EAST WEST INSTIUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

DEPARTMENT OF ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

Project Management with Git

BCS358C – SEMESTER 3

OBSERVATION BOOK

EXPERIMENT-1

Setting Up and Basic Commands

Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

Solution:

To initialize a new Git repository in a directory, create a new file, add it to the staging area, and commit the changes with an appropriate commit message, follow these steps:

1. Open your terminal and navigate to the directory where you want to create the Git repository.
2. Initialize a new Git repository in that directory:
   $ git init
3. Create a new file in the directory. For example, let's create a file named "my_file.txt" You can use any text editor or command-line tools to create the file. (touch my_file.txt).
4. Add the newly created file to the staging area. Replace "my_file.txt" with the actual name of your file:
   $ git add my_file.txt
   This command stages the file for the upcoming commit.
5. Commit the changes with an appropriate commit message. Replace "Your commit message here" with a meaningful description of your changes:
   $ git commit -m "Your commit message here"
   Your commit message should briefly describe the purpose or nature of the changes you made.
   For example:
   $ git commit -m "Add a new file called my_file.txt"

After these steps, your changes will be committed to the Git repository with the provided commit message. You now have a version of the repository with the new file and its history stored in Git.

1. **git init**

Definition: The git init command initializes a new Git repository in a project directory. It sets up all the necessary metadata and files that Git uses to track changes in your project.

Explanation: When you run git init, it creates a hidden .git folder in your project's root directory. This folder contains all of the information about your Git repository, including configurations, history of commits, and branches.

Use Case: If you have a project and want to start tracking changes with Git, you would use git init to make that directory a Git repository.

Desktop/ my-project/

git init

2. **git add**

Definition: The git add command stages changes (modifications, additions, or deletions) to be included in the next commit.

Explanation: When you modify files in your working directory, Git notices those changes but doesn't track them yet. To tell Git which changes should be part of the next commit, you use git add. This moves the changes to the staging area, preparing them to be committed. The changes won't be saved until you actually commit them with the git commit command.

Use Case: After modifying or creating a file, you use git add to stage it for the next commit.

git add file.txt

Or to add all changed files:

git add .

3. **git commit**

Definition: The git commit command captures the state of the project in a snapshot by saving changes from the staging area into the repository's history.

Explanation: When you run git commit, Git takes all the changes you've added to the staging area and creates a new commit. Each commit has a unique identifier (hash) and a message describing the changes made. This is how Git tracks the history of the project.

Use Case: After staging your changes with git add, you commit them with git commit to save those changes to the repository.

git commit -m "Added new feature"

In summary, git init starts tracking a project, git add stages changes, and git commit saves those staged changes into the repository.

EXPERIMENT-2

Creating and Managing Branches

Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."

Solution:

To create a new branch named "feature-branch," switch to the "master" branch, and merge the "feature-branch" into "master" in Git, follow these steps:

1. Make sure you are in the "master" branch by switching to it:
   $ git checkout master
2. Create a new branch named "feature-branch" and switch to it:
   $ git checkout -b feature-branch
   This command will create a new branch called "feature-branch" and switch to it.
3. Make your changes in the "feature-branch" by adding, modifying, or deleting files as needed.
4. Stage and commit your changes in the "feature-branch":
   $ git add .
   $ git commit -m "Your commit message for feature-branch"
   Replace "Your commit message for feature-branch" with a descriptive commit message for the changes you made in the "feature-branch."
5. Switch back to the "master" branch:
   $ git checkout master
6. Merge the "feature-branch" into the "master" branch:
   $ git merge feature-branch
   This command will incorporate the changes from the "feature-branch" into the "master" branch.

Now, your changes from the "feature-branch" have been merged into the "master" branch. Your project's history will reflect the changes made in both branches.

1. **git branch**

Definition: The git branch command is used to create, list, or delete branches in a Git repository.

Explanation: Branches in Git allow you to work on different versions of your project simultaneously. The git branch command shows the current branches in the repository and also allows you to create new branches or delete old ones. A branch is essentially a pointer to a specific commit. You can create a new branch to work on a feature, bug fix, or experiment without affecting the main codebase.

