäytössä?

“Sometimes \( \pi = 3.14 \) is (a) infinitely faster than the "correct" answer and (b) the difference between the "correct" and the "wrong" answer is meaningless.”

–Torvalds, Linus
Foreword

First, I would like to give thanks to Polycon Oy, my Advisor Magnus Sandberg and supervisor Vesa Hirvisalo for the excellent opportunity to research and write a thesis of Git Blame. Thanks to you writing this thesis was actually quite fun. I am also grateful to those co-workers who helped me with proofreading.

I must express my very profound gratitude to my wife Helena for providing me with unfailing support and continuous encouragement throughout my years of study and while writing this thesis. This accomplishment would not have been possible without you. I love you!

In addition, I would like to express my gratitude to my family, who have brought me up, helped me get this far and who have been there for me for my support. I also give my loving thanks to all my friends in Ristin kilta. You made my years at the University Great. Thanks for Saara, Samu, Maaria, Samppa, Daniel, Meri, Jyri, Tero and many others. Sorry I could not list you all.

Leaving the most important to the last, finally I want to thank my loving saviour Jesus Christ. You died for my sins and are on my side even if I am not worth it.

Otaniemi, April 18, 2018

Olli Raula
Contents

1 Introduction .................................................. 1
1.1 Research goals ........................................... 2
1.2 Thesis contribution ..................................... 2
1.3 Thesis outline ........................................... 3

2 Background .................................................. 4
2.1 Concepts of version control systems ................. 4
2.1.1 Diff and delta ....................................... 4
2.1.2 Actions and terms in version control system ....... 5
2.1.3 Branches .............................................. 6
2.2 Version control system types ......................... 6
2.2.1 Centralized version control systems .............. 7
2.2.2 Decentralized version control systems .......... 7
2.3 Repository mining ....................................... 8

3 Environment ................................................. 11
3.1 Git ......................................................... 11
3.1.1 Commit and hash .................................... 11
3.1.2 Commit version tree ................................ 11
3.1.3 Tags and branches ................................... 12
3.1.4 Git functions ........................................ 13
3.2 Gerrit ...................................................... 14
3.2.1 Scalability .......................................... 15
3.3 GitLab ..................................................... 15
3.4 Other frameworks ....................................... 15

4 Methods ....................................................... 16
4.1 Caching .................................................... 16
4.2 Diff ......................................................... 17
4.2.1 Mayers diff algorithm ................................ 17
4.3 Blame ....................................................... 18
4.4 Process .................................................... 18
Chapter 1

Introduction

Software projects today are very complicated. There can be hundreds of developers in developing multi-million line software. Software systems are getting bigger because increasing data masses. The number of users is rising rapidly, and these users want more features. Often, the software is also connected to many other systems. Increasing complexity has made version control systems critical in today's software projects. There are different version control systems. Common to all of them is that they store the whole history of the files with the information about change times and the changes themselves. There are two types of version control systems. Centralized version control systems like SVN store the version history in the one place. Decentralized version control systems like Git store the whole history also in developers computers. They can work without central a server though that is very practical to have it in many cases.

The most important reason to use a version control system is that you can collaborate much more efficiently. Changes to the code are distributed to other programmers only when it is needed, and the programmer can do the distribution quickly. In a bigger team, simultaneous programming very often introduces conflicts. A good version control system helps to resolve those problems. The second good point of using a version control system is to keep track of who has changed what, when and why. In many cases, external documentation is not complete, and it is good to know why something was changed. For example, if the team realizes that some change has introduced a bug, it is good to see why that change has been made to prevent just going back and maybe reintroducing the old bug. Without the version control system, the developer cannot find the reason for the change and cannot find the info why those changes are made. [1] Blame is a tool in version control systems which list all lines of the file with information of last editor and change time. Many studies use Blame and other tools when they study how the source code is made and how the programmer's dynamics help to produce a quality source. For example, Foyzur Rahman and Premkumar Devanbu[2] use Blame to detect ownership of the file and use that information to predict errors. However, there is a problem with large files because Blames in Git from them are very slow to generate. Slowness makes
the error prediction algorithms slower. Also in everyday work, it is a big problem if viewing a blame takes too long. This thesis is made for a company which have several files which Git blame takes minutes, so blame generation time needs to be decreased.

1.1 Research goals

The primary goal of this thesis is to reduce the blame generation time. The sub-goals are to keep disk space usage and initialization time in usable numbers. Reasonable blame generation time is necessary because finding the reason for the bug often needs several blame views, and if each of them uses 5 minutes, it makes work very slow.

This thesis uses heavy caching and modifying the stored older blame with partial blame and diff to archive its goals. Diff is the function which shows differences between two versions of the file. Different amounts of caching are used. With this kind of procedure, the latency should be decreased and only concerning the file size but not the revision count. After the thesis, the final plan is to make a web program which serves the blame results to developers. This thesis stays in library code with simple command line interface because it is the primary topic and web interface is not needed for testing that. The main questions are:

- Is it possible to generate blames properly from the intermediate result?
- How does the amount of caching speed up the process?
- How much data needs be stored and how long it takes?
- Which caching level is best for production?

1.2 Thesis contribution

The thesis author created a way to speed up git blame generation by caching intermediate results. The time of the Blame decreased from 60 seconds to below second. The caching levels are also studied to make possible to use less disk space but still archive significant speed improvements.

It is possible to generate blame from previous blame with partial Blame and diff. The increased amount of cached data increased blame speed linearly. With doubling the used disc space, the Blame speed decreased to, and with twenty-fold disk space, the blame speed dropped to below one percent. The data usage was lower than expected so with not enormous sized repositories it would be good to cache as much as possible and the caching should be set to maximum in most of the cases in production.
1.3 Thesis outline

This thesis is divided into two parts. The first half describes background information and how the speed increase is meant to be archived. First Chapter 2 gives background information about different version control systems and how they work. It also tells how scientists use public repositories in the literature. Chapter 3 describes how Git works and what has been done for this problem before. Chapter 4 describes how the goal is planned to be archived and how the needed version control systems functions work. The second half has the description of the implementation and the results. Chapter 5 describes how the result program is implemented. Chapters 7 and 7.2 measures the created system and discuss how it should be understood and used. A summary of this thesis is presented in Chapter 8.
Chapter 2

Background

This chapter describes background concepts in version control systems. First, the building blocks of the version control systems are outlined. The second part discusses the differences between centralized and decentralized version control systems. The last part describes how version control systems are used in the literature to study software development processes.

