Version Control System

Motivation

The one thing that is constant in software is change. Changes in codedue to changes in understanding, requirements and many other reasons is a fact of life. Version control software provides a mechanism to manage such changes to a file – typically code – or set of files over time, so that you can recall specific versions later. That is VCS allows you torevert files to a previous state, revert the entire project top revious state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue andwhen, and more.

Goals of a version control system

There are three basic goals of a version control system (VCS):

- We want people to be able to work simultaneously, not serially.
- When people are working at the same time, we want their changes to not conflict with each other.
- We want to archive every version of everything that has ever existed ever. And who did it. And when. And why.

Benefits

Collaboration - A team of developers (which could be distributed geographically) may work on the same set of files without interrupting each other's work.

Change Management - Changes can be inspected, reviewed or rolled back as needed.

Tracking Evolution - Every change is associated with a short description, so you just need to pull a change log of some code to see how it evolved and who did what.

Branching - it is possible to create a copy (branch) of the entire project for some specific purpose; for example prototyping or developing a fix for an important customer - without disrupting development workflow.

Types of Version Control system

- Local
- Centralized
- Distributed

Local Version Control System Many people copy files into another directory (perhaps a time-stamped directory), to save a version. This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you are in and accidentally write to the wrong file or copy over files you do not mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

Centralized Version Control System

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it is far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they are working on. If the hard disk of the central database becomes corrupted, and proper backups have not been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem – whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control System

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients do not just check out the latest snapshot of the files, they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every checkout isreally a full backup of all the data.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that are not possible in centralized systems, such as hierarchical models.

Git

Git is a distributed version control system. Git originates from the Linux kernel development and is used by many popular Open Source projects, for example the Android and the Eclipse developer teams, as well as many commercial organizations. The core of Git was originally written by Lins Torvalds – in a mixture of C proggrams and shell scripts.

Advantages of Git

Free and open source - Git is released under GPL; it is freely available over the internet, you can download its source code and also perform changes according to your requirements. You can use Git to manage propriety projects without payment.

Fast and small - As most of the operations are performed locally, it gives huge benefit in terms of speed. Git does not rely on central server that is why for every operation there is no need to interact with remote server. Core part of the Git is written in C, which avoids runtime overhead associated with the other high level languages. Though Git mirrors entire repository, size of the data on the client side is small. This illustrates that how efficient it is at compressing and storing data on client side.

Implicit backup - The chances of losing data are very rare when there are multiple copies of it. Data present on any client side is mirror of the repository, hence it can be used in the event of crash or disk corruption.

Security - Git uses common cryptographic hash function called secure hash function (SHA1) to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. Meaning that, it is impossible to change file, date, commit message and any other data from Git database without knowing Git.

No need of powerful hardware - In case of CVCS, the central server needs to be powerful enough to serve request of the entire team. For smaller team, it is not an issue but as team size grows, the hardware limitation of the server can be a performance bottleneck. In case of DVCS, developers do not interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

Easier branching - CVCS uses cheap copy mechanism, means if we create new branch it will copy all code to the new branch, so it's time consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time consuming. But branch management with Git is very simple. It takes only few seconds to create, delete and merge branches.

Git States

Git has three main states that your files can reside in "committed, modified, and staged".

Committed - means that the data is safely stored in your local database.

Modified - means that you have changed the file but have not committed it to your database yet.

Staged - means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project "the Git directory, the working directory, and the staging area".

Git directory - is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

Working directory - is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

Staging area - is a file, generally contained in your Git directory, that stores information about what will go into your next commit. It's sometimes referred to as the "index", but it is also common to refer to it as the staging area.

The basic Git workflow is as follows:

You modify files in your working directory.

You stage the files, adding snapshots of them to your staging area.

You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Creating a local project

You create a project by using the "git init", before that you create a specific directory when you want to create the project.

```
mkdir <dir_name>
cd <dir_name>
git init <project_name>
```

Git init - An Empty git repository is initialized in the SampleRepo directory.

Adding file to Project

Let us add a file to the project you just created. Create a file named "TSDaily-Log.rst" which consists of your daily log of TS module in which you write your daily updates. To add a file to project follow below steps::

```
cd wise-ejd
vim TSDailyLog.rst // Write some text in the file
git add TSDailyLog.rst
```

Easier branching - CVCS uses cheap copy mechanism, means if we create new branch it will copy all code to the new branch, so it's time consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time consuming. But branch management with Git is very simple. It takes only few seconds to create, delete and merge branches. TSDailyLog.rst file is added to your project wise-ejd.

Checking the status

After adding the file to the project, we can check the status of project by using the "git status" command. After adding the file in the project, we can check the status of project by using the "git status" command.

```
git status
```

It displays the message "changes to be committed", "new file: TSDaily-Log.rst", as we have not commited the changes you have done in the project.

