Groups

1) Data Structures (2)
2) Scaling and Distributed Systems (3 and 4)
3) KV/Redis (5 and 6)
4) Document/Mongo (7 and 8)
5) Graph/Neo4j/Other (9 and 10)

Questions/Changes

- *Possibly all images*
- Module 2 → AVL, B+, and Hash Tables use more
- Module 6 → All redis commands
- Module 7 and 8 → Could add more

Module 1 - Introduction

End of Course Takeaways

- Understand the efficiency-related concepts (including limitations) of RDBMSs
- Understand data replication and distribution effects on typical DB usage scenarios
- Understand the use cases for and data models of various NoSQL database systems, including storing and retrieving data. Data models include document-based, key-value stores, graph based among others.
- Access and implement data engineering and big-data-related AWS services

Tentative List of Topics

- Thinking about data storage and retrieval at the data structures level
- How far can we get with the relational model?
- NoSQL Databases
    - Document Databases (Mongo)
    - Graph Databases (Neo4j)
    - Key/Value Databases
    - Maybe Vector Databases
- Data Distribution and Replication
- Distributed SQL DBs & Apache Spark/SparkSQL
- Big Data Tools and Services on AWS

Module 2 - Data Structure's Effects on Performance

Searching

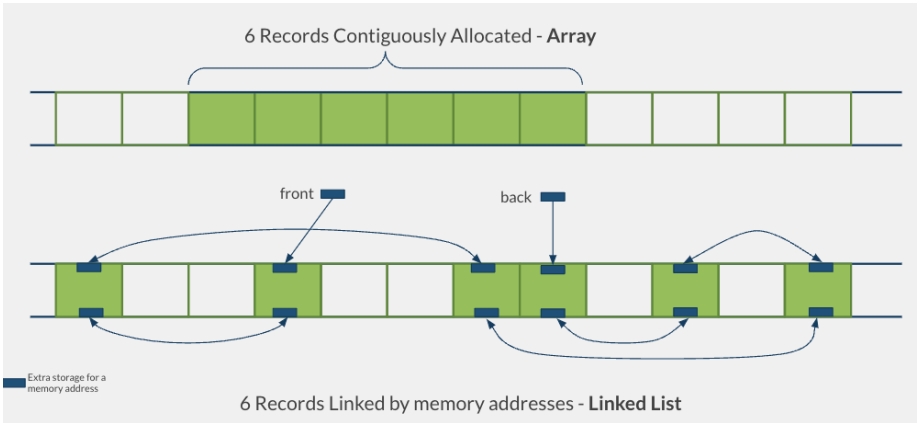
- Searching is the most common operation performed by a database system
- In SQL, the SELECT statement is arguably the most versatile / complex.
- Baseline for efficiency is Linear Search
    - Start at the beginning of a list and proceed element by element until:
        - You find what you're looking for
        - You get to the last element and haven't found it
    - Time Complexity:
        - Best case: O(1) - target found at the first position
        - Worst case: O(n) - target not found or at the last position

        - Average case: O(n/2) ≈ O(n)
- Related Terminology
    - Record - A collection of values for attributes of a single entity instance; a row of a table
    - Collection - a set of records of the same entity type; a table
        - Trivially, stored in some sequential order like a list
    - Search Key - A value for an attribute from the entity type
        - Could be >= 1 of the attribute

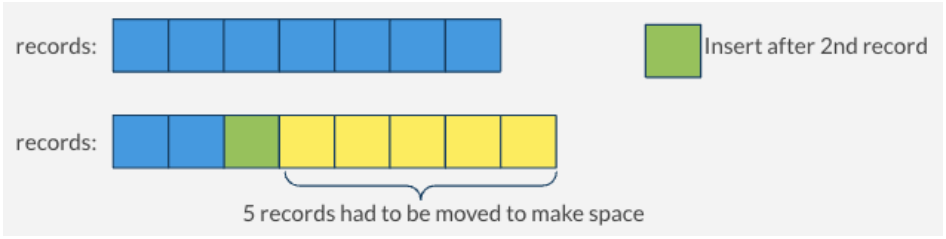
List of Records

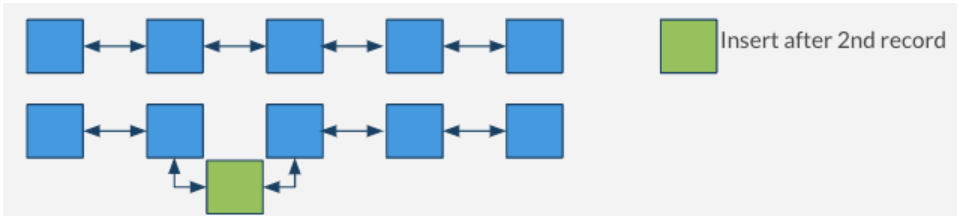
- Memory Allocation
    - If each record takes up x bytes of memory, then for n records, we need n*x bytes of memory.
- Contiguously Allocated List
    - All n*x bytes are allocated as a single “chunk” of memory
        - Characteristics:
            - Fast random access (constant time) using index
            - Slow for insertions/deletions except at the end
            - Requires memory reallocation when resizing
            - Excellent memory locality (benefits CPU caching)
- Linked List
    - Each record needs x bytes + additional space for 1 or 2 memory addresses
    - Individual records are linked together in a type of chain using memory addresses
        - Characteristics:
            - Slow random access (linear time)
            - Fast insertions/deletions anywhere in the list
            - No need for contiguous memory allocation
            - Poor memory locality (may cause cache misses)