2.1 Concepts of version control systems

This section lists and represents the main concepts and functions of version control systems.

2.1.1 Diff and delta

A diff is a program which returns differences between two files. It returns a list of changes and tries to keep the list minimal. In mathematical terms, it solves the longest common subsequence problem. In many cases, there is not only one feasible solution if there are many similar lines. Because of this, sometimes the diff claims different lines have been added than you have actually added. [3].

Listing 2.1: Example file

```
line a
line b
line a
```

For example, in listing 2.1, a developer has added line b and the upper line a. The diff algorithm cannot determine which one of the lines was added and which is the old one.
The result of the diff can also be described with the word 'diff'. The diff is an essential building block of version control systems.

2.1.2 Actions and terms in version control system

Version control systems have a lot of different actions. Here I introduce the ones which are the most important for this thesis. Not all appear in all version control systems, but Git at least has them all.

Working copy

A working copy is a copy of the files which are user-editable at a particular time. The user can edit this version but does not have the history or any other information.[4]

Checkout

A checkout occurs when a programmer takes a version of the code from the repository. Most of the times this version is the newest commit, but sometimes the user needs an older one. After the checkout, the checkouted version becomes the working copy.

Commit

A commit means one changeset in the repository. A commit tells who has changed which files, when and how. It also includes a commit message where the programmer can tell why he has made the change. To commit also means creating the commit.

Repository

A version control repository is the data storage where all the information dwells. A repository keeps track of changes and can contain access control. In many cases repository is not readable by a human and needs a client program. The repository can be on a server or local computer.[4]

Revision

A revision is one stage of the file or repository. Revisions are tagged in many cases with an integer, but Git uses hashes for this purpose. Hashes are described in more detail in chapter 3.1.1. [4]
Blame and annotate

Blame, also known as annotate, tells from which revision or commit a particular line originates from. It can be calculated from all repositories.

Merge

A merge is a commit which merges two branches. Merge commit has two or more predecessors, and it may fix conflicts from those branches.

Conflict

A conflict can be caused by many actions in a version control system. For example, if two branches have edited same lines differently, the result is a conflict. In many cases, the programmer has to decide manually if he chooses one of those or something completely different.

2.1.3 Branches

Version control systems have branches. There is a main branch which contains the most recent version of the code. The name of the main branch differs between different version control systems. In some version control systems like Git, the name of main branch is simply decided by the team. In other systems the system itself forces the name.

Most of the version control systems may contain limitless number of branches. The branches other than the main branch can be used for stabilizing release or for developing new features. For example, when a company decides to create a release, they can create a release branch from the main branch and only add new features to main branch and bugfixes to both. By doing this the team introduces less new bugs to the released code and the team can release smaller update versions with less testing. This way bugs coming from new features do not affect the hotfix versions, and they can be adequately tested before releasing the next large update. Branches can also be used for developing new features. Developers can distribute their non-working changes in a way that others do not get affected by them if they do not want to. After the new feature is ready, it can be merged to the branch which is used to store the 'main' version of the software. [4]

2.2 Version control system types

Version control systems can be divided into two different types: centralized and decentralized systems. They have similarities but also big differences.
2.2.1 Centralized version control systems

Centralized version control systems are called centralized because they have only one repository. This repository is usually on the server, but it can also be on a local computer if all users are logged in on that computer. User's local copy consists only of the working copy of the files. History of the code is saved to the server and is browsable in there. The user can also checkout older versions. The figure 2.1. [4] shows the structure of a centralized version control system.

Concurrent Versions System (CVS) and Subversion (SVN) are the most used centralized version control systems. This thesis focuses on the CVS because the git repository of the company (see section 1) has commits migrated from CVS to git. Subversion is a successor for CVS[4].

CVS development has started in the 80s[4] (original source has been taken offline). The CVS has many shortcomings mainly due to the legacy formats from previous version control systems. For example, it does not have atomic commits, which would enable the commit to be stored in the repository only partially. This default could corrupt the code in the repository. For example, if a three-file-commit has a conflict in the last file, two first files get committed, but the last one does not. As a consequence, in many cases the program does not compile correctly and needs manual fixing. Additionally, the CVS does not have an email address in its commits, which differs from the Git in which this is mandatory.[4]

In CVS tagging adds information to all files and branching uses tagging. Because of this, branching is very resource expensive in CVS and is thus not commonly done. In SVN branches are not costly, but they copy all the data, which leads to more disk space usage. In conclusion, the centralized version systems are not automatically ineffective in branching, but most commonly used implementations have different kinds of problems.[4]

2.2.2 Decentralized version control systems

Decentralized version control systems do not need a central repository. Of course, in many cases, it is beneficial. Every user has their own copy of the whole repository including history and all commit messages. This is shown in figure 2.2. The increased disk space usage is not a problem because disk space is nowadays cheap and systems can use heavy packing.[4]

In a decentralized version control system, most of the actions are made locally. Online operations are only used for updating branches to the same state in which the sender branch is. Local actions are in many cases much faster than online versions in centralized version control systems.[4] There are several decentralized version control systems on market, such as Git[5], Mercurial[6], Bazaar[7] and BitKeeper[8]. The decentralized version control system is a considerably newer concept than centralized. The first studies about decentralized version control systems are from 1997[9].
There are several reasons why many projects are moved or are moving to decentralized version control systems. The most practical reason is the possibility to create local branches and commits. That allows saving the current status while doing something else in the meantime or creating partial and unfinished commits for backup purposes. It is also much easier to distribute these experimental changes without committing them to the main branch. Merging is also easier because it happens on the user’s computer and is thus much more controllable. Decentralized version control systems also allow better off-line operations. [10]

In many operations, the most used decentralized version control system is more efficient than SVN or CVS. Because Git stores full files rather than a sequence of diffs, checkout and diff are much faster. The compression makes repositories smaller in Git than SVN even if SVN stores only diffs. [11]

2.3 Repository mining

In the past decade, an active topic has been what can be learned from different repositories. For example, Zimmermann et al. have used mined source history to predict what else should be changed with already changed line[12]. This technique is used to prevent errors caused by a programmer, who has not noticed to do all the required changes and to help the programmer to browse the source code. Git blame can be used as a poor man’s tool for this. The programmer can use blame to check for other lines that need to be changed when adding code.
Another way to process the history and changes is to classify the changes by the reason for the change. E. Burton Swanson[13] divides the maintenance related modifications into three categories. The first category consists of corrective changes, which fix the failure of previous revisions. Failure can be related to processing, performance or implementation. The second category is adaptive changes, which adapt to external changes; for example, new operating system or export format in other programs. The third category contains the perfective changes, which improve performance or maintainability without fixing a critical bug. In addition, commits from initial implementation can be classified as the fourth category.