Use Cases:

- Listing branches: You can list all the branches in your repository.
- Creating a new branch: You can create a new branch to isolate work.
- Deleting a branch: After merging, you might want to delete a branch to keep your repository clean.

Examples:

To list all branches:

git branch

To create a new branch:

git branch new-feature

To delete a branch:

git branch -d new-feature

2. **git checkout**

Definition: The git checkout command is used to switch between branches or restore files in your working directory.

Explanation: When you have multiple branches in a Git repository, you can switch between them using git checkout. This command changes the working directory to reflect the state of the branch you want to work on. Additionally, it can be used to restore files to a previous state (like reverting changes).

Use Cases:

- Switching branches: You can use git checkout to move from one branch to another.
- Restoring files: You can use it to restore a specific file to a previous state.

Examples:

To switch to a branch named new-feature:

git checkout new-feature

To create and switch to a new branch in one command:

git checkout -b new-feature

To revert changes in a file:

git checkout -- filename.txt

3. **git merge**

Definition: The git merge command merges changes from one branch into another branch.

Explanation: When you are done working on a feature in a separate branch, you can use git merge to combine the changes into your main branch (typically master or main). The merge takes the commits from one branch and applies them to the current branch, incorporating the history and changes.

Use Case: After completing work in a feature branch, use git merge to combine those changes into the main branch or another target branch.

Example:

To merge a branch named new-feature into the main branch:

git checkout main

git merge new-feature

In summary:

- git branch is used for creating, listing, and deleting branches.
- git checkout is for switching between branches or restoring files.
- git merge is for combining changes from one branch into another.

These commands are key in Git's branching model, allowing you to work on separate features and then merge them into the main project.

EXPERIMENT-3

Creating and Managing Branches

Write the commands to stash your changes, switch branches, and then apply the stashed changes.

Solution:

To stash your changes, switch branches, and then apply the stashed changes in Git, you can use the following commands:

1. Stash your changes:
   $ git stash save "Your stash message"
   This command will save your changes in a stash, which acts like a temporary storage for changes that are not ready to be committed.
2. Switch to the desired branch:
   $ git checkout target-branch
   Replace "target-branch" with the name of the branch you want to switch to.
3. Apply the stashed changes:
   $ git stash apply
   This command will apply the most recent stash to your current working branch. If you have multiple stashes, you can specify a stash by name or reference (e.g., git stash apply stash@{2})if needed.

If you want to remove the stash after applying it, you can use git stash pop instead of git stash apply.
Remember to replace "Your stash message" and "target-branch" with the actual message you want for your stash and the name of the branch you want to switch to

git stash is a command in Git that temporarily saves changes in your working directory that you're not ready to commit yet. This allows you to work on something else, while keeping your local modifications safe for later use. Here are the most commonly used git stash commands:

1. git stash
Description: Stashes your changes (tracked files) and returns your working directory to the last committed state. The changes are stored in a hidden stack.
Usage: **git stash**

2. git stash save "message"
Description: Stashes your changes with a descriptive message, making it easier to identify later.
Usage: **git stash save "work-in-progress for feature X"**

3. git stash list
Description: Lists all the stashed changes, showing the stash index and any messages saved with the stash.
Usage: **git stash list**

4. git stash pop
Description: Applies the latest stashed changes back to your working directory and removes them from the stash list.
Usage: **git stash pop**

5. git stash apply
Description: Applies the latest stashed changes (or a specific stash, if specified) back to your working directory but keeps them in the stash list.
Usage: **git stash apply**

6. git stash drop
Description: Removes the latest stash (or a specific stash) from the stash list.
Usage: **git stash drop**

7. git stash clear
Description: Clears all stashes from the stash list, removing them permanently.
Usage: **git stash clear**

These commands provide flexibility in how you manage your work-in-progress changes in Git, allowing you to stash, apply, or drop changes as needed while working on other features or resolving issues.

EXPERIMENT-4

Collaboration and Remote Repositories
Clone a remote Git repository to your local machine.

Solution:

To clone a remote Git repository to your local machine, follow these steps:

1. Open your terminal or command prompt.
2. Navigate to the directory where you want to clone the remote Git repository. You can use the cd command to change your working directory.
3. Use the **git clone** command to clone the remote repository. Replace <repository_url> with the URL of the remote Git repository you want to clone. For example, if you were cloning a repository from GitHub, the URL might look like this:
   $ git clone <repository_url>

   Here's a full example:
   $ git clone https://github.com/username/repo-name.git
   Replace https://github.com/username/repo-name.git with the actual URL of the repository you want to clone.

