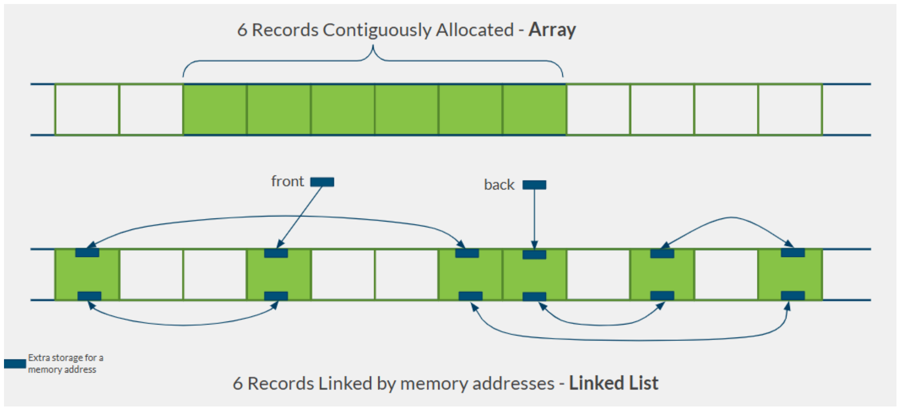
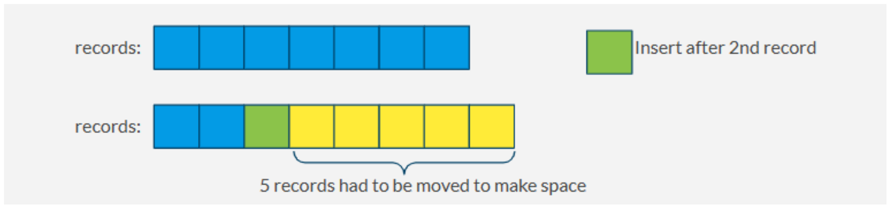
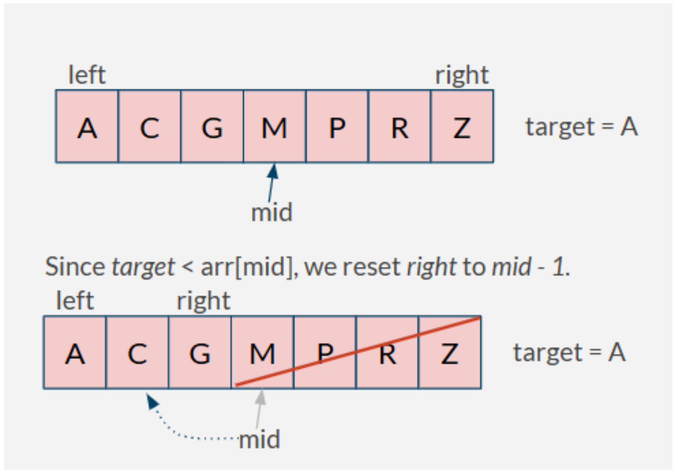
1/8/2025 - Foundations

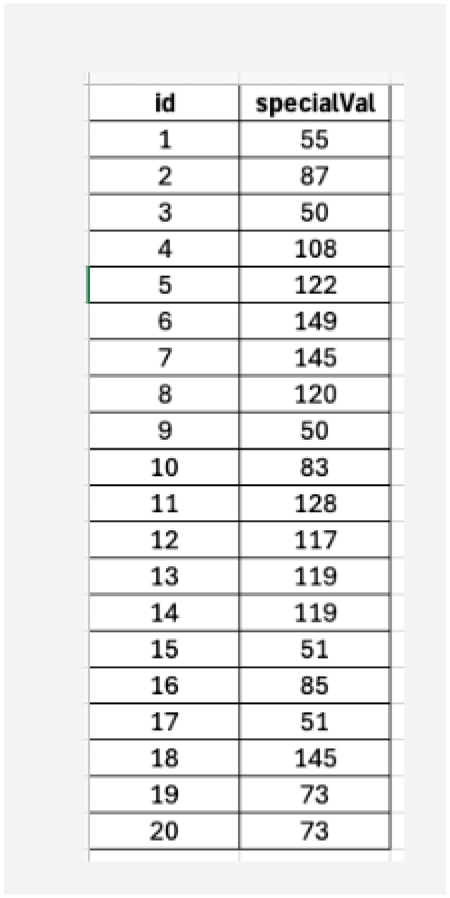
- Searching
  - Most common operation performed by a database system
  - Searching is why we have databases
  - SQL Select is most versatile/complex
  - Baseline for efficiency is Linear Search
    - Start at beginning of a list and proceed element by element until:
      - You find what you are looking for or you get the last element in the list
  - Record - a collection of values for attributes of a single entity instance; row in a table
  - Collection - a set of records of the same entity type; table
  - Search key - a value for an attribute from the entity type
    - Could be >= 1 attribute
- List of records
  - If each record takes up x bytes of memory, then for n records, we need n*x bytes of memory
  - Contiguous allocated list
    - All n*x bytes are allocated as a single "chunk" of memory
  - Linked list
    - Each record needs x bytes + additional space for 1 or 2 memory addresses
      - That quantity times n is how much space it takes uo
    - Individual records are linked together in a type of chain using memory addresses
  - 
6 Records Contiguously Allocated - Array
front
back
Extra storage for a memory address
6 Records Linked by memory addresses - Linked List
- Pros and Cons
  - Arrays are faster for random access, but slow for inserting anywhere but the end
    - 

- Linked lists are faster for interesting anywhere in the list, but slower for random access
    - *Insert*
- Numpy is arrays that are contiguously allocated

- Observations
  - Arrays
    - Fast for random access
    - Slow for random insertions
  - Linked lists
    - Slow for random access
    - Fast for random insertions
- Binary Search
  - Input: array of values in sorted order, target value
  - Output: the location (index) of where target is located or some value indicating target was not found
  - Go in the middle, can eliminate half if the item is smaller or larger
  - Recursive in nature but can be dangerous in large data
    - Probably over 30k recursive calls
  - 

  - Worst case of most searches needing to be done to find the value or determine it isn't in the set is log base 2 (n) - $\log_2(n)$
  - Example in slide is iterative version
- Time complexity
  - Linear search
    - Best case: target is found at the first element, only 1 comparison
    - Worst case: target is not in the array; n comparisons
    - Therefore, in the worst case, linear search is O(n) time complexity
  - Binary search
    - Best case: target is found at *mid*; 1 comparison (inside the loop)
    - Worst case: target is not in the array; $\log_2(n)$ comparisons
    - Therefore, in the worst case, binary search is $O(\log_2(n))$ time complexity
    - Can't perform binary search on an unsorted array
- Back to database searching

- Assume data is stored on disk by column id's value
- Searching for a specific id = fast
  - But what if we want to search for a specific specialVal?
    - Only option is linear scan of that column
    - Cannot store data on disk sorted by both id and specialVal (at the same time)
      - Data would have to be duplicated – inefficient
    - We need an external data structure to support faster searching by specialVal than a linear search



id	specialVal
1	55
2	87
3	50
4	108
5	122
6	149
7	145
8	120
9	50
10	83
11	128
12	117
13	119
14	119
15	51
16	85
17	51
18	145
19	73
20	73

- What do we have in our arsenal?
  - An array of tuples (specialVal, rowNumber) sorted by specialVal
    - We could use Binary Search to quickly locate a particular specialVal and find its corresponding row in the table
    - But, every insert into the table would be like inserting into a sorted array - slow
  - A linked list of tuples (specialVal, rowNumber) sorted by specialVal
    - Searching for [cont]
    -
- Something with fast insert and fast search?
  - Binary search tree - a binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent
    - Recursive data structure which lends itself to recursive formulas

