Version Control System

Reece Donovan

B.Sc. Computer Science - Final Year Project

Prof. Ken Brown

University College Cork

February 6, 2023

Abstract

Declaration of Originality

Acknowledgements

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 What is a Version Control System

A version Control System (VCS) saves modifications done by individual software developers, making the process more accessible since they can track their work over time. In addition, it helps share data between nodes where each node can be kept up to date with updated versions, so there is no need for any merge conflicts.

The advantages provided by VCS include: aiding in collaboration among programmers, improved efficiency when working on larger groups of mixed products; providing an audit trail; easy branching functionality; simplifying team-based development; understanding who has worked on specific pieces or sections during what period in time but also making sure that all changes are appropriately tracked providing transparency within teams while maintaining optimal performance levels through streamlined management processes, helping avoid errors arising from inconsistency between versions such as mismatched documents or programming problems resulting from conflicting edit operations.

2.1.1 What is the purpose of Version Control Systems

For almost all software projects, the source code is an asset that must be protected. For most software teams, the source code is a repository of invaluable knowledge and understanding about the problem domain that developers have collected and refined through careful effort. Version control protects this priceless resource in many ways: it prevents catastrophe or casual degradation from occurring and prevents human error or unintended consequences from affecting its quality.

For example, software developers working in teams continually write new source code and change existing code; these are organized into file folders called "file trees".

One developer may be working on a new feature while another fixes an unrelated bug by changing parts of the file tree; each developer may change several parts at once.

Version control can help teams solve these kinds of problems because every individual change made by each contributor is tracked, helping prevent concurrent work from conflicting with one another so that any incompatibility should be discovered and solved without blocking team members' progress further down the line (this will also ensure that any changes being made do not introduce bugs).

Further still, testing for new versions cannot occur until some development work is done beforehand. This means both processes move forward together until they reach their newest version - thus decreasing risk to both sides due to errors along either path.

2.2 Types of VCS

The two approaches to VCSs are Centralized and Distributed. Both approaches are in widespread use today. The centralized approach is based on the client-server model, where a single central repository stores the history of all files. In contrast, the distributed approach provides each user with a full repository copy.

While there is no clear answer for which approach is best, it's important to note that the centralized approach features a single point of failure but also more challenging scaling than its counterpart (Distributed).

2.2.1 Centralized Version Control Systems (CVCS)

A centralized Version Control System is a system that enables developers to work together on the same project by storing the primary copy of files in a central repository. This system keeps track of all files and saves information in the local repository. CVCS are called centralized because there is only one central server or repository.

The server maintains a complete record of issues, while clients only maintain a local copy of the shared documents. All developers make their modifications to the repository through checkout. However, only the last version of the files is retrieved from the server, meaning any modifications made will automatically be shared with other developers.

Users can modify in parallel with their local copy of shared documents and sync with the central server to release their contributions and make them visible to other collaborators. However, because centralized version control systems rely on one repository that includes the correct project version, they must restrict write access so that only trusted contributors can commit modifications.

CVCS has some challenges, such as if the central server is inaccessible, users will not be able to merge their work or save the released modifications. Also, if the central repository is corrupted, everything will be lost. Contributors must be the ones who have writing permissions to perform basic tasks, such as reverting modifications to a previous state, creating or merging branches, or releasing modifications with complete revision history. This limitation affects participation and authorship for new contributors.

So, the main drawbacks of using CVCS are that it requires a network connection to work on the source code, developers must order to contribute to a project, and a single point of failure is an issue when using one server.

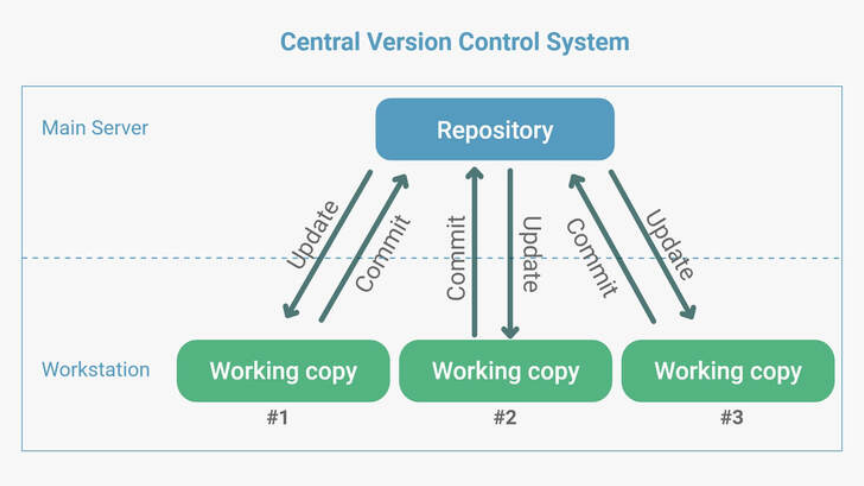

Figure 2.1: Centralized Version Control System

2.2.2 Distributed Version Control Systems (DVCS)

Distributed Version Control Systems (DVCS) were created to overcome the limitations of Centralized Version Control Systems (CVCS), which enable branching and merging, avoid local VCS operations and allow developer collaboration. Due to the limitations associated with using a centralized version control system, Open Source Software (OSS) projects today broadly adopt DVCS.

DVCS is designed to work in two ways: it keeps file histories locally on each device. However, it can also sync local user modifications with servers again when necessary so that these modifications can be shared with everyone else. In addition, in DVCS, developers can work separately or together on the same project, as they have access to all repositories needed for their task; any repository can be cloned from another, so there is no repository more important than others.

To provide a new way for versioning software artefacts, several Distributed Version Control Systems emerged in the software field, such as `Mercurial`, `Git`, `Bazaar`, etc. These tools have been adopted by many Open Source Software (OSS). The operations in DVCS are much faster than those found in CVSS because they are done locally, while CVSS operations require remote connection; some consider that distributed systems will soon replace centralized ones because they suit more substantial projects with more independent developers who want full functionality even without network connection available, offer advantages like earlier drafts of your work being saved without requiring you releasing them publicly or sharing them with other people etc.

Collaboration between team members and allowing individual developers to be servers or clients are the most important features of version control systems, so developers can work on source code without being connected to a central or remote repository.

DVCS introduces some challenges: it lacks a coherent version numbering system, where there is no centralized versioning server, and uses hash modifications or a unique GUID. So, the lack of a central server makes system backup difficult.

The two most prevalent complaints about the disadvantages of DVCS are that: pessimistic locks are not available and they have weak tools for binary. However, the reasons for the transition from centralized to decentralized version Control Systems are that developers can work offline and work incrementally efficiently because they can make several roles, such as developing new tasks or fixing errors; this also leads to exploratory coding, which gives them more freedom in their workflow while still retaining control over their code's release schedule.

