## Version Control System : is a software that helps software developers to work together and maintain a complete history of their work. A version control system (VCS) allows you to track the history of a collection of files. It supports creating different versions of this collection. Each version captures a snapshot of the files at a certain point in time and the VCS allows you to switch between these versions. These versions are stored in a specific place, typically called a repository.

* + **Functions of a VCS:**
    1. Allows developers to work simultaneously.
    2. Does not allow overwriting each other’s changes.
    3. Maintains a history of every version.
  + **Types of VCS:**
    1. **Centralized version control system (CVCS).**

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then we will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

* + 1. **Distributed/Decentralized version control system (DVCS).**

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

* **Git:** is a distributed revision control and source code management system with an emphasis on speed. It is a version control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source code management in software development, but it can be used to keep track of changes in any set of files. As a distributed revision control system it is aimed at speed, data integrity, and support for distributed, non-linear workflows.

Git was initially designed and developed by Linus Torvalds in 2005 for Linux kernel development. Git is a free software distributed under the terms of the GNU General Public License version 2. Git is currently the most popular implementation of a distributed version control system e.g., the Android or the Eclipse developer teams, as well as many commercial organizations. The core of Git was originally written in the programming language \_C, but Git has also been re-implemented in other languages, e.g., Java, Ruby and Python.

As with most other distributed version control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version tracking abilities, independent of network access or a central server.

A Git repository contains the history of a collection of files starting from a certain directory. The process of copying an existing Git repository via the Git tooling is called \_cloning. After cloning a repository the user has the complete repository with its history on his local machine. Of course, Git also supports the creation of new repositories.

If you want to delete a Git repository, you can simply delete the folder which contains the repository.

If you clone a Git repository, by default, Git assumes that you want to work in this repository as a user. Git also supports the creation of repositories targeting the usage on a server.

* + Bare repositories are supposed to be used on a server for sharing changes coming from different developers. Such repositories do not allow the user to modify locally files and to create new versions for the repository based on these modifications.
  + Non-bare repositories target the user. They allow you to create new changes through modification of files and to create new versions in the repository. This is the default type which is created if you do not specify any parameter during the clone operation. A local non-bare Git repository is typically called local repository.

## Advantages of Git

### Free and open source

Git is released under GPL’s open source license. It is available freely over the internet. You can use Git to manage property projects without paying a single penny. As it is an open source, you can download its source code and also perform changes according to your requirements.

### Fast and small

As most of the operations are performed locally, it gives a huge benefit in terms of speed. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the 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 mirrors the repository, hence it can be used in the event of a crash or disk corruption.

### Security

Git uses a 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. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.

### No need of powerful hardware

In case of CVCS, the central server needs to be powerful enough to serve requests of the entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware limitations of the server can be a performance bottleneck. In case of DVCS, developers don’t 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, If we create a new branch, it will copy all the codes to the new branch, so it is 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 a few seconds to create, delete, and merge branches.

### Working offline

Some version control applications popular in the past like CVS and SubVersion used the concept of central repository accessible somewhere in the network where all the changes needed to be commited.

Git uses the concept of local repository where you have a copy of the “complete repository” of your project. You can commit the changes you make to the files of your project to the local repository. So Git allows you to work completely offline, i.e. even when you do not have access to a remote repository. Later you can synchronize or sharing the changes you made when you have online access to the remote repository.

### Fast to Work With

Most of the Git operations are fast, mainly because they are performed on your local repository copy.

### Repositories Are Smaller

A typical [Git repository is smaller than for instance one using SubVersion](https://git.wiki.kernel.org/index.php/GitSvnComparison#Smaller_Space_Requirements). If you want to compare, you can try this [test page](http://vcs.atspace.co.uk/2012/11/05/which-repository-is-more-compact-git-or-svn/). You can read more about this learning [how Git packs information](http://git-scm.com/book/en/v2/Git-Internals-Packfiles).

### Moving or Adding files

If you want to move a file inside your repository Git automatically track the moves. This was not possible in old version control applications like CVS. Moving a file would typically require to create a new file and remove the old one, thus losing the changes history.

Also if you want to add only certain files with some extension with Git you can use wildcards. For example to add only .php files you can run:

git add '\*.php'

### Ignore Certain Files

Sometimes you have files being generated by your project, like for instance log files or files generated by your IDE, that you do not want to store in project repository because they are not really part of your source code. You can tell Git to ignore certain files in the local repository directories using a file named .gitignore. The files listed in .gitignore are excluded from the version control process as if they are not there. You can share those rules committing the file or just keep it locally.