1/16/2025 -
Practical outline

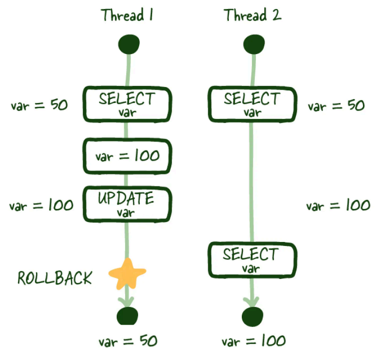
- A.json
    - ‘Preprocessed_text’: [‘bank, ‘finance’, etc.]
- Finance articles
    - Jan 2018 etc. up to May
    - Each folder has 50k+ json files
- Give program path to root folder and then have to recursively descend through the folder
- BST_NODE
    - Key is going to be a word
        - Values will have the files
            - If bank is in a.json and c.json, key would be bank, value would be a.json, c.json
    - Add-values: goes to values list and appends the new value
        - Self.values
        - Consider if bank appears twice in the file
            - Add it twice, add it only once
            - Prob not valuable information to have it twice
    - Put preprocessed texts into a set to to remove duplicates
    - Use string comparison to determine greater or less than
- BST_index
    - Abstract index is a type of inheritance defined in abstrac_ index.py
        - Abstract is its own class
        - Can change abstract
    - Abstract method (class that implements) have to provide implementation for abstract method
    - Line 145 insert function
        - Takes a key and a single value
            - Value is name of file currently being parsed
    - Line 28 insert recursive
        - Look up recursive
- AVL_TREE_NODE
    - Has everything BST has
    - Insert recursive
        - Height is starting at 1
    - If not root:
    - Node = AVLNode(key)
    - node.add_value(value)
        - Return node
    - Elif key < root.key:
        - Root.left = self.insert_recursive(current.left, key value)
    - Elif key > root.key
        - Root.right = self_insert_recurisve(root.left, key, value)
    - Elif key == root,key

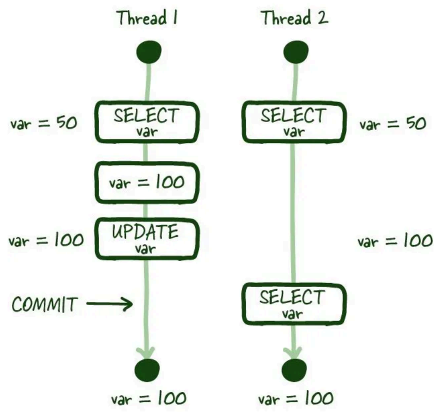
        - Update height
            - Height_left = (0 if not root.left else root.left.height)
            - Height_right = (0 if not root.right else root.right.height)
            - Root.heigh = 1 +(height_left if (height_left > ehgith_right) else height_right)
                    - Equivalent of max function
            - Balance = hieght_left - height_right
            - If balance > 1:
                - If key< root.left.key:
                    - Return self._rotate_right(root)
                - Else:
                    - Root.left = self._rotate_left(root.left)
                    - Return self.rotateright(root)
            - If balance < -1:
                - If key > root.right/key:
                    - Return self.rortateleft(root)
                - Else:
                    - Root.right self.rotateright(root.right)
                    - Return self.rotateleft(root)
            - Return root
- Rotate left
    - Y = x.right
    - Tw = y.left
    - Y.left = x
    - X.right = tw
    - Ht_x_left = (0 if not x.left else x.left_h
    - Should be using functions and not all if checkers
        - If statement is faster than a function call
        - Make sure its not taking too long
- Use pickle
    - Library in python that serializing objects to disk so that it can read it in much faster

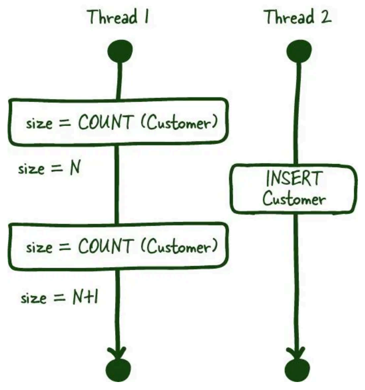
1/27/25

- Benefits of the relational model
    - (mostly) standard data model and query language
    - ACID compliance
        - Atomicity, consistency, isolation, durability
    - Works well will highly structured data
    - Can handle large amounts of data
    - Well understood, lots of tooling, lots of experience
    - A transaction is a unit of work for a database
- Relational database performance
    - Many ways that a RDBMS increases efficiency:

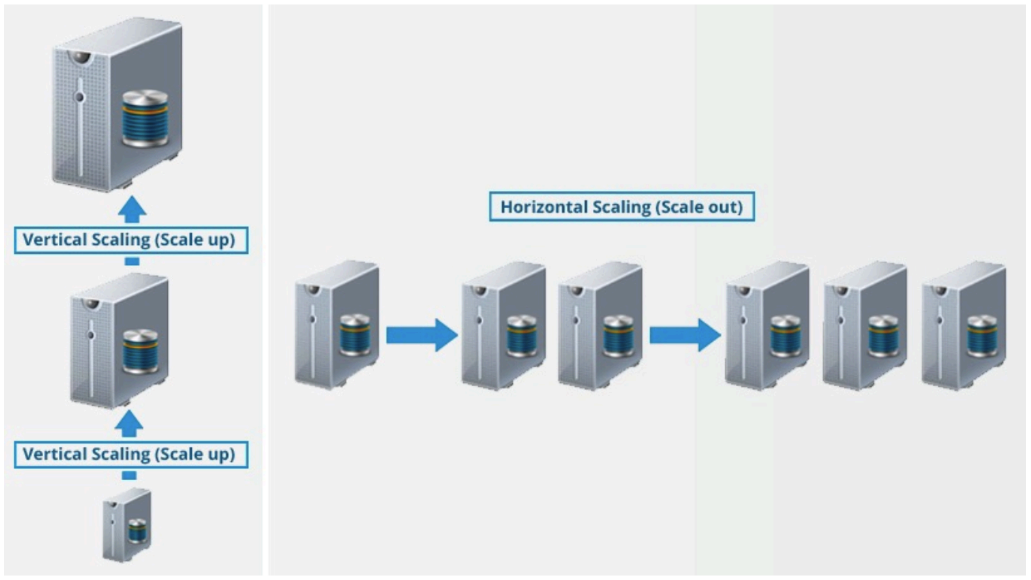
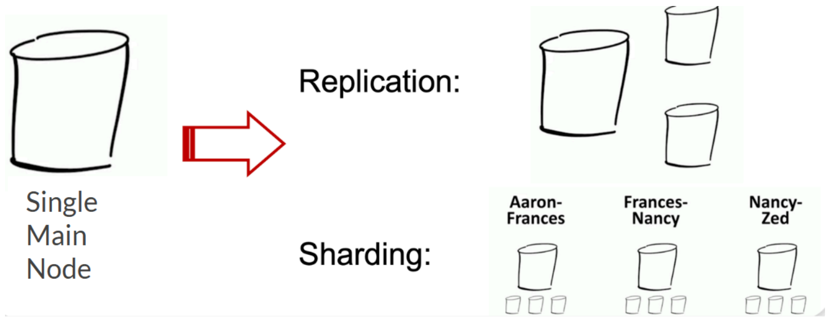
        - Indexing (focus of class)
        - Directly controlling storage
        - Column oriented storage vs row oriented storage
        - Query optimization
        - caching/prefetching
        - materialized views
        - Precompiled stored procedures
        - Data replication and partitioning
- Transaction processing
    - *Transaction* - a sequence of one or more of the CRUD operations performed as a single, logical unit of work
        - Either the entire sequence succeeds (COMMIT)
        - OR the entire sequence fails (ROLLBACK or ABORT)
    - Help ensure
        - Data integrity
        - Error recovery
        - Concurrency control
        - Reliable data storage
        - Simplified error handling
- Acid properties
    - Characteristics or properties of transactions that ensure the safety of database and the integrity of the database that would be detrimental if don't exist
    - Atomicity
        - Transaction is treated as an atomic unit - it is fully executed or no parts of it are executed
    - Consistency
        - A transaction takes a database from one consistent state to another consistent state
        - Consistent state - all data meets integrity constraints
    - Isolation
        - Two transactions T_1 and T_2 are being executed at the same time but cannot affect each other
        - If both T_1 and T_2 are reading the data - no problem
        - Ensured through "locking"
        - If T_1 is reading the same data that T_2 may be writing, can result in:
            - Dirty read
                - A transaction T_1 is able to read a row that has been modified by another transaction T_2 that hasn't yet executed a COMMIT



