UMD DATA605 - Big Data Systems

Git

Data Pipelines

Dr. GP Saggese
gsaggese@umd.edu

with thanks to Prof.
Alan Sussman
Amol Deshpande

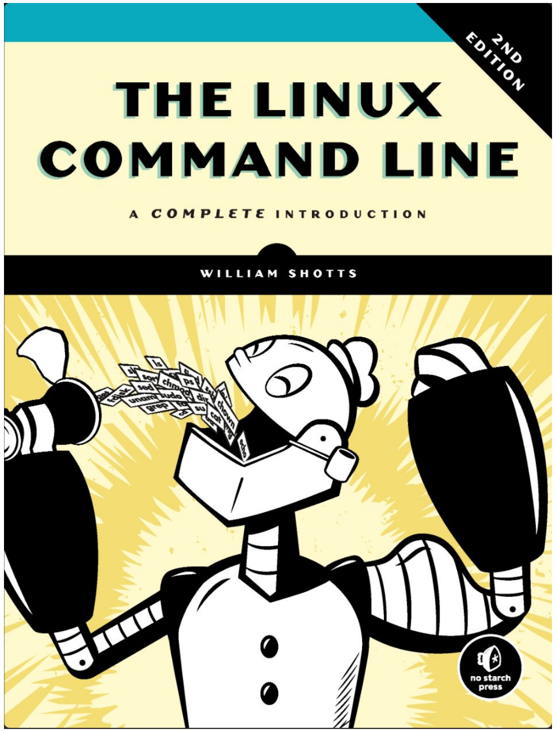
UMD DATA605 - Big Data Systems

Git

Data Pipelines

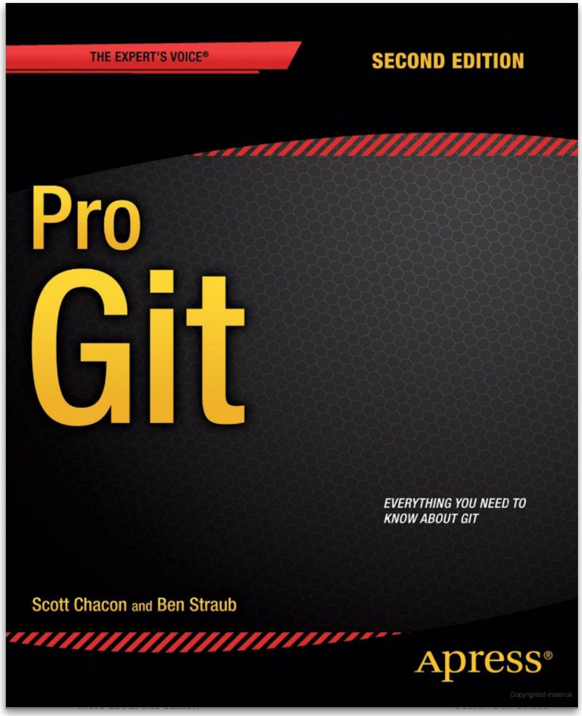
Bash / Linux: Resources

- Command line
  - https://ubuntu.com/tutorials/command-line-for-beginners
  - `find, xargs, chmod, chown`
  - symbolic and hard links
- How Linux works
  - Process
  - File ownership and permission
  - Virtual memory
  - How to administer a Linux box as root
- Mastery
  - https://linuxcommand.org/tlcl.php



Git: Resources

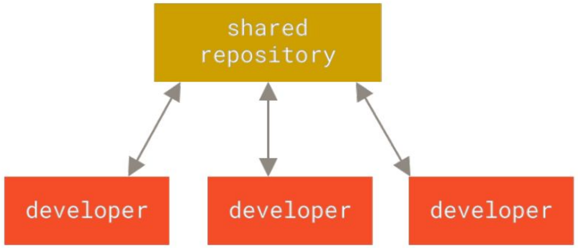
- Concepts in the slides
- Tutorial: tutorial_git
- We will use Git during the project
- Mastery: Pro Git (free)
- Web resources:
  - dangitgit.com (aka Oh Sh*t, Git!?!)

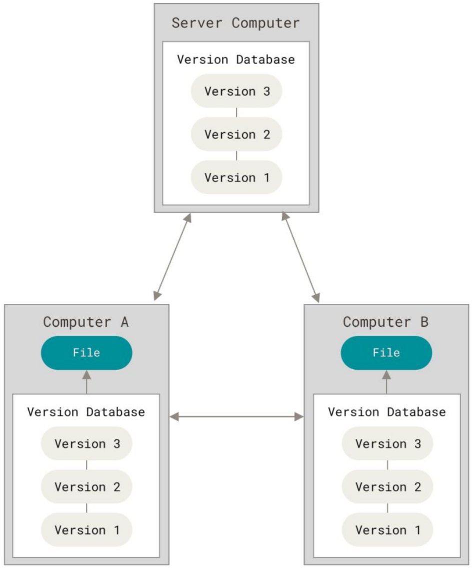


Version Control Systems

- A **VCS** is a system that allows to:
  - Record changes to files
  - Recall specific versions later (like a time-machine)
  - Compare changes over time
  - Track who changed what and when

- **Simplest "VCS"**
  - Make a copy of a dir and add `_v1` (bad) or add a timestamp `_20220101` (better)
  - It kind of works for one person, but doesn't scale
- **Centralized VCS**
  - E.g., Perforce, Subversion
  - There is a server storing the code, clients that connect to it
  - 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 client is a repo server