- Contiguous vs. Linked



    - Pros and Cons
        - Arrays are faster for random access, but slow for inserting anywhere but the end

- 

    - Linked Lists are faster for inserting anywhere in the list, but slower for random access
- 

    - Comparison Summary

Operation	Array	Linked List
Random Access	O(1)	O(n)
Insertion (beginning)	O(n)	O(1)
Insertion (middle)	O(n)	O(1)*
Insertion (end)	O(1)**	O(1)***
Deletion	O(n)	O(1)*
Memory Usage	n×x bytes	n×(x+p) bytes

*Assuming we have a pointer to the insertion/deletion location
**Assuming array has available capacity
***Assuming we have a tail pointer

Binary Search
- About
    - “Dictionary Search” splits in half, and recursively narrows down
    - Must be sorted
    - 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
    - The collection must be sorted
    - The collection must support random access (like arrays)

- Code Example

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # Target not found
```

- Visual Example
    - For an array `[A, C, G, M, P, R, Z]` searching for target = 'A':
        1. Initialize: left=0, right=6
        2. Calculate mid=3, arr[mid]='M'
        3. 'A' < 'M', so set right=2
        4. Calculate mid=1, arr[mid]='C'
        5. 'A' < 'C', so set right=0
        6. Calculate mid=0, arr[mid]='A'
        7. 'A' == 'A', return index 0
- Time (Linear Search vs. Binary Search)
    - 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; log2 n comparisons
        - Therefore, in the worst case, binary search is O(log2n) time complexity.
- Practical Database Searching Example
    - Original case
        - 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
        - Can't store data on disk sorted by both id and specialVal (at the same time)
            - data would have to be duplicated → space inefficient
    - Other Options
        - 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 a specialVal would be slow - linear scan required
        - But inserting into the table would theoretically be quick to also add to the list.

Binary Search Tree

- About
    - Fast insert and fast search
- Process
    - First insertion is the root of the tree
    - Insert value to the left or right based on if it is larger or smaller
    - Continue this iteratively
- Traversals
    - Pre Order, Post Order, In order, and Level Order
    - Output of Level Order Traversal
        - Write out numerically, top down, and left to right by level
- Order of insertions
    - The order of insertion affects the shape of the tree
    - If you go in order it will just be a sorted linked list
    - Balanced trees will be faster as you scale up, so you want to minimize the height of the tree
    - To minimize, you should fill each row
- BST Properties
    - Every node has at most two children (left and right)
    - All nodes in the left subtree have values less than the parent node
    - All nodes in the right subtree have values greater than the parent node
- BST Operations
    - Search: O(h) where h is the height of the tree
        - Best case (balanced tree): O(log n)
        - Worst case (degenerate tree): O(n)
    - Insert: O(h) - find the appropriate leaf position and add the node
    - Delete: O(h) - find the node and restructure the tree as needed
- BST Advantages for Database Indexing
    - Provides a compromise between arrays and linked lists
    - Supports both reasonably fast searches and insertions
    - Can be balanced to ensure O(log n) performance (leading to structures like B-trees and B+ trees)

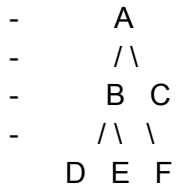
Creating/Inserting into a Binary Search Tree

- Tree Traversals
    - By “traversal” we mean visiting all the nodes in a graph. Traversal strategies can be specified by the ordering of the three objects to visit: the current node, the left

subtree, and the right subtree. We assume that the left subtree always comes before the right subtree.

- Types of traversals:
  - Pre Order Traversal
    - Visit the root node first.
    - Recursively traverse the left subtree.
    - Recursively traverse the right subtree.

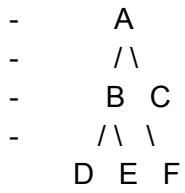
    Example:



    Preorder Traversal Output: `A B D E C F`

  - Post Order Traversal
    - Recursively traverse the left subtree.
    - Recursively traverse the right subtree.
    - Visit the root node.

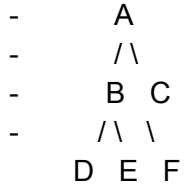
    Example:



    Postorder Traversal Output: `D E B F C A`
    (Used in applications like deleting a tree or evaluating expressions.)

  - In Order Traversal
    - Recursively traverse the left subtree.
    - Visit the root node.
    - Recursively traverse the right subtree.

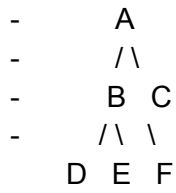
    Example:



Inorder Traversal Output: `D B E A C F`
(Commonly used for Binary Search Trees, as it results in sorted order.)

- Level Order Traversal
  - Visit nodes level by level from top to bottom.
  - Uses a queue data structure.

Example:



Level Order Traversal Output: `A B C D E F`
(Used in shortest path algorithms, such as in graphs.)

AVL Tree

- About
  - Approximately balanced binary search tree.
  - Self-balancing
    - It reroute the tree, so if it's going to double down, it rearranges the 3 to not go into a line
- Process
  - Maintains balance factor
    - |h(LST)-h(RST)| <= 1
    - LST = Left Sub Tree
    - RST = Right Sub Tree
  - Node of imbalance
    - Nodes without children are always balanced
    - Node of imbalance is the first one that goes 2 up without balance (Also; referred to as alpha)
  - 4 Cases of imbalance
    - LL (Left Left): Goes down left twice *Single Rotation*
    -

Then, there is an insertion somewhere in T1 that causes A's height to increase by 1. Now, height(C.left) and height(C.right) differ by 2.

Important Observations:
- All values in T1 are smaller than both A and C.
- All values in T2 are bigger than A but smaller than C
- All values in T3 are bigger than C.

--> Since all values in T2 are bigger than A but smaller than C, that subtree could live to the right of A or the left of C

-

We can adjust for this imbalance with a SINGLE ROTATION by rotating the node of imbalance with its left child.

Notice after the re-balancing, A has the same height as C did in the original tree.

