



UMD DATA605 - Big Data Systems

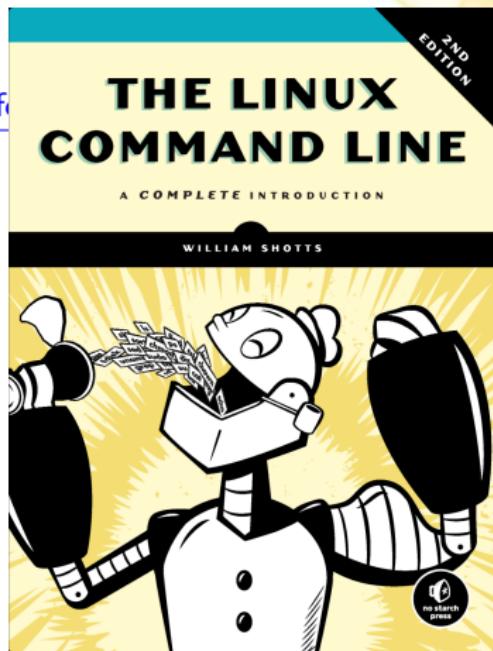
Git, Data Pipelines

Instructor: Dr. GP Saggese - gsaggese@umd.edu

- *Git*

Bash / Linux: Resources

- Command line
 - <https://ubuntu.com/tutorials/command-line-fundamentals>
- E.g., find, xargs, chmod, chown, symbolic, and hard links
- How Linux works
 - Processes
 - File ownership and permissions
 - Virtual memory
 - How to administer a Linux box as root
- Mastery
 - <https://linuxcommand.org/tlcl.php>
 -



Version Control Systems (1/2)

- A **Version Control System** (VCS) is a system that allows to:
 - Record changes to files
 - Recall specific versions later (like a “file time-machine”)
 - Compare changes to files over time
 - Track *who* changed *what* and *when* and *why*
- **Simplest "VCS"**
 - Make a copy of a dir and add
 - _v1 (bad) or
 - a timestamp _20220101 (better)
 - **Cons:** It kind of works for one person, but doesn't scale



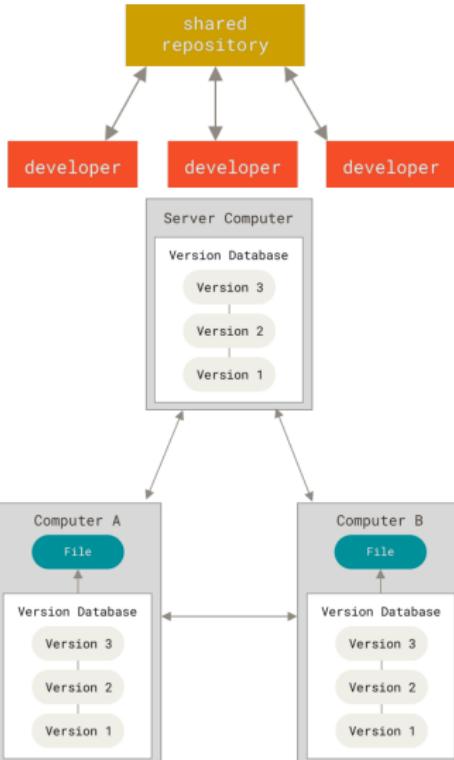
Version Control Systems (2/2)

- **Centralized VCS**

- E.g., Perforce, Subversion
- A server stores the code, clients connect to it
- **Cons:** If the server is down, nobody can work

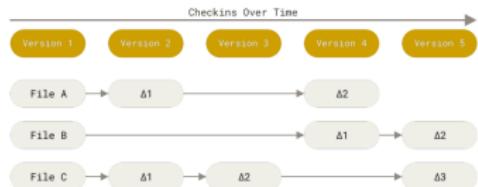
- **Distributed VCS**

- E.g., Git, Mercurial, Bazaar, Dart
- Each client has the entire history of the repo locally
- Each node is both a client and a server
- **Cons:** complex



VCS: How to Track Data

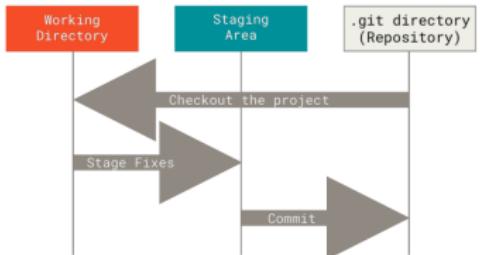
- Consider a directory with project files inside
- How do you track changes to the data?
- Delta-based VCS
 - E.g., Subversion
 - Store the data in terms of patches (changes of files over time)
 - Can reconstruct the state of the repo by applying the patches
- Stream of snapshots VCS
 - E.g., Git
 - Store data in terms of snapshots of a filesystem
 - Take a “picture” of what files look like
 - Store reference (hash) to the snapshots
 - Save link to previous identical files



- Almost everything is local for Git
 - The history is stored locally in each node
 - Diff-ing is done locally
 - For centralized VCS you need to access the server
 - You can commit to your local copy
 - Upload changes when there is network connection
- Almost everything is undoable in Git
 - No data corruption
 - Everything is checksummed
 - Nothing can be lost
 - Disclaimer:
 - As long as you commit (at least locally) or stash
 - As long as you don't precipitate in "git hell"
 - You need to know how to do it
- Git is a mini key-value store with a VCS built on top
 - This is actually exactly true
 - Two layers:
 - "porcelain": key-value store for a file-system
 - "plumbing": VCS layer

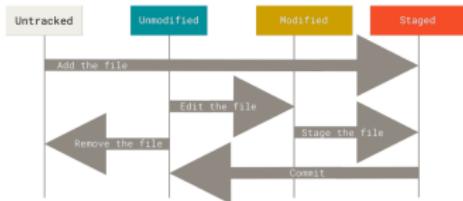
Sections of a Git Project

- There are 3 main sections of a Git project
 - Working tree (aka checkout)
 - Version of the code placed on the filesystem for the user to use and modify
 - Staging area (aka cache, index)
 - A file in .git that stores information about the next commit
 - Git directory (aka .git)
 - Store metadata and objects (like a DB)
 - It is the repo itself with all history
 - When you clone, you get the .git of the project