4. Git will clone the repository to your local machine. Once the process is complete, you will have a local copy of the remote repository in your chosen directory.
   You can now work with the cloned repository on your local machine, make changes, and push those changes back to the remote repository as needed.

The git remote command is used to manage connections to remote repositories in Git. Remote repositories are versions of your project hosted on the internet or a network that allows collaboration with others. The git remote command allows you to view, add, remove, and modify these connections.

Here are some key usages of git remote:

List all remotes:

Command: **git remote or git remote -v**
Explanation: This lists the remote repositories associated with the local repository. The -v option displays the URLs for the remotes (fetch and push URLs).

Example:
git remote -v

Output might be something like:
origin https://github.com/username/repo.git (fetch)
origin https://github.com/username/repo.git (push)

Add a new remote:

Command: **git remote add <name> <url>**
Explanation: This adds a new remote repository to your project, giving it a name (e.g., origin) and linking it to a URL where the remote repository is hosted.

Example:
git remote add origin https://github.com/username/repo.git

Remove a remote:

Command: **git remote remove <name>**
Explanation: This removes the association with a remote repository.

Example:
git remote remove origin

Rename a remote:

Command: **git remote rename <old-name> <new-name>**
Explanation: This renames an existing remote repository.

Example:
git remote rename origin upstream

Show detailed information about a remote:

Command: **git remote show <name>**
Explanation: This shows detailed information about a specific remote, such as the URLs and tracking branches.

Example:
git remote show origin

These commands help you manage how your local Git repository interacts with remote repositories, which is crucial in distributed version control systems like Git.

git remote add origin:

The git remote add origin command is used to link your local Git repository to a remote repository, assigning the remote repository the name origin. The term "origin" is an alias for the URL of the remote repository and is commonly used as the default name, but you can choose any name you like.

Purpose: To establish a connection between your local project and a remote repository (usually hosted on platforms like GitHub, GitLab, etc.) for the first time.

Syntax:

git remote add origin <remote-repository-URL>

Example:

git remote add origin https://github.com/username/repo.git

Explanation: In this example, the local repository is linked to the remote repository hosted on GitHub. Once this link is established, you can push your code to origin (the remote repository) using commands like git push.

git remote set-url origin:

The git remote set-url origin command is used to change the URL of an existing remote repository. This is helpful when the URL of the remote repository has changed (e.g., if you've moved a repository to a new location or changed its protocol from HTTP to SSH).

Purpose: To update or change the remote repository URL associated with a remote name (in this case, origin).

Syntax:

git remote set-url origin <new-remote-URL>

Example:

git remote set-url origin git@github.com:username/repo.git

Explanation: In this example, the URL of the remote repository origin is changed from an HTTPS URL to an SSH URL. This can be done if you want to switch the access method to the remote repository or if the repository's location changes.

Both commands are critical for managing remote repositories effectively. The first is for setting up the initial remote link, and the second is for updating it as needed.

git push:

The git push command is used to upload the local repository's content to a remote repository. It sends the changes made locally to a remote server, usually to a branch you specify. In simpler terms, it "pushes" your committed changes from your local Git repository to a remote one like GitHub or GitLab.

Example:
git push origin master
This pushes changes from your local main branch to the main branch in the remote repository named origin.

git pull:

The git pull command is used to fetch changes from a remote repository and merge them into the current branch. It is a combination of two actions: git fetch (which downloads the changes) and git merge (which merges them into your current branch).

Example:
git pull origin master
This fetches and merges changes from the remote main branch of the repository named origin into your local branch.

Both commands are essential in collaborating on projects, ensuring synchronization between the local and remote repositories.

