GIT

29-Jan-19

Version Control System

- **Version Control System (VCS)** is a software that helps software developers to work together and maintain a complete history of their work.
- Allows developers to work simultaneously.
- Does not allow overwriting each other's changes.
- Maintains a history of every version.
- Types of VCS
  - Centralized version control system (CVCS).
  - Distributed/Decentralized version control system (DVCS).

A Brief History of Version Control

- First Generation
  - Single-file
  - No networking
  - e.g. SCCS, RCS
- Second Generation
  - Multi-file
  - Centralized
  - e.g. CVS, VSS, SVN, TFS, Perforce
- Third Generation
  - Changesets
  - Distributed
  - e.g. Git, Hg, Bazaar, BitKeeper

GIT BASIC CONCEPTS

- Git is a distributed revision control and source code management system with an emphasis on speed.
- Git was initially designed and developed by Linus Torvalds for Linux kernel development.
- Git is a free software distributed under the terms of the GNU General Public License version 2.

About Git

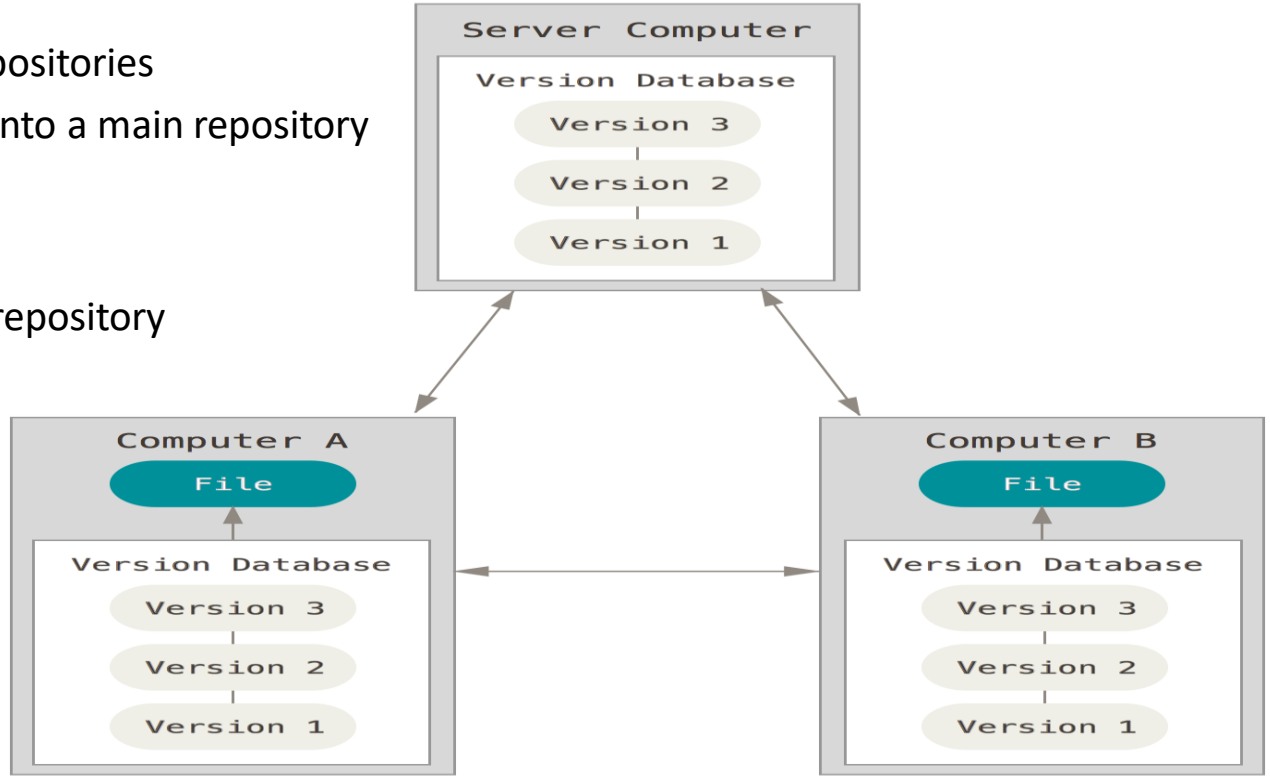
- Created by Linus Torvalds, who also created Linux
- Prompted by Linux-BitKeeper separation
- Started in 2005
- Written in Perl and C
- Runs on Linux, OS X, Windows, and many other operating systems
- Design goals Speed
  - Simplicity
  - Strong branch/merge support
  - Distributed
  - Scales well for large projects