States of a File in Git

- Each file can be in 4 states from Git point-of-view
 - **Untracked:** files that are not under Git version control
 - **Modified:** you have changed the file, but not committed to DB yet
 - **Staged:** you have marked a modified file to go into your next commit snapshot
 - **Committed:** data is safely stored in your local DB



Git Tutorial

- Git tutorial on class repo
 - Read me
- How to use a tutorial
 - Type the commands from the tutorial one-by-one
 - Do not copy paste
 - Look at the results
 - Understand what each line means
 - Play with things
 - “What happens if I do this?”
 - “Does the result match my mental model?”
 - Learn command line before switching to the GUI
 - GUIs hide details and you become dependent on it
- Use one or more tutorials on-line
 - Ideally go through the Git book and the examples
- Build your own cheat sheet
 - Reusing other people cheat sheet works only if you already know
- Strive to achieve mastery of the basic tools of the trade
 - Bash, Git, editor
 - Python
 - Pandas

Git: Daily Use

- Check out a project (git clone) or start from scratch (git init)
 - Only once per Git project client
- Daily routine
 - Modify files in working tree (vi ...)
 - Add files (git add ...)
 - Stage changes for the next commit (git add -u ...)
 - Commit changes to .git (git commit)
- Use a branch to group commits together
 - Isolate your code from changes in master
 - Merge master into your branch
 - Isolate master from your changes
 - Pull Request (aka PR) to get the code reviewed
 - Merge PR into upstream

Git Remote

- Remote repos are versions of the project hosted on Internet/remote file system
 - To collaborate you need to manage remote repos
 - Push / pull changes
- You can have multiple forks of the same repo with different policies
 - E.g., read-only, read-write
- `git remote -v`: show what are the remotes
- `git fetch`: pull from the remote repo all the data (e.g., branches, commits) that you don't have locally
- `git pull`: short hand from `git fetch origin + git merge master --rebase`
- `git push <REMOTE> <BRANCH>`: push local data to a remote
 - E.g., `git push origin master`
- If somebody pushed to the remote, you can't push your changes right away, but you need to:
 - Fetch the changes
 - Merge changes to the branch in your client
 - Resolve conflicts, if needed
 - (Test project sanity, e.g., by running unit tests)
 - Push changes to the remote

Git Tagging

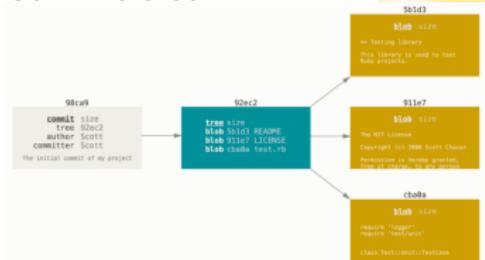
- Git allows to mark specific points in history with a tag
 - E.g., release points
- You can check out a tag
- You get in detached HEAD state
 - If you commit your change won't be added to the tag or to the branch
 - The commit will be "unreachable", i.e., reachable only by the commit hash



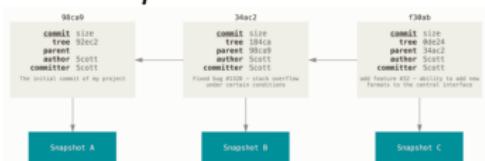
Git Internals

- You can understand Git only if you understand the data model
 - Git is a key-value store with a VCS user interface on top of it
 - Key = hash of a file
 - Value = content of a file
- Git objects
 - Commits: store pointers to the tree and commit metadata
 - Trees: represent directories and mapping between files and blobs
 - Blobs: content of files
- Refs:
 - Easy: Understanding Git — Data Model
 - Hard-core: Git internals

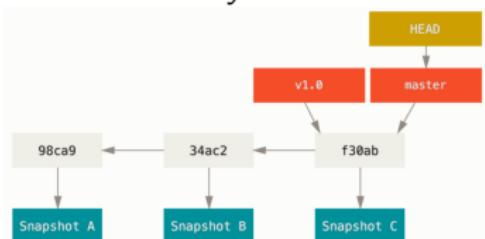
Commit tree



Commit parents

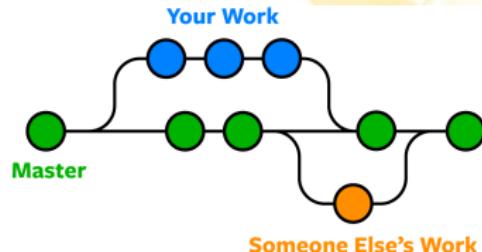


Commit history of a branch



Git Branching

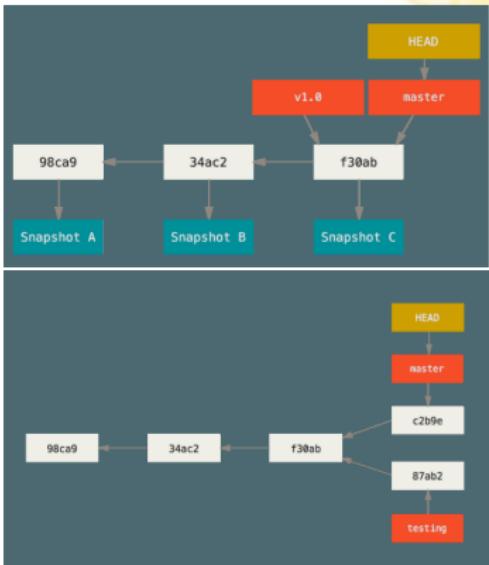
- **Branching**
 - = diverging from the main line of development
- **Why branch?**
 - Work without messing the rest of the code
 - Work without being affected by changes in the main branch
 - Merge the code downstream to pick up the changes
 - Once you are done working in your branch, merge code upstream
- **Git branching is lightweight**
 - It's instantaneous
 - A branch is just a pointer to a commit
 - Git doesn't store data as difference of files, but as a series of snapshot
- **Git workflows branch and merge often**
 - Even multiple times a day
 - This is surprising for people used to distributed VCS