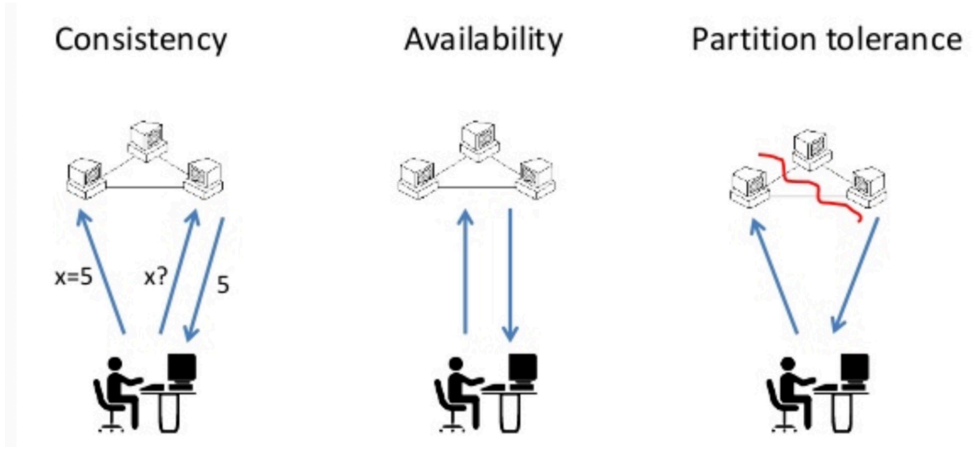
            - Non-repeatable read
                - Two queries in a single transaction T_1 execute a SELECT but get different values because another transaction T_2 has changed data and COMMITTED
                - 

            - Phantom read
                - When a transaction T_1 is running and another transaction T_2 adds or deletes rows from the set T_1 is using
                - 

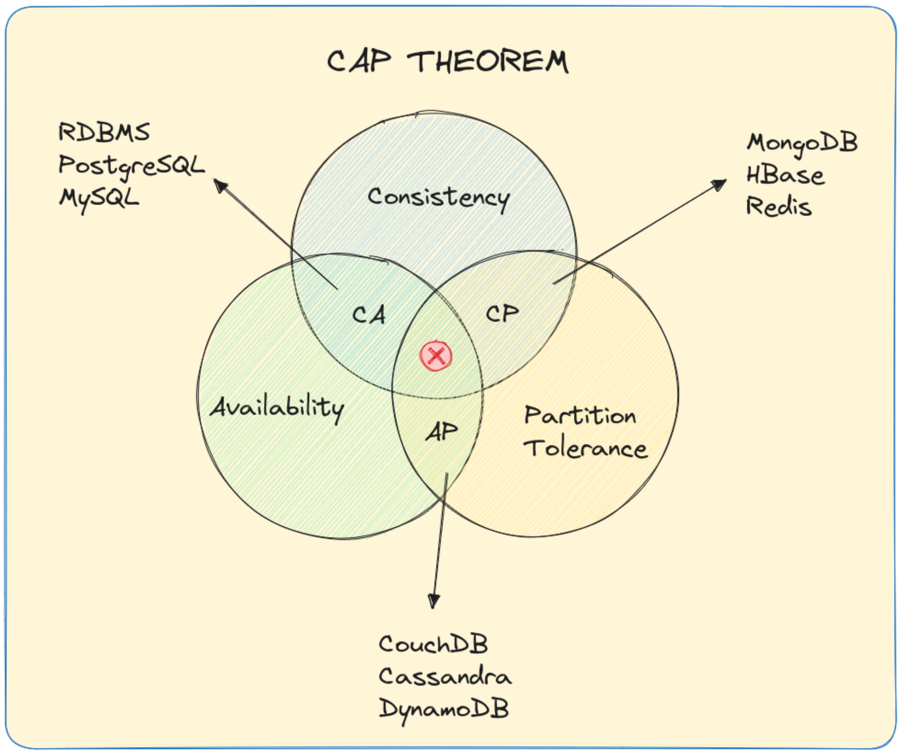
    - Durability
        - Once a transaction is completed and committed successfully, its changes are permanent
        - Even in the event of a system failure, committed transactions are preserved
        - If data disappears, want to be able to get it back
- Relational Databases may not be the solution to all problems:
    - Sometimes, schema evolve over time
    - Not all apps may need the full strength of ACID compliance
    - Joins can be expensive
    - A lot of data is semi-structured or unstructured (JSON, XML, etc)
    - Horizontal scaling presents challenges
    - Some apps need something more performant (real time, low latency systems)
- Scalability up or out
    - Conventional wisdom:

        - Scale vertically (up, with bigger, more powerful systems) until the demands of high-availability make it necessary to scale out with some type of distributed computing model
    - But why?
        - Scaling up is easier - no need to really modify your architecture. But there are practical and financial limits
    - However
        - There are modern systems that make horizontal scaling less problematic
    - 
Vertical Scaling (Scale up)
Vertical Scaling (Scale up)
Horizontal Scaling (Scale out)
    - More power is not always the answer however it is the easiest thing to do
- So what Distributed data when scaling out
    - A distributed system is "a collection of independent computers that appear to its users as one computer"
    - Characteristics of distributed system
        - Computers operate concurrently
        - Computers fail independently
        - No shared global clock
- Distributed storage - 2 directions
    - 

- Distributed data stores
    - Data is stored on > 1 node, typically replicated
        - I.e. each block of data is available on N nodes

  - Distributed databases can be relational or non-relational
    - MySQL and PostgreSQL support replication and sharding
    - CockroachDB - new player on the scene
    - Many NoSQL systems support one or both models
  - But remember: Network partitioning is inevitable
    - Network failures, system failures
    - Overall system needs to be a partition tolerant
      - System can keep running even with network partition
- The CAP Theorem
  - 

  - Can always have two of these but can never have all three
  - Consistency here is not necessarily same consistency as ACID
  - Availability is can i get to it all the time
  - If something breaks up system, can i still read and write from the partitions
  - <u>CAP Theorem</u> states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:
    - <u>Consistency</u>: every read receives the most recent write or error thrown
      - Every user of the DB has an identical view of the data at any given instant
    - <u>Availability</u>: every request receives a (non-error) response - but no guarantee that the response contains the most recent write
      - In the event of a failure, the database remains operational
    - <u>Partition tolerance</u> - the system can continue to operate despite arbitrary network issues
      - The database can maintain operations in the event of the network's failing between two segments of the distributed system

- 

- <u>Consistency + Availability</u>: System always responds with the latest data and every request gets a response, but may not be able to deal with network issues (network partitions)
    - If you make a request and network has been partitioned, don't know what you will get back or could get an error
- <u>Consistency + Partition tolerance: </u>If system responds with data from a distributed store, it is always the latest, else data request is dropped
    - "Else..dropped" is lack of availability
    - If gets partitioned, data may not be consistent and can't always access it
- <u>Availability + Partition Tolerance</u>: system always sends responses based on distributed data store, but may not be the absolute latest data
    - "Stale data"