-

```
fun rotateWithLeftChild(C, parentOfC):
    A = C.left
    C.left = A.right
    A.right = C

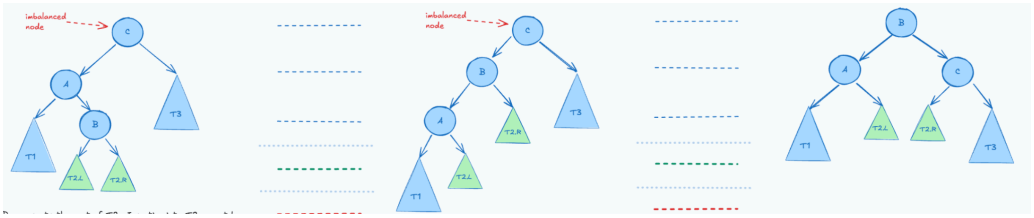
    if C == root of tree:
        root of tree = A
    else:
        if parentOfC.left = C:
            parentOfC.left = A
        else
            parentOfC.right = A

    updateHeight(C)
    updateHeight(A)
```

-
- Adding underneath T1
- You can rebalance like this, because T2 is less than C and greater than A, and it stays like that in the new form

- LR (Left Right): Goes down left then right *Double Rotation*

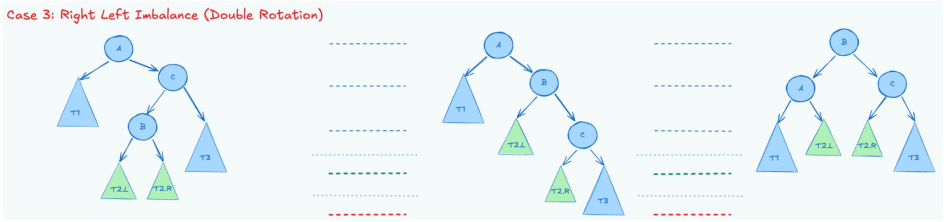
-

- 

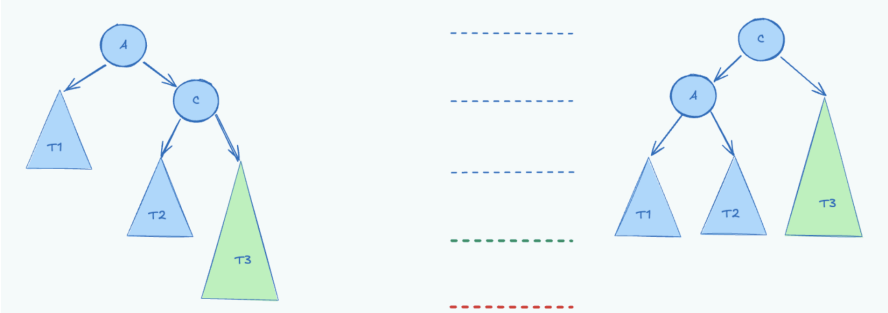
- B represents the root of T2. Inserting into T2 cannot be fixed by a single rotation (shown above). T2.L and T2.R are shown as half way to the next level because only 1 (where the insertion happened) would extend down to the red --- level.

  Observations:
  - All values in T2.L fall between A and B.
    - So, T2.L could be the left child of B or the right child of A.
  - All values in T2.R fall between B and C.
    - So, T2.R could be the right child of B or the left child of C.

  To fix:
  Step 1: Rotate A with its right child (B). (same rotation as RR imbalance)
  Step 2: Rotate C with its left child (now it is B after step 1).
- After this rotation, either T2.L or T2.R will be as deep as T1 and T3, but not both.
- ```
  fun doubleRotateWithLeftChild(C, parentOfC):
      rotateWithRightChild(A, parentOfA)
      rotateWithLeftChild(C, parentOfC)
  ```
- Adding underneath T2
- Need a double rotation. In a simple sense, because it's in the middle. But also, you need to fix the nodes of imbalance one by one

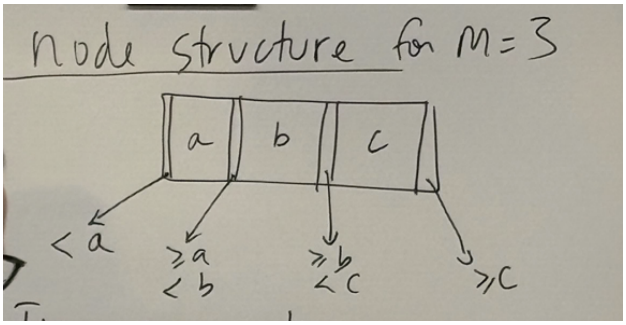
- RL (Right Left): Goes down right then left
  - Mirror of Case 2 (LR)
  - 

- RR (Right Right): Goes down left then right
  - Mirror of Case 1 (LL)
  ```