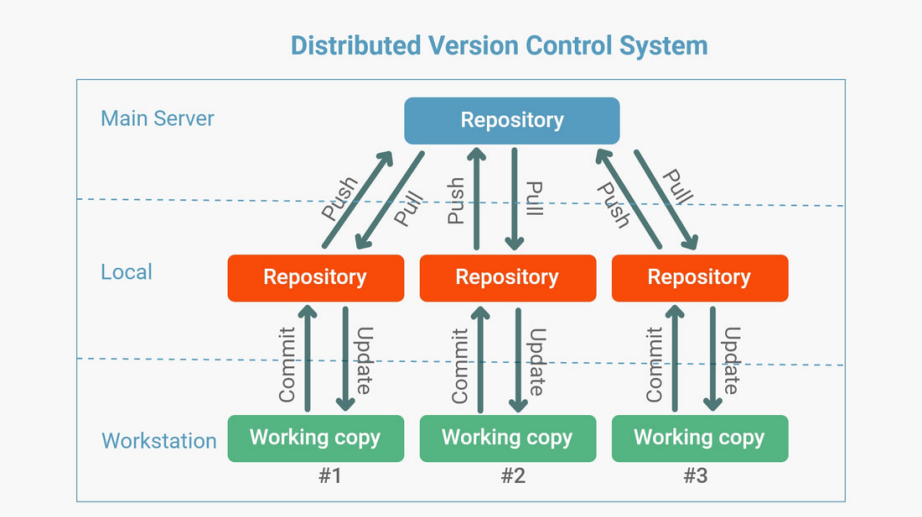

Figure 2.2: Distributed Version Control System

2.3 Existing Version Control Systems

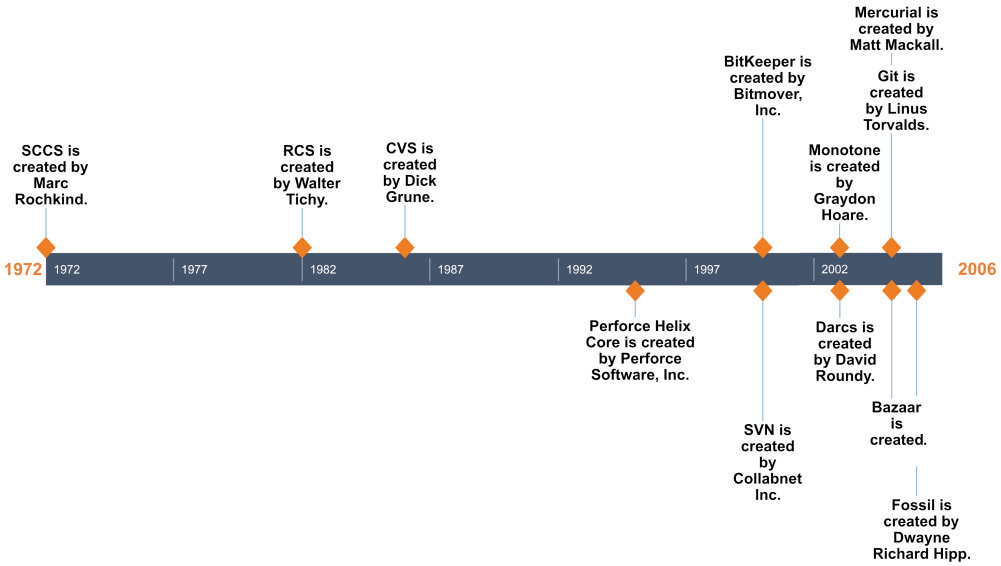

Figure 2.3: Timeline of the Creation of Version Control Systems [1]

2.3.1 Local Version Control Systems

Early VCS were intended to track changes for individual files and checked-out files could only be edited locally by one user at a time. They were built on the assumption that all users would log into the same shared Unix host with their own accounts, but it was not always possible. As you can imagine, these early systems made it easier for small teams to revisit code states from various points in history.

Source Code Control System (SCCS)

SCCS was released in 1972 and is one of the first successful VCS tools. It was written by Marc Rochkind at Bell Labs, who wanted to solve the problem of tracking file revisions. The tool made tracking down bugs introduced into a program significantly more manageable. SCCS is worth understanding at a basic level because it helped set up modern VCS tools that developers use today.

Architecture Much like modern VCS tools, SCCS has a set of commands that allow developers to work with versioning of files. The basic command functionality are:

- Check in files to track their history.
- Check out specific file versions for review.
- Check out specific file versions for editing.
- Check in new file versions with comments explaining the changes.
- Revert changes made to a checked out file.
- Basic branching and merging of changes.
- Print a log of a file's version history.

A special type of file called a `s-file` or a `history file` is created when a file is tracked by SCCS. This file is named with the original filename prefixed with a `s.`, and is stored in a subdirectory called `SCCS`.

So a file called `test.txt` would get a history file created in the `./SCCS/` directory with a name of `s.test.txt`. When created, the `s-file` contains the original file contents, a header that contains the file's version number, and some other metadata. There are also checksums stored in the `s-file` that are used to verify the integrity of the file (i.e. to make sure that the file has not been tampered with). The `s-file` content is not encoded or compressed in any way, which is a clear difference from modern VCS tools.

Since the original file's content is now stored in the history file, it can be retrieved into the working directory for review, compilation, or editing. Further changes made to this new copy, such as line additions, modifications and removals, can be checked back into a revised version of the history file, which increments its revision number.

Subsequent SCCS check-ins only store only deltas or changes to a file instead of storing entire contents each time; when a check-in is made, subsequent revisions are added onto existing delta tables inside an amended history file (history files do not use compression). This decreases the size of these large histories since they are not using compression on their files, so they take up more space than just having one complete copy that you are tracking with no differences like Word docs etc.

As previously mentioned, SCCS uses the Delta method known as Interleaved Deltas, which allows constant-time checkout regardless of how old your checked-out revision is - i.e., older revisions do not take longer than newer ones.

It is important to note that all files are tracked and checked in separately; there is no way to check in changes on multiple files as a part of one atomic unit - like a commit in Git. Each tracked file has its corresponding history file, which stores revisions. Generally, this means that the version numbers of different files cannot match each other. However, matching revision numbers can be achieved by editing

every file at once (even if not all of the changed areas have fundamental changes) and checking them all together. This will increment revision numbers for any modified content, so they'll be consistent with their revisions—but it's not comparable to including lots of different pieces within a single commit like you would in Git. In SCCS, these make individual check-ins across separate history folders rather than one extensive report containing everything at once.

When a file is checked out for editing in SCCS, it is locked so that anyone else cannot edit it. This prevents changes from being overwritten by other users and limits development since only one user can edit the file simultaneously.

The software supports branches that store sequences of changes to specific files - these can then be merged back into the original versions or with copies of other branched versions of the same parent branch.