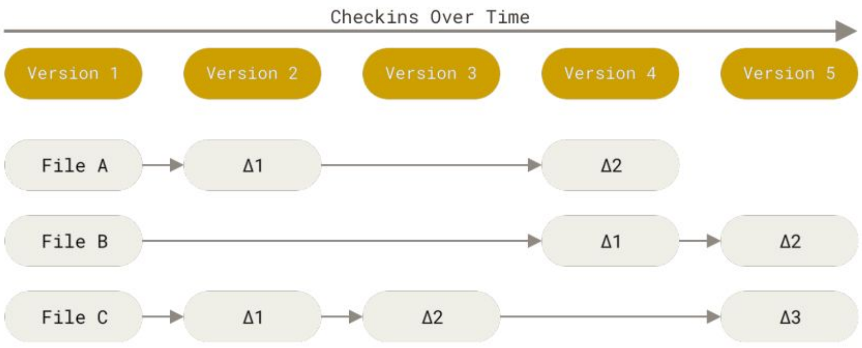




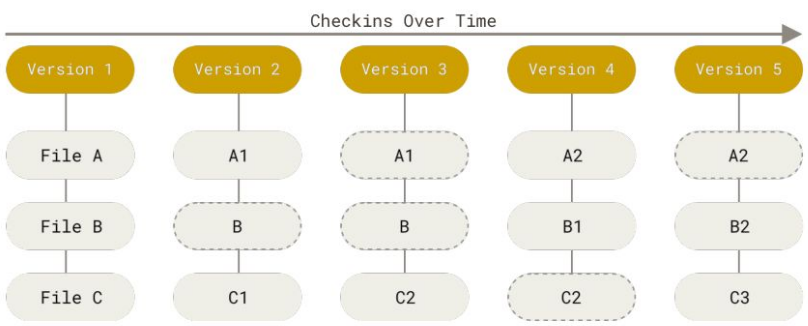
VCS: How to Track Data

- There is a directory with project files inside
- How do you track changes?

- **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
  - Data in terms of snapshots of a filesystem
  - Take a "picture" of what files look like
  - Store reference to the snapshots
  - Save link to previous identical files



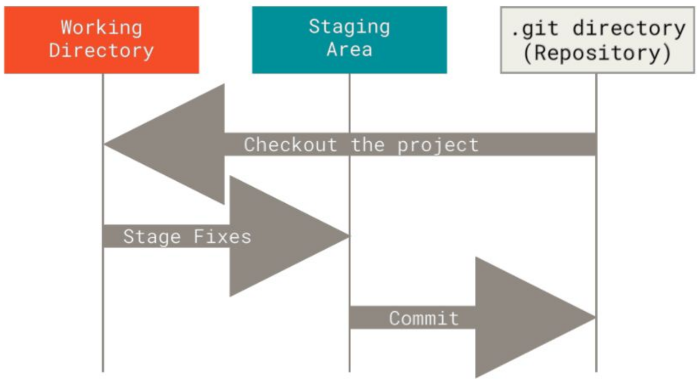
Git

- Almost everything is **local** for Git
  - History is stored locally
  - Diff-ing is done locally
  - 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
    - As long as you commit
    - Ideally, besides “git hell”
    - You need to know how to do it
- Git is like a mini file-system / key-value store with a VCS built on top
  - Actually this is exactly true
  - Two layers: “porcelain” and “plumbing”

States of a File in Git

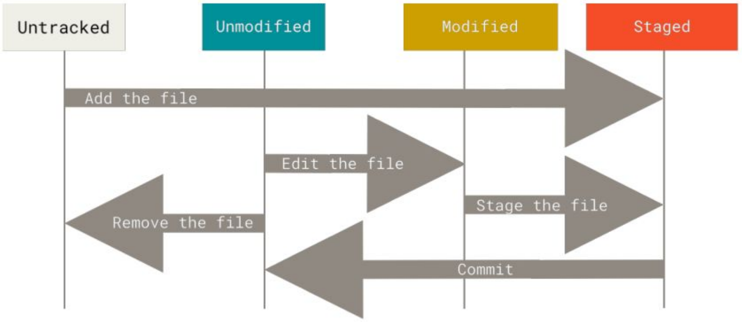
There are 3 main sections of a Git project:

- **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



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 in its current version to go into your next commit snapshot
- **Committed**: data is safely stored in your local DB



Git Tutorial

- [Git tutorial]

Git: Daily Use

- Check out the project (`git clone`) or start from scratch (`git init`)
  - Only once per project or per 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
  - PR to get the code reviewed
  - Merge PR into upstream

Git Remote

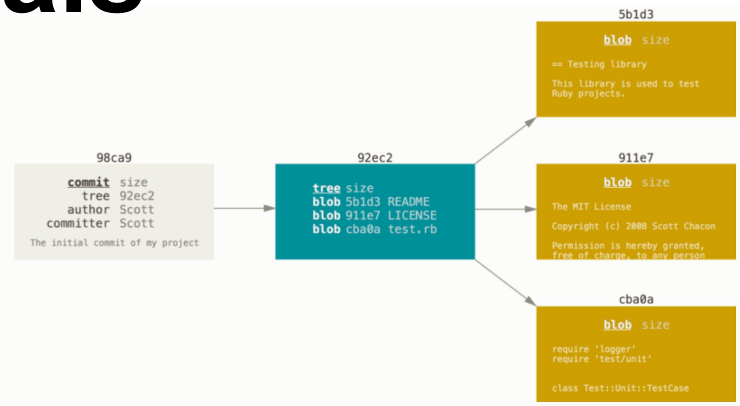
- Remote repos are versions of the project hosted on Internet or on a “remote” file system
  - To collaborate you need to manage remote repos
  - Push / pull changes
- You can have multiple repos with different policies
  - E.g., read-only, read-write
- `git remote -v`: show what are the remotes
- `git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

Git Tagging

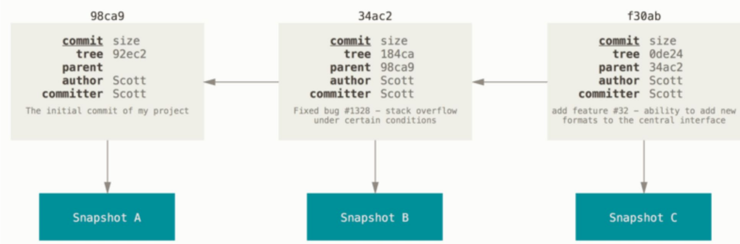
- Git allows to mark specific points in history with a tag
  - E.g., release points
- You can check out a tag
  - You get in detached HEAD state
  - If you commit your change won't be added to the tag or to the branch
  - The commit will be unreachable (only by the commit hash)