Todd L. Graves et al. uses repository mining to predict the number of errors in files[14]. Their best estimation called weighted time damp model calculates all changes and their sizes and weights down older changes. The older changes are most probably already fixed if there have been changes. They parsed the whole history with a special tool. However, Blame can also be used to observe the age of each line of the file.

Many of the studies parse history of their own, but some of them use blame directly. For example, Foyzur Rahman and Premkumar Devanbu[2] use blame for detecting the ownership of the file and analyze different kinds of questions related to the faults and other characteristics of the code. Matteo Orrù and Michele Marches study how file authorship effects quality of refactoring and they use git blame to examine how much
each developer has changed the file[15]. Based on the literature, it is not clear if only changes visible in the blame should be used for ownership detection, or if it is better to use all changes from the log.
Chapter 3

Environment

This chapter focuses on the current software related to the goal of this thesis. First, it describes how Git is done and how it can be used. Git has been chosen because the actual work is done on it. After that, the thesis describes other systems which are used to help a developer.

3.1 Git

Git is the most popular version control system today[16]. Git was started by Linus Torvalds in 2005 for Linux kernel development[4].

3.1.1 Commit and hash

Everything in Git is tagged with a Sha1 hash. Sha1 is calculated from the whole Git item. Commit hash is the most visible hash for the user. Commit has several fields such as the hash of the corresponding tree. The tree is described later in section 3.1.2. There is also a hash of the parent. Parent is the previous commit in branch sequence. After that there is the committer's name and email address and the commit message. Commit message is simply a text which the committer has written. It may also contain other information. The commit hash is hashed from all this data in a specific format. Every item in git is immutable. Unlike many other VSC, Git does not use diff encoding but stores full snapshots of the files. [17, 4]

3.1.2 Commit version tree

From figure 3.1 one can see how commits and trees work. Every commit has a predecessor which is called the parent. From the predecessor tree, the developer can see how the
current code is constructed. Every commit has a link to the tree starting from the root. From the root tree node, there are links to files which are called blobs and folders which are called trees. New commits create only new blobs and trees of changed files and folders. Those folders and files that are not changed are linked directly to the same object as the previous commit, like file 1 is linked to commits one, two and three. [18]

Revisions in git are named with the latest commit. This means that revision and commit in many cases mean the same.[4]

### 3.1.3 Tags and branches

Tags and branches are pointers to commits, which a human being can remember. Because commits are labeled only with sha1, it is convenient that a user can create branches and tags to them. Tags are locked to a specific commit. For example, released versions can be tagged to keep in mind which code is published and which is not. A branch is automatically updated when code updates. Every time the user commits new code the current branch goes to that new commit. There can be an unlimited number of branches and tags to the commit. In addition to normal tags there is HEAD, which is a special tag that points to commit of the current working copy.
3.1.4 Git functions

Git has several functions. Here is a list of the functions that are needed later in this thesis.

Diff

Listing 3.1: Sample diff from Teleirc[19]

diff --git a/tg.js b/tg.js
index 4d18f5a..c8b0e2a 100644
--- a/tg.js
+++ b/tg.js
@@ -128,10 +128,11 @@ module.exports = function(config, sendTo)
    {msg.voice || msg.contact || msg.location) && !
      config.showMedia) {
    return;
    }

-    var text;
+    var text;
    if (msg.reply_to_message && msg.text) {
        text = msg.text .replace(/ \n/g , ',n< ' + getName(
            msg.from, config) + ': ');
        sendTo.irc(channel.ircChan, ',< ' + getName(msg.from
            config) + ': ' +
-            '@' + getName(msg.reply_to_message.from,
                config) + ', ' + msg.text);
+            '@' + getName(msg.reply_to_message.from,
                config) + ', ' + text);
    } else if (msg.audio) {
        sendTo.irc(channel.ircChan, ',< ' + getName(msg.from
            config) + ': ' +
            '(Audio)');
@@ -181,7 +182,8 @@ module.exports = function(config, sendTo) {
            sendTo.irc(channel.ircChan, getName(msg.
                left_chat_participant, config) +
            ' was removed by: ' + getName(msg.from, config
                )});
    } else {
        sendTo.irc(channel.ircChan, ',< ' + getName(msg.from
            config) + ': ' + msg.text);

    13
+        text = msg.text.replace(/\n/g, '\n<') + getName(
+            msg.from, config) + '>' + text);
+        sendTo.irc(channel.ircChan, '<' + getName(msg.from
+            ) + '>' + text);

Git diff shows changes between two commits. Git diff uses the same format that is possible in Linux diff. For example, diff in listing 3.1 cannot show how it is constructed. The first row shows which file is diffed. The next three lines are uninteresting for the purposes of this thesis.

The line starting with double @ tells the context. First, there are the lines from where a hunk starts. Hunk means changes near each other in one block in the diff. It is displayed in line numbers in the original file and the resulting file. Then the Git tries to detect the function name to show which function is changed. This detection does not always work, but luckily it is not interesting information for this project. After that, there are some lines of context. The lines with minus and plus tell the lines that have been removed and added. If a line is changed, there are both minus an plus lines for one change. In this example, there is also another hunk.

Git diff can also show multiple files. In such a case, the other file starts similarly after the first file.[20]

**Blame**

Blame in version control systems is a view in which every line of the file is marked with the information of the commit in which the line has been changed last or added. The data can be anything that is in the commit: info, message, committer or time stamp.