Basic Commands

- `sccs create <filename.ext>` - Check in a new file to SCCS and create a new history file for it (in the `./SCCS/` directory by default).
- `sccs get <filename.ext>` - Check out a file from from its corresponding history file and place it in the working directory in readonly mode.
- `sccs edit <filename.ext>` - Check out a file from the corresponding history file for editing. Locks the history file so no other users can modify it.
- `sccs delta <filename.ext>` - Check in the modifications to the specified file. Will prompt for a comment, store the changes in the history file, and remove the lock.
- `sccs prt <filename.ext>` - Display the revision log for a tracked file.
- `sccs diffs <filename.ext>` - Display the differences between the current working copy of a file and the state of the file when it was checked out.

Revision Control System (RCS)

The Revision Control System (RCS) was released in 1982; it was written in C by Walter Tichy as an alternative to SCCS, which was still closed-source at the time. RCS was released under the GNU General Public License, which allowed it to be used in open-source projects.

> "RCS manages revisions of text documents, in particular source programs, documentation, and test data. It automates the storing, retrieval, logging and identification of revisions." – Walter Tichy [2]

Architecture RCS shares many traits with its predecessor, including:

- Handling revisions on a file-by-file basis.
- Changes across multiple files can't be grouped together into an atomic commit.
- Tracked files are intended to be modified by one user at a time.
- No network functionality.
- Revisions for each tracked file are stored in a corresponding history file.
- Basic branching and merging of revisions within individual files.

When a file is set checked into RCS for the first time, a corresponding history file is created for it in the local `./RCS/` directory. This file is postfixed with a `,v` so a file named `test.txt` would be tracked by a file called `test.txt,v`.

RCS uses a `reverse-delta` scheme for storing file changes. When a file is checked in, the history file contains the complete snapshot of its contents. When that same file is modified and checked in again, RCS calculates only one delta–the difference between the new version of that particular revision and the old version as recorded previously by RCS–and saves it along with an older snapshot if necessary.

This is called `reverse-delta` because to check out an earlier revision, RCS starts from what is newest and applies consecutive deltas until getting back to our desired revision; starting at what is newest allows for quick checkout times since we always have access to current revisions (since their snapshots are saved).

However, suppose you wanted to go back further than just one recent update on your project (1 or 2 old versions). In that case, things get considerably more complicated because those versions' snapshots would need to be calculated against each other before they can be applied to what is new.

This is not the case with SCCS since it takes the same amount of time to fetch any revision. In addition, no checksum is stored in RCS history files; therefore, file integrity cannot be ensured.

Basic Commands

- `ci <filename.ext>` - Check in a new file to RCS and create a new history file for it (in the `./RCS/` directory by default).
- `co <filename.ext>` - Check out a file from from its corresponding history file and place it in the working directory in readonly mode.
- `co -l <filename.ext>` - Check out a file from the corresponding history file for editing. Locks the history file so no other users can modify it.

- `ci <filename.ext>` - Check in file changes and create a new revision for it in its corresponding history file.
- `merge <file-to-merge-into.ext> <parent.ext> <file-to-merge-from.ext>` - Merge changes from two modified children of the same parent file.
- `rcsdiff <filename.ext>` - Display the differences between the current working copy of a file and the state of the file when it was checked out.
- `rcsclean` - Removes working files that don't have locks on them.

2.3.2 Centralized Version Control Systems

Version control technology continued to evolve, leading to centralized repositories that contained the 'official' versions of their projects. This was good progress since it allowed multiple users to checkout, work with the code simultaneously, and commit to this central repository. Furthermore, network access was required for people who wanted to commit changes they had made locally.

Concurrent Versions System (CVS)

Dick Grune developed the Concurrent Versions System in 1986. It was written in C, and its goal was to add a networking element to version control. As a result, the Concurrent Versions System became the first widely used VCS tool that allowed multiple users to work on the same project simultaneously from different locations. This kicked off the second generation of VCS tools, allowing geographically dispersed development teams to work together on a project.

Architecture CVS operates as a frontend for RCS - it provides a set of commands to interact with files in a project but uses the RCS history file format and commands behind the scenes.

This meant multiple developers could check out and work on the duplicate files for the first time in history. It did this by utilizing a centralized repository model.

The first step is to set up a centralized repository on a remote server using CVS. Projects can then be imported into the repository. When a project is imported into CVS, each file is converted into a `,v` history file and stored in a central directory known as a `module`. The repository generally lives on a remote server which is accessible over a local network or the Internet.

A developer checks out a copy of the module, which is copied to a working directory on their local machine. No files are locked in this process, so there is no limit to the number of developers that can check out the module at one time. Developers can

modify their checked-out files and commit their changes as needed. If a developer commits a change, other developers need to update their working copies via a (usually) automated merge process before committing their changes. Occasionally merge conflicts will need to be manually resolved before the commit can be made. CVS also provides the ability to create and merge branches.

Basic Commands

- `export CVSROOT=<path/to/repository>` - Sets the CVS repository root directory so it doesn't need to be specified in each command.
- `cvs import -m 'Import module'<module-name> <vendor-tag> <release-tag>` - Import a directory of files into a CVS module. Before running this browse into the root directory of the project you want to import.
- `cvs checkout <module-name>` - Copy a module to the working directory.
- `cvs commit <filename.ext>` - Commit a changed file back to the module in the central repository.
- `cvs add <filename.txt>` - Add a new file to track revisions for.
- `cvs update` - Update the working copy by merging in committed changes that exist in the central repository but not the working copy.
- `cvs status` - Show general information about the checked out working copy of a module.
- `cvs tag <tag-name> <files>` - Add an identifying tag to a single file or set of files.
- `cvs tag -b <new-branch-name>` - Create a new branch in the repository (must be checked out before working on it locally).
- `cvs checkout -r <branch-name>` - Checkout an existing branch to the working directory.
- `cvs update -j <branch-to-merge>` - Merge an existing branch into the local working copy.

Apache Subversion (SVN)

Subversion (SVN) was developed by CollabNet in 2000 and is now maintained by the Apache Software Foundation. It is written in C and was designed to be a more robust centralized solution than CVS.

Architecture SVN, like CVS, uses a centralized repository model. Therefore, remote users must have a working network connection to commit their changes to the central repository.

Subversion introduced atomic commits, which ensure that a commit will either fully succeed or be completely abandoned if an issue occurs.

If a commit operation in CVS fails, for example, due to a network outage, the repository can be corrupted and inconsistent. In Subversion, a commit or revision can include multiple files and directories. This allows users to track sets of related changes together as a grouped unit instead of separately for each file, like in past storage models.

The current storage model that Subversion uses for tracked files is called `FSFS` or `File System atop the File System`. This name was chosen since it creates its database structure using a file and directory structure that match the operating system filesystem it is running on. The unique feature of the Subversion filesystem is that it is designed to track not only the files and the directories it contains, but the different versions of these files and directories and they change over time. It is a filesystem with an added time dimension. In addition, folders are first class citizens in Subversion. Empty folders can be committed in Subversion, whereas in the rest (even Git) empty folders are unnoticed.