- *Cap in reality*
    - What it is really saying:
        - If you cannot limit the number of faults, requests can be directed to any server, and you insist on serving every request, then you cannot possibly be consistent
    - But it is interpreted as:
        - You must always give up something: availability, consistency, tolerance to failure

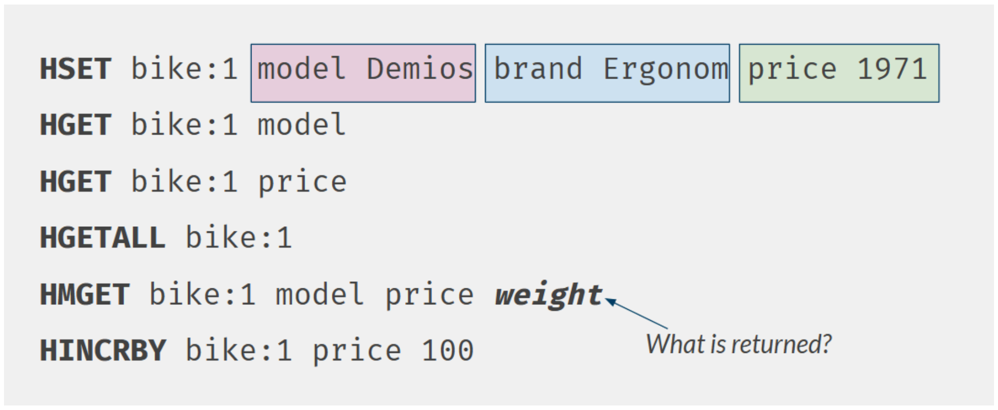
2/3/25

- Distributed DBs and ACID - Pessimistic Concurrency
    - ACID transaction

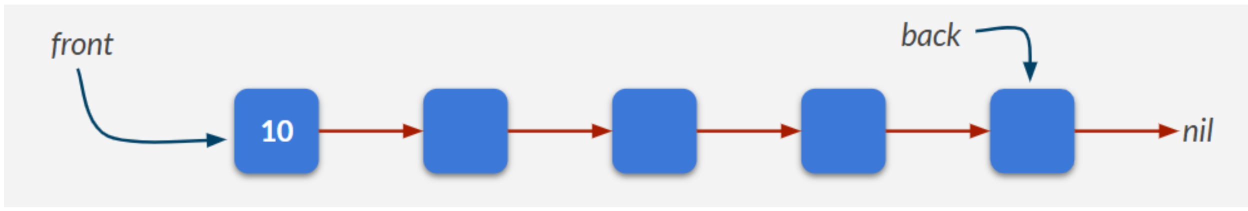
    - If something bad could happen, it will, and attempts to prevent a conflict from happening
    - Focuses on "data safety"
    - Considered a <u>pessimistic concurrency model</u> because it assumes one transaction has to protect itself from other transactions
        - IOW, it assumes if something can go wrong it will
    - Conflicts are prevented by locking resources until a transaction is complete (there are both read and write locks)
    - Write Lock Analogy → borrowing a book from a library…If you have it, no one else can
    - Typically when you start a transaction, transaction processing unit can look and see what needs to be locked if anything
- Optimistic concurrency
    - Transactions don't obtain locks on data when they read or write
    - *Optimistic* because it assumes conflicts are unlikely to occur
        - Even if there is a conflict, everything will be ok
    - But how?
        - Add last update timestamp and version number to columns to every table… read them when changing. THEN, check at the end of transaction to see if any other transaction has caused them to be modified
    - Low conflict systems (backups, analytical dbs, etc)
        - Read heavy systems
        - The conflicts that arise can be handled by rolling back and re-running a transaction that notices a conflict
        - So optimistic concurrency works well - allows for higher concurrency
    - High conflict Systems
        - Rolling back and rerunning transactions that encounter a conflict → less efficient
        - So, a locking scheme (pessimistic model) might be preferable
    - When lots of people involved, may not want to use this model
- NOSQL
    - First used to describe a relational database system that did not use SQL
        - 1998 - Carlo Strozzi
    - More common, modern meaning is "Not Only SQL"
    - But, sometimes thought of as "non-relational databases"
    - Idea originally developed, in part, as a response to processing unstructured web-based data
- ACID Alternative for Distribution Systems - BASE
    - <u>B</u>asically <u>A</u>vailable
        - Guarantees the availability of the data (per CAP), but response can be "failure"/"unreliable" because the data is an inconsistent or changing state
        - System appears to work most of the time
    - <u>S</u>oft State

        - The state of the system could change over time, even without input. Changes could be the result of eventual consistency
            - Data stores don't have to be write-consistent
            - Replicas don't have to be mutually consistent
    - Eventual Consistency
        - The system will eventually become consistent
            - All writes will eventually stop so all nodes/replicas can be updated
    - Many DB systems that support replication of nodes will support the three things of BASE to some degree
- Categories of NoSQL DBs
    - Document databases
        - Like json
    - Graph databases
        - Things with lots of relationships between the different points??
    - Key-value databases
    - Columnar databases
    - Vector databases
- Key Value Stores
    - Or key value databases
    - Feel similar to operating a hash table in python
    - *key = value*
    - Designed around 3 things
        - Simplicity
            - The data model is extremely simple
            - Comparatively, tables in a RDBMS are very complex
            - Lends itself to simple CRUD ops and API creation
        - Speed
            - Usually developed as in-memory DB
            - Retrieving a *value* given its *key* is typically a O(1) op b/c hash tables or similar data structures used under the hood
            - No concept of complex queries or joins…they slow things down
        - Scalability
            - Horizontal scaling is simple - add more nodes
            - Typically concerned with *eventual consistency*, meaning in a distributed environment, the only guarantee is that all nodes will *eventually* converge on the same value
- KV DS Use Cases
    - EDA/Experimentation Results Store
        - Store intermediate results from data preprocessing and EDA
        - Store experiment or testing (A/B) results w/o prod db
    - Feature Store
        - Store frequently accessed feature → low-latency retrieval for model training and prediction
    - Model Monitoring

        - Store key metrics about performance of model, for example, in real-time inferencing
- KV SWE Use Cases
    - Caching scenarios
    - Storing Session Information
        - Everything about the current session can be stored via a single PUT or POST and retrieved with a single GET … very fast
    - User Profiles & Preferences
        - User info could be obtained with a single GET operation … language, TZ, product or UI preferences
    - Shopping Cart Data
        - Cart data is tied to the user
        - Needs to be available across browser, machines, session
    - Caching Layer
        - In front of a disk-based database
- Consider efficiency of KV because we can keep everything memory resident
- Redis DB
    - <u>R</u>emote <u>D</u>irectory <u>S</u>erver
        - Open source, in-memory database
        - Sometimes called a data structure store
        - Primarily a KV store, but can be used with other models: Graph, Spatial, Full Text Search, Vector, Time Series
    - It is considered an in-memory database system but….
        - Supports durability of data by
            - Essentially saving snapshots to disk at specific intervals OR
            - Append-only file chick is a journal of changes that can be used for roll-forward if there is af failure
            - Can be very fast .. >100,000 SET ops/second
            - Rich collection of commands
            - Does NOT handle complex data. No secondary indexes. Only supports lookup by Key
- Redis Data Types
    - Keys:
        - Usually strings but can be binary sequence
    - Values:
        - Strings
        - Lists (linked lists)
        - Sets (unique unsorted string elements)
        - Sorted sets
        - Hashes (string → string)
        - Geospatial data
- Redis Databases and Interaction
    - Redis provides 16 databases by default
        - They are numbered 0 to 15