**Checkout and reset**

Checkout and reset change the HEAD. The difference is that checkout creates a new branch or uses the old one and moves HEAD there. Reset takes an active branch and moves it to another commit. If the previous commit is not linked with another branch or tag, it is deleted later. Git keeps non-branch commits for a month or two and removes them after that. The user can access those by relog.

**3.2 Gerrit**

Gerrit is a code review tool for Git and also includes Git browsing functionality[21]. Gerrit history starts from Google's Modrian project. Modrian is a code reviewing tool similar to Gerrit built on top of the Perfore version control system. It is a closed
source project. When parts of Modrian were open-sourced and integrated with SVN, Rietveld was born. Gerrit started as a fork from Rietveld for Git. All these projects were coded in Python, but in 2009 an entirely rewritten Java-based Gerrit 2.0 was released\cite{22,23}. Gerrit has many notable users, for example Android\cite{24}, Chromium Os\cite{25} and LibreOffice\cite{26}.

3.2.1 Scalability

In many ways, Gerrit scales very well. It can handle 40-gigabit repositories with thousands of users. It can also handle twenty thousand pushes a day and twenty thousand projects on the same server. \cite{27}

Although the blame performance in Gerrit has been updated, Gerrit caches only newest version of every file\cite{28}. At least the first version of the caching engine refreshes cache just once in every 12 hours, which makes recently changed files slower for the first load. Gerrit does not cache log history, so finding the cached entry takes time. Gerrit developers thought that the shortcomings would be fixed later. However, no information about the recent situation was available on the time of writing this thesis. Gerrit is written with Java and uses JGit as Git library. There are also bugs in JGit library which are bypassed in Gerrit.\cite{29}.

3.3 GitLab

Our company uses Gitlab as our company Git review and browse tool. In this thesis, the speed of the Blame is compared mainly between the created software and the command line Git, although Gitlab is also used as an alternative reference point. Gitlab has a very rich user interface with many non-Git related features, such as task management.

Gitlab was launched in 2011\cite{30}. Gitlab is written mostly with Ruby, but some parts use Go instead. Our company hosts Gitlab in our premises. There is also a cloud-hosted version available.

3.4 Other frameworks

There are several other systems similar to Gerrit and Gitlab, such as Github. Github is a closed source software, so it is not examined in the thesis.
Chapter 4

Methods

This chapter describes the background techniques and research which are used in the implementation.

The primary goal of this thesis is to make Git blame generation much faster. The primary method is to use stored blame results and update them to needed revision. This way it is possible to store only some part of the blames but also be able to generate all blames quickly.

4.1 Caching

The key point to speed up the blame generation is to store at different points in the commit log. There are many options to do this because the data structure is straightforward. The simplest way is just to use directory structure with simple text files. Because the commit and file counts are not enormous, this would work.

The second option is a database. Because of the simple data structure, the program is not limited to any kind of database. Performance is the only thing which matters. Y. Li and S. Manoharan have studied database speeds and concluded that any database type does not make it better than any other type[31]. The database is needed to be selected based on operations the software uses because different databases perform better in different actions. This blamer tool uses reading the most. The writing is done at night or after delivering the result to user so small delay is not a problem. Packing is also useful to have because the system stores quite significant amounts of infrequently used data.
4.2 Diff

The diff is also generated externally. The diff is needed to know which lines are changed between the result and the stored blame. Git uses the Myers diff algorithm by default[20]. The default is also used in the project because it should mimic the standard git as much possible.

4.2.1 Mayers diff algorithm

Eugene W. Myers developed the Mayers diff algorithm in 1986. The algorithm uses $O(ND)$ time where $N$ is the sum of both files lengths and $D$ the minimum edit distance. Because of the $O$-value, the algorithm is efficient in diffs where the difference is small. [32]

Figure 4.1 visualizes how the Myers algorithm works in its basic form. On the left, there is the original file, and on the top is the edited file. In the graphical form, the algorithm needs to find the optimal route from the north-west corner to the south-east corner. Using path horizontally means that the character in that column is added. When going vertically, the character in that row is deleted. Diagonal path means that the character is not changed. [32]

In this example, the algorithm has three optimal routes marked blue and red. All other routes are also possible but not optimal. Both branches of the red and the blue route
have one deletion and two additions. Red adds A and C and removes A. The blue adds B and C and removes B. This is the reason why diff is ambiguous. [32]

4.3 Blame

The program uses standard blame for its data. It needs full blame to be stored and fetched into the database. It also requires partial blame to update the stored blame for the result. In partial blame, every line which is introduced between the limits is same than in full blame[33].

Git does not use caching. When a user asks a Blame, Git runs diff for all consequent pairs in file history until if find the origin to all lines[34]. However, Git does not try to find the ultimate result for small uninteresting line hunks like "\}"
. They are combined with the nearest bigger hunk even they are not really originating from there[35]. Git parses history in a tree shape and uses queue and task splitting in it. Because the diff is ambitious, the blame can also have differences.[36]

4.4 Process

The process has three steps. Fetch the original data, edit it with the diff result and update the edited blame list with the partial blame.

First, it is needed to determine and fetch the latest prior revision before the revision that user is asking. This information is found in version control system log. It can also be being pre-stored to the database. This option is not discussed in this thesis. The diff between those two revisions is also needed.

The updating of the older blame starts with the modifications from the diff. Because the diff does not allow editions, there are only added or removed lines. If the line is deleted from the file, it is also deleted from the blame listing. Every blame row is tight to that one specific source line. Same way the added line also adds a change to the blame listing. This way the algorithm can use diff of the files as a diff of the blame listing. Of course, this blame-diff does not include the information of what is changed, only what lines are added and what is removed. After parsing the diff, the partial blame result has the same count of the lines than in the file, and all those lines which are not changed have the right information. Those lines that are changed have wrong or empty info.