Git Internals

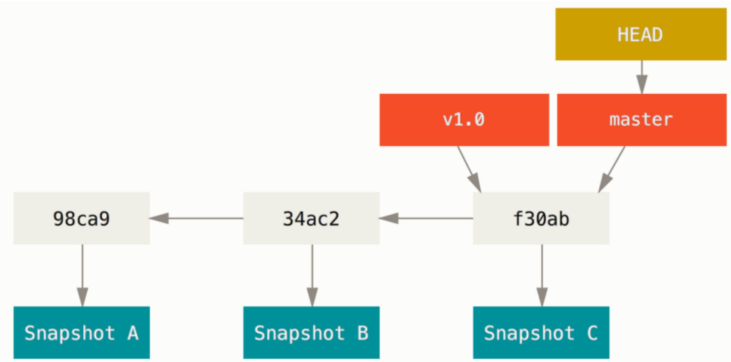
- You can understand Git only if you understand the data model
- Git is a key-value store with a VCS user interface on top of it
  - Key = hash of a file
  - Value = content of a file
- Git objects
  - Blobs: content of files
  - Trees: represent directories and mapping between files and blobs
  - Commits: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



Commit tree



Commit parents



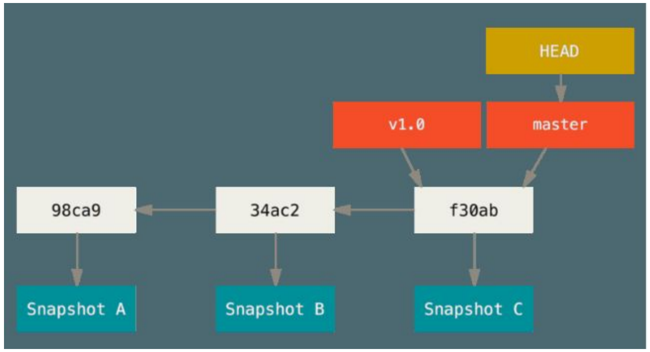
Commit history of a branch

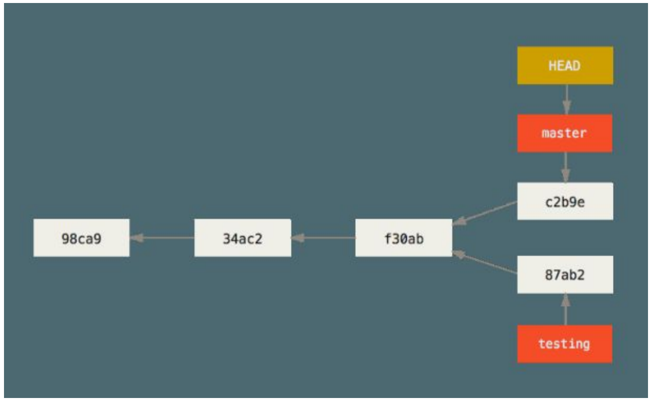
Git Branching

- Branching = diverging from the main line of development
- Why?
  - Work without messing the rest of the code
  - Work without being affected by changes in the main branch
  - Once you are done with working in the 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 branching workflows branch and merge often
  - Even multiple times a day

Git Branching

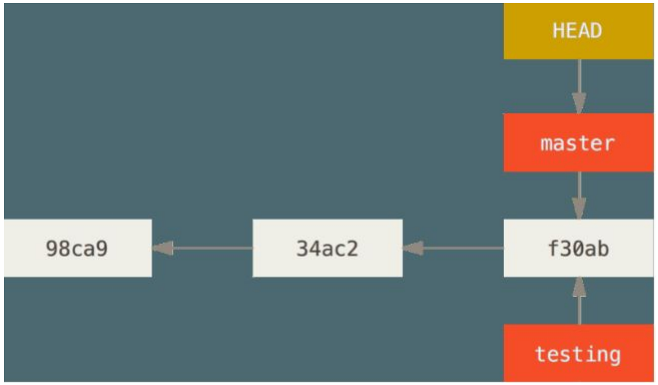
- \`master\` (or \`main\`) is just a normal branch
  - \`master\` is a pointer to the last commit
  - As you commit the pointer moves forward
- \`HEAD\`
  - Pointer to the local branch you are on
  - E.g., \`master\`, \`testing\`
- \`git branch testing\`
  - Create a new pointer \`testing\`
  - Point to the commit you are on
  - Can be moved around
- Divergent history
  - When work progresses in two branches

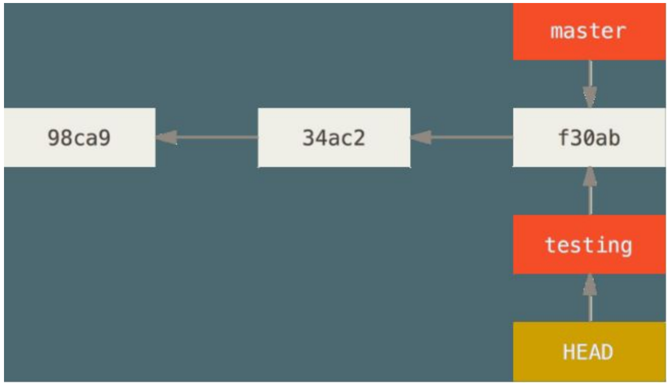


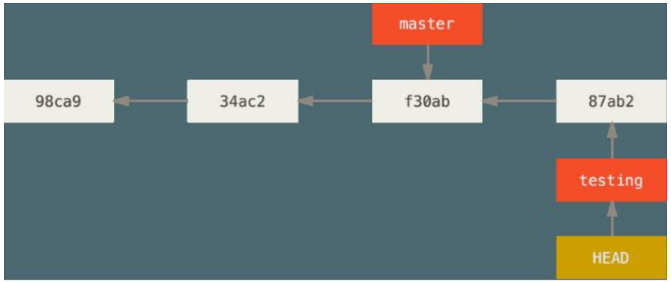


Git Checkout

- `git checkout` switches branch
  - Move `HEAD` to the new branch
  - Change the files in the working dir to match the state corresponding to the branch pointer
- `git checkout testing`
- Then you can keep working by committing on `testing`