Configuring the user details

Git comes with a tool called **git config** that lets you get and set configuration variables that control all aspects of how Git looks and operates. To add the user details to project like username and email Id, follow the below steps.

```
git config user.name "Your full name"
git config user.email "Yourid@yourdomain.com"
```

Commiting the changes

Now you can commit the added file to the project.

```
git commit -m <The Commit Message>
```

Once commited it displays the number of files added, number of lines inserted and number of lines deleted.

Maintaining different versions of file

If you want to do some changes in the existing file named "TSDailyLog.rst", we can open the file, do the required changes and add it back to the project and commit it. git maintains different versions of file for each commit. Follow the below steps to change the file "TSDailyLog.rst".

```
vim TSDailyLog.rst
```

Now, add the todays log in the file

```
git add TSDailyLog.rst
git commit -m "Added todays log"
```

Git maintains two versions of TSDailyLog.rst

Viewing the Commit History

After you have created several commits, you will probably want to look back to see what changes performed in the project. The most basic and powerful tool to do this is the "git log" command.

```
git log
```

It shows a list of commits including the details like author's name and e-mail, the date written, and the commit message.

Display difference between versions

The "git log" will display all the changes performed in the project, if you want to see the changes between two specific commits use "git diff" command and pass the commit messages for which you want to see the difference.

You can use "get log –abbrev-commit" command, which displays partial prefix commit id instead of showing the full 40-byte hexadecimal commit id.

```
>>>>>>> b2056cff7fb49b5c0f40d8b3925c21829867393a
git diff c0311d2 7125dca
```

Working With A Remote Repository

A remote repository is generally a bare repository - a Git repository that has no working directory. Because the repository is only used as a collaboration point, there is no reason to have a snapshot checked out on disk, it's just the Git data. In the simplest terms, a bare repository is the contents of developer project's ".git" directory and nothing else.

Remote repositories are mirrored within the local repository. It is possible to work with multiple remote repositories. Each remote repository is identified with a local alias. When working with a unique remote repository, it is usually named “origin”.

Adding a remote repository

To add a new remote repository, use the ”git remote add” command, in the directory your repository is stored at. The ”git remote add” command takes two arguments

name - is a local alias, identifying the remote repository, After adding a remote, you will be able to use “name” as a convenient shortcut for ”url” in other Git commands.

url - is the location of the remote repository

```
git remote add origin https://gitlab.com/priyanka88/my_learnings.git
```

Pushing to remote repository

Pushing is how you transfer commits from your local repository to a remote repository. The most common use case for “git push” is to publish your local changes to a central repository. After you have accumulated several local commits and are ready to share them with the rest of the team, then push them to the central repository.

The “git push” command takes two arguments:

- A remote name, for example, “origin”
- A branch name, for example, “master”

Cloning A Repository

Clone When working on a project with others, you will need to clone the global git repository. This command will copy the current state of the git repository from the server to your workstation. The repository is completely self-contained and can be copied from one directory to another and from one computer to another. As mentioned above, it contains the entire development history of the project.

To get this copy of the global git repository onto your computer, do the following:

```
git clone <<repository.git>>
```


There are several different protocals that you can use to connect to a git repository (e.g., http, ftp, ssh, rsync). For example, to clone the my_learnings repository, you could do:

```
mkdir GitRepo
cd GitRepo
git clone https://gitlab.com/priyanka88/my_learnings.git
```

Now you have a copy of the git repository on your computer. Now what? Well, if you are a developer, you may want to actually edit files in the repository and then share the changes you have made with the rest of the team. As this will represent the majority of your interactions with git, it seems like a good place to start.

Add and Commit

Let us edit a file named "TSDailyLog.rst" which consist of your daily log of TS module.

```
vim TSDailyLog.rst
```

After having edited files, you are ready to create a commit. A commit is a logical change; it can be as short as one character or as long as you want. More than that, it helps everyone working on the project better understand exactly what was changed without being obliged to look at the code.

"git add" adds an edited file to the staging area, preparing it to be commited.

Note: if you modify the file after doing "git add", you will need to add it again in order for those changes to be included in the commit.

"git commit" creates the logical commit on your current branch (master for now). master moves forward by one node, leaving origin / master behind. Your local repository is now ahead of the global repository.

Note: A commit is denoted by an SHA.

Push

Now you have updated your repository, committed some changes and are ready to share those changes with everyone else. Before you do so, update your repository again! Do this just to make sure that in the period between when you last updated your repository and when you want to push your changes, someone else has not introduced changes to the repository.

```
git push
```


Pull

The “git pull” fetch from and integrate with another repository or a local branch. It incorporates changes from a remote repository into the current branch.

```
git pull
```