        - There is no other name associated
    - DIrect interaction with Redis is through a set of commands related to setting and getting k/v pairs (and variations)
    - Many language libraries available as well
- Foundation Data Type - String
    - Sequence of bytes - text, serialized objects, bin arrays
    - Simplest data type
    - Maps a string to another string
    - Use cases:
        - Caching frequently accessed HTML/CSS/JS fragments
        - Config. Settings, user settings info, token management
        - Counting web page/app screen views or rate limiting
- Basic command
    - SETNX will only set a key if it doesn't already exist in the table
    - SET someValue 0
    - INRC someValue
        - Considered atomic operations, won't end up have two transactions trying to increment same number at same time
    - INCRBY someValue 10
    - DECR someValue
    - DECRBY someValue 5
        - INCR parses the value as an int and increments (or adds to value)
    -
- Hash Type
    - Value of KV entry is a collection of field-value pairs
    - Use cases:
        - Can be used to represent basic objects/structures
            - Number of field/value pairs per has is 2^31-1
            - Practical limit: available system resource (e.g. memory)
        - Session information management
        - User/Event tracking (could include TTL)
        - Active Session Tracking (all sessions under one hash key)
- Hash Commands
    - 

```
HSET bike:1 model Demios brand Ergonom price 1971
HGET bike:1 model
HGET bike:1 price
HGETALL bike:1
HMGET bike:1 model price weight
HINCRBY bike:1 price 100
```

- As long as for data in table, only searching for primary key, this can become close to what a relational database is like
    - Search only by username
    - But if ever want to search by email, can't do it in redis
- List Type
    - Value of KV Pair is **linked lists** of string values
    - Use cases:
        - Implementation of stacks and queues
        - Queue management and message passing queues (producer/consumer model)
        - Logging systems (easy to keep in chronological order)
        - Build social media streams/feeds
        - Message history in a chat application
        - Batch processing by queueing up a set of tasks to be executed sequentially at a later time
- Linked Lists crash course
    - Sequential data structure of linked nodes (instead of contiguously allocated memory)
    - Each node points to the next element of the list (except the last points to nil/null)
    - O(1) to insert new value at front or insert new value at the end
    - 

    - push/pop from same side operates like a stack
    - push/pop from alternate sides operates like a queue
- JSON Type
    - Full support of the JSON standard
    - Uses JSONPath syntax for parsing/navigating a JSON document
    - Internally, stored in binary in a tree-structure → fast access to sub elements
- Set Type
    - Unordered collection of unique strings (members)
    - Use cases:
        - Track unique items
        - Primitive relation
        - Access control list for users and permission structures
        - Social network friends list
    - Supports set operations
- Write a python script that will parallelize the parsing of json files to maximize the throughput that redis can handle

MongoDB Lecture

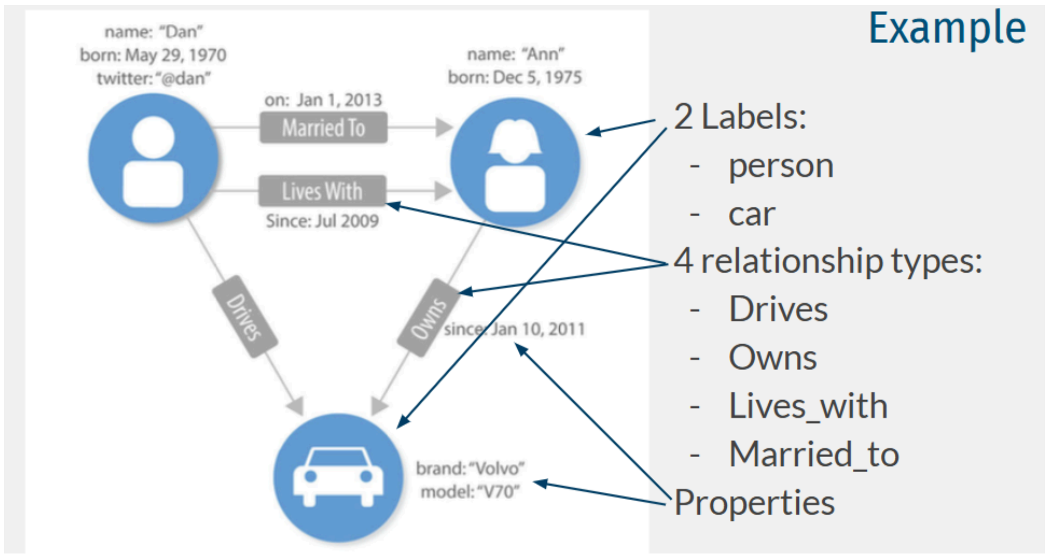
- Document Database

- A non-relatinal database that stores data as structured documents usually in json
- JSON = JavaScript Object Notation
    - A lightweight data-interchange format
    - It is easy for humans to read and write
    - Its easy for machines to parse and generate
- JSON is built on two structure
    - A collection of name/value pairs. In various languages, this is operationalized as an object, record, struct, dictionary, hash table, keyed list, or associative array
    - An ordered list of values. In most languages, this is operationalized as an array, vector, list, or sequence
- These are two universal data structures supported by virtually all modern programming languages
    - Thus, json makes a great data interchange

- Binary Json, BSON
    - BSON → Binary Json
        - Binary encoded serialization of a JSON-like document structure
        - Supports extended types not part of basic JSON (e.g. data, BinaryDAte, etc.)
        - Lose the human readability aspect
        - Lightweight - keep space overhead to a minimum
        - Traversable - designd to be easily traversed, which is vitially important tot a document DB
- XML (extensible markup language)
    - Precursor to JSON as data exchange format
    - XML + CSS → web pages that separated content and formatting
    - Structurally similar to HTML, but tag set is extensible
- Why Document Databases
    - Document databases address the impedance mismatch problem between object persistence in OO systems and how relational databases structure data
        - OO programming → inheritance and composition of types
        - How do we save a complex object to a relational database?
            - We basically have to deconstruct it
    - The structure of a document is self-describing
    - They are well-aligned with apps that use JSON/XML as a transport layer

Introduction to the Graph Data Model

- *What is a graph database*
    - Data model based on the graph data structure
    - Composed of nodes and edges
        - Edges connect nodes
        - Each is uniquely identified
        - Each can contain properties (e.g. name, occupation, etc)

            - Supports queries based on graph-oriented operations
                - Traversals
                - Shortest path
                - Lots of others
- *Where do graphs show up?*
    - Social networks
        - Modeling social interactions in fields like psychology and sociology
        - Social media as well
    - The web
        - It is just a big graph of “pages” nodes connected by hyperlinks (edges)
    - Chemical and biological data
        - Systems biology, genetics, etc.
        - Interaction relationships in chemistry
- *What is a graph?*
    - <u>Labeled property graph</u>
        - Composed of a set of node (vertex) objects and relationship (edge) objects
        - Labels are used to mark a node as part of a group
        - Properties are attributes (think KV pairs) and can exist on nodes and relationships
        - Nodes with no associated relationships are OK.
        - Edges not connected to nodes are <u>not</u> permitted
- *Example*
    - 