The final part is to check the lines that are changed from the partial blame and update them to our result. Those lines that are not changed are still right because of the stored blame. There can be some differences in this generated blame and the blame which can be generated directly, but they are only some not interesting lines because of differences in diffs. The reason for this is described in section 4.2.1. These differences are the reason incremental algorithm could not be used to populate the database.
Chapter 5

Implementation

5.1 Requirements

The main requirement for the program is the response time of the Blame. The response time should be only a few seconds. 15 seconds is still acceptable, but 30 is not because the user gets frustrated. In normal web interfaces already 4 seconds is too much [37], but this system is used for professionals who use it often enough to know it but rarely enough to not get frustrated. In small questionnaire performed in the company, 11 of 13 responders think that 10 seconds or more is the limit when they get frustrated. The same percentage thinks that 5 seconds or below is a good result. Based on the small internal questionnaire, the blame is used several times a week but not daily, and when it is needed, they need to use it on average 2.5 times.

Another requirement is that the system can be run from a decent server. Database initialization should be doable in a maximum two months. It should not use an excessive amount of disk space. Several terabytes are still usable, but it should not be more than 10 TB. The usage of the other resources should also be reasonable.

5.2 Approach

My general approach is to use heavy caching to show the blame view faster. I store blame results to a NoSql database and create the needed results for the user incrementally from that data. The main idea is to find the first commit before the commit from the database and take the diff from the version of that commit and the version that is needed. I also take partial blame between those two versions. With the diff and the partial blame, the algorithm can edit a copy of the stored blame to show blame for the needed version.

Terminal and web interfaces should be implemented later. The web interface should be so simple that browsers can handle blames of files with a size of 100000 lines. It should
also include simple features for browsing files and commits. In this thesis, the speed is the only aspect of the usability of the program that is considered.

5.3 Libraries and software used

5.3.1 Python

Python was chosen as a programming language for several reasons. First of all, it has all the needed libraries, and they are well maintained. Python is also good for text processing [38]. [39]

5.3.2 PyGit2

PyGit2 is the natural choice for Git library for python. There are several other libraries as well, but it looked most promising. Pygit2 is set of bindings to libgit2 C-library. [40] LibGit2 is a library for Core Git functions written in C. LibGit2 is used in the biggest git repository providers such as GitHub, GitLab, and BitBucket. LibGit2 has bindings to many programming languages.[41] Implementing process encountered quite significant problems with PyGit2. Some of the issues are data size dependent, for example, blame command. Other issues are some missing features. Issues are described later in this section 5.3.2.

Blame

The blame command in Pygit2 supports showing the blame between 2 commits and partial blame between 2 lines of the file. With small files, it works fine, but when trying with a large file, I realized that it is much slower and resource hungry than command-line git. Moreover, limiting blame by commits or lines does not make command any faster. Blame command with our example file of 7000 commits and 60000 lines consumes all available memory and crashes even if I restricted to one commit and line. The 350-line file took 35 seconds with Pygit2 while standard git uses only 2.8 seconds.

PyGit2 blame also had another problem. The company repository has many commits migrated from an old CVS repository. There are no email addresses in commits coming from CVS, and the migration script that was used added none. There is a bug in the pygit2 library which crashed the program if it tried to get the committer name from a commit with an empty email address. I filed a bug-report and planned to get the bug fixed until I decided not to use the blame feature of the library.

These problems were fixed by running standard Git blame with python exec. Running as exec means that a standard git program is run as subprogram and the output is read
back to the Python side. The result is the same as the library provides, but faster. It is almost as fast as the Git blame run from command line.

Standard Git output was tried. It was easier to parse in our case, but there was a problem for showing specific char combinations in the hash, so it was needed to use no porcelain format. No porcelain format is git format made for programs. The porcelain format has more info than standard format and, in many cases, it is easier to parse. The downside is that it needs more memory in the reading side because every line has only hash and the line and other info are emitted only once. The special porcelain format emits other info also many times, but then it uses much more lines. [42]

Diff

The diff is created using the PyGit2 library. The library outputs the same output than command line Git diff. The only problem with the library is that specific file diffs cannot be requested, but only all files. This issue is not a significant problem because the parsing functionality just skips the uninteresting parts. There are no performance issues in pygit2 diff.

Log

The log feature in PyGit2 is not very well implemented. There is no easy way to get every commit of one file. Because of this, I use exec to run log feature of the Git directly. The use of exec is not a big issue because the output is right and needs only minimal parsing.

5.3.3 Threading

Because the ultimate goal of this project is providing a practical tool for the whole company, it is needed to think about how our code works in simultaneous use. Other components are made for a web environment, but Git is not. Pygit2 commands log, and file content fetch need the repository to be set to specific state. These are the only problematic commands that need a specific repository state. Other commands work well concurrently because they do not need specific state and they are only reading the data.

Log needs that the repository is reset to the most recent commit. The repository needs to be reset to that commit to get file contents for a specific commit.

5.3.4 MongoDB and pyMongo

MongoDB and pyMongo are chosen because the system stores large masses of text. It also performs well for fetching the data[31]. MongoDB has a new engine called WiredTiger
which compresses data very well[43]. In their tests, they got 75% off of the size of an email database. Long similar lists of commits are more likely to get compressed even better. Results about this are presented in the evaluation section, even though this is not the main thesis topic.

MongoDB is a NoSQL engine. Lines in MongoDB and other NoSQL systems are called documents. Documents have fields which are like traditional database columns, but every document can have different fields. The field can have almost anything as their data in addition to text or integers such as arrays or binaries. In this program, every document has similar fields, and data is only text.

The primary data in MongoDB is on the blames table. It has documents which consist of the file path, the commit hash, the list of hashes as a string and a string of committer names. Here it is possible to add commit times, but this is left for later development. These lists are long line break spaced lists to allow easy copying to other programs for testing purposes. When MongoDB is moved to production, all the requirements should be checked properly. For example, XFS-filesystem is recommended. In the test setup, these are not considered. Also, SSD-discs and much ram make the DB fly. [44]

5.4 DB initialization

The DB initialization is done in a background process when the user interface is not in use. The main idea is to run it once and after that update new commits every night. Adding the changes of the previous day is not a resource intensive task and can be done fast.

5.5 Incremental blame

When new blame needed to be generated incrementally, first the system needs to check if the exact blame is in the database. If the system found it, it can return the result.