The git clone command is used to create a copy of a remote repository on your local machine. It clones all files, branches, and commits, allowing you to work with the repository locally. Below are the various git clone commands and options:

1. **git clone <repository-url>**
   Description: This command clones the repository from the given URL (which could be HTTP, HTTPS, or SSH). It creates a new directory with the name of the repository and copies all the content, including commits and branches.
   Usage:
   git clone https://github.com/user/repository.git

2. **git clone <repository-url> <directory-name>**
   Description: Clones the repository into a specific directory. Instead of the default repository name, you can specify a custom directory.
   Usage:
   git clone https://github.com/user/repository.git custom-directory

3. **git clone --branch <branch-name> <repository-url>**
   Description: Clones a specific branch of the repository instead of the default branch (usually main or master).
   Usage:
   git clone --branch develop https://github.com/user/repository.git

EXPERIMENT-5

Collaboration and Remote Repositories

Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

Solution:

To fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch in Git, follow these steps:

1. Open your terminal or command prompt.'
2. Make sure you are in the local branch that you want to rebase. You can switch to the branch using the following command, replacing **<branch-name>** with your actual branch name:
   $ git checkout <branch-name>
3. Fetch the latest changes from the remote repository. This will update your local repository with the changes from the remote without merging them into your local branch:
   $ git fetch origin
   Here, origin is the default name for the remote repository. If you have multiple remotes, replace origin with the name of the specific remote you want to fetch from.
4. Once you have fetched the latest changes, rebase your local branch onto the updated remote branch:
   $ git rebase origin/<branch-name>
   Replace **<branch-name>** with the name of the remote branch you want to rebase onto. This command will reapply your local commits on top of the latest changes from the remote branch, effectively incorporating the remote changes into your branch history.
5. Resolve any conflicts that may arise during the rebase process. Git will stop and notify you if there are conflicts that need to be resolved. Use a text editor to edit the conflicting files, save the changes, and then continue the rebase with:
   $ git rebase –continue
6. After resolving any conflicts and completing the rebase, you have successfully updated your local branch with the latest changes from the remote branch.
7. If you want to push your rebased changes to the remote repository, use the git push command. However, be cautious when pushing to a shared remote branch, as it can potentially overwrite other developers' changes:
   $ git push origin <branch-name>
   Replace **<branch-name>** with the name of your local branch. By following these steps, you can keep your local branch up to date with the latest changes from the remote repository and maintain a clean and linear history through rebasing.

The git fetch origin command is used to retrieve updates from a remote repository (usually referred to as origin) without modifying your working directory or current branch. It downloads any new commits, branches, or changes from the remote repository but does not merge them into your local branch. Below are different ways to use git fetch origin and their explanations:

1. **git fetch origin**

   Description: Fetches all branches, tags, and commits from the remote repository (origin) that are not present in your local repository. This command updates your local references to the remote branches but does not modify your working directory.

   Usage:

   git fetch origin

2. git fetch origin <branch-name>

   Description: Fetches updates only for the specified branch from the remote repository (origin) without affecting your local branch or files.

   Usage:

   git fetch origin main

The git rebase command is used in Git to integrate changes from one branch into another by applying commits in sequence, without creating a merge commit. This keeps the history linear, making it cleaner and easier to follow. Below are some commonly used git rebase commands and their explanations:

1. git rebase <upstream-branch>

   Description: Rebases your current branch on top of the specified upstream branch. It applies your branch's commits after the latest commit from the upstream branch, essentially moving your branch's starting point to the latest commit on the upstream branch.

   Usage:

   git rebase main

2. git rebase --interactive (or git rebase -i)

   Description: Initiates an interactive rebase, allowing you to modify, reorder, squash, or edit commits during the rebase process. This is useful for cleaning up commit history before sharing it.

   Usage:

   git rebase -i HEAD~3

   (This example will interactively rebase the last 3 commits.)

3. git rebase --onto <new-base> <upstream> <branch>

   Description: Rebases a branch onto a new base while skipping certain commits. It moves the commits on the <branch> that are not part of <upstream> onto <new-base>. This is useful when you want to move a branch to a different base but exclude intermediate commits.

   Usage:

   git rebase --onto main feature-branch bugfix-branch

4. git rebase --continue

   Description: If a rebase results in conflicts, Git pauses and asks you to resolve them. Once resolved, this command continues the rebase process from where it stopped.

   Usage:

   git rebase --continue

5. git rebase --abort

   Description: Aborts the rebase process and returns your branch to the state it was in before the rebase started. This is useful if the rebase becomes complicated or you change your mind.

