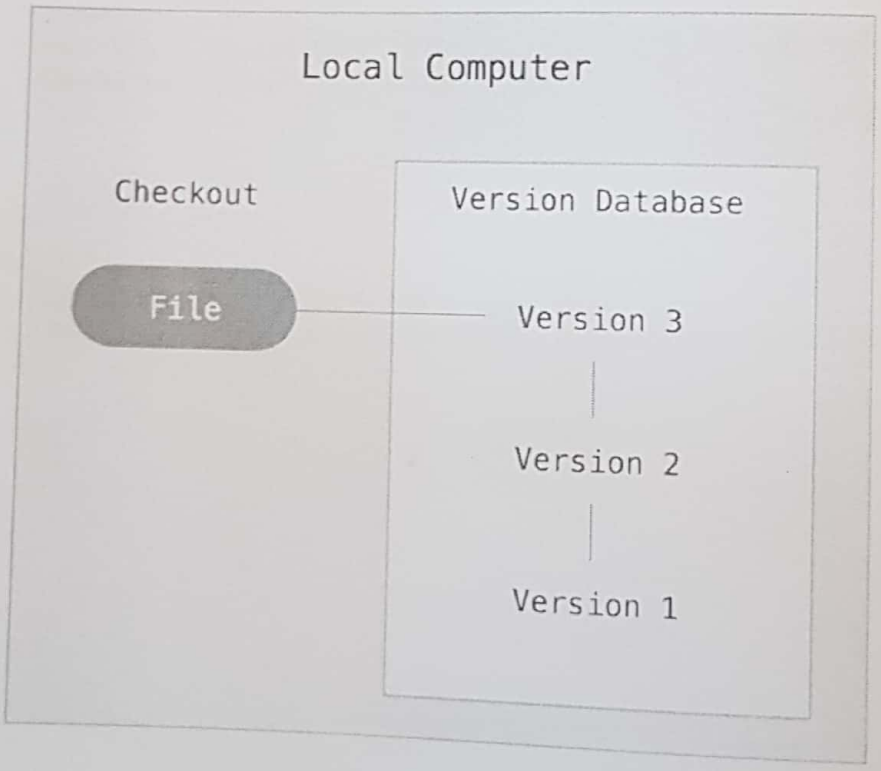
7) Version Control System setup and usage using GIT.

- Creating a repository
- Checking out a repository
- Adding content to the repository
- Committing the data to a repository
- Updating the local copy
- Comparing different revisions
- Revert
- Conflicts and Solving a conflict

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

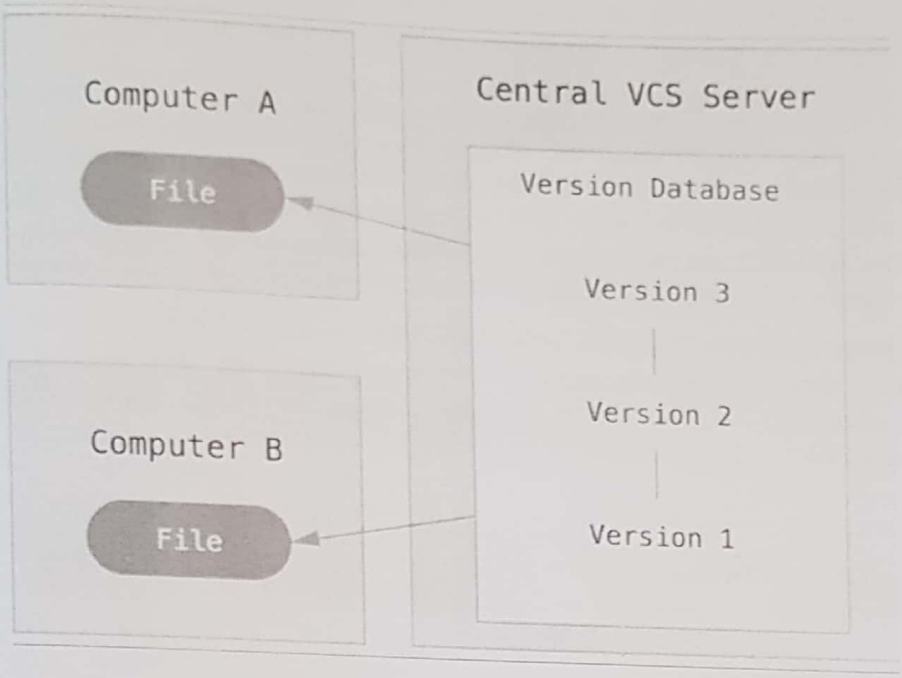
Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.



