**1)Git, Distributed Version Control System Introduction**

Git-Distributed Version Control System

VCS-Version Control System

GIT Software is developed by Linus Torvalds, who developed Linux. He created the GIT for the development of the Linux Kernel for the contribution with other kernel developers

VCS is a software that is designed to record changes made to the file over time.

Git gives us the ability to revert the files or the set of files you made changes, so we can get the previous file at that specific point of time and the changes that were made to it.

Git not only records the source code but also tracks images, research papers or any types of files.

There are three kinds of version control systems

1)Local Version Control System

2)Centralized Version Control System

3)Distributed Version Control System

**Local Version Control System**

* Many People use this method.
* This method is to copy files into another directory.
* They keep the files with the timestamp directory.
* It is very simple that we maintain the backup

It is like we have a set of files in one folder. Now we want to make changes to the files, so we copy the files into another folder and name it with the date in the name.

Only one person can use this system.

**Centralized Version Control System**

* To collaborate with other developers this method is developed.
* These systems have a single server that has all versioned files, and several clients can take the files from that central place.
* For many years this has been the standard for version control.
* CVS, Subversion and Perforce are the examples of this method

In a CVCS each developer fetches the latest version of the code from server and the developers uploads latest version on the top of previous version.

**Advantages Centralized Version Control System**

* This has advantages over local VCS.
* Everyone can know what is going on in the project.
* Administrators can have control that who can do what.
* Everyone can get the project update immediately as all are linked to the one centralized server.

**Disadvantages Centralized Version Control System**

* If the Centralized server goes down for an hour, nobody can collaborate at all or save the changes to the centralized server.
* If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything.
* Local VCSs 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**

* Clients not only check out the snapshot of the files, but they can also fully mirror the repository including its full history.
* In a client’s system also, you can maintain the versions so that when the server is up you can fully copy back to the server.
* Every Clone is a full backup of all the data

In a centralized system we take only the latest version, they take the entire repository and the older versions too. The versions are stored in client’s system too so when server is down and gets back up everything will be uploaded to server.

**2)Working, Staging and Repository Area in GIT Project**

**How Git Works**

Compared to other VCS like subversion. Most operations in GIT require local files and local resources to operate.

If you're used to a CVCS where most operations have that network latency overhead, (means we use network to connect to files)

So, in Git, unlike CVCS we do not deal with server much but only work with files on our local pc so we will not face issues with network latency.

* For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you.
* It simply reads it directly from your local database.
* If you want to see the changes between the current working file and the file a month ago.
* Git can look up the file a month ago and do a local difference calculation.
* There is no need to ask a remote server to do it or pull an older version of the file from the remote server to do it locally.
* This means you can do your work happily until you get to a network connection go upload.
* in Subversion and CVS, you can edit files, but you can't commit changes to your database (because your database is offline).

**Three States in GIT**

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

* 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. (It means this modified file should go to commit in the next version)
* Committed means that the data is safely stored in your local database.

**Working Directory**

The working tree (project we are working on) 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**

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git is the "index", but the phrase "staging area" works just as well.

**Repository Area**

The 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.

**GIT Workflow**

The basic Git workflow goes something like this:

* You modify files in your working tree.
* You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the 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.

**3)Installation of Git**

<https://git-scm.com/download/win> (Windows)

**There are a lot of different ways to use Git.**

* Using Command Line Tools (Mostly used)
* Using Graphical User Interfaces

**Check GIT Version**

After installation we can check the version of GIT using command:

git --version

**Git Bash**

Go to Gitbash, it is an app from git a command prompt exclusively for git.

**4)Git Configuration Levels Explore Local, User and System Level Config Files**

**Git Configuration**

* After Installing GIT in our system.
* The first thing we need to do is set our username and email address.
* This is because when we use git to change the files in the project. Git uses this information to identify who has made the changes to the file.
* The command to add the username and address is Git config

Git uses a series of configuration files to determine the behavior.

GIT has multiple levels of configuration (3 levels)

* Repository/Project Level (Local) Priority1
* User Account (Global Level) Priority2
* System Level (Git Installation) Priority3

**Git Config Locations**

**Local (Repository/Project Level)**

repository/.git/config

**Global (User Level):**

Users/username/.gitconfig

**System (Git Installation)**

/usr/local/etc/gitconfig

If we want to know the location of where the above configurations are located

Check Complete Git Config

Git config --list --show-origin

Remove a specific setting for a specific level of config

Git config --global --unset user.name

Remove the specific section

Git config --global --remove-section user

**5)Add Git Username and user email in configuration file using Git Config**

mkdir gitprojects creates a folder named gitprojects in the directory

cd.. root directory or one level up

cd filepath to go to that folder

DIR list of files

Type nul filename.extension creates that file with no data

git init to add git into this project(empty git repository is added to folder)

The folder created is a hidden folder

There will be a config file in the hidden folder

It has no username and email id

git config user.email to get the email in the file

git config -- local user.email [yakalaanirudh@gmail.com](mailto:yakalaanirudh@gmail.com) to add email

git config user.name anirudh to add name globally

git config --local user.name anirudh to add name locally

git config user.name to get the name in the file

git config --unset user.email

git config --local --remove-section user username and email removed

We can add these changes to the config file in the project manually too.