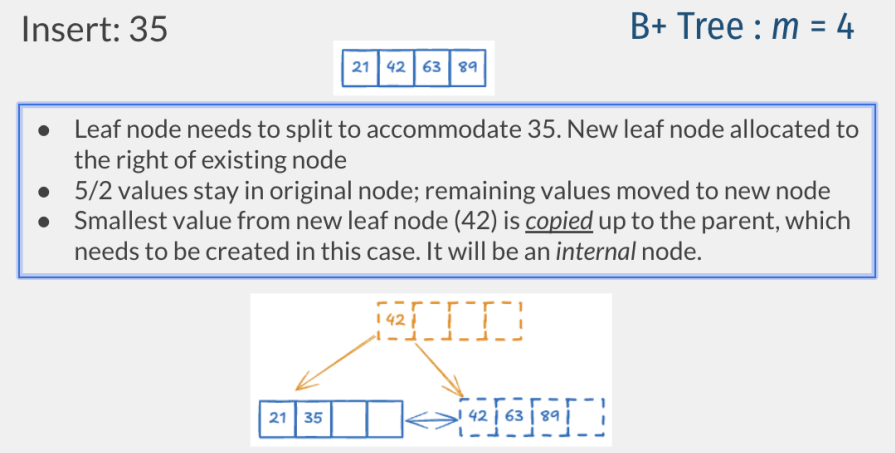
- 

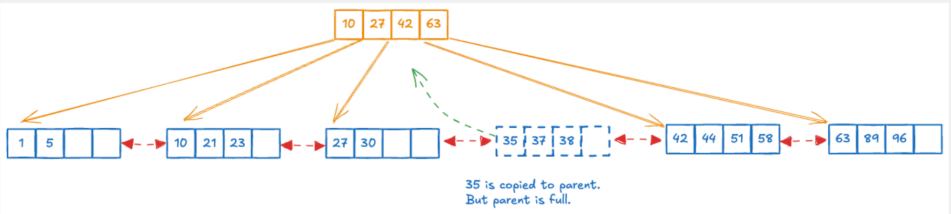
    - General Rotation Methodology
        - Identify node of imbalance
        - Check the height of the subtrees, the one that is has subs off by more than 1 is imbalance
        - Only rotate 3 at a time (or at least fully move, even when up in the tree)
        - Repeat until balanced

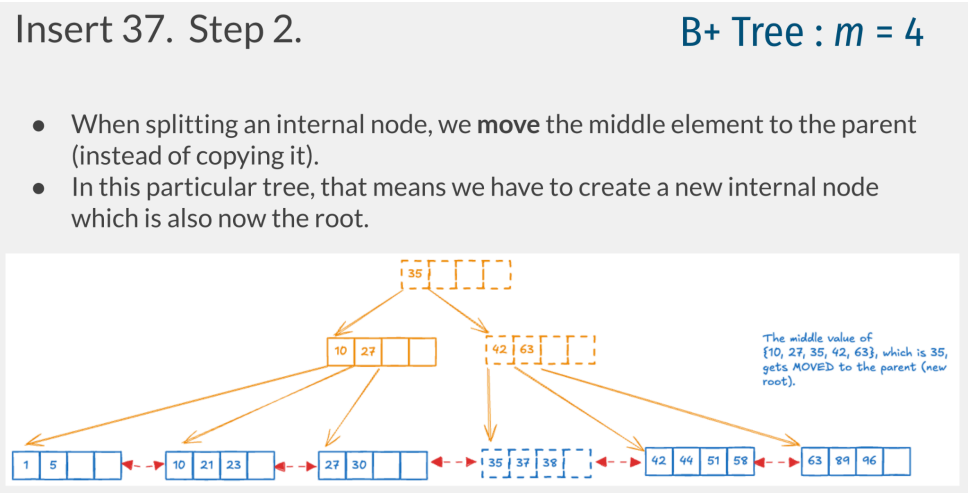
B+ Tree

- About
    - Not the same as a B Tree
    - Properties
        - Self balancing tree
        - Maintains sorted data
        - Allows for efficient insertion/deletion and searching
        - Bottom row is memory references
        - Optimized for disk based indexing
        - Minimizing disk accesses for indexing
    - M-way tree with order M
        - M → maximum # of keys in each node
        - M+1 → max children of each node
        - 

- Process
    - All nodes (except the root) must be ½ full minimum (meaning # of children)
    - Insertions always in leaf nodes first. Then copied into the internal nodes for indexing
    - All values are stored at the leaf nodes, while internal nodes only store keys to guide the search process.
    - Node structure
        - First node is a root node