   Usage:

   git rebase --abort

6. git rebase --skip

Description: Skips the current commit during a rebase if you want to bypass a specific commit that's causing conflicts. The rebase continues with the next commit.

Usage:

git rebase --skip

7. git rebase --preserve-merges

Description: Preserves merge commits during a rebase. This option maintains the existing merge commits while rebasing the branch, but it is generally less commonly used.

Usage:

git rebase --preserve-merges main

Key Points:

- Rebase vs Merge: While merging keeps the commit history with explicit merge commits, rebasing creates a linear commit history, which can be cleaner but more complex when conflicts arise.
- Interactive Rebasing: It is a powerful tool for rewriting commit history, letting you squash, edit, or reorder commits.
- Rebase Carefully: Avoid rebasing branches that have already been shared with others, as it rewrites history and may cause issues with collaborators.

These git rebase commands give you control over how you integrate changes between branches, offering a way to keep your project's history clean and well-organized.

EXPERIMENT-6

Collaboration and Remote Repositories

Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.

Solution:

To merge the "feature-branch" into "master" in Git while providing a custom commit message for the merge, you can use the following command:

$ git checkout master
$ git merge feature-branch -m "Your custom commit message here"

Replace "Your custom commit message here" with a meaningful and descriptive commit message for the merge. This message will be associated with the merge commit that is created when you merge "feature-branch" into "master.

The command git merge feature-branch -m "Your custom commit message here" is used to merge changes from one branch (in this case, feature-branch) into the current branch, while allowing you to include a custom commit message for the merge commit.

Here’s a breakdown of each part:

- **git merge:** This is the command that merges changes from the specified branch into the current branch.
- **feature-branch:** This is the name of the branch whose changes you want to merge into the current branch. It could be any branch you're working with (e.g., development, bug-fix, etc.).
- **-m "Your custom commit message here":** The -m option allows you to specify a custom commit message for the merge commit. This commit message is added to the merge commit created when the changes are merged. If you don't provide this option, Git opens the default editor and prompts you to enter a commit message manually.

Notes:

- A merge commit is created by Git when changes from two branches are merged. It contains information about both parent branches.
- The -m option helps streamline the process when you want to automate or quickly add a descriptive message about the merge.

EXPERIMENT-7

Git Tags and Releases

Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

Solution:

To create a lightweight Git tag named "v1.0" for a commit in your local repository, you can use the following command:

$ git tag v1.0

This command will create a lightweight tag called "v1.0" for the most recent commit in your current branch.

If you want to tag a specific commit other than the most recent one, you can specify the commit's SHA-1 hash after the tag name.

For example:

$ git tag v1.0 <commit-SHA>

Example:

$ git tag t1.0 <commit-SHA>

Replace <commit-SHA> with the actual SHA-1 hash of the commit you want to tag.

git tag Command

The git tag command is used to create, list, or manage tags in Git. Tags are references to specific points in Git history (commits), commonly used to mark important milestones like version releases (e.g., v1.0.0, v2.0.0).

Git supports two types of tags:

- Lightweight Tags: Simple pointers to a commit, acting like a label.
- Annotated Tags: Full Git objects with additional metadata (author, date, message), used for more permanent references like official releases.

Common git tag Usages:

- List tags: **git tag** OR **git tag --list**
- Create a lightweight tag: **git tag <tagname>**
- Create an annotated tag: **git tag -a <tagname> -m "Message"**
- Push tags to a remote: **git push origin <tagname> or git push origin --tags**

Tags are static and do not change once created, making them ideal for marking versions in a project.

EXPERIMENT-8

Advanced Git Operations

Write the command to cherry-pick a range of commits from "source-branch" to the current branch.

Solution:

To cherry-pick a range of commits from "source-branch" to the current branch, you can use the following command:

$ git cherry-pick <start-commit>^..<end-commit>

Replace **<start-commit>** with the commit at the beginning of the range, and **<end-commit>** with the commit at the end of the range.

The ^ symbol is used to exclude the <start-commit> itself and include all commits after it up to and including <end-commit>.

This will apply the changes from the specified range of commits to your current branch.