E.g., you would start branching before going for lunch

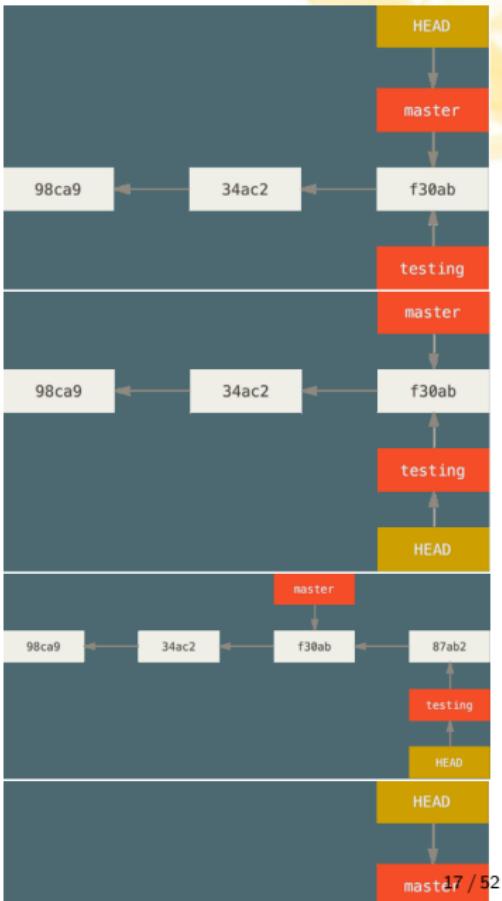
Git Branching

- master (or main) is just a normal branch
 - master is a pointer to the last commit
 - As you commit the pointer moves forward
- HEAD
 - Pointer to the local branch you are on
 - E.g., master, testing
 - git checkout <BRANCH> moves you across branches
- git branch testing
 - Create a new pointer testing
 - Point to the commit you are on
 - Pointer can be moved around
- Divergent history
 - When work progresses in two branches that are “split”



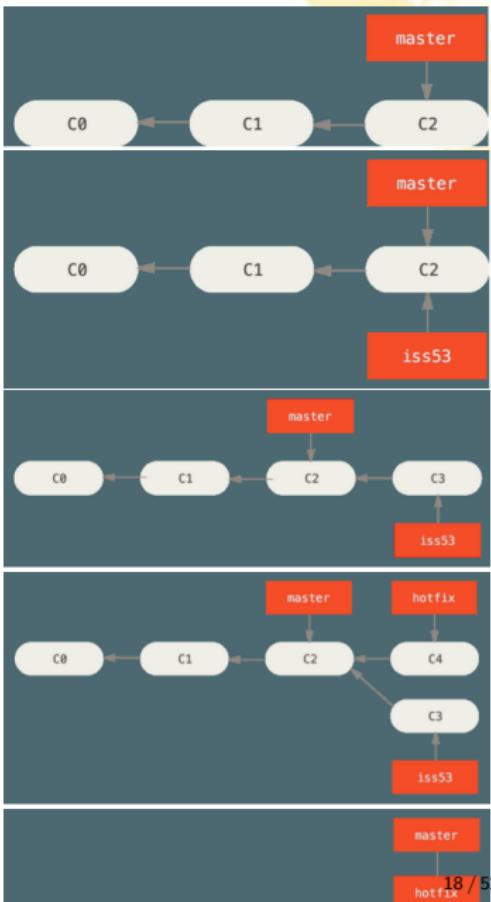
Git Checkout

- git checkout switches branch
 - Move HEAD pointer to the new branch
 - Change the files in the working dir to match the state corresponding to the branch pointer
- E.g., there are two branches master and testing
 - You are on master
 - git checkout testing
 - The pointer moves, the working dir is changed (not really in this case)
 - Then you can keep working by committing on testing
 - The pointer to testing moves forward (no divergent history)



Git Branching and Merging

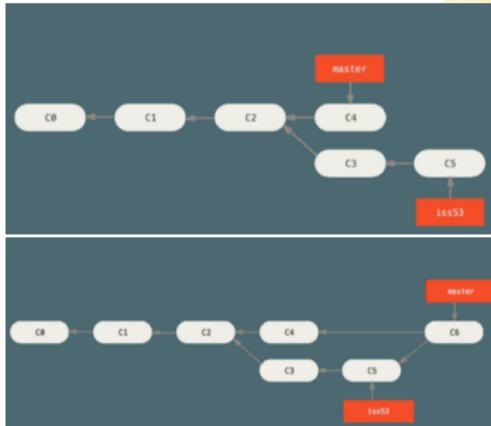
- Tutorials: Work on main, Hot fix
- Start from a project with some commits
- Branch to work on a new feature "Issue 53"
 - > git checkout -b iss53
 - **work ... work ... work**
 - > git commit
- Need a hotfix to master
 - > git checkout master
 - > git checkout -b hotfix
 - **fix ... fix ... fix**
 - > git commit -am "Hot fix"
 - > git checkout master
 - > git merge hotfix
- Fast forward
 - Now there is a divergent history between master and iss53



Git Branching and Merging

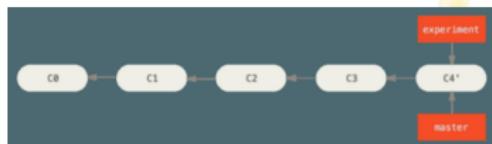
```
> git checkout iss53  
**work ... work ... work**
```