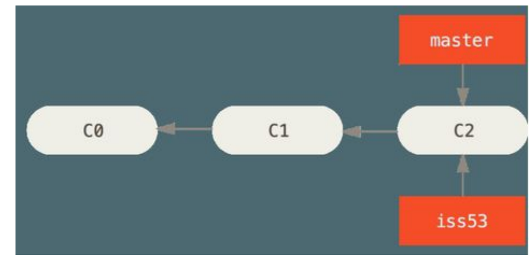


Git Branching and Merging

Start from a project with some commits

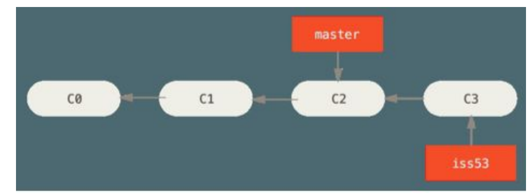
Branch to work a new feature "Issue 53"

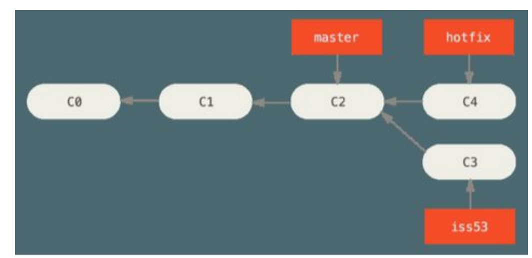
```
> git checkout -b iss53
work ... work ... work
> git commit
```



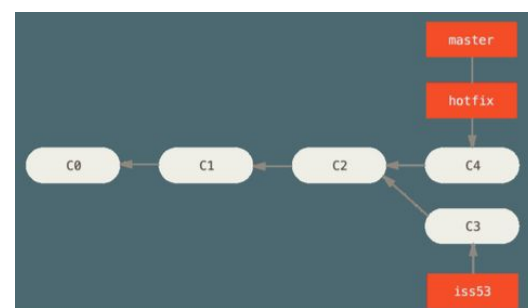
Need a hot-fix to master

```
> git checkout master
> git checkout -b hotfix
fix ... fix ... fix
> git commit -am "Hot fix"
> git checkout master
> git merge hotfix
Fast forward
```





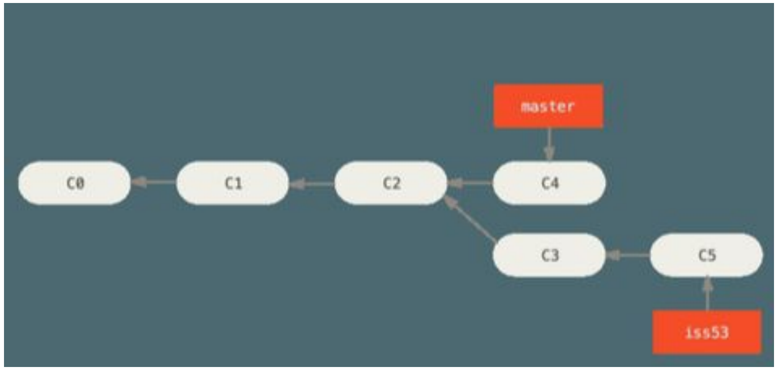
There is a divergent history between `master` and `iss53`



Git Branching and Merging

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

The branch keeps diverging

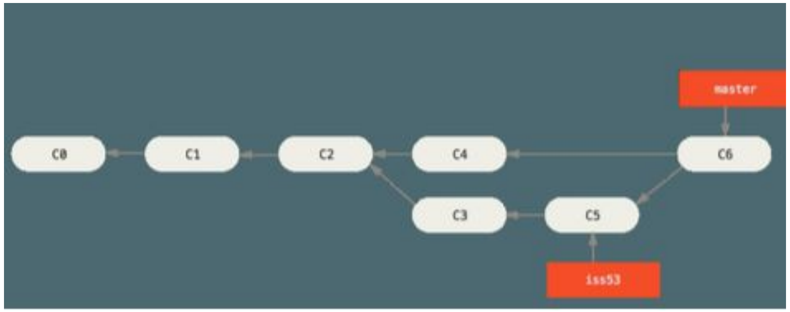


At some point you are done with iss53
You want to merge your work back to master

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

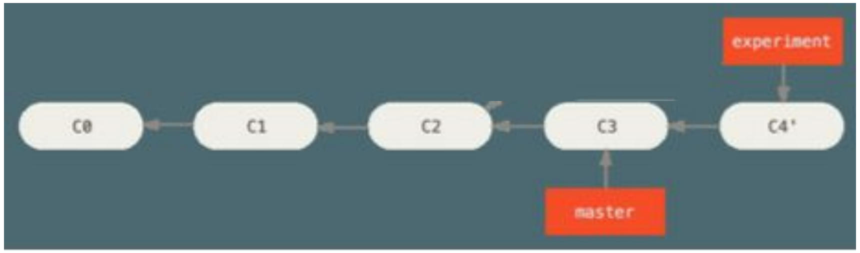
```
> git branch -d iss53
```

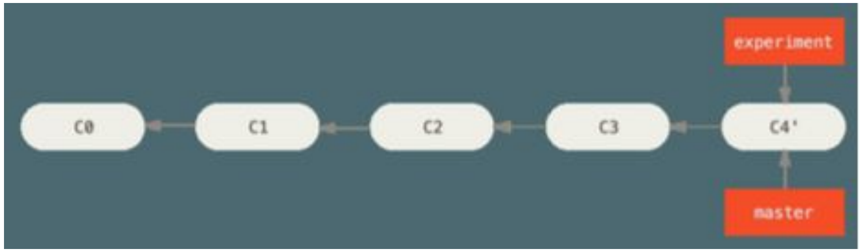
Fast Forward Merge

- 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
  - E.g., C4’ is reachable from C3

```
> git checkout master
> git merge experiment
```