When a Subversion repository is created, a (nearly) empty database of files and folders is created as a part of it. A directory called `db/revs` is created in which all revision tracking information for the checked-in (committed) files is stored. Each commit (which can include changes to multiple files) is stored in a new file in the `revs` directory and is named with a sequential numeric identifier starting with 1. When a file is committed for the first time, its full content is stored. Future commits of the same file will store only the changes - also called the `diffs` or `deltas` - in order to conserve space. In addition, the deltas are compressed using `lz4` or `zlib` compression algorithms to further reduce their size.

By default, this is actually only true to a point. Although storing file deltas instead of the whole file each time does save on storage space, it adds time to checkout and commit operations since all the deltas need to be strung together in order to recreate the current state of the file. For this reason, by default Subversion stores up to 1023 deltas per file before storing a new full copy of the file. This achieves a nice balance of both storage and speed.

SVN does not use a conventional branching and tagging system. A normal Subversion repository layout is to have three folders in the root:

- `trunk/`
- `branches/`

- `tags/`

The `trunk/` folder is used for the production version of the application. The `branches/` folder is used to store subfolders that correspond to individual branches. The `tags/` folder is used to store tags which represent specific (usually significant) project revisions.

Basic Commands

- `svn create <path-to-repository>` - Create a new, empty repository shell in the specified directory.
- `svn import <path-to-project> <svn-url>` - Import a directory of files into the specified Subversion repository path.
- `svn checkout <svn-path> <path-to-checkout>` - Copy a stored repository path to the desired working directory.
- `svn commit -m 'Commit message'` - Commit a set of changed files and folders along with a descriptive commit message. These can be used as notes for future developers to understand what changes were made. The message of the initial commit is typically set to 'Initial commit'.
- `svn add <filename.txt>` - Add a new file to track revisions for.
- `svn update` - Update the working copy by merging in committed changes that exist in the central repository but not the working copy.
- `svn status` - Show a list of tracked files that have been changed in the working directory (if any).
- `svn info` - Show a list of general details about the checked-out copy.
- `svn copy <branch-to-copy> <new-branch-path-and-name>` - Create a new branch by copying an existing one.
- `svn switch <existing-branch>` - Switch the working directory to an existing branch. This will checkout the specified branch.
- `svn merge <existing-branch>` - Merge the specified branch into the current branch checked out in the working directory. Note this needs to be committed afterwards.
- `svn log` - Show the commit history and associated descriptive messages for the active branch (useful for devs to find details of previous changes).

Perforce Helix Core

Perforce Helix Core is a proprietary VCS created, owned, and maintained by Perforce Software, Inc. It is typically set up using a centralized model although it does offer a distributed model option. It is written in C and C++, and was initially released in 1995. It is primarily used by large companies that track and store a lot of content using large binary files, as is the case in the video game development industry. Although Helix Core is typically cost prohibitive for smaller projects, Perforce offers a free version for teams of up to 5 developers.

Architecture Perforce Helix Core is set up as a server/client model. The server is a process called `p4d` which waits and listens for incoming client connections on a designated port, typically port 1666. The client is a process called `p4` which comes in both command-line and GUI flavors. Users run the `p4` client to connect to the server and issue commands to it. Support is available for various programming language APIs, including Python and Java. This allows automated issuance and processing of Helix Core commands via scripts. Integrations are also available for IDEs like Eclipse and Visual Studio, allowing users to work with version control from within those tools.

The Helix Core Server manages repositories referred to as depots, which store files in directory trees, not unlike Subversion (SVN). Clients can checkout sets of files and directories from the depots into local copies called **workspaces**. The atomic unit used to group and track changes in Helix Core depots is called the **changelist**. Changelists are similar to Git commits. Helix Core implements two similar forms of branching - **branches** and **streams**. Branches are conceptually similar to what we are used to - separate lines of development history. A stream is a branch with added functionality that Helix Core uses to provide recommendations on best merging practices throughout the development process.

When a file is added for tracking, Helix Core classifies it using a **file type** label. The two most commonly used file types are text and binary. For binary files, the entire file content is stored each time the file is stored. This is a common VCS tactic for dealing with binary files which are not amenable to the normal merge process, since manual conflict resolution is usually not possible.

For text files, only the deltas (changes between revisions) are stored. Text file history and deltas are stored using the RCS (Revision Control System) format, which tracks each file in a corresponding `,v` file in the server depot. This is similar to CVS, which also leverages RCS file formats for preserving revision history. Files are often compressed using gzip when added to the depot and decompressed when synced back to the workspace.

Basic Commands Below is a list of the most common Perforce commands.

2.3.3 Distributed Version Control Systems

In a distributed version control system, all copies of the repository are created equal - there is no central copy of the repository. This design principle encourages commits, branches and merges to be created locally without network access and pushed to other repositories as needed.

Git

Git was created in 2005 by Linus Torvalds (also the creator of Linux) and is written primarily in C combined with some shell scripts. It is widely considered the best VCS tool due to its features, flexibility, and speed. Linus Torvalds originally wrote it for the Linux codebase and it has grown to become the most popular VCS in use today.

"You can do a lot of things with Git, and many of the rules of what you should do are not so much technical limitations but are about what works well when working together with other people. So Git is a very powerful set of tools, and that can not only be overwhelming at first, it also means that you can often do the same (or similar) things different ways, and they all 'work.'" - Linus Torvalds

Git repositories are commonly hosted on local servers as well as cloud services. Git forms the backbone of a broad set of DevOps tools available from popular service providers including GitHub, BitBucket, GitLab, and many others.

Architecture Git is a distributed VCS. This means that no copy of the repository needs to be designated as the centralized copy - all copies are created equal. This is in stark contrast to the second generation VCS which rely on a centralized copy for users to checkin and checkout from.

What this means is that developers and coding partners can share changes with each other directly before merging their changes into an official branch. This allows team collaboration to take on a flexible distributed workflow, if desired.

Furthermore, developers can commit their changes to their local copy of the repository without any other repositories knowing about it. This means that commits can be made without any network or Internet connection. Developers can work locally offline until they are ready to share their work with others. At that point, the changes can be pushed to other repositories for review, testing, or deployment.

When a file is added for tracking with Git, it is compressed using the `zlib` compression algorithm. The result is hashed using a SHA-1 hash function. This yields a unique hash value that corresponds specifically to the content in that file. Git stores this in an `object database` which is located in the hidden `.git/objects` folder.

The name of the file is the generated hash value, and the file contains the compressed content. These files are called Git blobs and are created each time a new file (or changed version of an existing file) are added to the repository.