Distributed Version Control System

Centralized VCS	Distributed VCS
• One central repository • Must be capable of connecting to repo • Need to solve issues with group members making different changes on the same files	• Everyone has a working repo • Faster • Connectionless • Still need to resolve issues, but it's not an argument against DVCS

Distributed Version Control System - Advantages

- Different topologies
  - Centralized
    - Developers push changes to one central repository
  - Hierarchical
    - Developers push changes to subsystem-based repositories
    - Sub-system repositories are periodically merged into a main repository
  - Distributed
    - Developers push changes to their own repository
    - Project maintainers pull changes into the official repository
- Backups are easy
  - Each clone is a full backup



Advantages of Git

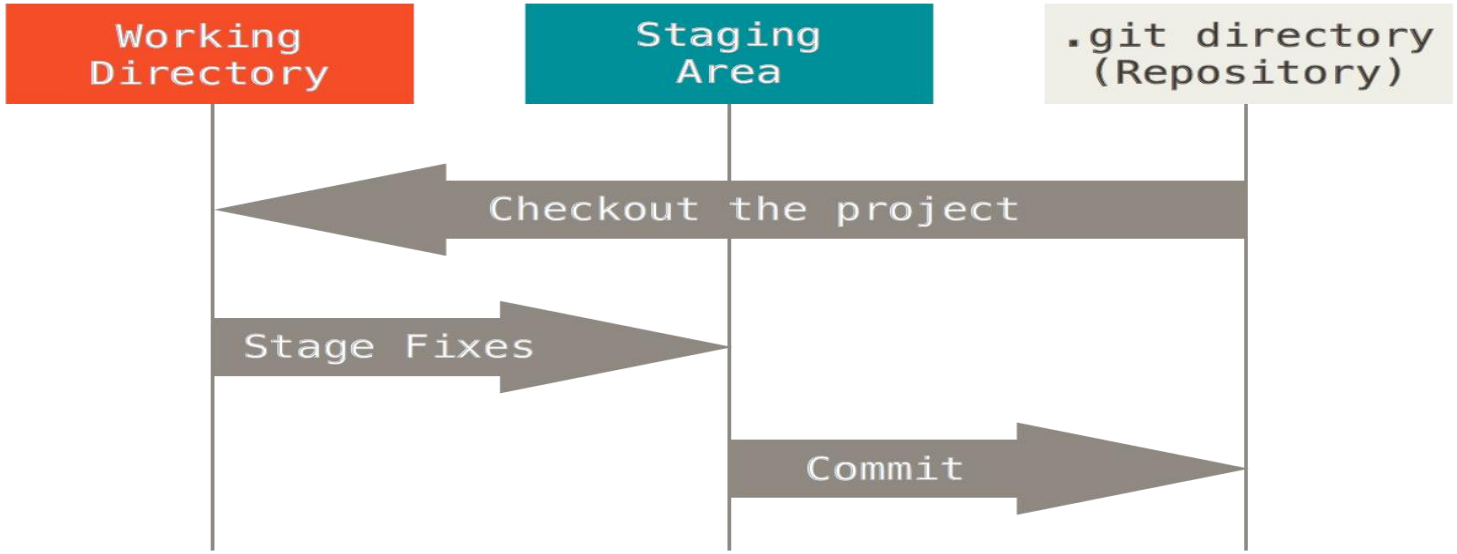
- **Free and open source -** GPL's open source license.
- **Fast and small**
- **Implicit backup**
- **Security -** SHA1
- **No need of powerful hardware**

DVCS Terminologies

- Local Repository
- Working Directory and Staging Area or Index
- Blobs
- Trees
- Commits
- Branches
- Tags
- Clone
- Pull
- Push
- HEAD
- Revision
- URL

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.