**6)Understand Git Help and status command details**

git help

gives a list of the most common used commands and its documentation

git help -a

gives a list of all commands and its documentation

commands are divided into two categories

porcelain commands -mostly used

plumbing commands -rarely used

git help add

documentation for this specific command

git status it tells present status of the project

tells us like in which branch we are like master branch, number of commits and what committed

**7)Add and Commit File into Git Staging Area and Repository Area**

* Let’s create a file named index.html in the working directory.
* The file is not committed and when we type status there will be an untracked file to track the file, we do git add index.html.
* The add command moves the file from the working directory to staging area and the file will be a tracked one.
* If we want to remove the file from the staging area back to the working directory, we need to do git rm --cached.
* To commit the file from the staging area to the repository we do git commit.
* We can add a message to the commit like initial commit.
* If we are making edits in the file from the cmd we can exit the file by :wq means write and quit.
* Now lets say we make changes to the index.html file from the cmd we can do it by vi index.html
* We can also add a message while committing by using git add index.html(to move to staging and then) git commit -m “type message here”.
* It tells us all the commits we have done to the repository. git log. The data in the log is (username, email id, time and date, and the commit message).

**8)GIT Diff command**

**Git Diff command**

Diff command is used in git to track the difference between the changes made on a file.

Diff command takes two inputs and reflects the differences between them.

git diff -Between Working Area and Staging Area

git diff--staged -Between Staging Area and Repository (Committed)

git diff head -Between Working Area and Repository

**Let's analyze a simple example**

The format is basically the same as the diff -u unified diff.

We start with numbers from 1 to 16 and remove 2, 3, 14 and 15:

diff -u <(seq 16) <(seq 16 | grep -Ev '^(2|3|14|15)$')

Output:

@@ -1,6 +1,4 @@

1

-2

-3

4

5

6

@@ -11,6 +9,4 @@

11

12

13

-14

-15

16

@@ -1,6 +1,4 @@ means:

* -1,6 means that this piece of the first file starts at line 1 and shows a total of 6 lines. Therefore, it shows lines 1 to 6.
* 1
* 2
* 3
* 4
* 5
* 6

- means "old", as we usually invoke it as diff -u old new.

* +1,4 means that this piece of the second file starts at line 1 and shows a total of 4 lines. Therefore it shows lines 1 to 4.

+ means "new".

We only have 4 lines instead of 6 because 2 lines were removed! The new hunk is just:

1

4

5

6

@@ -11,6 +9,4 @@ for the second hunk is analogous:

* on the old file, we have 6 lines, starting at line 11 of the old file:
* 11
* 12
* 13
* 14
* 15
* 16
* on the new file, we have 4 lines, starting at line 9 of the new file:
* 11
* 12
* 13
* 16

Note that line 11 is the 9th line of the new file because we have already removed 2 lines on the previous hunk: 2 and 3.

**9)How Git Stores the Data (SHA1)**

**How GIT stores the data**

* Git stores the data in the form of keys and values
* Values are nothing but the contents of the file.
* You give the value, and it will calculate a key for it, that is nothing but hash.
* Git calculator the hashes with SHA1 algorithm.
* Every piece of content has its own hash.
* SHA1 is 20 bytes in hexadecimal format.
* Not only the content of the files, the directories and so on commits has their own SHA-1.
* Every object in GIT has its own SHA1.

If we create a file in a sample project and write some lines in the file and commit it. Before committing in the git folder in objects subfolders only two folders will be present, but after committing new folders will be created.

When we git log

A hash code will be visible 19351a94ac85c402 882a81c38ce5a088

In objects subfolder in git folder there will be a folder named 30

When we do git cat-file 19351a94ac85c402 882a81c38ce5a088 -p

We get tree c8035d85bc6f7bbfd8f95797122fe746cdc8798f

In objects subfolder in git folder there will be a folder named c8

When we do git cat-file c8035d85bc6f7bbfd8f95797122fe746cdc8798f -p

We get blob 305df835cd143f25affdb27e653e3124d838ef25.

In objects subfolder in git folder there will be a folder named 30

Blob means it is the end.

When we do git cat-file 305df835cd143f25affdb27e653e3124d838ef25-p

We will see the data that has been committed. (That is the data we edited)

**10)Rename and Restore files in Git Repository using mv and the restore commands**

Let’s say we have a file channel.txt.

We renamed the file to channel.html by mv channel.txt channel.html.

Now if check git status, we see that channel.txt is deleted and channel.html is added.

Now if we add .html file to staging area by git add channel.html , and when we do git status we will see that html file is staged and .txt file is deleted in working area.

So, we add txt file by git add channel.txt, and when we do git status we will see that html file is staged and .txt file is not logged.

Now commit using git commit -m” Renamed File”.

But in git itself we have git mv command

git mv channel.html channel.txt

git status

Now we can see that the renamed channel.txt is in the staging area by git automatically.

Now commit git commit -m” Rename 2”

**Restore command-**

**To move back files from staging to working directory and undo changes in working directory.**

Let’s say there is a file named res.txt.

We modified its content (123 to 1234567) and added it to staging area by git add res.txt.

If we do git restore --staged res.txt.

The file will be moved back from the staging area to working directory

If we do git restore res.txt. again,

The changes will be reverted (1234567 to 123)