"The object database is literally just a content-addressable collection of objects. All objects are named by their content, which is approximated by the SHA1 hash of the object itself. Objects may refer to other objects (by referencing their SHA1 hash), and so you can build up a hierarchy of objects." - Linus Torvalds

Git implements a `staging index` which acts as an intermediate area for changes that are getting ready to be committed. As new changes are staged for commit, their compressed contents are referenced in a special index file - which takes the form of a `tree` object. A `tree` is a Git object that connects blob objects to their real file names, file permissions and links to other trees, and in this way represents the state of a particular set of files and directories. Once all related changes are staged for commit, the index tree can be committed to the repository, which creates a `commit` object in the Git object database.

A commit references the head tree for a particular revision as well as the commit author, email address, date, and a descriptive commit message. Each commit also stores a reference to its parent commit(s) and so over time a history of project development is established.

As mentioned, all Git objects - blobs, trees, and commits - are compressed, hashed, and stored in the object database based on their hash value. These are called `loose objects`. At this point no diffs have been utilized to save space which makes Git very fast since the full content of each file revision is accessible as a loose object.

However, certain operations such as pushing commits to a remote repository, storing too many objects, or manually running Git's garbage collection command can cause Git to repackage the objects into `pack files`. In the packing process, reverse diffs are taken and compressed to eliminate redundant content and reduce size. This process results in `.pack` files containing the object content, each with a corresponding `.idx` (or index) file containing a reference of the packed objects and their locations in the pack file.

These pack files are transferred over the network when branches are pushed to or pulled from remote repositories. When pulling or fetching branches, the pack files are unpacked to create the loose objects in the object repository.

Basic Commands

- `git init` - Initialize a Git repository in the current directory (creates the hidden `.git` folder and its contents).
- `git clone <git-url>` - Download a copy of the Git repository at the specified URL.

- `git add <filename.ext>` - Add an untracked file or changed file to the staging area (creates corresponding entries in the object database).
- `git commit -m 'Commit message'` - Commit a set of changed files and folders along with a descriptive commit message.
- `git status` - Show information related to the state of the working directory, current branch, untracked files, modified files, etc.
- `git branch <new-branch>` - Create a new branch based on the current checked-out branch.
- `git checkout <branch>` - Checkout the specified branch into the working directory.
- `git merge <branch>` - Merge the specified branch into the current branch checked-out in the working directory.
- `git pull` - Update the working copy by merging in committed changes that exist in the remote repository but not the working copy.
- `git push` - Pack loose objects for local active branch commits into pack files and transfer to remote repository.
- `git log` - Show the commit history and associated descriptive messages for the active branch.
- `git stash` - Save all uncommitted changes in the working directory to a cache so that they can be retrieved later.

Mercurial

Mercurial was created in 2005 by Matt Mackall and it is written in Python. It was also started with the goal of hosting the codebase for Linux, but Git was chosen instead. It is the second most popular distributed VCS after Git, but is used far less often.

Architecture Like Git, Mercurial is a distributed version control system that allows any number of developers to work with their own copy of a project independently from others.

Mercurial leverages many of the same technologies as Git, such as compression and SHA-1 hashing, but does so in different ways.

When a new file is committed for tracking in Mercurial, a corresponding `revlog` file is created for it in the hidden directory `.hg/store/data/`. You can think of a `revlog` (or revision log) file as a modernized version of the `history files` used by the older VCS like CVS, RCS, and SCCS.

Unlike Git, which creates a new blob for every version of every staged file, Mercurial simply creates a new entry in the revlog for that file. To conserve space, each new entry only contains the delta (changes) from the previous version. Once a threshold number of deltas is reached, a full snapshot of the file is stored again.

This reduces the lookup time when applying many deltas to reconstruct a particular file revision.

These file revlogs are named to match the files that they track, but are postfixed with `.i` and `.d` extensions. The `.d` files contained the compressed delta content. The `.i` files are used as indexes to quickly track down different revisions inside the `.d` files. For small files with low numbers of revisions, both the indexes and content are stored in `.i` files. Revlog file entries are compressed for performance and hashed for identification. The hash values are referred to as `nodeids`.

Whenever a new commit is made, Mercurial tracks the all file revisions in that commit in something called the `manifest`. The manifest is also a revlog file - it stores entries that correspond to particular states of the repository.

However, instead of storing individual file content like the file revlogs, the manifest stores a list of filenames and nodeids that specify which file revision entries exist in each revision of the project. These manifest entries are also compressed and hashed. The hash values are again referred to as `nodeids`.

Lastly, Mercurial uses one more type of revlog called a `changelog`. The changelog contains a list of entries that associate each commit with the following information:

- Manifest nodeid - Identifies the full set of file revisions that exist at a particular time.
- Parent commit nodeid(s) - This allows Mercurial to establish a timeline or branch of project history. One or two parent ID's are stored depending on the type of commit (normal vs merge).
- Commit author
- Commit date

Each changelog entry also generates a hash known as its `nodeid`.

Basic Commands

- `hg init` - Initialize the current directory as a Mercurial repository (creates the hidden `.hg` folder and its contents).
- `hg clone <hg-url>` - Download a copy of the Mercurial repository at the specified URL.
- `hg add <filename.ext>` - Add a new file for revision tracking.
- `hg commit -m 'Commit message'` - Commit a set of changed files and folders along with a descriptive commit message.
- `hg status` - Show information related to the state of the working directory, untracked files, modified files, etc.
- `hg update <revision>` - Checkout the specified branch into the working directory.
- `hg merge <branch>` - Merge the specified branch into the current branch checked out in the working directory.
- `hg pull` - Download new revisions from remote repository but don't merge them into the working directory.
- `hg push` - Transfer new revisions to remote repository.
- `hg log` - Show the commit history and associated descriptive messages for the active branch.

BitKeeper

Monotone

Bazaar

Fossil

2.4 Summary of Key Features

2.4.1 Repositories

2.4.2 Commits

2.4.3 Branching and Merging

2.4.4 Pulling and Pushing

Chapter 3

Design

In this chapter, we examine various data structures and algorithms with key aspects that may make them suitable for incorporation into a `Version Control System`. We also assess what metrics are relevant to facilitate the comparison of these data structures and algorithms.

3.1 Data Structures

Data structures are objects that can be used to store, organize, and manipulate large amounts of data. They play a crucial role in computer science and are fundamental building blocks of many software systems, including `Version Control Systems`. The choice of data structure is critical to the overall performance and scalability of a `Version Control System`.

In order to evaluate the suitability of a data structure for implementation in a `Version Control System`, it is necessary to consider several core aspects. Firstly, `operational efficiency`, which refers to the time and space complexity of the basic operations performed by the data structure, is a crucial consideration. The efficiency of these operations can have a significant impact on the overall performance of the `Version Control System`.

Another important aspect is the data structure's `structural specificity`, which encompasses how data is stored and organized within the structure. This is critical because the structural specifics can affect the ease of implementation and the ability to efficiently perform operations such as `insertions`, `deletions`, and `updates`.