    - Internal nodes hold keys. Acts more as an index
    - Leaf nodes hold data. Only bottom row is leaf nodes.
- Leave are stored as a DLL (Double linked list)
- Keys and nodes are kept sorted
- Steps/Cases
    - When you cause a split at the leaf node, you copy the left of the new node up to the parent
    - 

    - When you have a 2 layer split, you do the first level the same way. When you split the parent, you don't copy up to the top, just add one to the root.
    - 

    - 

Hash Tables

- About
    - Like a Python Dictionary
    - Give each element an address (index)

        - Benefits and Drawbacks
            - Constant time work to hash and calculate new index
        - Programming Principles
            - Don't make it a python list (the table itself), but it can be a numpy array with elements as lists
- Process
    - Take the key (index)
    - Hash it with the chosen hash function (modulo(%) is the usual function)
        - hashIndex = k % n
        - Insert element into table at new index
        - Add a tuple, original index, and element
    - Other Information
        - Table size=m
        - k=current value you want to insert (also known as key)
        - Number of inserted values=n
        - λ load factor = n/m
            - Want to keep it below .9
            - tables may be cluttered and have longer search times and collisions if the load factor is high. An ideal load factor can be maintained with the use of a good hash function and proper table resizing
            - Dynamic resizing:
                - This feature enables the hash table to expand or contract in response to changes in the number of elements contained in the table. This promotes a load factor that is ideal and quick lookup times
        - More buckets → less likelihood of buckets having multiple things
    - Applications of Hash Table:
        - Hash tables are frequently used for indexing and searching massive volumes of data. A search engine might use a hash table to store the web pages that it has indexed.
        - Data is usually cached in memory via hash tables, enabling rapid access to frequently used information.
        - Hash functions are frequently used in cryptography to create digital signatures, validate data, and guarantee data integrity.
        - Hash tables can be used for implementing database indexes, enabling fast access to data based on key values.

Other Information

- Python Collections
    - Not a queue, but a deque
    - You can insert and remove from both ends
- Binary Tree Class
    - Attributes of a node
        - Value: int

        - Left: Binary Tree Node
        - Right: Binary Tree Node
- Hardware Structure
    - Structure (descending my capacity)
        - CPU
        - Registers
        - L1 Cache
        - L2 Cache
        - RAM
        - SDD/HDD
            - Lots of storage
            - Persistent
            - Very slow
            - Specific memory conventions
                - 2048 byte block size
                - DB systems have minimum access
    - Can use an AVL tree to search memory

Module 3 - Moving Beyond the Relational Model

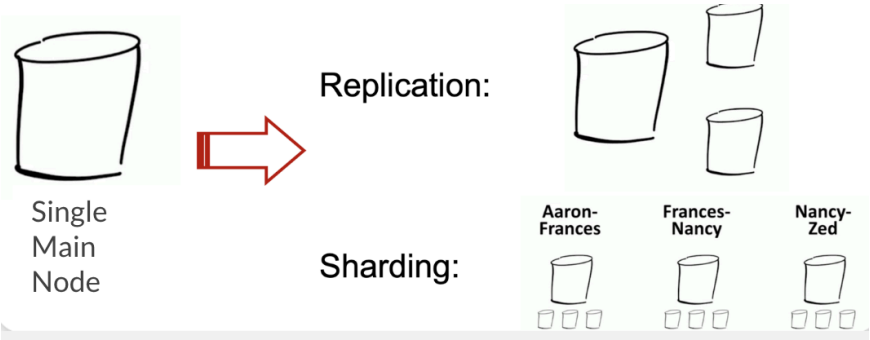
Benefits of Relational Model

- Mostly standard data model and query language
- ACID Compliance (more on this in a second)
    - Atomicity, Consistency, Isolation, Durability
    - Atomicity
        - Transaction is treated as an atomic unit. It is fully executed or no parts of it are executed
    - Consistency
        - Transaction takes a database from one consistent state to another consistent state
        - Consistent state → all data meets integrity constraints
    - Isolation
        - Two transactions T1 and T2 are being executed at the same time but cannot affect each other
            - If both T1 and T2 are reading the data - no problem
            - If T1 is reading the same data that T2 may be writing, can result in:
        - Types of bad reads
            - Dirty Read: a transaction T1 is able to read a row that has been modified by another transaction T2 that hasn't yet executed a COMMIT
            - Non-repeatable Read: two queries in a single transaction T1 execute a SELECT but get different values because another transaction T2 has changed data and COMMITTED