### Branches

Sometimes you need to work on new experimental features without interfering with the main code of your project. You can achieve this by creating new branches to try the code of those experimental features. Branches also allow different developers to work on different features without interfering with each other work. Then when the features are ready, they can merge the branch changes in the main branch.

### Check the Status of Your Changes

Check the status of the changes you made to your repository is pretty straightforward. The **git status** command lets you see what would happen if you committed your changes at a given moment. It can help to avoid the mistakes of using different externals branches of your project.

### Stash Branches

If you are working on a branch of your project but you do not want to commit the changes, you can save the current status of that branch to return to it in the future. You can switch your work to another branch and insert the stashed modification in it. With **git stash** command, you can save the current status. With **git stash pop** command you will apply the stashed modifications.

### Cherry Pick Changes from Branches

Git allows to pick one commit from some branch and apply it into the current branch. This operation is helpful for testing purposes. Imagine you want to test some temporary modification or pick some commits done in other branches.

### Find version that Introduced a bug using Binary Search

If you have an issue in your code and you want to know when it was introduced and what it is, with the **git bisect** command you could go back to every commit till you will find the bad one one which the issue was introduced.

## DVCS Terminologies

### Local Repository

Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

1. **Repository**

A *repository* contains the history, the different versions over time and all different branches and tags. In Git each copy of the repository is a complete repository. If the repository is not a bare repository, it allows you to checkout revisions into your working tree and to capture changes by creating new commits. Bare repositories are only changed by transporting changes from other repositories.

This description uses the term *repository* to talk about a non-bare repository. If it talks about a bare repository, this is explicitly mentioned.

### Working Directory and Staging Area or Index

The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

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 tree, and the staging area.



Figure . Working tree, staging area, and Git directory.

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.

The working tree 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.

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 parlance is the “index”, but the phrase “staging area” works just as well. Or The *staging area* is the place to store changes in the working tree before the commit. The *staging area* contains a snapshot of the changes in the working tree (changed or new files) relevant to create the next commit and stores their mode (file type, executable bit).

**Basic workflow of Git**

**Step 1** : You modify a file from the working directory.

**Step 2** : You add these files to the staging area.

**Step 3** : You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository.



Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

First commit

$ git add sort.c

adds file to the staging area

$ git commit –m “Added sort operation”

Second commit

$ git add search.c

Adds file to the staging area

$ git commit –m “Added search operation”

*Index* is an alternative term for the *staging area*.

### Blobs

Blob stands for **B**inary **L**arge **Ob**ject. Each version of a file is represented by blob. A blob holds the file data but doesn’t contain any metadata about the file. It is a binary file, and in Git database, it is named as SHA1 hash of that file. In Git, files are not addressed by names. Everything is content-addressed.

### Trees

Tree is an object, which represents a directory. It holds blobs as well as other sub-directories. A tree is a binary file that stores references to blobs and trees which are also named as **SHA1** hash of the tree object. The *working tree* contains the set of working files for the repository. You can modify the content and commit the changes as new commits to the repository.

### Commits

When you commit your changes into a repository this creates a new *commit object* in the Git repository. This *commit object*uniquely identifies a new revision of the content of the repository.

This revision can be retrieved later, for example, if you want to see the source code of an older version. Each commit object contains the author and the committer. This makes it possible to identify who did the change. The author and committer might be different people. The author did the change and the committer applied the change to the Git repository. This is common for contributions to open source projects.

Commit holds the current state of the repository. A commit is also named by **SHA1** hash. You can consider a commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.

### Branches

A *branch* is a named pointer to a commit. Selecting a branch in Git terminology is called *to checkout a branch. If you are working in a certain branch, the creation of a new commit advances this pointer to the newly created commit.*

*Each commit knows their parents (predecessors). Successors are retrieved by traversing the commit graph starting from branches or other refs, symbolic references (for example: HEAD) or explicit commit objects. This way a branch defines its own line of descendants in the overall version graph formed by all commits in the repository.*

*You can create a new branch from an existing one and change the code independently from other branches. One of the branches is the default (typically named \_master* ). The default branch is the one for which a local branch is automatically created when cloning the repository.

Branches are used to create another line of development. By default, Git has a master branch, which is same as trunk in Subversion. Usually, a branch is created to work on a new feature. Once the feature is completed, it is merged back with the master branch and we delete the branch. Every branch is referenced by HEAD, which points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit.