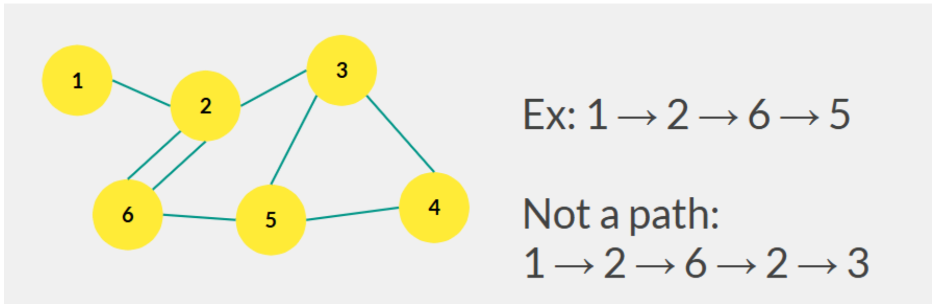
Example

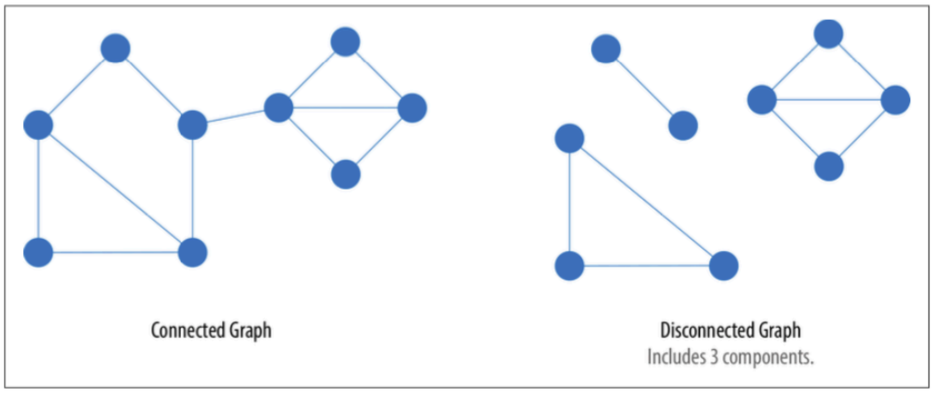
2 Labels:
- person
- car

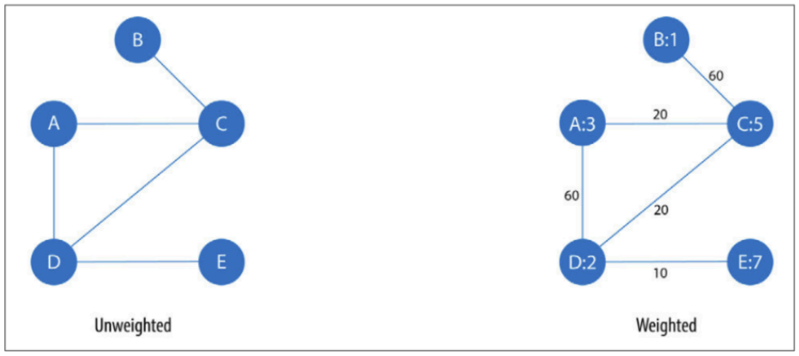
4 relationship types:
- Drives
- Owns
- Lives_with
- Married_to

Properties

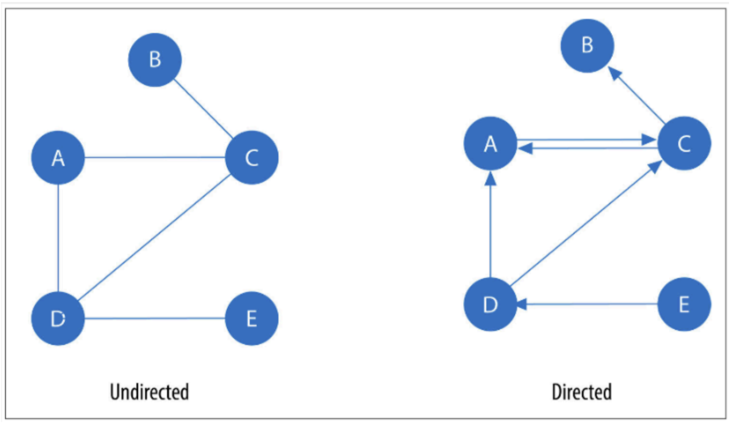
- *Paths*
    - A path is an ordered sequence of nodes connected by edges in which no nodes or edges are repeated

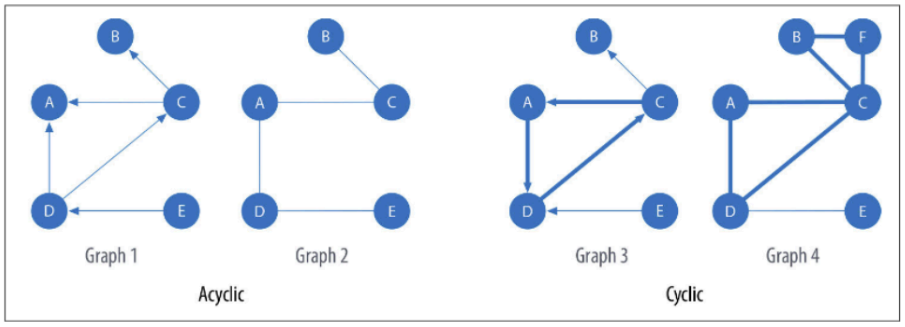
- 

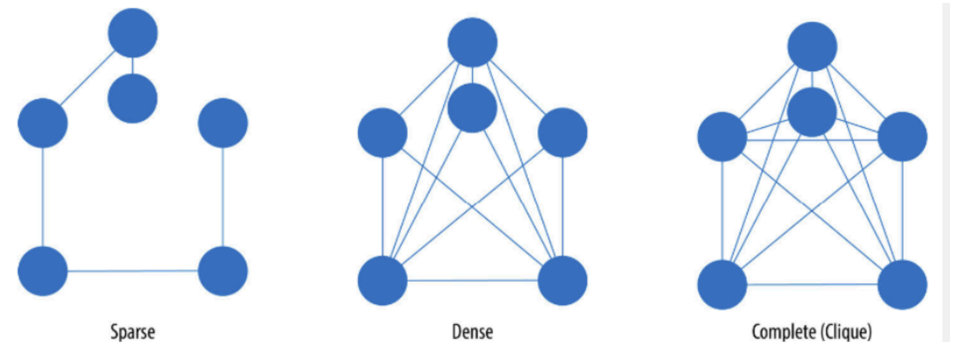
- *Flavors of Graphs*
  - Connected (vs Disconnected): there is a path between any two nodes in the graph
    - N choose 2
    - No representation of how long or direct but can follow lines from one node to any other nodes
    - 

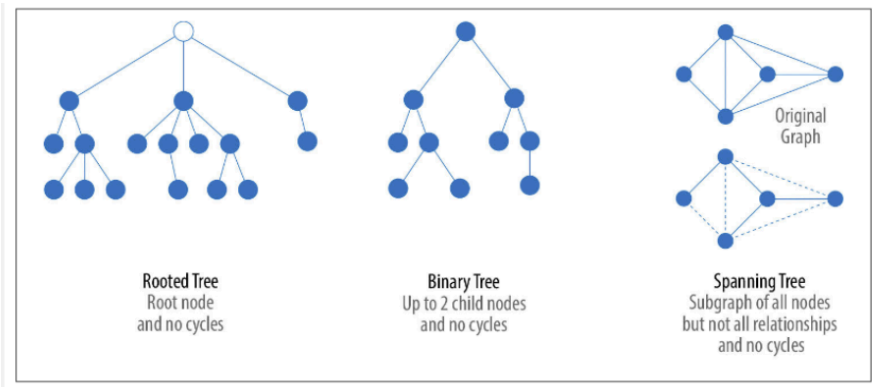
  - Weighted (vs unweighted) - edge has a weight property (important for some algorithms)
    - If an unweighted graph, can consider it a weighted graph where they all have even weights
    - 