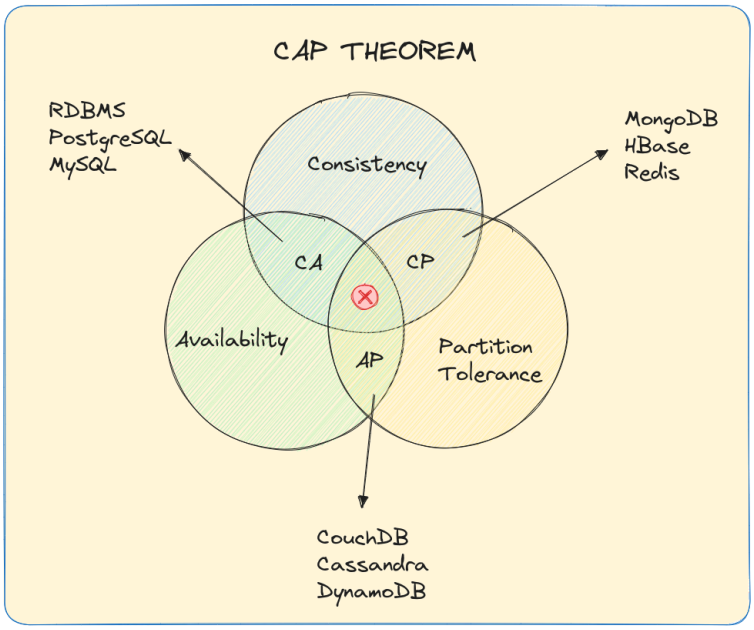
        - Phantom Reads: when a transaction T1 is running and another transaction T2 adds or deletes rows from the set T1 is using
    - Durability
        - Once a transaction is completed and committed successfully, its changes are permanent.
        - Even in the event of a system failure, committed transactions are preserved
- Ways they increase efficiency
    - indexing (the topic we focused on)
    - 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 (CRUD = Create, Read, Update, or Delete)
        - Either the entire sequence succeeds (COMMIT)
        - OR the entire sequence fails (ROLLBACK or ABORT)
- Scalability
    - 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.

Distributed Storage

- About
    - A distributed system is “a collection of independent computers that appear to its users as one computer.” -Andrew Tennenbaum
    - Characteristics of Distributed Systems:
        - Computers operate concurrently
        - Computers fail independently
        - No shared global clock
    - 2 Directions



    -
    - Sharding (Fragmenting) is when different things are stored in different places
- More details
    - 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
    - Network partitioning is inevitable
        - Network failures, system failures
        - Overall system needs to be Partition Tolerant
            - System can keep running even w/ network partition
- Advantages of Distributed Database System :
    - There is fast data processing as several sites participate in request processing.
    - Reliability and availability of this system is high.
    - It possesses reduced operating cost.
    - It is easier to expand the system by adding more sites.
    - It has improved sharing ability and local autonomy.
- Disadvantages of Distributed Database System :
    - The system becomes complex to manage and control.
    - The security issues must be carefully managed.
    - The system require deadlock handling during the transaction processing otherwise
    - the entire system may be in an inconsistent state.
    - There is a need for some standardization for processing of distributed database systems.
- CAP Theory
    - Consistency, Availability, and Partition Tolerance → You can never have all 3 (2 not 3)
        - Consistency: Every read receives the most recent write or error thrown
        - Availability: Every request receives a (non-error) response - but no guarantee that the response contains the most recent write
        - Partition Tolerance: The system can continue to operate despite arbitrary network issues.



-
    - Consistency + Availability: System always responds with the latest data and every request gets a response, but may not be able to deal with network partitions
    - Consistency + Partition Tolerance: If the system responds with data from the distrib. system, it is always the latest, else data request is dropped.
    - Availability + Partition Tolerance: System always sends are responds based on distributed store, but may not be the absolute latest data
    -
- 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: consistency, availability, or tolerance to failure.

Module 4 - Data Replication

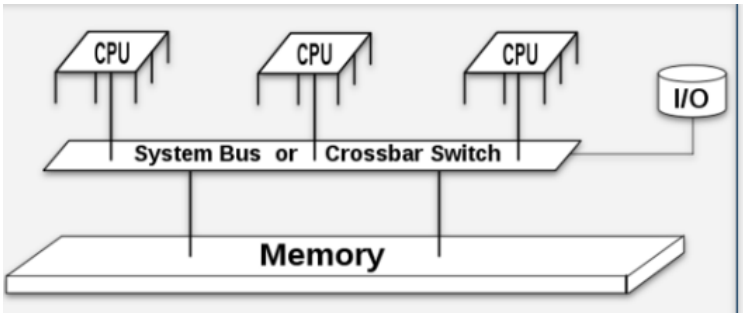
About Data Replication

- Used in distributed systems
- Benefits of Distributing Data
    - Scalability / High throughput: Data volume or Read/Write load grows beyond the capacity of a single machine
    - Fault Tolerance / High Availability: Your application needs to continue working even if one or more machines goes down.
    - Latency: When you have users in different parts of the world you want to give them fast performance too
- Challenges