ENVIRONMENT SETUP

- **Installing Git in Windows**
  - *http://git-scm.com/download/win*

Configuring Git

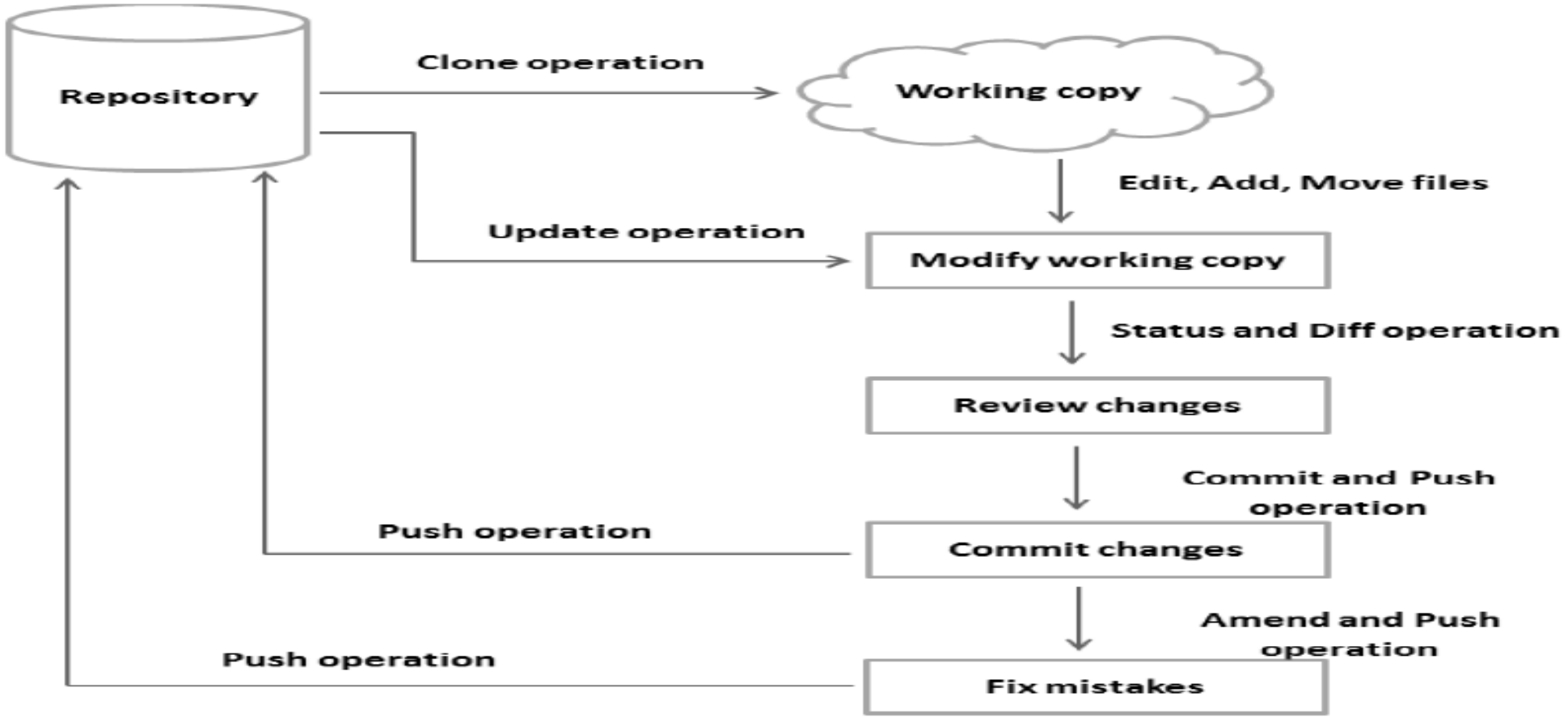
- Git provides the git config tool, which allows you to set configuration variables.
- Git stores all global configurations in **.gitconfig** file, which is located in your home directory.
- System-level configuration
  - git config --system
  - Stored in /etc/gitconfig or c:\Program Files (x86)\Git\etc\gitconfig
- User-level configuration
  - git config --global
  - Stored in ~/.gitconfig or c:\Users\<NAME>\.gitconfig
- Repository-level configuration
  - git config
  - Stored in .git/config in each repo

Configuring Git

- **Listing Git settings -** git config --global --list
- **Setting username -** git config --global user.name "Ganesh Babu“
- **Setting email id -** git config --global user.email ganeshbabum@yahoo.com
- **Color highlighting -** git config --global color.ui auto
- **Setting default editor –** git config --global core.editor Notepad
- Autocorrect - git config –global help.autocorrect 1

- **Reset** - git config --global -unset <>

LIFE CYCLE



WORKING LOCALLY WITH GIT

- Creating a local repository
- Adding files
- Committing changes
- Viewing history
- Viewing a diff
- Working copy, staging, and repository
- Deleting files
- Cleaning the working copy
- Ignoring files with .gitignore

WORKING LOCALLY WITH GIT

- **Initialize a new repository** - git init
- **Check status of repository** – git status
- **To add a file to repository** – git add <<filename>>
- **To add all the files** – git add –A
- **To add all updated files** – git add -u
- **Committing** - git commit
- **Committing with comment** - git commit -m 'Initial commit'
- **Check log file**
  - git log
  - git log --oneline
- **Add modified file** – git add –u
- **View diffs** –
  - git diff <sha1>..<sha2>>
  - git diff HEAD~1..HEAD
  - git diff HEAD~1..