- Git simply moves the branch pointer forward from X to Y





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

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

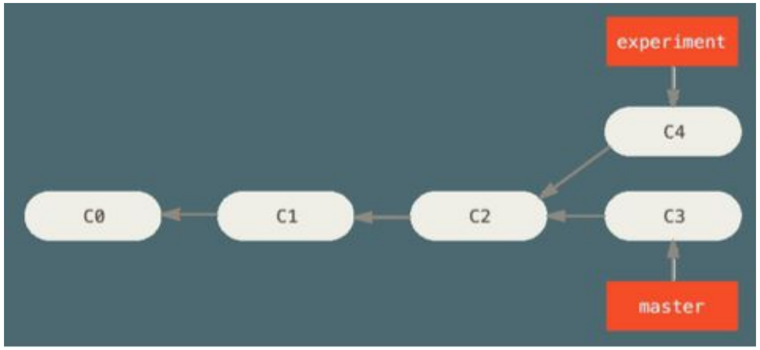
```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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")
```

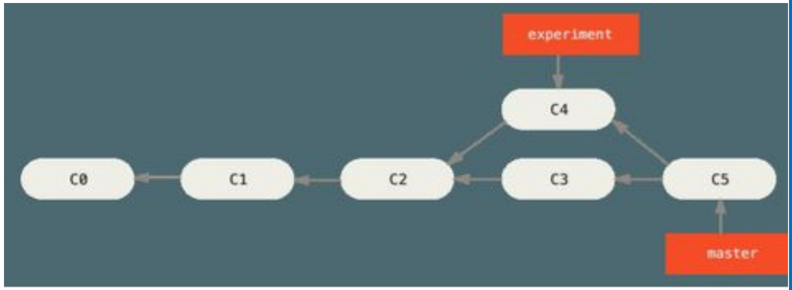
```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

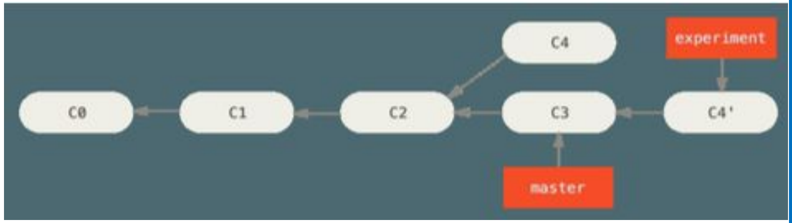
```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Git Rebasing

- In Git there are two ways of merging divergent history
  - E.g., `master` and `experiment`
  - Have a common ancestor C2
- **Merge**
  - Create a new snapshot C5 and commit
  - Go to the target branch
  - `> git checkout master`
  - `> git merge experiment`
- **Rebase**
  - Go to the branch to rebase
  - `> git checkout experiment`
  - `> git rebase master`
  - In words rebasing is like:
    - 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

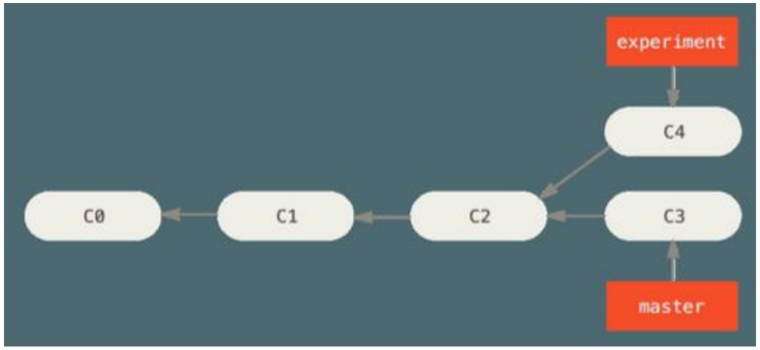


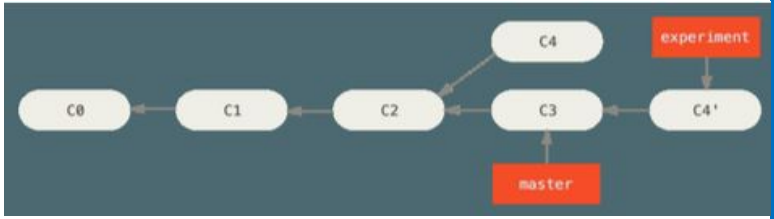




Uses of Rebase

- 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
  - 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





Golden Rule of Rebasing

- Rebasing means abandoning existing commits and creating new ones that are similar but different

- The problem is:
  1. You push your commits somewhere
  2. Others pull your commits down and base their work on them
  3. You rewrite those commits with `git rebase` and push them again with `git push --force`
  4. Your collaborators have to re-merge their work

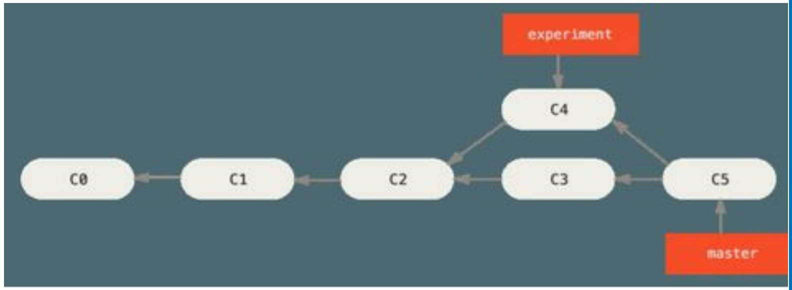
- Solution

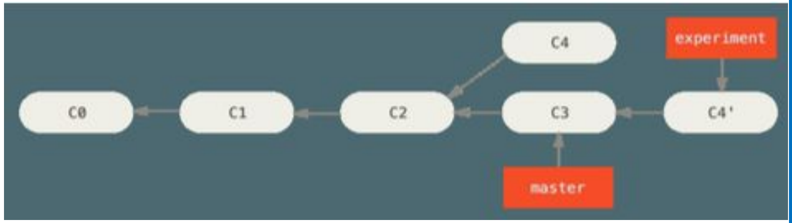
  Strict version: "Do not rebase commits that exist outside your repository"