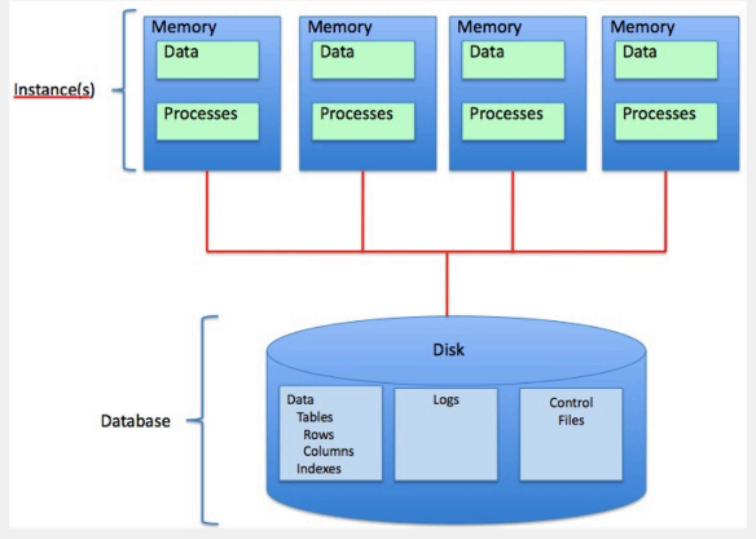
    - Consistency: Updates must be propagated across the network.
    - Application Complexity: Responsibility for reading and writing data in a distributed environment often falls to the application.
- Common Strategies for Replication (one at a time)
    - Single leader model
    - Multiple leader model
    - Leaderless model
- Leader-Based Replication
    - All writes from clients go to the leader
    - Leader sends replication info to the followers
    - Followers process the instructions from the leader
    - Clients can read from either the leader or followers
- Transmission to Followers
    - Statement-based: Send INSERT, UPDATE, DELETEs to replica. Simple but error-prone due to non-deterministic functions like now(), trigger side-effects, and difficulty in handling concurrent transactions.
    - Write-ahead Log (WAL): A byte-level specific log of every change to the database. Leaders and all followers must implement the same storage engine and make upgrades difficult.
    - Logical (row-based) Log: For relational DBs: Inserted rows, modified rows (before and after), deleted rows. A transaction log will identify all the rows that changed in each transaction and how they changed. Logical logs are decoupled from the storage engine and easier to parse.
    - Trigger-Based: Changes are logged to a separate table whenever a trigger fires in response to an insert, update, or delete. Flexible because you can have application specific replication, but also more error prone.
- Synchronous vs. Asynchronous Replication
    - Synchronous: Leader waits for a response from the follower
    - Asynchronous: Leader doesn't wait for confirmation.
- Challenges (If the Leader Fails)
    - How do we pick a new Leader Node?
        - Consensus strategy – perhaps based on who has the most updates?
        - Use a controller node to appoint new leader?
    - If asynchronous replication is used, new leader may not have all the writes
    - How do we recover the lost writes? Or do we simply discard?
    - After (if?) the old leader recovers, how do we avoid having multiple leaders receiving conflicting data? (Split brain: no way to resolve conflicting requests.
    - Leader failure detection. Optimal timeout is tricky.
- Replication Lag
    - Replication Lag refers to the time it takes for writes on the leader to be reflected on all of the followers.
    - Synchronous replication: Replication lag causes writes to be slower and the system to be more brittle as num followers increases.

    - Asynchronous replication: We maintain availability but at the cost of delayed or eventual consistency. This delay is called the inconsistency window.
- Read-after-Write Consistency
    - Ex: You're adding a comment to a Reddit post… after you click Submit and are back at the main post, your comment should show up for you.
    - Method 1: Modifiable data (from the client's perspective) is always read from the leader.
    - Method 2: Dynamically switch to reading from leader for "recently updated" data. For example, have a policy that all requests within one minute of last update come from leader.
- Monotonic Read Consistency
    - Monotonic read anomalies: occur when a user reads values out of order from multiple followers.
    - Monotonic read consistency: ensures that when a user makes multiple reads, they will not read older data after previously reading newer data.
- Consistent Prefix Reads
    - Reading data out of order can occur if different partitions replicate data at different rates. There is no global write consistency.
    - Consistent Prefix Read Guarantee - ensures that if a sequence of writes happens in a certain order, anyone reading those writes will see them appear in the same order.

Vertical scaling

- Increasing the capability of one unit or machine
- Shared Memory architectures
    - Geographically Centralized server
    - Some fault tolerance (via hot-swappable components)
    - 

- Shared Disk Architecture
    - Machines are connected via a fast network
    - Contention and the overhead of locking limit scalability (high-write volumes) … BUT ok for Data Warehouse applications (high read volumes)
    -

- 

Horizontal Scaling

- Each node has its own CPU, memory, and disk
- Coordination via application layer using conventional network
- Geographically distributed
- Commodity hardware

Replication vs. Partitioning

- Replicates have same data as Main
- Partitions have a subset of the data

Module 5 - NoSQL and KV DBs

Pessimistic Concurrency (Distributed DBs and ACID)

- ACID transactions
    - Focuses on “data safety”
    - Considered a pessimistic concurrency model because it assumes one transaction has to protect itself from other transactions. In other words, it assumes that 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.

Optimistic Concurrency

- Transactions do not 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 still be OK.
- Management
    - Add last update timestamp and version number columns to every table… read them when changing. THEN, check at the end of the transaction to see if any other transaction has caused them to be modified.