WORKING LOCALLY WITH GIT

- Undoing the changes in the file – git checkout <<filename>
- Undoing all changes - git reset –hard
- Reset commit
  - git reset –-soft<<Commit id>> : Does not touch the index file or the working tree at all
  - git reset –-hard <<Commit id>> :Resets the index and working tree. Any changes to tracked files in the working tree since <commit> are discarded.
  - git reset –-mixed<<Commit id>> : Resets the index but not the working tree

- Cleaning unwanted files
  - git clean –n
  - git clean –f <<file name>>
  - git clean –f
- Ignoring files with gitignore
  - Configure .gitignore
  - Create .gitignore file and add /logs

WORKING REMOETLY WITH GIT

- Cloning a remote repository
- Listing remote repositories
- Fetching changes from a remote
- Merging changes
- Pulling from a remote
- Pushing changes remotely
- Working with tags

WORKING REMOETLY WITH GIT

- Proxy setup
  - git config --global http.proxy http://sonata\\ganesh.bm:pwd@172.23.0.99:8080
  - git config --global https.proxy https://sonata\\username:pwd@172.23.0.99:8080
- Cloning a remote repository
  - git clone gitURL
  - git clone https://github.com/jquery/jquery.git
- Basic Repository Statistic
  - To see the number of commits git log --oneline | wc –l
  - git log –oneline –graph
  - git shortlog
  - git shortlog –sne (summary, numeric decreasing, email)
- Git repository provides graph options (https://github.com/jquery/jquery)
- To see last change – git show HEAD

WORKING REMOETLY WITH GIT

- To see the remote location
  - git remote
  - git remote -v

Git Protocols

Protocol	Port	Example	Notes
http(s)	80/443	https://github.com/jquery/jquery.git	Read-write Password for auth Firewall friendly
git	9418	git://github.com/jquery/jquery.git	Read-only Anonymous only
ssh	22	git@github.com:jquery/jquery.git	Read-write SSH keys for auth
file	n/a	/Users/James/code/jquery	Read-write Local only

Fetching from a Remote

- Adding to remote repository
  - git remote add origin https://github.com/ganeshbabum/training.git
  - git push -u origin master
- Fetch from remote repository
  - git fetch
  - git fetch origin
- To check the status in remote branch
  - git log origin/master
- Combines branch into current local branch
  - git merge origin/master

Pulling/Pushing from/to a Remote

- Downloads bookmark history and incorporates changes
- git pull origin master (it is exactly like git fetch; git fetch origin/master)
- git remote rm origin (to remove the origin)

Branching

- Branch operation allows creating another line of development.
- We can use this operation to fork off the development process into two different directions.
- Creating a local branch
  - git branch feature1
- To move to branch
  - git checkout feature1
- Create and checkout
  - git checkout –b <<branchname>>
- To see the log with branch
  - git log --graph --oneline --all (all branches, -- decorate adds the labels like tags, head etc…)
  - You may create alise instead of typing all -
    - git congif --global alias.lga "log --graph --oneline --all --decorate“
  - Use git lga.

Branching

- To create a brach from a particular commit (SHA1)
  - Git branch fix1 <sha1 # - 5 chars>
- Renaming branch (moving the branch from one name to another)
  - git branch –m <b1> <b2>
- Delete branch
  - git branch -d <<branch name>>
  - The branch may not get deleted if the branch is not merged with the master
  - To delete forcefully - git branch -D bug1234
- To undelete the branch
  - git reflog
  - git branch bug1234 88cc4be
- Git will keep deleted commits for 30 days

Merging branches.

- Combines branch into current local branch
- git checkout master – to go to master branch
- git merge <<branch name>>
- git branch –d <<branch name>> - to delete the branch
- git mergetool – allows us to use different merge tools to resolve conflicts.

Creating/Deleting a remote branch

- git push origin <<branch>>
- git push origin <<localbranch:new name for remote branch>> - to give another name for remore branch, else git will create the branch with the same name as local.
- To see the remote branches – git branch –r

- To delete remote branch – git push origin :<<remote branch>>

Tags

- Tag operation allows giving meaningful names to a specific version in the repository.
- By default, git tag will create a tag on the commit that HEAD is referencing.
- Lightweight Tags
- Annotated tags
- Listing Tags
  - git tag
  - git tag -l "v6.7.0*"
- Create Lightweight Tags
  - git tag v6.7
  - git show v6.7
- Annotated tags are stored as full objects in the Git database. They’re checksummed; contain the tagger name, email, and date; have a tagging message;
  - git tag -a v6.7 -m "my version 6.7"