For example, if you want to cherry-pick a range of commits from "source-branch" starting from commit ABC123 and ending at commit DEF456, you would use:

$ git cherry-pick ABC123^..DEF456

Make sure you are on the branch where you want to apply these changes before running the cherry-pick command.

git cherry-pick Command
The git cherry-pick command allows you to apply the changes from a specific commit (or set of commits) from one branch to another, without merging the entire branch. It is a way to "pick" specific commits and apply their changes selectively.

Common git cherry-pick Usages:

Cherry-pick a single commit:
git cherry-pick <commit-hash>

Cherry-pick a range of commits:
git cherry-pick <start-commit-hash>^..<end-commit-hash>

Abort cherry-pick if conflicts arise:
git cherry-pick --abort

Use Cases:

- Applying a bug fix from one branch (e.g., main) to another (e.g., release).
- Selectively integrating specific features or fixes into a different branch without merging unrelated changes.
- This command helps maintain a clean history while bringing only the necessary changes into your working branch.

EXPERIMENT-9

Analysing and Changing Git History

Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?

Solution:

To view the details of a specific commit, including the author, date, and commit message, you can use the git show or git log command with the commit ID. Here are both options:

1. Using git show
   $ git show <commit-ID>
   Replace **<commit-ID>** with the actual commit ID you want to view. This command will display detailed information about the specified commit, including the commit message, author, date.

2. Using git log:
   $ git log -n 1 <commit-ID>
   The -n 1 option tells Git to show only one commit. Replace **<commit-ID>** with the actual commit ID. This command will display a condensed view of the specified commit, including its commit message, author, date, and commit ID.

Both of these commands will provide you with the necessary information about the specific commit you're interested in.

This command will display the details of the commit, including:

- Author: The name and email of the person who made the commit.
- Date: The date and time when the commit was made.
- Commit Message: The message associated with the commit.
- Changes: A diff showing the changes made in that commit.

git show

The git show command is used to display detailed information about a specific commit, including the changes (diff) introduced in that commit. It shows the commit metadata (author, date, commit hash) along with the actual content differences (file changes) introduced.

Usage:
git show [commit]

Example:
git show 1a2b3c4d

This will display details of the commit with hash 1a2b3c4d, including the commit message, files changed, and the diff for each file.

Key Features:

- Displays the changes made in a single commit.
- Shows the commit message, author details, and the patch (diff) showing the changes.
- Useful for reviewing the content of a specific commit in detail.

git log

The git log command provides a history of commits in a repository. It lists all previous commits in reverse chronological order (most recent commit first), including commit messages, author information, and timestamps.

Usage:
git log

Example:
git log

This will display the commit history with details like commit hash, author, date, and commit message.

Key Features:

- Lists the history of commits.
- Can be customized to show additional information such as file changes, commit stats, and more.
- Useful for tracking the progress and changes made in a project over time.

EXPERIMENT-10

Analysing and Changing Git History

Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."

Solution:

To list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31" in Git, you can use the git log command with the --author and --since and --until options. Here's the command:

$ git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"

This command will display a list of commits made by the author "JohnDoe" that fall within the specified date range, from January 1, 2023, to December 31, 2023. Make sure to adjust the author name and date range as needed for your specific use case.

The given command:

git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"

This command filters and displays the commit history in a Git repository based on specific criteria. Let's break down each part:

Components:

- **git log:** This is the base command that shows the commit history of a Git repository.
- **--author="JohnDoe":** This option filters the log to show only the commits made by the specified author, in this case, "JohnDoe". Git matches the author's name against the name in the commit metadata.
- **--since="2023-01-01":** This option specifies the starting date for the commit history. Commits made on or after January 1, 2023, will be included in the log.
- **--until="2023-12-31":** This option specifies the end date for the commit history. Commits made on or before December 31, 2023, will be included.

Summary:
This command will display a list of commits made by "JohnDoe" between January 1, 2023, and December 31, 2023. It will only show the commits within that time range and filter out others. The output includes the commit details such as the commit hash, message, author, and date for the selected commits.

Use Case:
This is useful when you want to see the work done by a specific contributor within a specific time frame, such as during a year or a particular project phase.

EXPERIMENT-11

Analysing and Changing Git History

Write the command to display the last five commits in the repository's history.