- The branch keeps diverging
- At some point you are done with iss53
- You want to merge your work back to master
- Go to the target branch
 - > git checkout master
 - > git merge iss53
- Git can't fast forward
- Git creates a new snapshot with the 3-way "merge commit" (commit with more than one parent)
- Delete the branch > git branch -d iss53



Fast Forward Merge

- Fast forward merge = merge a commit X with a commit Y that can be reached by following the history of commit X**
- There is not divergent history to merge
- Git simply moves the branch pointer forward from X to Y
- **Mental model:** a branch is just a pointer that says where the tip of the branch is**
- E.g., C4' is reachable from C3
 - > git checkout master
 - > git merge experiment
- Git moves the pointer of master to C4'



Merging Conflicts

- Tutorial:
 - Merging conflicts
- Sometimes Git can't merge, e.g.,
 - The same file has been modified by both branches
 - One file was modified by one branch and deleted by another
- Git:
 - Does not create a merge commit
 - Pauses to let you resolve the conflict
 - Adds conflict resolution markers
- User merges manually
 - Edit the files `git mergetool`
 - `git add` to mark as resolved**
 - `git commit`
 - Use PyCharm

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
please contact us at support@github.com
</div>
>>>>> iss53:index.html
```

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   index.html
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)
```

```
Changes to be committed:
```