If exact blame was not found, the system finds the latest prior blame stored in the database. The system uses that as a base for our processing. In this example (Figure 5.1) the system generates the blame for version sha15(*1) and have blame for version sha9(*2) in the database. Different sha numbers mean some commits. Sha number in the blame, tells that the line comes from that version. In this figure, codeXX implies some code in the line, and it can be thought like real code.

In this example, the program only cares about hashes. Usually, the committer name and the commit time are needed. Those are easily be generated while selecting the hashes. Just keep committer and date info with hash info or fetch them from the commit later.
Figure 5.1: Example incremental blame generation

Then the system generates diff(*3) between versions sha5 and sha9. The diff format of the example listing is simplified compared to git diff, but it includes the same information. In the diff, line numbers are given in result file. The system does not need the line numbers in starting file. Diff lines have to be read in the order. For example, last removed line 8 is the line 8 in the result file after all previous lines in the data are modified. It is line with code7, not the 8th line of the original blame(code8).

The last piece what is needed is the partial blame between versions sha9 and sha15(*4). In this blame can be fetched from Git, all the lines that are unchanged after sha9 are marked with sha9. It cannot determine whether it is sha9 actually or an older version, but fortunately, it is not needed to do so.

The program starts generating the result line by line. First, it checks if the diff marks the next line as deleted. If it does, the system does not do anything else, just do not move that line. In this example, code2 and code7 are removed like this.

If the next line is marked as addition, the program checks the sha from the partial blame (*4). This sha is inserted into the resulting blame. Lines code11 and code12 are examples of this.

If diff does not say anything about the next line, the algorithm still needs to check the partial blame. If the line is changed two times between our two commits (in this example
sha9 and sha15), diff does not show any change. From the partial blame, it can be seen if that line hash is different from the starting hash (this time sha9). In this example, code9 line behaves like this. If the hash is the starting hash(sha9), the algorithm knows that the line is not changed after our starting point.

The last and most common option is to keep the line from the original blame and continue. After the algorithm has checked every line in all of the inputs, our result blame is ready and can be combined with the file content and sent to the user.

This method introduces small differences to the standard blame. This partial blame cannot be used for database initialization because these differences multiply and get bigger after each step.
Chapter 6

Evaluation setup

6.1 Systematic approach

A good way to create a meaningful performance analysis is to create it systematically. Jain[45] describes it in 10 steps.

State Goals and Define the System

It is important to define the measured system properly and select relevant metrics. The system should be limited to only those parts which can be controlled and which are in the study area. For example, if we are studying different CPU’s we do not want to pipe all data through internet and time that. When selecting which parts are in the system, we also can specify the coals and what we want to archive. These goals should be specific enough. [45]

List Services and Outcomes

Another necessary step is to specify the system services. It is needed to know what outputs system can have, what results are desirable and what are not. These are used for metrics and workloads. [45]

Select Metrics

It is important to use the right metrics to get meaningful results. Based on the services we want to select metrics so that they describe our goals well. Normally metrics are related to performance, accuracy or availability. Throughput, response time and latency are examples of performance-related metrics. Response time is time between user request
<table>
<thead>
<tr>
<th>Criterion</th>
<th>Analytical Modeling</th>
<th>Simulation</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage</td>
<td>Any</td>
<td>Any</td>
<td>Post prototype</td>
</tr>
<tr>
<td>Time required</td>
<td>Small</td>
<td>Medium</td>
<td>Varies</td>
</tr>
<tr>
<td>Tools</td>
<td>Analysts</td>
<td>Computer languages</td>
<td>Instrumentation</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Low</td>
<td>Moderate</td>
<td>Varies</td>
</tr>
<tr>
<td>Trade-off evaluation</td>
<td>Easy</td>
<td>Moderate</td>
<td>Difficult</td>
</tr>
<tr>
<td>Cost</td>
<td>Small</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Scalability</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 6.1: Evaluation techniques

to system response. It is needed to specify the exact places that are timed. For example, if the user interface is not studied, we can start timing when the actual request is sent. Mean is all that is needed in many cases. Variance is, of course, important at least if it is large. In a shared system, it is also needed to study individual and global metrics. If some change speeds up one feature significantly but at the same time slows down every another user, it is not good. [45]

List of parameters

The list of parameters can be divided into two groups. Workload parameters include parameters that affect the input of the program. For example, how users use the system or how big files users store. The second part of the parameters specifies how the system is tuned. CPU count or amount of RAM are good examples. [45]

Select Factors to study

Factors are parameters which are varied during the study. Different values of the factors are called levels. Selecting a right set of factors and levels is crucial. We do not want to select too many factors and levels because testing resources are not infinite. We want to select those factors which are relevant to the study but also have a real impact on the results.[45]

Select Evaluation Technique

Selecting the evaluation technique needs to be done based on the measured system, time and resources. Goals can also influence this. The three possible ways are analytical modeling, simulation, and measuring a real system.

In all of these techniques have pros and cons. Jain[45] describes these in an array 6.1 from most important to least important.
This thesis mostly uses measurements and then do some very basic statical analysis. It is also a good idea to use two or even all three techniques to minimize the risk of failure. Every analysis result should be suspected until proven true. [45]

Select Workload

The test workload should be selected as near as possible to the real workload. The goal can also specify the workload. The workload is purely mathematical in static analysis. In simulation or measurement workload is real requests or generated requests. [45]

Design Experiments

It is necessary to specify which factors and levels we test in which order. For example, it would be nice to first test more factors with only two levels and then select most interesting factors and test them with more levels. This step also includes running the tests. [45]

Analyze and Interpret Data

It is needed to analyze the data properly. The tests need to be repeated and statistically analyzed to prevent random variability. [45]

Present Results

The final part is presenting the results. The results should be presented clearly in an easily understood way. Graphs are good and do not need to present too much statistical jargon. [45]

Static analysis

The systematic approach is better utilized in simulation or measuring real system. This thesis uses static analysis shortly and straightforwardly. Rough O-values are calculated by analyzing the execution flow. The data is also very simplified, and only two steps in the program and the native Git are used.

6.2 Measurement metrics

I selected three metrics to be metered. The first and most important is response time because that makes the difference in my program compared to others. Two others are
initialization time and disk usage. These are interesting, and we need to be sure that they are feasible.