  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

- Rebase and merge are depend on the interpretation of the repository commit history
- What does the **commit history** of a repo mean?
- **a) A record of what actually happened**
- History should not be tampered with
- What if there is a series of messy merge commits?
  - This is how it happened
  - The repo should preserve this
- Use `git merge`
- **b) The story of 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
- Tell the history in the way that is best for future readers
- Use `git rebase` and filter-branch
- **Best of the merge-vs-rebase approaches**
- Rebase changes you've made in your local repo but haven't pushed yet `git pull --rebase` to clean up your history
- Merge to master to preserve the history of how something was built
- In practice squash-and-merge branches





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\`
  - Can't change the remote branch (e.g., \`origin/master\`)
  - Can change tracking branch (e.g., \`master\`)
  - Git updates them when you do `git fetch origin` (or `git pull`)
- To share code in a local branch you need to push them to a remote
  - `> git push origin serverfix`
- To work on it
  - `> git checkout -b serverfix origin/serverfix`

Git Workflows

Workflows

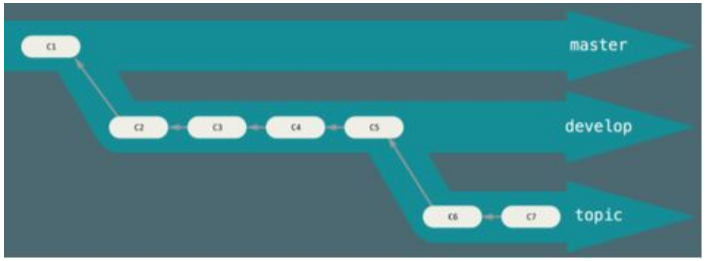
- = ways of working and collaborating using Git

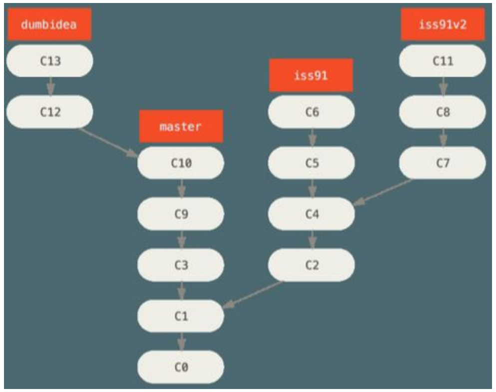
Long-running branches

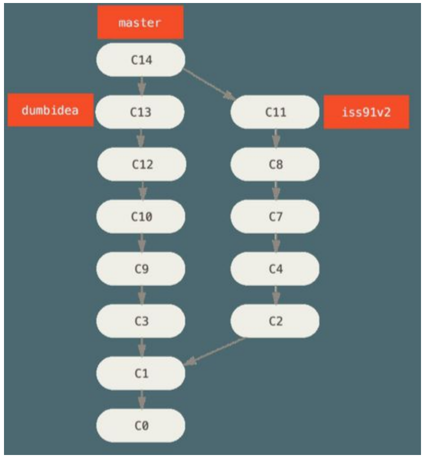
- Branches at different level of stabilities, that are always open
- 1. `master` is always ready to be released
- 2. `develop` to develop in
- 3. Topic / feature branches
- When branches are "stable enough" they are merged up

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



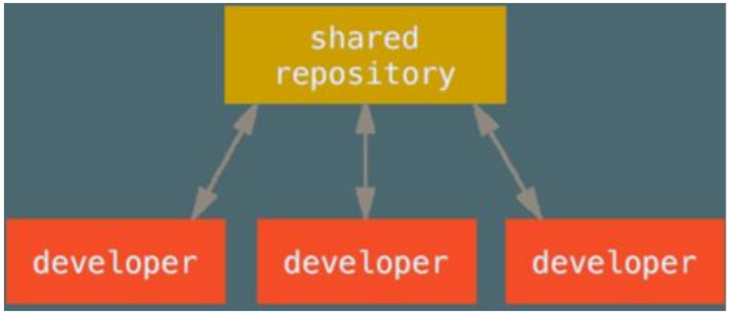




Centralized Workflow

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)

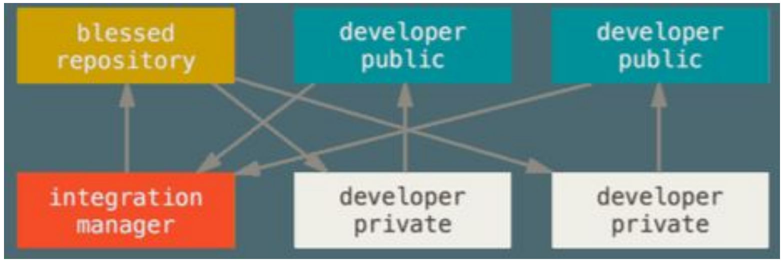
Forking Workflows

- Typically devs don't have permissions to update directly branches on a project
  - Read-write for contributors
  - Read-only for anybody else
- The solution is based on “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

Integration-Manager Workflow

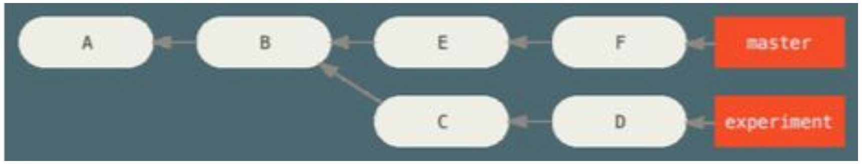
- This is the classical model for open-source development
  - E.g., Linux, GitHub fork

- Each dev has:
  - Write access to their own public repo
  - Read access to everyone else's public repo
- 1. One repo is the "official" project
  - Only the project maintainer pushes to the public repo
- 2. Each contributor:
  - Forks the project into a private copy
  - 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 official repo



Git log

- `git log` reports info about commits
- `refs` are references to
  - HEAD (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 range of commits that are reachable from 2 but not from 1
  - `git log master..experiment -> D,C`
  - `git log experiment..master -> F,E`
- Triple-dot notation
  - 1...2 commits that are reachable from either of the two differences but not from both
  - `git log master...experiment -> F,E,D,C`

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 script-able way
    - E.g., change your email, remove a file (with passwords or large file)
  - Checks out each version, runs the command, commits the result
- hooks
  - Run scripts before each commit, merging, ...