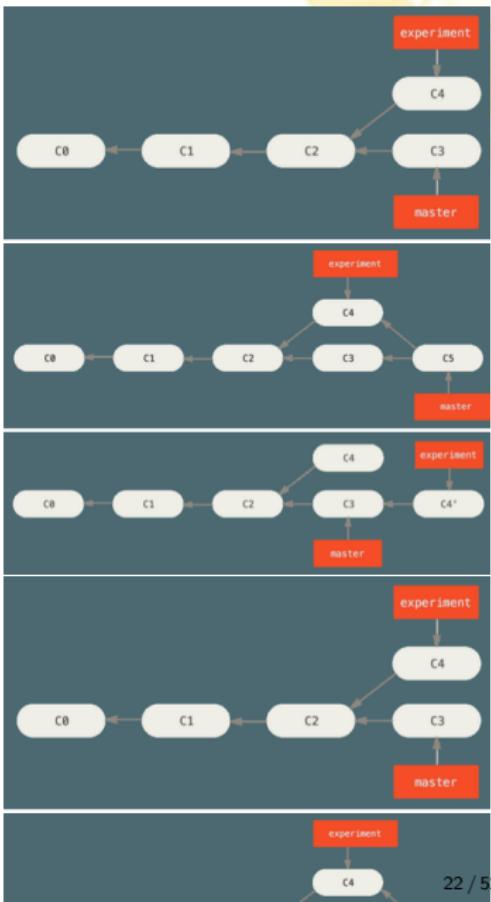
```
modified:   index.html
```

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
please contact us at support@github.com
</div>
>>>>> iss53:index.html
```



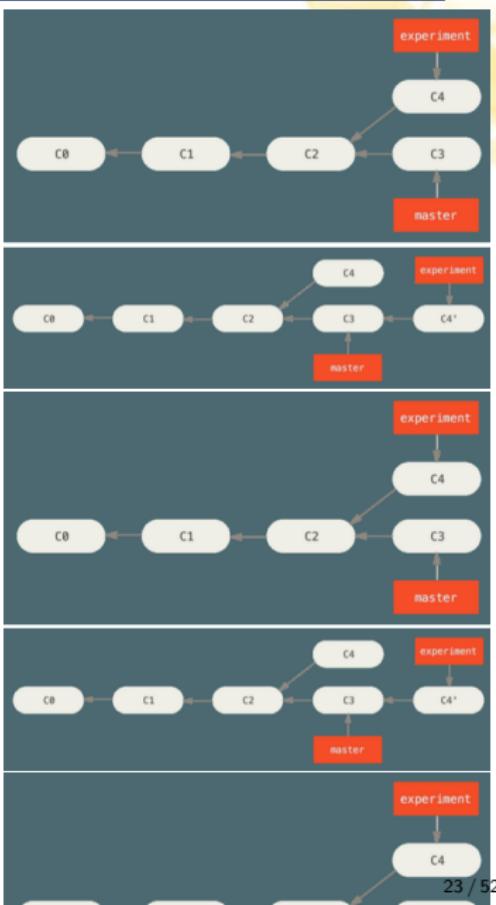
Git Rebasing

- In Git there are two ways of merging divergent history
 - E.g., master and experiment have a common ancestor C2
- Merge**
 - Go to the target branch
 - > git checkout master
 - > git merge experiment
 - Create a new snapshot C5 and commit
- Rebase**
 - Go to the branch to rebase
 - > git checkout experiment
 - > git rebase master
 - Rebase algo:
 - Get all the changes committed in the branch (C4) where we are on (experiment) since the common ancestor (C2)
 - Sync to the branch that we are rebasing onto (master at C3)
 - Apply the changes C4'
 - Only the branch where we are is affected
 - Finally fast forward master



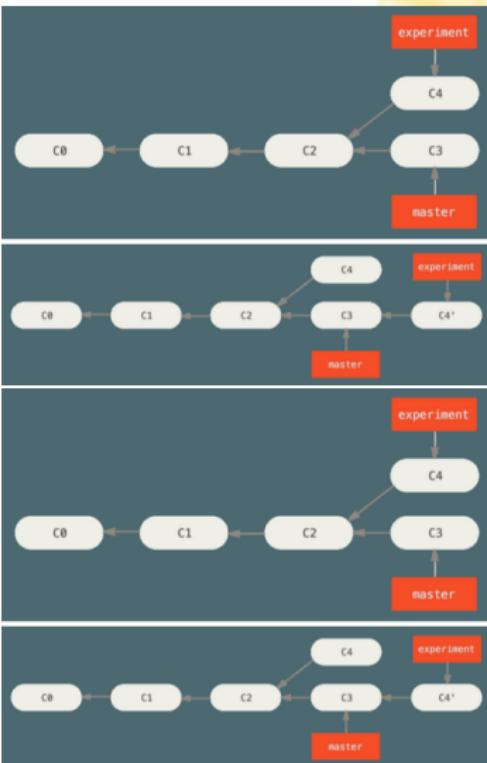
Uses of Rebase

- Rebasing makes for a cleaner history
 - The history looks like all the work happened in series
 - Although in reality it happened in parallel to the development in master
- Rebasing to contribute to a project
 - Developer
 - You are contributing to a project that you don't maintain
 - You work on your branch
 - When you are ready to integrate your work, rebase your work onto **origin/master**
 - The maintainer
 - Does not have to do any integration work
 - Does just a fast forward or a clean apply (no conflicts)



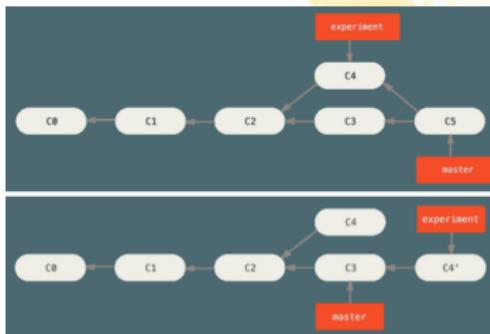
Golden Rule of Rebasing

- **Remember:** rebasing means abandoning existing commits and creating new ones that are similar but different
- **Problem**
 - You push your commits to a remote somewhere
 - Others pull your commits down and base their work on them
 - You rewrite those commits with `git rebase`
 - You push them again with `git push --force`
 - Your collaborators have to re-merge their work
- **Solution Strict version:** “Do not **ever** rebase commits that exist outside your repository”
Loose version: “It’s ok to rebase your branch even if you pushed to a server, as long as you are the only one to use it”



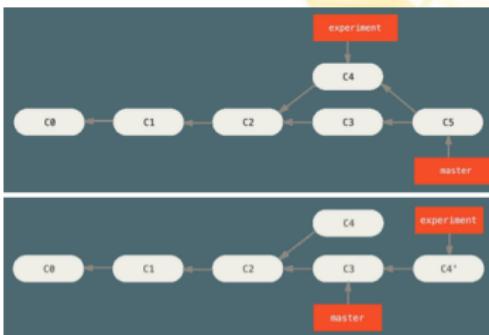
Rebase vs Merge: Philosophical Considerations

- Rebase-vs-merge depend on the answer to the question:
 - What does the commit history of a repo mean?
 - a) History is the record of what actually happened
 - *“History should not be tampered with!”**
 - Q: What if there is a series of messy merge commits?
 - A: This is how it happened. The repo should preserve this
 - Use git merge
 - b) History represents how a project should have been made
 - “You would not publish a book as a sequence of drafts and correction, but rather the final version”
 - You should tell the history in the way that is best for future readers
- Use git rebase and filter-branch**



Rebase vs Merge: Philosophical Considerations

- Many man-centuries have been wasted discussing rebase-vs-merge at the watercooler
 - Total waste of time! Tell people to get back to work!
- When you contribute to a project often people decide for you based on their preference
- **Best of the merge-vs-rebase approaches**
- Rebase changes you've made in your local repo
 - Even if you have pushed but you know the branch is yours
 - Use `git pull --rebase` to clean up the history of your work
 - If the branch is shared with others then you need to definitively `git merge`
- Only `git merge` to master to preserve the history of how something was built
- **Personally**
- I like squash-and-merge branches to master
 - My commits are just my checkpoints
 - Rarely they are "complete"



Remote Branches

- Remote branches are pointers to branches in remote repos

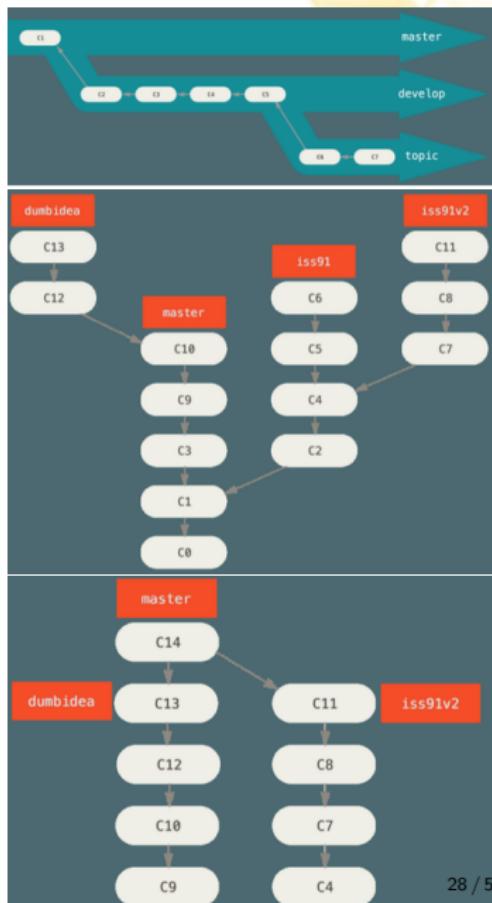
```
git remote -v
origin  git@github.com:gpsaggese/umd_data605.git (fetch)
origin  git@github.com:gpsaggese/umd_data605.git (push)
```

- Tracking branches

- Local references representing the state of the remote repo
- E.g., master tracks **origin/master**
- You can't change the remote branch (e.g., **origin/master**)
- You can change tracking branch (e.g., **master**)
- Git updates tracking branches when you do `git fetch origin` (or `git pull`)
- To share code in a local branch you need to push it to a remote
 - > `git push origin serverfix`
- To work on it
 - > `git checkout -b serverfix origin/serverfix`

Git Workflows