6.2.1 Response time

The most important metric is query response time for the massive file. This response time is measured from user terminal interface because also the normal git is terminal only. Modern frameworks do not add big overhead if we use simple layout. We are pleased about results of below 5 seconds with the 70 thousand line and 8000 commit file. 10 seconds is still usable and 30 seconds is in the limits. This metric is also compared to normal git without caching.

6.2.2 Disk usage

Disk usage is critical metric especially when it grows tremendous. This thesis study how much disk space different configurations use when compared to the plain Git repository. Because we use a compressed database, we also deliver an uncompressed size as a reference. The excellent result is below 500 gigabytes, and three terabytes are near the limit. These limits are chosen based on disk space availability and price.

6.2.3 Initialization time

Initialization time is important metric but only if it differs much. Initialization is done only once, so it does not matter if it takes days or some weeks. The main thing is that it is doable at some suitable time. Less than one week is ok, one month is usable and two months is in the limits in a good machine.

6.3 Factors

The only factor in our program is part of the commit blame stored in DB. This factor changes all our three metrics. This factor is an important factor because at least disk space usage should be changed easily with it.

6.4 Evaluation technique

Measurement is mostly used because true world values are needed, and we have a working prototype. We also want to compare the program to native git blame, so testing is the only real way to do that. We also have a small static analysis of our program. That can make further development easier.
6.5 Workload

Real data is used as a workload. Because our program is easily scaled out and there are few users we do not test how it scales to a big amount of simultaneous users.

We use our company Git repository. It has about 4000 files. Size of the working copy is about 180 megabytes. The sum of all files row counts is approximately 2,6 million. In git repository, there are about 56000 commits. In last months our team has produced about 15-20 new commits a day to our repository. The initialization test just initializes the whole repository. The Disk usage test is similar in size of that initialized repository.

The biggest and most problematic file has about 75000 lines, and size is about 4 megabytes. The file has about 7700 commits. We use this file in our response time tests because in small files we do not get reasonable differences and they work fast already. Response time test checks the average and all quartiles of big enough count of different blames. Size of that asked list is chosen so that there are all possible distances to stored blame in the DB because we expect that that distance makes blaming slower. Because we have stored minimum one in 500 commits, we use 1000 consecutive blames.

6.6 Design experiment

We use several levels in our factor. Every 500, 100, 50, 5 and one blame are used as our levels. Every one option is not using the incremental feature at all. We add more levels if they sound interesting. We run every level manually. First, initialize the database with current factor level and time it. Then we take database usage of that.

The last thing is to run shell-script which runs needed workload and time it. For timing, we use Linux time command[46] and time every round separately to allow calculation of quartiles. The time command provides time usage in real time and a CPU-time. We use CPU time because then we can omit everything uninteresting (for example Linux UI).

6.7 Static analysis

We make a very superficial static analysis to compare our program to the native Git. Like in section 4.3 we can see that Git has no cache. It has to find the origin of all lines. Git has to check every line commit by commit until it finds the origin of the line.

We can approximate that half of the added lines in the commit are entirely new and another half changes some older line. Of course first commit is only add. This way the average age of the line is somewhere between commit count /4 and commit count /2. Commit count / 4 comes from the situation where edits always edit the oldest commits.
The commit count / 2 comes from the situation where commits always rewrite newest commits.

With this assumption we can approximate that git blame speed is $O(c^l)$ where $c$ is the commit count and $l$ is lines.

Here we think only our implementation with step size 50 and 1 for keeping it simple enough. The depth of the blame is maximum 49 or 0, so the speed of this is $O(1)$. Fetching the data is only dependent on file size with a small static factor. Generating the diff with small changes depends mostly on file size. The number of changes is quite static, commit sizes variable but not that much. The diff is $O($File size$)$. This means that the speed is overall $O($filesize$)$ with both step sizes, of course, the step size 1 is faster in reality.

### 6.8 Comparing to GitLab

We planned to make comparison also to Gitlab. In the end, we did not get any results because GitLab user interface is so heavy that no browser can handle it. Every browser crashed after waiting about five minutes. From this, we can guess that it uses native git command, but delivery fails because of so much sweet looking CSS and HTML on the page.

This problem in Gitlab is good to know because we are also going to make a web interface to this program. Because of this problem, the web-interface should be made as simple as possible. We do not use any layout markings, just plain text with links and very simple CSS like fonts and colors.

### 6.9 Measurement setup

Initialization time is measured with normal Linux Time command. With the command, we can measure how much time the system used for processing the program. This makes it more exact than external stopwatch because Linux system always does other tasks in the middle of the program and we do not want them to be in the results.

The disk space usage is measured from the MongoDB stats command[47]. It calculates database size and other statistics. It also calculates data size without WiredTiger packing(section 5.3.4). We add the size of the git repository because it is also needed in the normal Git and to run the caching program. The size of the repository is approximately 330 megabytes.

The measurements are done in one virtualized Linux environment. Virtual environment has two 3.3GHz 64bit cores. LsCpu command describes it with 6600PogoMips. The
processor has 32K l1 memory, 256K l2 memory, and 6144K l3 memory. All cores have NUMA node0 support. The mongo uses that. The system has 8,9Gb DRAM-3 memory.
Chapter 7

Evaluation

This chapter presents taken measurements. First, the section 7.1 describes how the partial blame of standard Git was measured. Next, these results are used to select the right factors for the created program. After that, there are the measurements that show how the step size affects the response time, disc space usage and initialization time. Then the disc space usage is compared to response time. Finally, the section 7.2 discusses the results and how they fit the goals.

7.1 Measurement results

7.1.1 Partial blame speed

![Graph showing the relationship between depth of partial blame and time in seconds.](image)

Figure 7.1: Speed of partial blame
Before selecting the factors, the blame time was tested compared to its depth. In graph 7.1 one can see how the blame speed decreases while depth increases. The blame is tested from the newest commit. Every depth is measured five times, and the average of those five measurements is saved. In the graph, there is one point for every 10 real data-points, calculated as average. From the graph can be seen that time increase is very linear and blame depth of 500 is an absolute limit, because it takes already 10 seconds. It is noteworthy that partial blame with a depth of 3000 is about as slow as complete blame even if the latter is about 8000 commits.