Git supports branching which means that you can work on different versions of your collection of files. A branch allows the user to switch between these versions so that he can work on different changes independently from each other.

For example, if you want to develop a new feature, you can create a branch and make the changes in this branch. This does not affect the state of your files in other branches. For example, you can work independently on a branch called production for bugfixes and on another branch called feature\_123 for implementing a new feature.

Branches in Git are local to the repository. A branch created in a local repository does not need to have a counterpart in a remote repository. Local branches can be compared with other local branches and with \_remote-tracking branches. A remote-tracking branch proxies the state of a branch in another remote repository.

Git supports the combination of changes from different branches. The developer can use Git commands to combine the changes at a later point in time.

### Tags

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is that tags are immutable. It means, tag is a branch, which nobody intends to modify. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, developers create tags for product releases.

A *tag* points to a commit which uniquely identifies a version of the Git repository. With a tag, you can have a named point to which you can always revert to. You can revert to any point in a Git repository, but tags make it easier. The benefit of tags is to mark the repository for a specific reason, e.g., with a release.

Branches and tags are named pointers, the difference is that branches move when a new commit is created while tags always point to the same commit. Tags can have a timestamp and a message associated with them.

### Clone

Clone operation creates the instance of the repository. Clone operation not only checks out the working copy, but it also mirrors the complete repository. Users can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.

### Pull

Pull operation copies the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances. This is same as the update operation in Subversion.

### Push

Push operation copies changes from a local repository instance to a remote one. This is used to store the changes permanently into the Git repository. This is same as the commit operation in Subversion. The Push operation stores data permanently to the Git repository. After a successful push operation, other developers can see

### HEAD

HEAD is a pointer, which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit. The heads of the branches are stored in **.git/refs/heads/** directory. Or *HEAD* is a symbolic reference most often pointing to the currently checked out branch.

Sometimes the *HEAD* points directly to a commit object, this is called *detached HEAD mode*. In that state creation of a commit will not move any branch.

If you switch branches, the *HEAD* pointer points to the branch pointer which in turn points to a commit. If you checkout a specific commit, the *HEAD* points to this commit directly.

### Revision

Revision represents the version of the source code. Revisions in Git are represented by commits. These commits are identified by **SHA1** secure hashes.

### URL

URL represents the location of the Git repository. Git URL is stored in config file. A URL in Git determines the location of the repository. Git distinguishes between *fetchurl* for getting new data from other repositories and *pushurl* for pushing data to another repository.

* **Basic commands in git**

1. **Push**

Sends local commits to the remote repository. git push requires two parameters: the remote repository and the branch that the push is for. The Push operation stores data permanently to the Git repository. After a successful push operation, other developers can see changes. Pushes all the modified local objects to the remote repository and advances its branches.

$ git push <remote\_URL/remote\_name> <branch>

Push all local branches to remote repository

$ git push —all

git push origin master

1. **Pull**

To get the latest version of a repository run git pull. This pulls the changes from the remote repository to the local computer. git pull, is used with a different goal in mind: to update your current HEAD branch with the latest changes from the remote server. This means that pull not only downloads new data; it also directly integrates it into your current working copy files. This has a couple of consequences. Fetches the files from the remote repository and merges it with your local one.

$ git pull <branch\_name> <remote\_URL/remote\_name>

Since "git pull" tries to merge remote changes with your local ones, a so-called "merge conflict" can occur. Check out our in-depth tutorial on [How to deal with merge conflicts](https://www.git-tower.com/learn/git/ebook/en/command-line/advanced-topics/merge-conflicts) for more information.

Like for many other actions, it's highly recommended to start a "git pull" only with a clean working copy. This means that you should not have any uncommitted local changes before you pull. Use Git's Stash feature to [save your local changes temporarily](https://www.git-tower.com/learn/git/ebook/en/command-line/branching-merging/stashing).

git pull origin

1. **Commit**

Commit changes to head (but not yet to the remote repository): Takes all of the changes written in the index, creates a new commit object pointing to it and sets the branch to point to that new commit. Commit any files you've added with git add, and also commit any files you've changed since then

git commit -m "Commit message"

git commit -a

git commit -m ‘committing added changes’

git commit -a -m ‘committing all changes, equals to git add and git commit’Examples:

1. **Rollback (undoing changes)**

Using the git revert Command to Rollback a Commit

git revert

1. **Status**

This command returns the current state of the repository. git status will return the current working branch. If a file is in the staging area, but not committed, it shows with git status. Or, if there are no changes it’ll return nothing to commit, working directory clean. Shows you the status of files in the index versus the working directory. It will list out files that are untracked (only in your working directory), modified (tracked but not yet updated in your index), and staged (added to your index and ready for committing).

git status

# On branch master #

# Initial commit #

# Untracked files: #

# (use "git add <file>..." to include in what will be committed) #

1. **Init**

This command turns a directory into an empty Git repository. This is the first step in creating a repository. After running git init, adding and committing files/directories is possible. Initializes a git repository – creates the initial *.git* directory in a new or in an existing project.

$ git init

1. **git add**

Adds files in the to the staging area for Git. Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area). There are a few different ways to use git add, by adding entire directories, specific files, or all unstaged files. Adds files changes in your working directory to your index. Example:

$ git add <file or directory name>

1. **branch**

To determine what branch the local repository is on, add a new branch, or delete a branch. Branch operation allows creating another line of development. We can use this operation to fork off the development process into two different directions. For example, we released a product for 6.0 version and we might want to create a branch so that the development of 7.0 features can be kept separate from 6.0 bug fixes.

|  |  |
| --- | --- |
| Create a new branch and switch to it: | git checkout -b <branchname> |
| Switch from one branch to another: | git checkout <branchname> |
| List all the branches in your repo, and also tell you what branch you're currently in: | git branch |
| Delete the feature branch: | git branch -d <branchname> |
| Push the branch to your remote repository, so others can use it: | git push origin <branchname> |
| Push all branches to your remote repository: | git push --all origin |
| Delete a branch on your remote repository: | git push origin :<branchname> |

1. **log**

The git log command to view the commit details.

1. **git config**

Sets configuration values for your user name, email, gpg key, preferred diff algorithm, file formats and more. Examples:

git config --global user.name "My Name"

git config --global user.email [user@domain.com](mailto:user@domain.com)

1. **git clone**

Creates a GIT repository copy from a remote source. Also adds the original location as a remote so you can fetch from it again and push to it if you have permissions. Example:

git clone [git@github.com:user/test.git](mailto:git@github.com:user/test.git)

1. **git rm**

Removes files from your index and your working directory so they will not be tracked. Example:

git rm filename

1. **git merge**

Merges one or more branches into your current branch and automatically creates a new commit if there are no conflicts. Example:

git merge newbranchversion

1. **git reset**

Resets your index and working directory to the state of your last commit. Example:

git reset --hard HEAD

1. **git tag**

Tags a specific commit with a simple, human readable handle that never moves.

1. **git remote**

Shows all the remote versions of your repository.

git remote origin

1. **git log**

Shows a listing of commits on a branch including the corresponding details.

git log commit

1. **git diff**

Generates patch files or statistics of differences between paths or files in your git repository, or your index or your working directory.

1. **git archive**

Creates a tar or zip file including the contents of a single tree from your repository.

1. **git gc**

Garbage collector for your repository. Optimizes your repository. Should be run occasionally.

git gc

1. **git fsck**

Does an integrity check of the Git file system, identifying corrupted objects.

git fsck

1. **git prune**

Removes objects that are no longer pointed to by any object in any reachable branch.

git prune

* **Significance and importance of git in development of applications**

As you may be aware, Git is a version control software application created by Linus Torvalds (the creator of Linux) with the collaboration of many other developers. A version control application keeps track of all the changes that you do in the files of your project. Every time you make changes to files of a project, you can push those changes to a repository. Other developers can pull your changes from the repository and continue to work with the improvements that you added to the project files. There are many version control applications.

* **Maintaining version of applications using git**

From your master branch since you are done v1.0 add a tag called v1.0.

git tag -a -m "Tagging release 1.0" v1.0

This way you can always come back to a specific version at any time by calling git checkout [tag\_name]. Another common practice is to use branches to work on features until they are stable.

git checkout -b [feature-branch]

That creates a new branch named whatever is in [feature-branch] and checks it out. Be sure to do this from where you want to start working on the feature (typically from master). Once stable they can then be safely merged into master. From master run:

git merge [feature-branch]

This way your master branch always stays in a working state and only completed items get added once ready. This will allow you to keep a working copy of the app at all times (ideally anyways) for testing, etc.

You could use branches for each version of the application however using tags makes it so you can't merge into another branch version by accident.