Git workflows - = ways of working and collaborating using Git **Long-running branches** - Branches at different level of stabilities, that are always open - **master** is always ready to be released - **develop** branch to develop in - *topic / feature* branches - When branches are “stable enough” they are merged up **Topic branches** - Short-lived branches for a single feature - E.g., **hotfix**, **wip-XYZ** - Easy to review - Silo-ed from the rest - This is typical of Git since other VCS support for branches is not good enough - E.g., - You start **iss91**, then you cancel some stuff, and go to **iss91v2** - Somebody starts **dumbidea** branch and merge to **master** (!) - **You squash-and-merge your iss91v2****



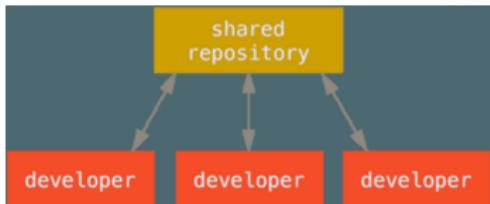
Centralized Workflow

Centralized workflow in centralized VCS -

Developers: - Check out the code from the central repo on their computer - Modify the code locally - Push it back to the central hub (assuming no conflicts with latest copy, otherwise they need to merge)

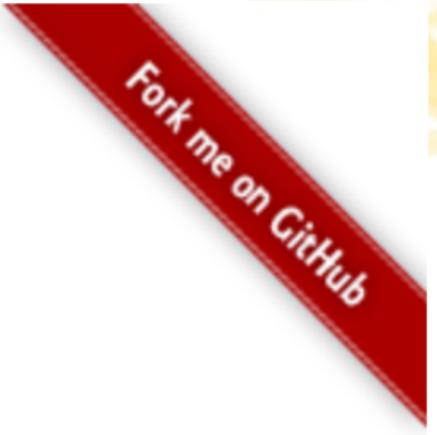
Centralized workflow in Git - Developers:

- Have push (i.e., write) access to the central repo - Need to fetch and then merge - Cannot push code that will overwrite each other code (only fast-forward changes)



Forking Workflows

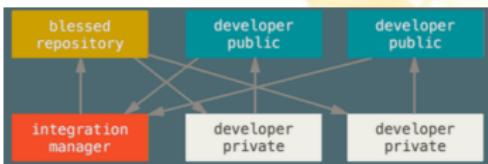
- Typically devs don't have permissions to update directly branches on a project
 - Read-write permissions for core contributors
 - Read-only for anybody else
- Solution
 - "Forking" a repo
 - External contributors
 - Clone the repo and create a branch with the work
 - Create a writable fork of the project
 - Push branches to fork
 - Prepare a PR with their work
 - Project maintainer
 - Reviews PRs
 - Accepts PRs
 - Integrates PRs
 - In practice it's the project maintainer that pulls the code when it's ready, instead of external contributors pushing the code
- Aka "GitHub workflow"
 - "Innovation" was forking (Fork me on GitHub!)
 - GitHub acquired by Microsoft for 7.5b USD



Fork me on GitHub

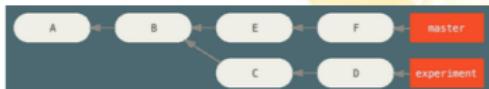
Integration-Manager Workflow

- This is the classical model for open-source development
 - E.g., Linux, GitHub (forking) workflow
- 1. One repo is the "official" project
 - Only the project maintainer pushes to the public repo
 - E.g., sorrentum/sorrentum
- 2. Each contributor
 - Has read access to everyone else's public repo
 - Forks the project into a private copy
 - Write access to their own public repo
 - E.g., [gpsaggese/sorrentum](#)
 - Makes changes
 - Pushes changes to his own public copy
 - Sends email to maintainer asking to pull changes (pull request)
- 3. The maintainer
 - Adds contributor repo as a remote
 - Merges the changes into a local branch
 - Tests changes locally
 - Pushes branch to the official repo



Git log

- git log reports info about commits**
- refs are references to:
 - HEAD (commit you are working on, next commit)
 - origin/master (remote branch)
 - experiment (local branch)
 - d921970 (commit)
- after a reference resolves to the parent of that commit
- HEAD = commit before HEAD, i.e., last commit
- 2 means
- A merge commit has multiple parents
- **Double-dot notation**
 - `1..2` = commits that are reachable from 2 but not from 1 (difference)**
 - `git log master..experiment \to D,C`
 - `git log experiment..master \to F,E`
- **Triple-dot notation**
 - `1...2` = commits that are reachable from either branch but not from both (union excluding intersection)**



Advanced Git

- stashing
 - Copy state of your working dir (e.g., modified and staged files), save it in a stack, to apply later
- cherry-picking
 - Rebase for a single commit
- rerere
 - = “Reuse Recorded Resolution”
 - Git caches how to solve certain conflicts
- tagging
 - Give a name to a specific commit (e.g., v1.3)
- submodules / subtrees
 - Project including other Git projects
- bisect
 - Sometimes a bug shows up at top of tree
 - You don't know at which revision it started manifesting
 - You have a script that returns 0 if the project is good and non-0 if the project is bad
 - `git bisect` can find the revision at which the script goes from good to bad**
- filter branch
 - Rewrite repo history in some scriptable way



SCIENCE
ACADEMY

GitHub

- GitHub acquired by MSFT for 7.5b
- **GitHub: largest host for Git repos**
 - Git hosting (100m+ open source projects)
 - PRs, forks
 - Issue tracking
 - Code review
 - Collaboration
 - Wiki
 - Actions (CI / CD)
- **"Forking a project"**
 - In open-source communities
 - It had a negative connotation