![Graph showing the relationship between depth and blame time](image)

**Figure 7.2: Speed of partial blame first 100**

In the figure 7.2 there is a closer look of the depths 1 to 100. There is nothing exceptional, but one can see that it is very linear already from the beginning. It is good to know that with a short blame depth the elapsed time is almost non-existent and does not increase extremely fast. The speed difference between the depths 1 to 20 is small and allows a relatively significant decrease of storable data with only small time increase. From the two figures, one can see that one should not test steps bigger than 500 because after that the blame itself takes too much time and tests should focus to small step sizes because the speed is the priority number one for the project.

### 7.1.2 Response time

In figure 7.3 one can see how the amount of caching affects the response times. The table 7.1 describes how the standard Git performs. One can imagine a line from the least cached level of the figure to the standard Git result. This imaginary line is very logical because the standard Git has to create blame for about 7000 commits deep.

From the graph can be seen that the response time of the blame drops fast from every 500 to every 100 commit. Every commit is faster mainly because the software does not
need to generate the commit log to find the right blame. There is a small measurement anomaly in 1/50: the average is bigger than the third quartile. There were about 30 exceptional long blames there, most probably caused by the computer running the tests.

Compared to the goals of the thesis, the results look good. With the longer storing step the speed is too slow compared to the aims of the thesis, but the speed with full caching is more than reasonable.

### 7.1.3 Disk usage

There are not many surprises in the graph of disk usage (7.4). The uncompressed usage grows in a very linear fashion compared to the portion of the blames stored. A happy surprise is how well WiredTiger compresses the data. These results are all excellent compared to our goals. From the graph can be seen that the compression ratio is a little
better with all blames stored than when a smaller portion of them is stored.

7.1.4 Initialization time

The initialization time in graph 7.4 is very similar to the database size, which sounds very logical, because the average blame depth and file size are similar in all blaming steps. Because the blames are run individually without the incremental algorithm, the time increases quite fast. With early tests with incremental database initialization speed increase was much slower.

7.1.5 Response time compared to database size

An interesting thing to consider is how much response time benefits from disk usage. Compressed usages are utilized because they will also be used in a real-world environment. The first point in the graph presents the standard Git. It and all other data usages include the size of the git repository. As we can see from the graph, the first three hundred megabytes over the plain Git repository are the most effective. After that, the effect is quite linear and goes to almost zero near six thousand megabytes.
It can be noticed from the graph 7.5 that the megabytes that are used first are the most efficient. After that, the saved seconds per megabyte decrease linearly.

### 7.1.6 Megabytes per saved second

The last figure 7.6 is fascinating. It tells how linearly the effect of cached data decreases. If it would be possible to cache more, it would be interesting to explore which caching level is most usable. In this case, the cache of full caching is still small enough to be used.
7.2 Discussion of the results

The first research question was about whether or not it is possible to generate blames properly from the intermediate result. The results verified that this type of caching works fine. The second research question was: How does the amount of caching speed up the process? The software reduced blaming speed significantly, below one percent with full caching. The last question was which caching level is the most suitable for production. Of course, it depends on the size of the repository, but full caching with this size of the repository is the best option. The full caching is fastest, and the other properties are good enough, so it is doable. It is also good that its result is exactly the same than the standard Git output.

The third research question was: How much data needs be stored and how long it takes? The database size of several gigabytes is very reasonable compared to what is archived. As can be seen from the goals, even several terabytes are still doable, so there is no reason to not to use full caching with massive repositories. In some very massive repositories this full caching might need too much disk space, but fortunately, this study shows that the database usage can be reduced efficiently. The about four day initialization time is also usable because it does not need static state of the repository and keeping it updated is easy.

There were also some possible improvements in the program. In this thesis, the stored commits are selected from every file similarly. More intelligent selecting would be more efficient. The need of blames per file can be used; users do not blame all files with equal frequency so less essential files would need less caching. The same can be done within inside files. For example, it is possible that newer revisions are blamed more. This customization could also be archived on the fly. If some blame is needed once, the program could store it in the database to speed up next time.

One useful future improvement would be storing hash lists from the log to the database. The full caching did not need it, and that is one of the reasons why that was so much faster than half caching. The file content could also be stored in the database. It will use more space than the hashes, but it should also speed up the process a little and reduce slowdowns because of less need of non-thread-safe functions.

The next project is to make the program work in the browser and keep the same speed. This way all developers can benefit it easily.
Chapter 8

Conclusions

Version control systems are a crucial part of today's software development processes. They can be used to help collaboration, but they also keep track of the changes and help to resolve and predict bugs. Git blame is one tool which helps a programmer to check when and why something has done in the way it is done. That information is also used in the research when it is studied how programs have been created.

This thesis developed a program which improves the speed of the Git blame. Git is the most used version control system, and the blame in the Git is very slow with bigger files with a long history. Some web-frontends crash and terminal git can take a minute or two.

The main idea in the implementation was to store some portion of the blames in the database and then use partial blame and diff to generate the actual result from it. It is possible to change the storing frequency and to choose the best frequency for each situation. The limiting factors are disk usage and initialization time.

The software developed in this thesis functioned well. The system provides working blame views although they had some minor differences to what Git delivers. The reason is that diff is ambiguous when there are similar lines. The system speed is quite linear to the portion of the blames stored. This case full caching is the best option because it does not use too much disk space and is fastest. With the full caching, the blame speed is about 0.2 seconds. The software written by the author is in production use.

The results were excellent and the reduced blame time was higher than what was expected by the developers of the company, as they considered anything below 5 seconds as a good improvement. The limitations of this study for research use is the amount of preparation needed. Many times the databases are parsed only once or twice, and it does not make any sense to make this caching because of that so the next step would make git blame generation faster without large initialization. Further studies could focus on finding ways to make incremental blame generation even faster.
Bibliography


    2015. URL: https://www.mongodb.com/blog/post/new-compression-options-
mongodb-30.

mongodb.com/manual/administration/production-notes/.

[45] Raj Jain. *The art of computer systems performance analysis - techniques for exper-
imental design, measurement, simulation, and modeling*. Wiley professional com-


docs.mongodb.com/manual/reference/method/db.stats/.

42