Solution:

To display the last five commits in a Git repository's history, you can use the git log command with the –n option, which limits the number of displayed commits. Here's the command:

$ git log -n 5

This command will show the last five commits in the repository's history. You can adjust thenumber after -n to display a different number of commits if needed.

The command:

git log -n 5

Explanation:
This command is used to display the last 5 commits in the Git commit history.

Components:

- **git log:** This is the base command to show the commit history of a repository, which lists previous commits with details like commit hash, author, date, and message.

- **-n 5:** The -n option limits the number of commits shown in the log output to the specified number, in this case, 5. So, -n 5 tells Git to display only the 5 most recent commits.

EXPERIMENT-12

Analysing and Changing Git History

Write the command to undo the changes introduced by the commit with the ID "abc123".

Solution:

To undo the changes introduced by a specific commit with the ID "abc123" in Git, you can use the git revert command. The git revert command creates a new commit that undoes the changes made by the specified commit, effectively "reverting" the commit. Here's the command:

$ git revert <commit>

Replace **<commit>** with the actual commit ID that you want to revert. After running this command, Git will create a new commit that negates the changes introduced by the specified commit. This is a safe way to undo changes in Git because it preserves the commit history and creates a new commit to record the reversal of the changes.

$ git revert abc123

git revert

The git revert command in Git is used to create a new commit that reverses the changes made by a previous commit, effectively undoing the changes introduced by that commit without altering the commit history.

Unlike git reset, which modifies the history by removing or modifying existing commits, git revert preserves the history by adding a new commit that negates the changes.

Usage:
git revert <commit>

Example:
git revert abc1234

This will create a new commit that undoes the changes made by the commit with the hash abc1234, while leaving all other commits intact.

Key Features:

- Does not erase history: git revert keeps the original commit history intact by creating a new commit.
- Safe for shared repositories: Since it doesn't rewrite history, it's safer to use in collaborative environments (unlike git reset or git rebase).
- Commits a new change: It doesn't delete the original commit but instead adds a new commit that undoes the changes.

Use Case:

- Used to undo specific changes in a safe manner while keeping the commit history intact.
- Helpful in collaborative projects where modifying commit history is undesirable.

GIT CONFLICTS & ADDITIONAL GIT COMMANDS

In Git, the commands --continue, --abort, and --skip are typically used in conjunction with operations that involve multiple steps, such as rebase, cherry-pick, merge, or amending commits. These options help you manage and recover from conflicts or issues that might arise during these processes.

1. **--continue**
The --continue option is used to resume an interrupted Git operation after resolving conflicts or completing necessary steps.

Example usage:
git rebase --continue
git cherry-pick --continue

Explanation:
When performing operations like git rebase or git cherry-pick, conflicts may arise if changes cannot be automatically merged. After manually resolving conflicts, you would use the --continue option to tell Git that you have resolved the conflicts and are ready to proceed with the remaining steps of the operation.

Common operations:

git rebase --continue
git cherry-pick --continue
git merge --continue

2. **--abort**
The --abort option is used to cancel or stop an ongoing Git operation and revert back to the state before the operation began.

Example usage:
git rebase --abort
git merge --abort

Explanation:
If something goes wrong during a rebase or merge (such as complex conflicts you don't want to resolve right now), --abort allows you to stop the process and return to the pre-operation state. This means all changes made during the operation will be undone.

Common operations:

git rebase --abort
git merge --abort
git cherry-pick --abort

3. **--skip**
The --skip option allows you to skip the current commit (in cases like rebase or cherry-pick) if you are unable or unwilling to resolve the conflicts for that particular commit.

Example usage:
git rebase --skip
git cherry-pick --skip

Explanation:
When you hit a conflict during a multi-step process like a rebase or cherry-pick, and you decide that the problematic commit can be omitted (or is not necessary), using --skip will bypass the current commit and move on to the next one in the sequence. Any changes in the skipped commit will not be applied.