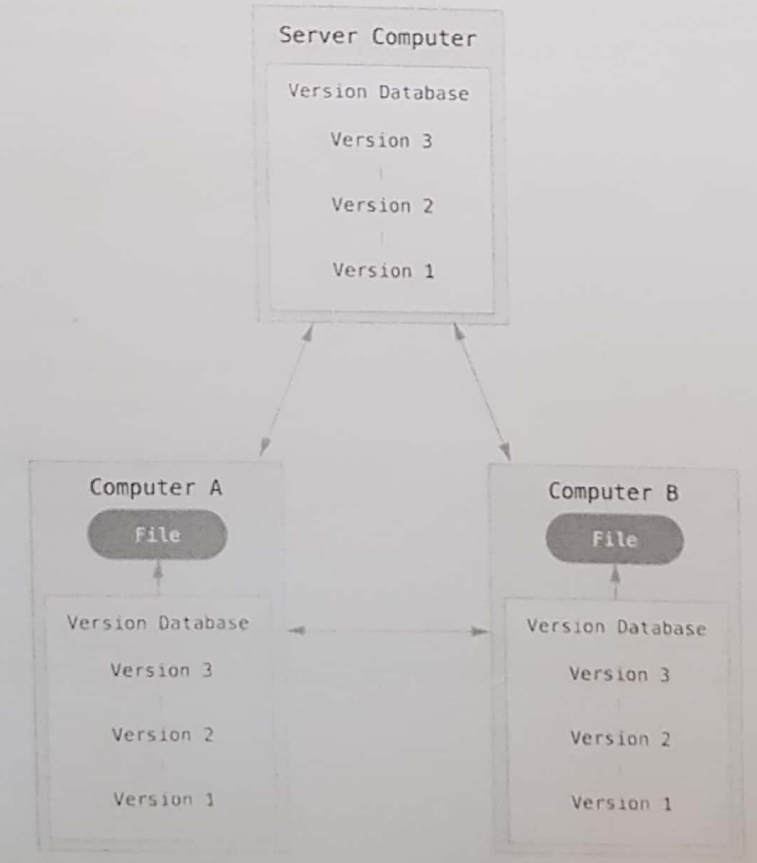
Centralized Version Control Systems

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



Distributed Version Control Systems

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



Git is the new fast-rising star of version control systems. Initially developed by Linux kernel creator Linus Torvalds, Git has recently taken the Web development community by storm. Git offers a much different type of version control in that it's a distributed version control system. With a distributed version control system, there isn't one centralized code base to pull the code from. Different branches hold different parts of the code. Other version control systems, such as SVN and CVS, use centralized version control, meaning that only one master copy of the software is used.

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time.

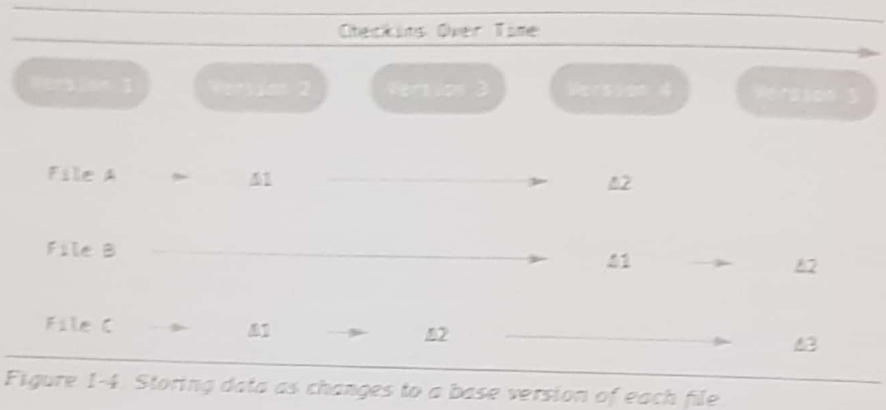


Figure 1-4. Storing data as changes to a base version of each file

Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.