  - Directed (vs Undirected) - relationships (edges) define a start and end node

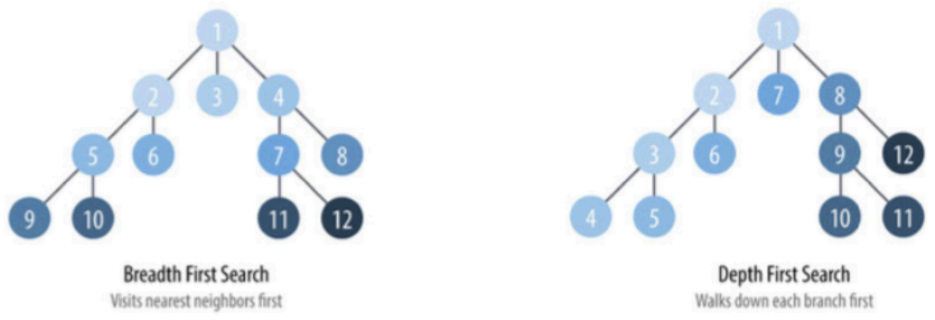
- 

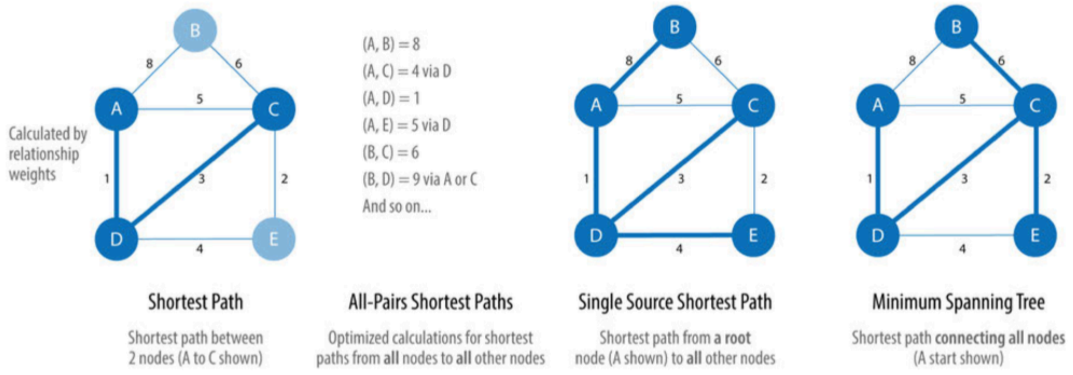
  - Acyclic (vs Cyclic) - graph contains no cycles
    - 

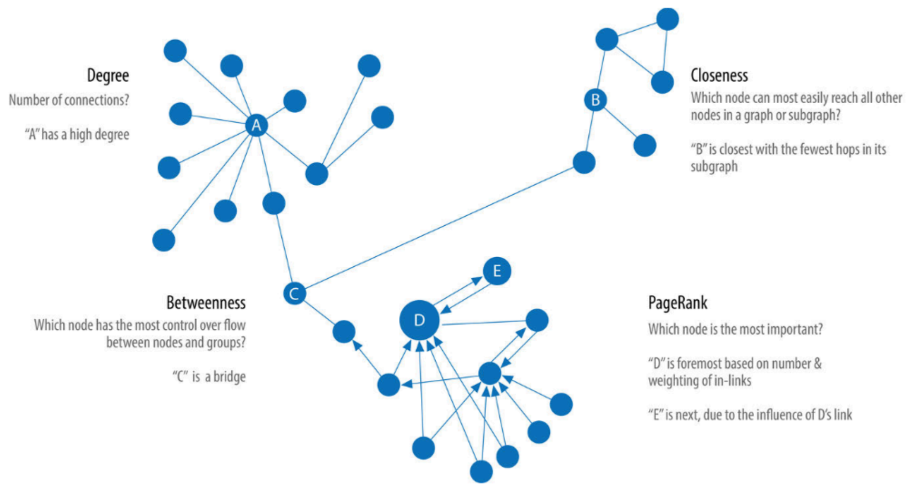
  - Sparse vs Dense
    - 

  - Trees
    - 

- *Types of Graph Algorithms - Pathfinding*
  - <u>Pathfinding</u>

        - Finding the shortest path between two nodes, if one exists, is probably the most common operation
        - "Shortest" means fewest edges or lowest weight
        - Average shortest path can be used to monitor efficiency and resiliency of networks
        - Minimum spanning tree, cycle detection, max/min flow… are other types of pathfinding
- *BFS vs DFS*
    - 

        - Moves from light to dark
- *Shortest path*
    - 
Calculated by relationship weights
A
B
C
D
E
(A, B) = 8
(A, C) = 4 via D
(A, D) = 1
(A, E) = 5 via D
(B, C) = 6
(B, D) = 9 via A or C
And so on...
Shortest Path
Shortest path between 2 nodes (A to C shown)
All-Pairs Shortest Paths
Optimized calculations for shortest paths from all nodes to all other nodes
Single Source Shortest Path
Shortest path from a root node (A shown) to all other nodes
Minimum Spanning Tree
Shortest path connecting all nodes (A start shown)
- *Types of Graph Algorithms - Centrality & Community Detection*
    - <u>Centrality</u>
        - Determining which nodes are "more important" in a network compared to other nodes
        - EX: Social Network Influencers?

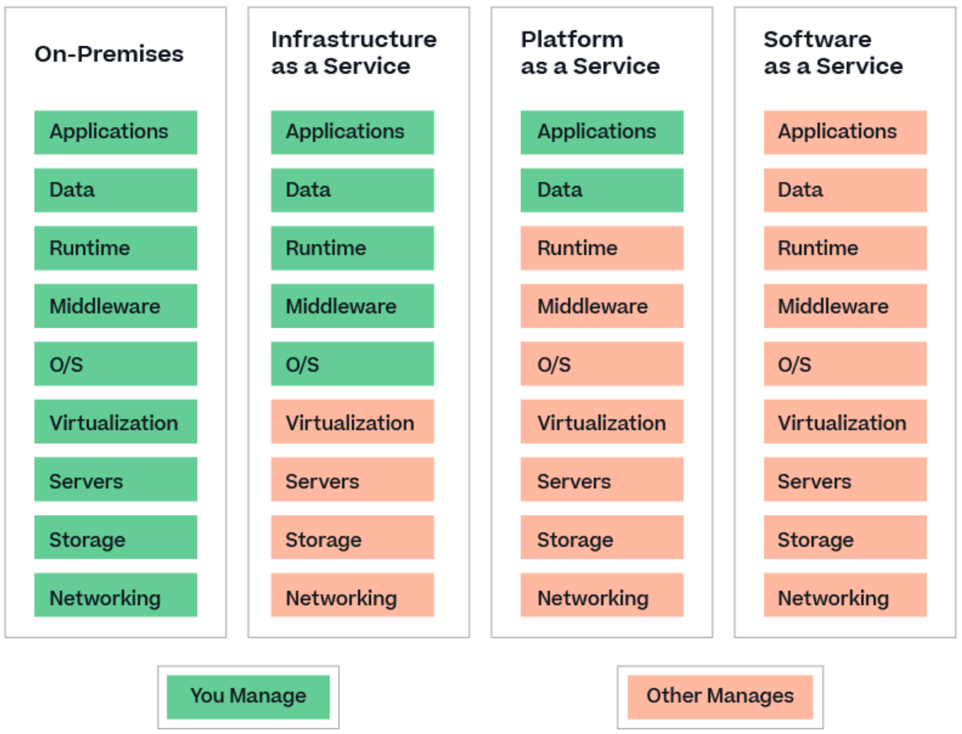
-
        - 
Degree
Number of connections?
"A" has a high degree
Closeness
Which node can most easily reach all other nodes in a graph or subgraph?
"B" is closest with the fewest hops in its subgraph
Betweenness
Which node has the most control over flow between nodes and groups?
"C" is a bridge
PageRank
Which node is the most important?
"D" is foremost based on number & weighting of in-links
"E" is next, due to the influence of D's link
A
B
C
D
E
    - Community Detection
        - Evaluate clustering or partitioning of nodes of a graph and tendency to 7strengthen or break apart
- *Some Famous Graph Algorithms*
    - Dijkstra's Algorithm - single-source shortest path algo for positively weighted graphs
    - A* Algorithm - similar to Dijkstra's with added feature of using a heuristic to guide traversal
    - PageRank - measures the importance of each node within a graph based on the number of incoming relationships and the importance of the nodes from those incoming relationships
- *Neo4j*
    - A graph database system that supports both transactional and analytical processing of graph-based data
    - Relatively new class of no-sql DBs
    - Considered schema optional (one can be imposed)
    - Supports various types of indexing
    - ACID compliant
    - Supports distributed computing
    - Similar: Microsoft CosmoDB, Amazon Neptune
- Maximum port number is 65535
- 0-1023 are ports reserved for root access
- Docker compose commands must be run in the same folder where the docker yml file is