Lastly, the `implementation details` must be considered, including the ease of implementation and compatibility with the programming language. A data structure that is straightforward to implement and easy to maintain/iterate upon will result in a more streamlined and efficient `Version Control System`.

3.1.1 Linked List

A `Linked List` is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. The `Linked List` is the most sought-after data structure when it comes to handling dynamic data elements [3].

There are two main types of `Linked Lists`: `Singly-Linked Lists (SLL)` and `Doubly-Linked Lists (DLL)`, but we will only be considering the `Doubly-Linked List (DLL)` when we reach the `Implementation` chapter.

Singly-linked lists (SLL)

- `SLL` nodes contain two fields: `data` field and `next` pointer field.
- Traversal of a `SLL` can be done using the `next` pointer field only. Meaning, the `SLL` can be traversed in only one direction, from the first node to the last node.
- The `SLL` occupies less memory than a `DLL` because it does not contain a `prev` pointer field.
- `SLL` is preferred over `DLL` when it comes to the execution of stack and queue operations.
- `SLL` is also preferred over `DLL` to save memory when a searching operation is not required.

Table 3.1: Efficiency Analysis of Singly-Linked List Operations

Operation	Worst Case	Average Case
Access	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$
Insert (at Head)	$O(1)$	$O(1)$
Insert (at Current)	$O(1)$	$O(1)$
Insert (at Tail)	$O(1)$	$O(1)$
Delete	$O(n)$	$O(n)$
Delete (at Head)	$O(1)$	$O(1)$
Delete (at Current)	$O(n)$	$O(n)$
Delete (at Tail)	$O(n)$	$O(n)$

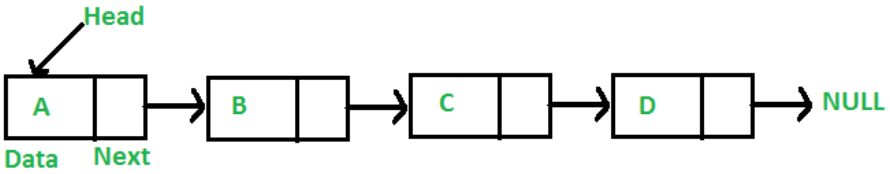

Figure 3.1: Singly Linked List (SLL) [4]

Doubly-linked lists (DLL)

- `DLL` nodes contain three fields: `data` field, `prev` pointer field and `next` pointer field.
- Traversal of a `DLL` can be done using the `next` pointer field or the `prev` pointer field. Meaning, the `DLL` can be traversed in both directions, from the first node to the last node and vice versa.
- The `DLL` occupies more memory than a `SLL` because it contains a `prev` pointer field.

Table 3.2: Efficiency Analysis of Doubly-Linked List Operations

Operation	Worst Case	Average Case
Access	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$
Insert (at Head)	$O(1)$	$O(1)$
Insert (at Current)	$O(1)$	$O(1)$
Insert (at Tail)	$O(1)$	$O(1)$
Delete	$O(n)$	$O(n)$
Delete (at Head)	$O(1)$	$O(1)$
Delete (at Current)	$O(1)$	$O(1)$
Delete (at Tail)	$O(1)$	$O(1)$

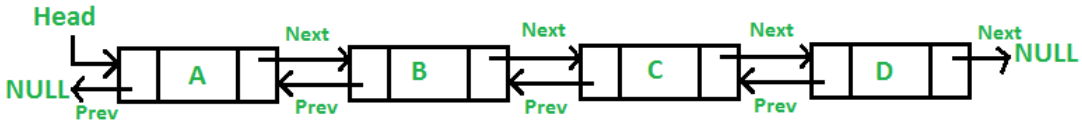

Figure 3.2: Doubly Linked List (DLL) [4]

Advantages

- Dynamic size: As new versions of a file are created, the `Linked List` can grow in size to accommodate the new data, without the need to pre-allocate memory.
- Efficient storage of file changes: Each node in the `Linked List` can store the entire contents of a file version, along with metadata such as the date and time the change was made. This allows for efficient storage of file changes over time.
- Easy traversal of file history: The `Linked List` structure allows for easy traversal of the file history, as each node contains a reference to the previous version of the file. This makes it easy to track changes and revert to previous versions of a file.
- Memory efficient: The `Linked List` structure is memory efficient, as each node only contains the current version of the file and changes made to it, along with simple references to the next and previous nodes in the list. This means that the `Linked List` structure does not need to store the entire file history in memory, which can be a significant amount of data.

Disadvantages

- Inefficient retrieval of specific file versions: Retrieving a specific version of a file can be slow, as the `Linked List` structure does not allow for random access to the file history. This means that the entire file history must be traversed from the most recent version of the file to the desired version.
- Limited scalability: For large file histories, the `Linked List` structure can be less efficient and will not scale as well as other data structures.
- Extra memory overhead: Each node in the `Linked List` structure contains a reference to the next and previous nodes in the list, which can add up when dealing with large file histories.

- Not suitable for concurrent access: The `Linked List` structure is not suitable for concurrent access, as it is not thread-safe and can lead to data corruption and race conditions.

Implementation Details Exercitation ullamco culpa velit excepteur aute esse amet. Quis adipisicing consequat quis sunt elit cupidatat sunt ipsum nostrud laborum aliqua veniam veniam commodo. Minim ullamco aute aliquip eiusmod officia cillum fugiat magna consectetur aute sunt aliqua labore. Reprehenderit dolore commodo deserunt laborum culpa laborum elit. Labore Lorem quis culpa amet adipisicing pariatur consequat eu proident in officia aute voluptate. Excepteur irure deserunt et ullamco deserunt labore anim dolor amet est est culpa. Magna eiusmod nulla ipsum esse anim nostrud mollit fugiat proident magna laboris.

Ut non aliquip aliquip Lorem reprehenderit nisi qui aliqua cupidatat enim adipisicing deserunt. Eiusmod est dolore ut ipsum Lorem et sunt est minim in Lorem. Reprehenderit ea proident officia anim dolore incididunt sunt labore sunt. Ea sunt incididunt anim tempor. Esse amet ut magna id irure ex.

Id do cillum ad ad officia dolor sunt deserunt amet ex. Pariatur aute sint anim aute id irure reprehenderit laboris non tempor id. Labore in ad sunt nulla dolore velit ut aliquip amet dolore quis voluptate nisi. Magna non magna nulla officia magna voluptate officia dolore Lorem ea sunt duis. Non dolore proident voluptate consequat consequat laborum aliqua veniam et occaecat amet ea dolore.

Summary A `Doubly-Linked List` is a data structure that consists of a sequence of nodes, each node having a data field and two pointers, one pointing to the next node in the list and the other pointing to the previous node.

One of the main advantages of using a `Doubly-Linked List` is that it allows for easy insertion and deletion of nodes, making it possible to add new file versions or remove outdated ones easily.