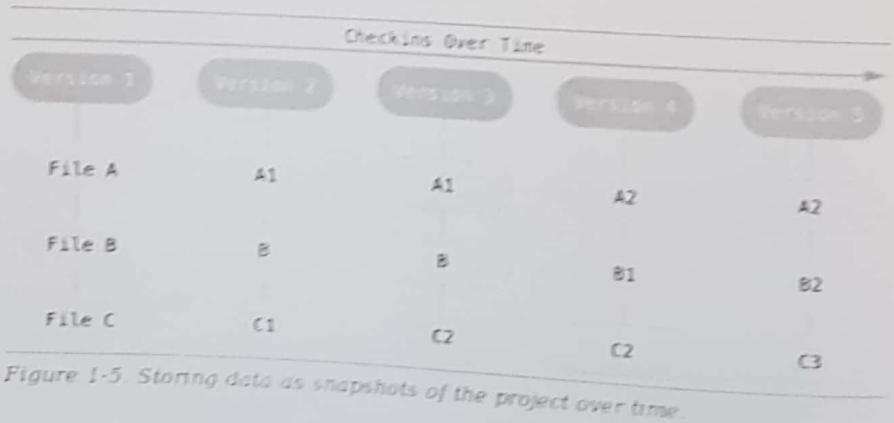


Figure 1-5. Storing data as snapshots of the project over time

Installing Git

$ sudo apt-get install git-all

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. /etc/gitconfig file: Contains values for every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.
2. ~/.gitconfig or ~/.config/git/config file: Specific to your user. You can make Git read and write to this file specifically by passing the --global option.
3. config file in the Git directory (that is, .git/config) of whatever repository you're currently using: Specific to that single repository.

Identity

The first thing you should do when you install Git is to set your user name and email address.

$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com

Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system's default editor.

If you want to use a different text editor, such as Emacs, you can do the following:

$ git config --global core.editor notepad++

Checking Your Settings

If you want to check your settings, you can use the git config --list command to list all the settings Git can find at that point:

git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...

You can also check what Git thinks a specific key's value is by typing git config <key>:
$ git config user.name
John Doe

Getting a Git Repository

You can get a Git project using two main approaches. The first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

Initializing a Repository in an Existing Directory

If you're starting to track an existing project in Git, you need to go to the project's directory and type:

$ git init
This creates a new subdirectory named .git that contains all of your necessary repository files – a Git repository skeleton. At this point, nothing in your project is tracked yet.

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'

Cloning an Existing Repository

If you want to get a copy of an existing Git repository – for example, a project you'd like to contribute to – the command you need is git clone.

$ git clone https://github.com/libgit2/libgit2

Checking the Status of Your Files

The main tool you use to determine which files are in which state is the git status command. If you run this command directly after a clone, you should see something like this:

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean

This means you have a clean working directory – in other words, there are no tracked and modified files.
You add a new file to your project, a simple README file. If the file didn't exist before, and you run git status, you see your untracked file like so:
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)

Tracking New Files

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

$ git add README

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

new file: README

Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system.

S cat .gitignore
*.[oa]
*~

Viewing Your Staged and Unstaged Changes

If the git status command is too vague for you – you want to know exactly what you changed, not just which files were changed – you can use the git diff command.

S git diff

Committing Your Changes

Now that your staging area is set up the way you want it, you can commit your changes.

S git commit
if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:
S git commit -m 'initial commit'
S git add forgotten_file
S git commit --amend

git diff-to find the change in current and previous version
git log-to list all the versions
git diff key1 key2-to find difference between two committed versions
git revert HEAD-revert the commit we just created