GitHub

- GitHub acquired by MSFT for $7.5b
- GitHub: largest host for Git repos
  - Git hosting (many 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
    - Forking means making a copy to a project so that you can contribute it even if you don't have push / write access

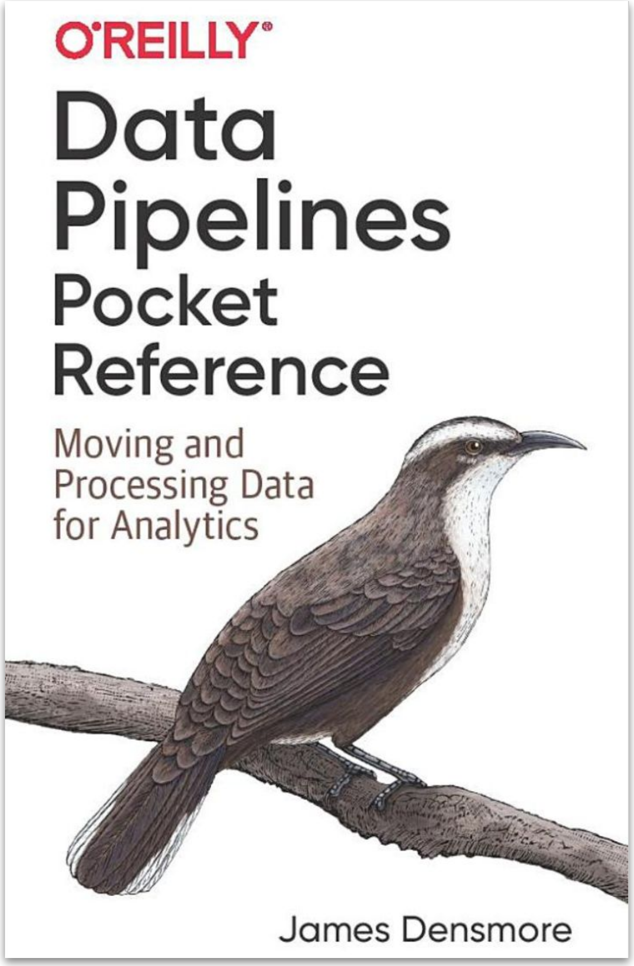
UMD DATA605 - Big Data Systems

Git

Data Pipelines

Data Pipelines - Resources

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

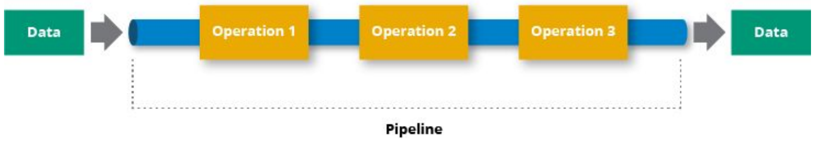


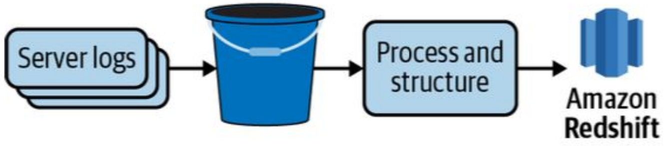
Data as a Product

- Many services today sell data (e.g., Google, Facebook, Amazon, Netflix)
- Data products, e.g.,
  - A recommendation engine
  - Personalized search engine
  - Sentiment analysis on user-generated reviews
- Services are typically powered by machine learning
- Many steps are required
  - Data ingestion
  - Data pre-processing
    - Cleaning, tokenization, feature computations
  - Model training
  - Model deployment
    - MLOps
  - Monitor model
    - Is it working?
    - Is it getting slower?
    - Are performance getting worse?
  - Feedback from deployment
    - E.g., recommendations vs what users bought
    - Ingest data back for future versions of the model

Data Pipelines

- "Data is the new oil"
- Data needs to be:
  - collected
  - pre-processed / cleaned
  - validated
  - processed
  - combined
- **Data pipelines**
  - Processes that move and transform data
  - Goal: derive new value through analytics, reporting, machine learning
- **Data ingestion**
  - Simplest data pipeline
  - Extract data (e.g., from REST API)
  - Load data into DB (e.g., SQL table)





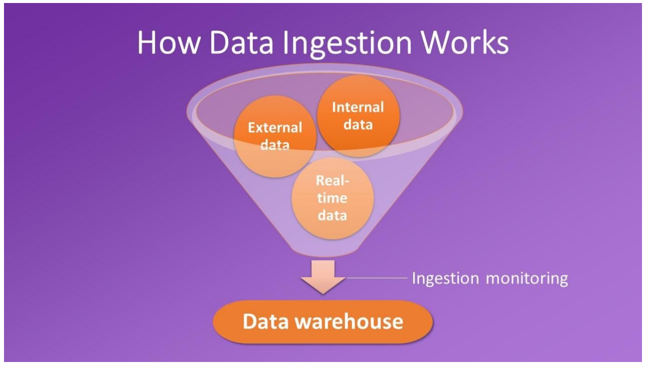
Roles in Building Data Pipelines

- **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 (e.g., Tableaux)

- **Problems in practice**
- Who is responsible for the data?
- Issues with scaling
- Build-vs-buy
  - Which tools?
  - Open-source vs proprietary
- Architecture
  - Who is in charge of it?
  - Conventions
  - Documented
- Getting stuff done
  - Done is better than perfect
- Service level agreement
- Talk to stakeholders on a regular basis