 - Take a project, modify it, and make it a competing project
 - In GitHub parlance
 - Make a copy of a project so that you can contribute to it even if you don't have push / write access



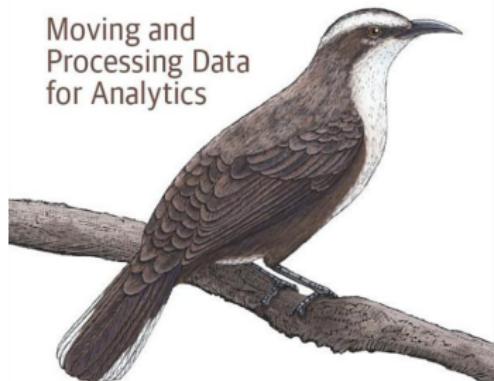
Data Pipelines - Resources

- Concepts in the slides
- Class project
- Mastery
 - Data Pipelines Pocket Reference: Moving and Processing Data for Analytics

O'REILLY®

Data Pipelines Pocket Reference

Moving and
Processing Data
for Analytics



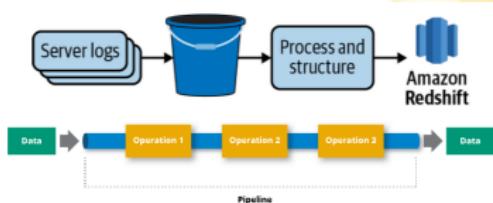
James Densmore

Data as a Product

- Many services today "sell" data
 - Services are typically powered by data and machine learning
 - Data products, e.g.,
 - Personalized search engine (Google)
 - Sentiment analysis on user-generated data (Facebook)
 - A recommendation engine + e-commerce (Amazon)
 - Streaming data (Netflix, Spotify)
- Several steps are required to generate data products
 - Data ingestion
 - Data pre-processing
 - Cleaning, tokenization, feature computation
 - Model training
 - Model deployment
 - MLOps
 - Model monitoring
 - Is model working?
 - Is model getting slower?
 - Are model performance getting worse?
 - Collect feedback from deployment
 - E.g., recommendations vs what users bought
 - Ingest data from production for future versions of the model

Data Pipelines

- “Data is the new oil”
 - ... but oil needs to be refined
- **Data pipelines**
 - Processes that move and transform data
 - **Goal:** derive new value from data through analytics, reporting, machine learning**
- Data needs to be:
 - Collected
 - Pre-processed / cleaned
 - Validated
 - Processed
 - Combined
- **Data ingestion**
 - Simplest data pipeline
 - Extract data (e.g., from REST API)
 - Load data into DB (e.g., SQL table)



Roles in Building Data Pipelines

- Data engineers
 - Build and maintain data pipelines
 - Tools:
 - Python / Java / Go / No-code
 - SQL / NoSQL stores
 - Hadoop / MapReduce / Spark
 - Cloud computing
- Data scientists
 - Build predictive models
 - Tools:
 - Python / R / Julia
 - Hadoop / MapReduce / Spark
 - Cloud computing
- Data analysts
 - E.g., marketing, MBAs, sales, . . .
 - Build metrics and dashboards
 - Tools:
 - Excel spreadsheets
 - GUI tools (e.g., Tableaux)
 - Desktop

• Recurring practical problems

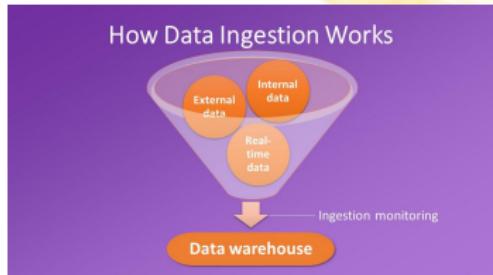
• Who is responsible for the data?

• Issues with scaling



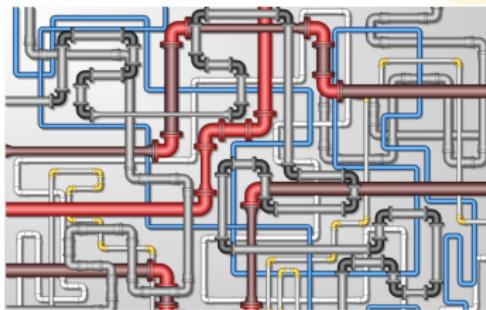
Data Ingestion

- Data ingestion
 - = extract data from one source and load it into another store
- Data sources / sinks
 - DBs
 - E.g., Postgres, MongoDB
 - REST API
 - Abstraction layer on top of DBs
 - Network file system / cloud
 - E.g., CSV files, Parquet files
 - Data warehouses
 - Data lakes
- Source ownership
 - An organization can use 10-1000s of data sources
 - Internal
 - E.g., DB storing shopping carts for a e-commerce site
 - 3rd-parties
 - E.g., Google analytics tracking website



Data Pipeline Paradigms

- There are several styles of building data pipelines
- Multiple phases
 - Extract
 - Load
 - Transform
- Phases arranged in different ways depending on philosophy about data / roles
 - ETL
 - ELT
 - EtLT



ETL Paradigm: Phases

- Extract

- Gather data from various data sources, e.g.,
 - Internal / external data warehouse
 - REST API
 - Data downloading from API
 - Web scraping

- Transform

- Raw data is combined and formatted to become useful for analysis step

- Load

- Move data into the final destination, e.g.,
 - Data warehouse
 - Data lake

- **Data ingestion pipeline = E + L**

- Move data from one point to another
- Format the data
- Make a copy
- Have different tools to operate on the data



SCIENCE
ACADEMY

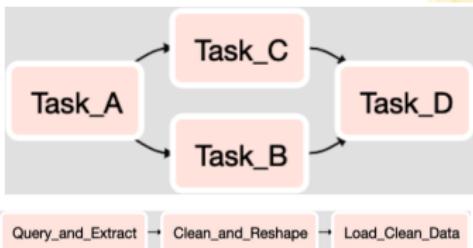
ETL Paradigm: Example

- Extract
 - Buy-vs-build data ingestion tools
 - Vendor lock-in
- Transform
 - Data conversion (e.g., parsing timestamp)
 - Create new columns from multiple source columns
 - E.g., year, month, day -> yyyy/mm/dd
 - Aggregate / filter through business logic
 - Try not to filter, better to mark
 - Anonymize data
- Load
 - Organize data in a format optimized for data analysis
 - E.g., load data in relational DB
 - Finally data modeling



Workflow Orchestration

- Companies have many (10-1000s) data pipelines
- Orchestration tools, e.g.,
 - Apache Airflow (from AirBnB)
 - Luigi (from Spotify)
 - AWS Glue
 - Kubeflow
- Schedule and manage flow of tasks according to their dependencies
 - Pipeline and jobs are represented through DAGs