However, Concurrent access to a `Doubly-Linked List` can lead to data inconsistencies and race conditions, and proper synchronization must be implemented to prevent these issues.

3.1.2 Binary Tree

A `Binary Tree` is a hierarchical data structure that consists of a set of nodes, where each node can have at most two children. The topmost node in the tree is called the root node, and the nodes that do not have any children are called leaf nodes. The nodes that have children are called internal nodes. The `Binary Tree` structure is a special case of the `Tree` data structure, where each node can have at most two children.

Each node in a `Binary Tree` contains the following elements:

1. Data: The data stored in the node.
2. Left child: A pointer to the left child node.
3. Right child: A pointer to the right child node.

There are several different types of `Binary Tree` structures, including:

1. Full Binary Tree: A `Binary Tree` where each node has either zero or two children.
2. Complete Binary Tree: A `Binary Tree` where all levels of the tree are completely filled, except for the last level. The last level of the tree is filled from left to right.
3. Balanced Binary Tree: A `Binary Tree` where the difference between the height of the left and right subtrees of any node is not greater than one.
4. Degenerate (or Pathological) Binary Tree: A `Binary Tree` that is not balanced, where the height of the tree is equal to the number of nodes in the tree.

Table 3.3: Efficiency Analysis of Binary Tree Operations

Operation	Worst Case	Average Case
Search	$O(n)$	$O(\log n)$
Insert	$O(n)$	$O(\log n)$
Delete	$O(n)$	$O(\log n)$

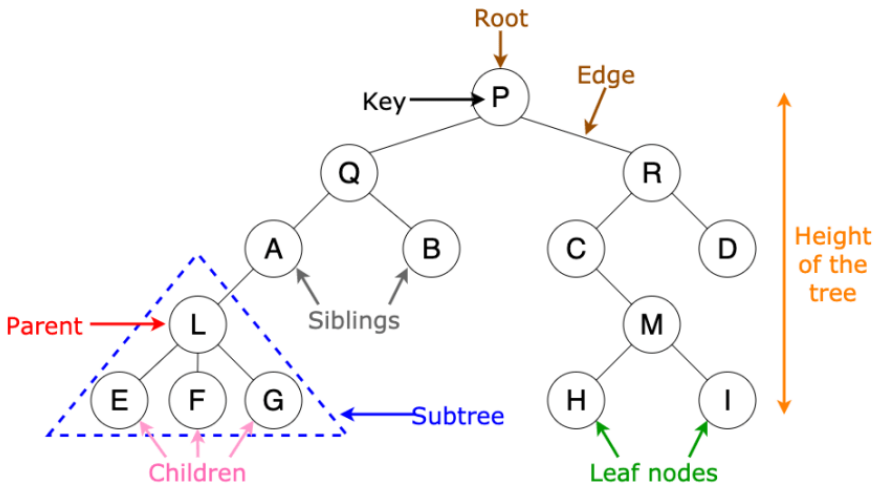

Figure 3.3: Binary Tree [5]

Advantages

- Efficient storage of file changes: Each node in the `Binary Tree` structure can store the entire contents of a file version, along with metadata such as the date and time the version was created. This allows for efficient storage of file changes, as the entire file history can be stored in a single `Binary Tree` structure.
- Easy traversal of file history: The `Binary Tree` structure allows for easy traversal of the file history, with the root node representing the most recent version of the file and the leaf nodes representing the oldest versions of the file. This makes it easy to track changes and revert to previous versions of the file.
- Flexibility: `Binary Tree` structures can be used to implement a variety of other data structures, such as `Binary Search Trees`, `AVL Trees`, `Heaps`, and others, which can be useful for other operations in a `Version Control System`.

Disadvantages

- Inefficient retrieval of specific file versions: The `Binary Tree` structure can be slow when retrieving specific file versions, as it requires traversing the `Binary Tree` to find the desired node. This can be inefficient when dealing with large file histories.
- Limited scalability: For large file histories, the `Binary Tree` structure can be less efficient and may not scale as well as other data structures.

- Extra memory overhead: Each node in the `Binary Tree` structure requires extra memory to store the pointers to the left and right child nodes. This can become significant when dealing with large file histories.
- Unbalanced trees can lead to poor performance: If the `Binary Tree` structure becomes skewed, the time complexity of searching, insertion, and deletion operations can become `O(n)`, where `n` is the number of nodes in the tree.
- Not suitable for concurrent access: The `Binary Tree` structure is not suitable for concurrent access, as it is not thread-safe. This can lead to data corruption and race conditions.

Implementation Details Nulla cillum laborum quis et cillum eu. Id exercitation ad aliquip ipsum elit excepteur tempor occaecat enim excepteur culpa aliqua ullamco pariatur. Et do elit nisi duis et aliquip consequat dolor labore.

Laborum consequat elit fugiat excepteur esse exercitation anim anim est eiusmod aliquip ad. Labore veniam cillum officia aute elit minim minim in laboris. Incididunt velit amet consequat officia nulla exercitation ex voluptate in duis ullamco Lorem.

Nostrud officia consectetur proident aliquip elit commodo do pariatur eu aliqua. Commodo irure deserunt tempor nisi cillum elit nulla dolore amet pariatur aliquip irure reprehenderit aute. Cillum minim cillum irure commodo cillum eu consequat et dolore sint sunt aliquip fugiat consequat. In laboris exercitation fugiat aute laboris mollit cupidatat laboris nostrud ut. Non enim aliquip anim ullamco non incididunt proident eu. Et officia tempor exercitation magna incididunt incididunt veniam reprehenderit.

Summary Magna fugiat consectetur magna adipisicing minim id elit Lorem culpa. Ut cillum adipisicing fugiat pariatur consectetur irure. Sit voluptate mollit nulla culpa incididunt minim velit non ex nostrud ad. Amet eiusmod magna voluptate nulla exercitation sit id. Nostrud in do id exercitation excepteur minim reprehenderit anim excepteur veniam eiusmod duis proident. Eiusmod aliquip laborum deserunt officia cillum Lorem excepteur Lorem ad.

3.1.3 Hash Table

A `Hash Table` is a data structure that uses a hash function to map keys to their corresponding values. It is an efficient way to implement an associative array, where keys are used to look up values.

The basic idea behind a `Hash Table` is to use a hash function to map a key to an index in an array, called a bucket, where the corresponding value can be found or stored. The process of mapping a key to an index is called `hashing`.

Each element in a `Hash Table` consists of:

1. Key: This is the value used to look up a corresponding element in the `Hash Table`.
2. Value: This is the value associated with the key that is stored in the `Hash Table`.

When a new element is added to a `Hash Table`, the key is passed through a `hash` function which produces an `index` (also called a hash value or bucket) where the element is stored. When a value is to be retrieved, the key is passed through the same hash function, and the resulting `index` is used to look up the corresponding value in the `Hash Table`.