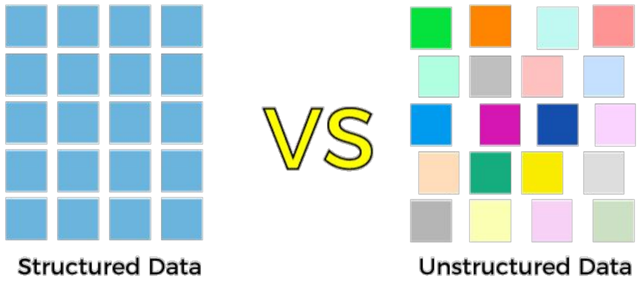
Data Ingestion

- **Data ingestion**
  - = extract data from one source and load it into another
- Data sources
  - DBs
    - E.g., Postgres, MongoDB
  - REST API (abstraction on top of DBs)
  - Network file system / cloud
    - E.g., CSV files, Parquet files
  - Data warehouses
  - Data lakes
- Source ownership
  - An organization can use 10-100s of data sources
  - Internal
    - E.g., Postgres DB storing shopping carts
  - 3rd-parties
    - E.g., Google analytics tracking website usage



Structure in Data (or Lack Thereof)

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



Data Cleaning

- Data cleanliness
  - Quality of source data varies greatly
  - Typically messy
    - Duplicated records
    - Incomplete or missing records
    - Inconsistent formats
      - E.g., phone with / without dashes
    - Mislabeled or unlabeled data
- When to clean it?
  - As soon as possible
  - In different stages
  - As late as possible
  - ETL vs ELT vs EtLT
- Mantras
  - Hope for the best, assume the worst
  - Validate data early and often
  - Don’t trust anything
  - Be defensive

OLAP vs OLTP Workloads

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

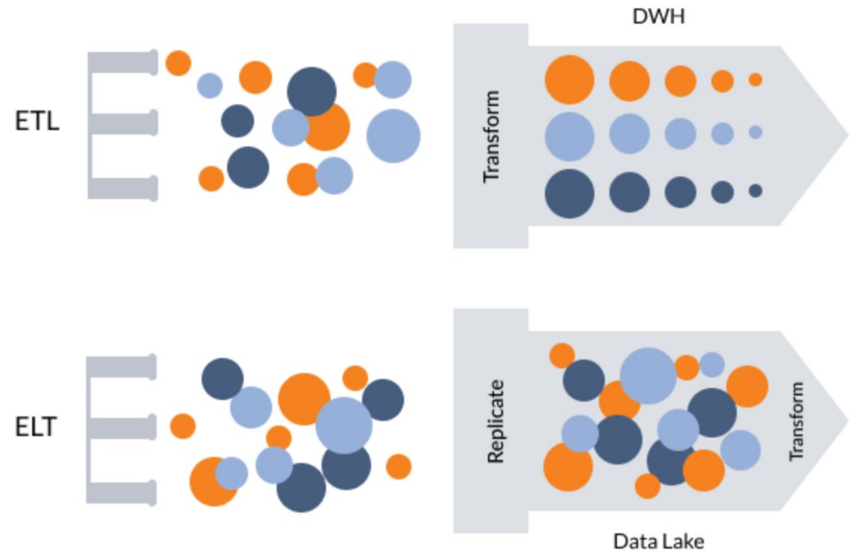
Data Volumes

- High-volume vs low-volume
  - Lots of small reads / writes
  - A few large reads / writes
- One-shot vs streaming

- Other challenges
  - API rate limits / throttling
  - Connection time-outs
  - Slow downloads

Data Warehouse vs Data Lake

- **Data warehouse**
  - = DB storing data from different systems in a structured way
  - E.g., a large Postgres instance with many DBs and tables



- **Data lake**
  - = data stored in a semi-structured or without structure
  - E.g., an AWS S3 bucket storing blog posts, flat files, JSON objects, 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 which data to use and how to access 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
  - Economy of scale
  - The flexibility has a cost (2x-3x more expensive than owning)
  - Vendor lock-in

ETL Paradigm

- **Extract**
  - Gather data from various data sources, e.g.,
    - external data warehouse
    - REST API
    - data downloading
    - 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

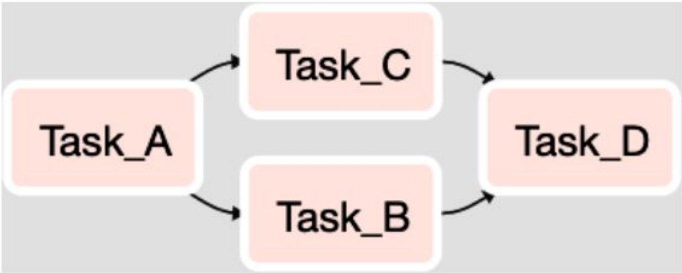
- Data ingestion = E + L
  - Move data from one point to another
  - Format the data
  - Make a copy
  - Have different tools to operate on the data

ETL Paradigm

- Data ingestion tools
  - Buy-vs-build
  - Vendor lock-in
- Data transformation, e.g.,
  - Data conversion (e.g., parsing timestamp)
  - Create new metrics from multiple source columns
  - Aggregate / filter through business logic
  - Anonymize data
- Data modeling
  - Structure data in a format optimized for data analysis
  - E.g., load data in relational DB

Workflow Orchestration

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



ELT paradigm

- ETL has been the standard approach for long time
  - Extract -> Transform -> Load
- Today ELT is becoming the pattern of choice
  - Extract -> Load -> Transform
- Enabled by cloud computing
  - Large storage to save all the raw data
  - Distributed data storage and querying (e.g., HDFS)
  - Columnar DBs
  - Data compression
- Allow to separate data engineers and data scientists / analysts
  - Data engineers focus on data ingestion (E + L)
  - Data scientists focus on transform (e.g., SQL, MongoDB)
  - No need to know how the data will be used
  - ETL requires to understand the data at ingestion time

Row-based vs Columnar DBs

- Row-based DBs
  - E.g., MySQL, Postgres
  - Optimized for reading / writing records
  - 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

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

E<u>tL</u>T

- ETL
  - Extract -> Transform -> Load
- ELT
  - Extract -> Load -> Transform
  - Transformations / 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