In Class 2/24
- Take class notes
- Generate embeddings
    - Chunk: file name, pdf page, vector
- Store in a redis-stack
- On other side with person:

        - Questio embedding
            - Get 10 most similar embeddings
- Retrieval Augmented Generation (RAG)

AWS Introduction

- Amazon Web Services
    - Leading Cloud Platform with over 200 different services available
    - Globally available via its massive networks of regions and availability zones with their massive data centers
        - Breaks everything into regions and then into availability zones
            - Availability zone think of like a giant data center
    - Based on a pay-as-you-use cost model
        - Theoretically cheaper than renting rackspace/servers in a data center…Theoretically
- History of AWS
    - Originally launched in 2006 with only 2 servies: S3 and EC2
    - By 2010, services had expanded to include SimpleDB, Elastic Block Store, Relational Database Service, DynamoDB, CloudWatch, Simple Workflow, CloudFront, Availability Zones, and others
    - CVN = content delivery network
    - Amazon had competitions with big prizes to spur the adoption of AWS in its early days
    - They've continuously innovated, always introducing new services for ops, dev, analytics, etc
- Cloud Models
    - IaaS - Infrastructure as a Service
        - Contains the basic services that are needed to build an IT infrastructure
    - PaaS - Platform as a Service
        - Remove the need for having to manage infrastructure
        - You can get right to deploying your app
    - SaaS - Software as a System
        - Provide full software apps that are run and managed by another party/vendor



- The Shared Responsibility Model - AWS
  - AWS Responsibilities (Security OF the cloud)
    - Security of physical infrastructure (infra) and network
      - Keep the data centers secure, control access to them
      - Maintain power availability, HVAC, etc.
      - Monitor and maintain physical networking equipment and global infra/connectivity
    - Hypervisor & Host OSs
      - Manage the virtualization layer used in AWS compute services
      - Maintaining underlying host OSs for other services
    - Maintaining managed services
      - Keep infra up to date and functional
      - Maintain server software (patching etc.)
  - Client responsibilities (Security IN the cloud)
    - Control of Data/Content
      - Client controls how its data is classified, encrypted, and shared
      - Implement and enforce appropriate data-handling policies
    - Access management and IAM
      - Properly configure IAM users, roles, and policies
      - Enforce the Principle of Least Privilege
    - Manage self-hosted APps and associated oSs
    - Ensure networks security to its VPC
    - Handle compliance and governance policies and procedures
- The AWS Global Infrastructure

    - Regions - distinct geographical areas
        - Us-east-1, us-west-1, etc
    - Availability Zones (AZs)
        - Each region has multiple AZs
        - Roughly equivalent to isolated data centers
    - Edge Locations
        - Locations for CDN and other types of caching services
        - Allows content to be closer to end user
    - Currently 36 regions totalling 114 availability zones with 700+ POPs (points of presence) with cloudFront
- Compute services
    - Compute resources
        - Resources where you create them and have them available
        - All the way to on-demand serverless instances
    - VM-based
        - EC2 and EC2 Spot - Elastic Cloud Compute
    - Container Based
        - ECS - Elastic container service
        - ECR - Elastic container registry
        - EKS - Elastic Kubernetes Service
        - Fargate - Serverless container service
    - Serverless: AWLS Lambda
- Storage Services
    - Each has its own unique icon
    - Amazon S3 - Simple Storage Service
        - Object storage in buckets; highly scalable; different storage classes
    - Amazon EFS - Elastic File System
        - Simple, serverless, elastic, “set and forget” file system
    - Amazon EBS - Elastic Block Storage
        - High-performance block storage service
    - Amazon File Cache
        - High-speed cache for datasets stored anywhere
    - AWS Backup
        - Fully managed, policy-based service to automate data protection and compliance of apps on AWS
- Database Services
    - Relational - Amazon RDS, Amazon Aurora
    - Key-value - Amazon DynamoDB
    - In-memory - Amazon MemoryDK, Amazon ElastiCache
    - Document - Amazon DocumentDB (compatible with MongoDB)
    - Graph - Amazon Neptune
- Analytics Services
    - Amazon Athena - Analyze petabyte scale data wher it lives (S3, for example)
    - Amazon EMR - Elastic MapReduce - Access APache Spark, Hive, Presto etc.

    - AWS Glue - Discover, prepare, and integrate all your data
    - Amazon Redshift - Data warehousing service
    - Amazon Kinesis - real-time data streaming
    - Amazon QuickSight - cloud-native BI/reporting tool
- ML and AI Services
    - Amazon SageMaker
        - Fully-managed ML platform, including Jupyter NBs
        - Build, train, deploy ML models
    - AWS AI Services w/ Pre-trained Models
        - Amazon comprehend - NLP
        - Amazon Rekognition - Image/Video analysis
        - Amazon Textract - text extraction
        - Amazon translate - machine translation
- Important Services for Data Analytics/Engineering
    - **EC2** and Lambda
        - Elastic Cloud Compute
        - Scalable Virtual Computing in the cloud
        - Pay as you go
        - Many instance types available
        - Multiple different operating systems
    - Amazon S3
    - Amazon RDS and DynamoDB
    - AWS Glue
    - Amazon Athena
    - Amazon EMR
    - Amazon Redshift
- Features of EC2
    - Elasticity - easily (and programmatically) scale instances up or down as needed
    - You can use one of the standard AMIs or provide your own AMI if pre-config is needed
    - Easily integrates with many other services such as S3, RDS, etc.
- EC2 Lifestyle
    - Launch - when starting an instance for the first time with a chosen configuration
    - start/stop - temporarily suspend usage without deleting the instance
    - Terminate - permanently delete the instance
    - Reboot - restart an instance without sling the data on the root volume
- Where can you store data in EC2
    - Instance Store: Temporary high-speed storage tied to the instance lifecycle
    - EFS (Elastic File System) Support - shared file storage
        - Like a thumb drive
    - EBS (elastic block storage) - persistent block-level storage
    - S3 - large data set storage or EC2 backups even
- Common EC2 Use Cases
    - Web Hosting - Run a website/web server and associated apps

    - Data processing - It’s a VM… you can do anything to data possible with a programming language
    - Machine Learning - Train models using GPU instances
    - Disaster Recovery - Backup critical workloads or infrastructure in the cloud

Lambdas 3/17/25

- Lambdas provide serverless computing
- Automatically run code in response to events
- Relieves you from having to manage servers - only worry about the code
- You only pay for execution time, not for idle compute time (different from EC2)

EXAM:

- Mongo query
- How to rotate/insert a tree
- Support mapping