Common operations:

git rebase --skip
git cherry-pick --skip

Other Similar Commands

git reset

Usage:
git reset --hard
git reset --soft
git reset --mixed

Explanation:
The git reset command is used to undo changes in the working directory and/or the staging area. It comes in three forms:

- **--hard:** Resets the working directory, staging area, and the current commit pointer.
- **--soft:** Only resets the commit pointer, leaving changes in the working directory and staging area.
- **--mixed:** Resets the commit pointer and the staging area, but leaves changes in the working directory.

git merge --abort

Usage:
git merge --abort

Explanation:
Used to stop a merge operation and return the repository to the state it was in before the merge was initiated, effectively canceling the merge.
git am --abort / --continue / --skip

Usage:
git am --abort, git am --continue, git am --skip

Explanation:
These options work similarly to those described for rebase or cherry-pick, but are used with git am, which applies patches from emails. --abort cancels the patch application, --continue resumes after conflicts are resolved, and --skip bypasses a problematic patch.
These commands are useful for managing complex operations, especially when you need to recover from conflicts or errors during rebase, merge, or cherry-pick operations.

COMMON GIT CONFLICTS

Here's a list of some of the most common Git errors and conflicts, along with solutions:

1. **Merge Conflicts**

Description: Occur when two or more branches modify the same line of a file differently, and Git doesn't know which change to keep.

Solution:

When you attempt to merge, Git will mark the conflicting areas in the files.

Open the conflicting files and look for conflict markers (<<<<<<<, =======, and >>>>>>>).

Manually resolve the conflict by editing the file. Choose which part to keep or combine both changes.

After resolving, mark the conflict as resolved using:

git add <filename>

Finally, complete the merge with:

git commit

2. **Detached HEAD**

Description: Occurs when you are working on a commit that is not the tip of any branch, which prevents you from easily making new commits on the branch.

Solution:

To return to the latest commit on a branch:

git checkout <branch-name>

If you want to keep the changes you made:

git checkout -b <new-branch-name>

3. **Uncommitted Changes When Switching Branches**

Description: Git will prevent you from switching branches if you have uncommitted changes.

Solution:

You can either:

Commit your changes:

git add .

git commit -m "Your message"

git checkout <branch-name>

Stash your changes:

git stash

git checkout <branch-name>

After switching branches, you can retrieve your changes with:

git stash apply

4. **'Non-fast-forward' Error**

Description: Happens when trying to push changes, but the remote branch has new commits that you don't have locally.

Solution:

First, pull the latest changes from the remote:

git pull origin <branch-name>

If no conflicts, then push again:

git push origin <branch-name>

If there are conflicts, resolve them as needed (merge conflicts) and commit the resolved code before pushing.

5. **'No Tracking Information' Error**

Description: Occurs when you try to push or pull, but your local branch isn't linked to a remote branch.

Solution:

You can link the local branch with the remote branch like this:

git branch --set-upstream-to=origin/<branch-name>

6. **Accidentally Deleted a Branch**

Description: You deleted a branch, but it had important commits that haven't been merged yet.

Solution:

Use Git's reflog to recover the branch:

git reflog

Find the commit hash where the branch was last, then recreate the branch:

git checkout -b <branch-name> <commit-hash>

7. **Rebase Conflicts**

Description: Conflicts occur when rebasing branches that have diverged significantly.

Solution:

When a conflict occurs during rebase, Git pauses the process and allows you to resolve the conflict.

Resolve the conflict like you would for a merge conflict.

After resolving, continue the rebase with:

git rebase --continue

If you want to abort the rebase:

git rebase --abort

8. **Push Rejected Due to Protected Branch**

Description: Some teams enforce protected branches (like master or main) to prevent accidental pushes.

Solution:

Create a pull request to push changes to the protected branch.

If it's urgent, and you're allowed, you may force-push (though generally discouraged):

git push --force

9. **Reverting or Resetting Changes**

Description: Mistakenly committed or pushed wrong changes and need to undo them.

Solution:

Revert: Create a new commit to undo a previous commit, while preserving history:

git revert <commit-hash>

Reset: Move the current branch backward to a previous state:

Soft reset (keeps your changes unstaged):

git reset --soft <commit-hash>

Hard reset (discards all changes):

git reset --hard <commit-hash>

10. **'Your Branch is Ahead of Origin/Branch'**

Description: This happens when you've made local commits but haven't pushed them to the remote.

Solution:

Push your changes:

git push origin <branch-name>