Advantages

- Efficient searching, insertion, and deletion: A well-implemented `Hash Table` allows for these operations to be performed in `O(1)` time, which is useful for a Version Control System as it needs to be able to quickly retrieve, insert, and delete file versions.
- Dynamic resizing: `Hash Tables` can grow or shrink in size as needed, which is useful for a Version Control System as the number of file versions can vary greatly.
- Low overhead: A `Hash Table` only requires a small amount of overhead for pointers and the hash function, and thus uses less memory than an array or a `Linked List` with the same number of elements.

Disadvantages

- Hash collisions: `Hash` functions can produce collisions, where two different keys produce the same `index`, leading to the same location in the `Hash Table`.
  - `Collision resolution techniques`, such as `open addressing` and `separate chaining`, can be used to handle collisions but it still increases the time complexity of the `Hash Table` operations.
- Clustering: When all the elements in a `Hash Table` are stored in the same bucket, it is called `clustering`. This can lead to a performance degradation leading the `Hash Table` operations to have the worst-case time complexity of `O(n)` for insertion, deletion, and retrieval.

Implementation Details Non veniam nostrud occaecat magna minim officia fugiat ad tempor nulla sunt laboris deserunt minim. Officia commodo ut anim culpa dolore eiusmod ex do tempor tempor Lorem dolore aute culpa. Fugiat aliqua tempor minim esse consequat cupidatat sint dolor mollit irure consectetur fugiat. Ipsum excepteur eu cillum ea sit cillum sunt excepteur exercitation fugiat Lorem ut fugiat ut. Nulla aliquip cillum pariatur dolore aute pariatur do. Excepteur labore veniam eiusmod dolore dolor mollit eu.

Ea aliqua incididunt consectetur id officia. Dolor enim commodo pariatur enim veniam anim. Culpa laboris eu aute anim aliquip. Voluptate laboris et minim labore exercitation aliqua.

Quis voluptate duis dolore ea proident duis et ad. Proident sunt irure sint minim veniam ut laboris culpa laboris eiusmod id. Duis proident id nulla dolor est dolore. Mollit exercitation ad eiusmod ad eiusmod duis occaecat fugiat elit occaecat eu. Incididunt nisi velit quis mollit dolore cupidatat qui fugiat labore in. Est do qui enim in adipisicing eiusmod pariatur enim sit exercitation dolore officia.

Summary Minim non consequat culpa eu officia aliquip officia officia veniam in. Velit ut Lorem ea do adipisicing velit aliquip laborum. Proident dolore pariatur aliquip excepteur consectetur culpa adipisicing enim irure tempor eiusmod. Deserunt minim aliquip occaecat cillum mollit. In nulla fugiat consectetur nisi aliquip incididunt proident excepteur ex excepteur voluptate laboris. Reprehenderit amet anim incididunt aliqua est minim labore ipsum duis consectetur amet. Do mollit quis eiusmod ad consectetur ea nulla dolore voluptate nisi ea laborum est.

3.1.4 Directed Acyclic Graph (DAG)

A `Directed Acyclic Graph (DAG)` is a type of graph that consists of a set of nodes and directed edges between pairs of nodes. The edges have direction and they connect one node to another. Unlike in a `Tree` structure, in a `DAG`, a node can have multiple parents and multiple children, but there cannot be any cycles in the graph.

A node in a `DAG` can represent any type of data, and the edges can represent any type of relationship between the nodes. Each node in a `DAG` contains the following elements:

1. Data: The data that the node represents.
2. Adjacency list: A list of the nodes that are connected to the current node by an edge.

Advantages

- Representing complex relationships: `DAGs` can be useful for representing complex relationships between different versions of a file, such as `branching` and `merging` of changes.
- Representing multiple paths: `DAGs` can be used to represent multiple paths or multiple possibilities of how a file can change over time.
- Flexibility: `DAGs` are very flexible and can be used to represent any type of data and any type of relationship between the data.

Disadvantages

- Complex traversal: Traversing a `DAG` can be more complex than traversing a `Tree` because there can be multiple paths to traverse, which makes it more difficult to retrieve specific versions of a file.
- Limited scalability: `DAGs` may not scale well for large file histories and a large number of versions because they can become very complex and difficult to traverse.
- Not good for searching, insertion, and deletion: `DAGs` are not good for searching, insertion, and deletion because they are not ordered and they do not have a root node. This means that the time complexity of these operations is `O(n)` due to the need to traverse the entire graph.

Implementation Details Qui ea velit dolor cupidatat esse anim est labore. Reprehenderit veniam quis magna Lorem ad id exercitation mollit pariatur culpa officia. Dolor commodo aute eu quis Lorem ea nulla eu reprehenderit elit proident. Aliquip nostrud consequat laboris velit anim id dolor fugiat. Aute ea cupidatat velit sunt mollit id proident nostrud incididunt dolore incididunt aliquip adipisicing. Cillum et labore anim est incididunt laborum dolore commodo ullamco mollit ut ut ullamco.

Est veniam aliquip non mollit. Duis aliqua pariatur in officia cillum labore excepteur aliqua nisi. Tempor adipisicing commodo consequat occaecat ad mollit ad fugiat aliquip consectetur aliquip fugiat et. Sunt voluptate exercitation sunt esse.

Ex pariatur labore mollit labore amet. Occaecat culpa commodo ad cillum occaecat voluptate Lorem officia. Deserunt ea velit consequat sunt ex velit cillum do. Dolor nostrud in veniam laboris pariatur fugiat Lorem. Est quis sint anim deserunt elit fugiat culpa occaecat est in anim sunt ipsum.

Summary Minim ullamco duis officia fugiat anim ad in ipsum adipisicing nostrud mollit exercitation duis cupidatat. Esse occaecat enim anim aliqua qui fugiat mollit deserunt. Lorem culpa laborum ut veniam culpa in nulla et irure. Tempor nostrud magna qui sunt magna aliquip nisi qui. Magna excepteur pariatur mollit sint. Minim non officia aliquip qui laboris sunt. Exercitation aute adipisicing occaecat duis laboris.

3.2 Algorithms

3.2.1 Traversal

Algorithm 1

Algorithm 2

3.2.2 Hashing

Algorithm 1

Algorithm 2

3.2.3 Diffing

Algorithm 1

Algorithm 2

Chapter 4

Implementation

Chapter 5

Evaluation

Chapter 6

Conclusion

Bibliography

[1] J. Stopak, "Version control systems: A technical guide to vcs internals," Nov 2019.

[2] W. F. Tichy, "Rcs — a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[3] A. S. Ravikiran, "Linked list in a data structure: All you need to know," Oct 2022.

[4] R. Vaghani, "Difference between singly linked list and doubly linked list," Jan 2023.

[5] C. McMahon, "Understanding binary search trees," Jul 2020.