Tags

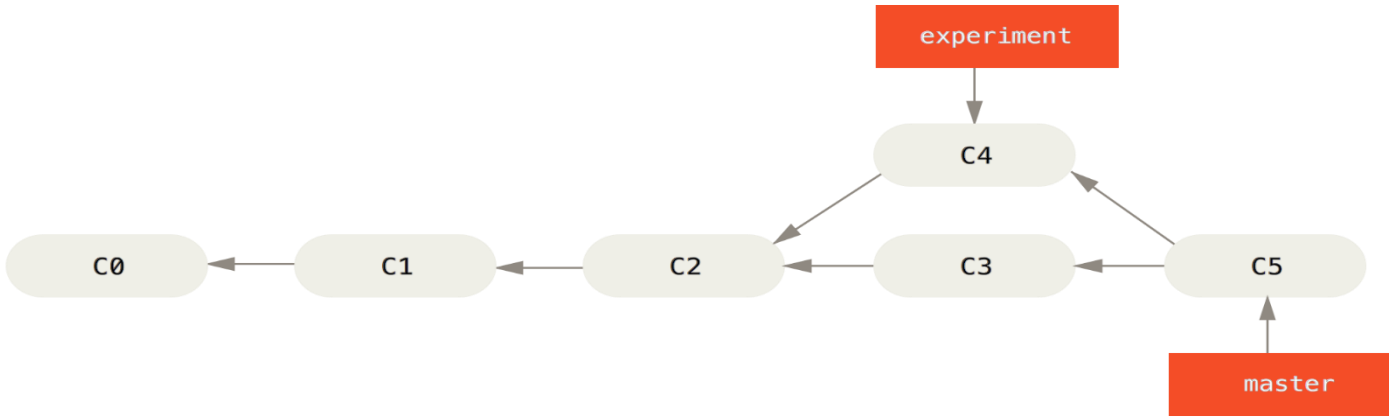
- Tagging Old Commits
  - git tag v0.6 <<commit id>>
- To verify the tag
  - git tag –v v1.0
  - Push - git push –tag
- To show
  - git show Release_1_0
- To delete tag
  - git tag -d Release_1_0
- View the state of a repo at a tag by using
  - git checkout Release_1_0
- ReTagging/Replacing Old Tags
  - git tag -a -f v1.4 15
  - Will get error if –f option is not used.

Rewriting history

- **Changing the Last Commit:**
  - The git commit --amend command is a convenient way to modify the most recent commit.
  - git commit --amend lets you take the most recent commit and add new staged changes to it.
  - git commit --amend

Rebasing Changes

- In Git, there are two main ways to integrate changes from one branch into another: the merge and the rebase.



- Take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called rebasing. With the rebase command, you can take all the changes that were committed on one branch and replay them on another one.

Rebasing Changes

- Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow.
- From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit.
- git checkout experiment
- git rebase master
- An interesting option it accepts is --interactive (-i for short), which will open an editor with a list of the commits which are about to be changed. This list accepts commands, allowing the user to edit the list before initiating the rebase action.

Cherry-picking

- Useful when we want to apply the selected commits to a branch.
- In this case merge and rebase is not useful.
- Good for moving patches
- git cherry-pick <commint sha>>

Stashing changes

- In Git, the stash operation takes your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can reapply at any time.
- you can store your partial changes and later on commit it.
- git stash - to stash the changes
- Git stash list
- Git stash apply - to pull the changes back
- Git reset –hard HEAD
- Git stash pop (like apply, but removes the item from stash)
- Git stash drop
- Git stash branch newbranch

Blame

- Show what revision and author last modified each line of a file
- git blame only operates on individual files
- git blame README.MD
- git blame -L 1,5 README.md - output to the requested line range.
- git blame -e README.md - shows the authors email address instead of username.
- git blame -w README.md -ignores whitespace changes
- git blame -M README.md -detects moved or copied lines within in the same file

Detached head

- This exact state - when a specific *commit* is checked out instead of a *branch* - is what's called a "detached HEAD".
- Git checkout <<commit>>
- Normally, when checking out a proper branch name, Git automatically moves the HEAD pointer along when you create a new commit. You are automatically on the newest commit of the chosen branch.
- When you instead choose to check out a *commit hash*, Git won't do this for you. The consequence is that when you make changes and commit them, these **changes do NOT belong to any branch**.
  This means they can easily get lost once you check out a different revision or branch

Git Revert

- Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.
- This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.
- A revert operation will take the specified commit, inverse the changes from that commit, and create a new "revert commit".
- git revert HEAD