- Monitor, retry, and send alarms



ELT paradigm

- ETL has been the standard approach for long time

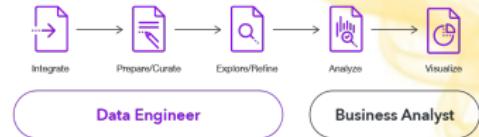
- Extract → Transform → Load
- **Cons**
 - Need to understand the data at ingestion time
 - Need to know how the data will be used

- Today ELT is becoming the pattern of choice**

- Extract → Load → Transform
- **Pro:**
 - No need to know how the data will be used
 - Separate data engineers and data scientists / analysts
 - Data engineers focus on data ingestion (E + L)
 - Data scientists focus on transform (T)

- **ETL → ELT enabled by new technologies**

- Large storage to save all the raw data (cloud computing)
- Distributed data storage and querying (e.g., Columnar DBs)



Row-based vs Columnar DBs

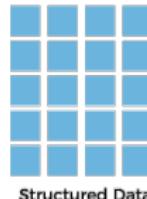
- Row-based DBs
 - E.g., MySQL, Postgres
 - Optimized for reading / writing rows
 - Read / write small amounts of data frequently
- Columnar DBs
 - E.g., Amazon Redshift, Snowflake
 - Read / write large amounts of data infrequently
 - Analytics requires a few columns
 - Better data compression

OrderId	CustomerId	ShippingCountry	OrderTotal
1	1258	US	55.25
2	5698	AUS	125.36
3	2265	US	776.95
4	8954	CA	32.16
Block 1	1, 1258, US, 55.25		
Block 2	2, 5698, AUS, 125.36		
Block 3	3, 2265, US, 776.95		
Block 4	4, 8954, CA, 32.16		

- ETL
 - Extract → Transform → Load
- ELT
 - Extract → Load → Transform
 - Transformation / data modeling ("T") according to business logic
- EtLT
 - Sometimes transformations with limited scope ("t") are needed
 - De-duplicate records
 - Parse URLs into individual components
 - Obfuscate sensitive data (for legal or security reasons)
 - Then implement rest of "LT" pipeline

Structure in Data (or Lack Thereof)

- **Structured data:** there is a schema
 - Relational DB
 - CSV
 - DataFrame
 - Parquet
- **Semi-structured:** subsets of data have different schema
 - Logs
 - HTML pages
 - XML
 - Nested JSON
 - NoSQL data
- **Unstructured:** no schema**
 - Text
 - Pictures
 - Movies
 - Blobs of data



Structured Data

VS

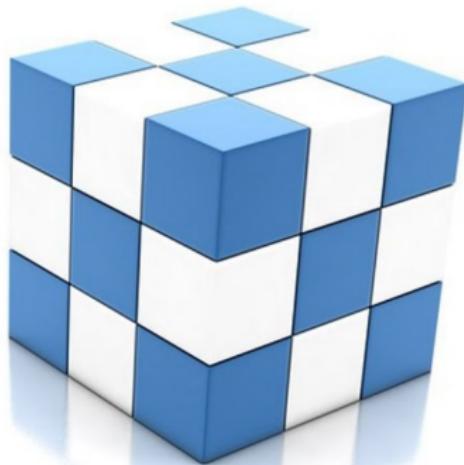


Unstructured Data

OLAP vs OLTP Workloads

- There are two classes of data workloads
- **OLTP**
 - On-Line Transactional Processing
 - Execute large numbers of transactions by a large number of processes in real-time
 - **Lots of concurrent small read / write transactions**
 - E.g., online banking, e-commerce, travel reservations
- **OLAP**
 - On-Line Analytical Processing
 - Perform multi-dimensional analysis on large volumes of data
 - **Few large read or write transactions**
 - E.g., data mining, business intelligence

OLAP



OLTP

Challenges with Data Pipelines

- High-volume vs low-volume
 - Lots of small reads / writes
 - A few large reads / writes
- Batch vs streaming
 - Real-time constraints
- API rate limits / throttling
- Connection time-outs
- Slow downloads
- Incremental mode vs catch-up

Data Warehouse vs Data Lake

- **Data warehouse**

- = DB storing data from different systems in a structured way
- Corresponds to ETL data pipeline style
- E.g., a large Postgres instance with many DBs and tables
- E.g.,
 - AWS Athena, RDS
 - Google BigQuery



- **Data lake**

- = data stored in a semi-structured or without structure
- Corresponds to ELT data pipeline style
- E.g., an AWS S3 bucket storing blog posts, flat files, JSON objects, `data605/lectures_source/images`

Data Lake: Pros and Cons

- Data lake = stores data in a semi-structured or without structure
- Pros
 - Storing data in cloud storage is cheaper
 - Making changes to types or properties is easier since it's unstructured or semi-structured (with no predefined schema)
 - E.g., JSON documents
 - Data scientists
 - Don't know initially how to access and use the data
 - Want to explore the raw data
- Cons
 - It is not optimized for querying like a structured data warehouse
 - There are tools that allow to query data in a data lake similar to SQL
 - E.g., AWS Athena, Redshift Spectrum

Advantages of Cloud Computing

- Ease of building and deploying:
 - Data pipelines
 - Data warehouses
 - Data lakes
- Managed services
 - No need for admin and deploy
 - Highly scalable DBs
 - E.g., Amazon Redshift, Google BigQuery, Snowflake
- Rent-vs-buy
 - Easy to scale up and out
 - Easy to upgrade
 - Better cash-flow
- Cost of storage and compute is continuously dropping
 - Economies of scale
- Cons
 - The flexibility has a cost (2x-3x more expensive than owning)
 - Vendor lock-in