Concurrency Scenarios

- 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

ACID Alternative

- ACID is very computationally expensive
- BASE Model (implements CAP Theory)
    - Some level of unreliability
    - BA → Basically Available
        - Guarantees the availability of the data (per CAP), but response can be “failure”/“unreliable” because the data is in an inconsistent or changing state
        - System appears to work most of the time
    - Soft State → The state of the system could change over time, even w/o input. Changes could be 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

NoSQL

- Originally, a relational DB system that does not use SQL
- Today means “Not Only SQL”

KV Databases

- Key = values
- Value can be any arbitrary data (string, number, JSON object, binary object, etc)
- Designed
    - 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 deployed as in-memory DB
        - retrieving a value given its key is typically a O(1) op b/c hash tables or similar data structs 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.
- 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 the model, for example, in real-time inferencing.
- 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 browsers, machines, sessions
- Caching Layer
    - In front of a disk-based database

Redis DB

- About
    - Redis → Remote Directory Server
    - Open source, in-memory database
    - Sometimes called a data structure store
    - It is considered an in-memory database system, but…
        - Supports durability of data by: a) essentially saving snapshots to disk at specific intervals or b) append-only file which is a journal of changes that can be used for roll-forward if there is a failure
    - Originally developed in 2009 in C++
    - Can be very fast … > 100,000 SET ops / second
    - Rich collection of commands (rather than SQL)
    - Primarily a KV store, but can be used with other models: Graph, Spatial, Full Text Search, Vector, Time Series
    - Does NOT handle complex data. No secondary indexes. Only supports lookup by Key. This is how it's different from MySQL, which is relational and by tables.
    - Redis Data Types
        - Keys:
            - usually strings but can be any binary sequence
        - Values:
            - Strings
            - Lists (linked lists)
            - Sets (unique unsorted string elements)
            - Sorted Sets

        - Hashes (string → string)
        - Geospatial data
    - Redis Database 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 are available as well.
- Basic Commands
    - Initial Basic Commands
        - SET /path/to/resource 0
        - SET user:1 “John Doe”
        - GET /path/to/resource
        - EXISTS user:1
        - DEL user:1
        - KEYS user*
        - SELECT 5
            - select a different database
    - More Basic Commands
        - SET someValue 0
        - INCR someValue #increment by 1
        - INCRBY someValue 10 #increment by 10
        - DECR someValue #decrement by 1
        - DECRBY someValue 5 #decrement by 5
            - INCR parses the value as int and increments (or adds to value)
        - SETNX key value
            - only sets value to key if key does not already exist
- Data Types
    - 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
    - Hash Type
        - Value of KV entry is a collection of field-value pairs (this is the data type)
        - Use Cases:
            - Can be used to represent basic objects/structures
                - number of field/value pairs per hash is 2^32-1
                - practical limit: available system resources (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
- List Type
    - Value of KV Pair is linked lists of string values
    - Use Cases:
        - implementation of stacks and queues
        - queue management & 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
    - About Linked Lists
        - Sequential data structure of linked nodes (instead of contiguously allocated memory)
        - Each node points to the next element of the list (except the last one - points to nil/null)
        - O(1) to insert new value at front or insert new value at end
    - List commands - queue
        - Queue-like Ops
        - LPUSH bikes:repairs bike:1
        - LPUSH bikes:repairs bike:2
        - RPOP bikes:repairs
        - RPOP biles:repairs
    - List Commands - Stack
        - Stack-like Ops
        - LPUSH bikes:repairs bike:1
        - LPUSH bikes:repairs bike:2
        - LPOP bikes:repairs
        - LPOP biles:repairs
    - List commands - others
        - Other List Ops
        - LLEN mylist
        - LRANGE <key> <start> <stop>
        - LRANGE mylist 0 3
        - LRANGE mylist 0 0

        - LRANGE mylist -2 -1
- 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 (IP addresses visiting a site, page, screen)
        - primitive relation (set of all students in DS4300)
        - access control lists for users and permission structures
        - social network friends lists and/or group membership
    - Supports set operations!!
    - Set commands
        - SADD ds4300 “Mark”
        - SADD ds4300 “Sam”
        - SADD cs3200 “Nick”
        - SADD cs3200 “Sam”
        - SISMEMBER ds4300 “Mark”
        - SISMEMBER ds4300 “Nick”
        - SCARD ds4300
        - SADD ds4300 “Mark”
        - SADD ds4300 “Sam”
        - SADD cs3200 “Nick”
        - SADD cs3200 “Sam”
        - SCARD ds4300
        - SINTER ds4300 cs3200
        - SDIFF ds4300 cs3200
        - SREM ds4300 “Mark”
        - SRANDMEMBER ds4300

Module 6 - Redis + Python

Redis Python Process and Setup

- Redis-py is the standard client for Python.
- Maintained by the Redis Company itself
- Full Command List https://redis.io/docs/latest/commands/

Redis Python Code

- Import and setup
    - import redis
    - redis_client = redis.Redis(host=’localhost’,
        port=6379,
        db=2,

decode_responses=True)

- #Port is the port mapping given when you created the container (probably the default 6379)
- #db is the database 0-15 you want to connect to
- #decode_responses → data comes back from server as bytes. Setting this true converter them (decodes) to strings.

- String Commands
  - # r represents the Redis client object
  - r.set(‘clickCount:/abc’, 0)
  - val = r.get(‘clickCount:/abc’)
  - r.incr(‘clickCount:/abc’)
  - ret_val = r.get(‘clickCount:/abc’)
  - print(f’click count = {ret_val}’)
- More String Commands
  - # r represents the Redis client object
  - redis_client.mset({'key1': 'val1',
    'key2': 'val2',
    'key3': 'val3'})
  - print(redis_client.mget('key1',
    'key2',
    'key3'))
  - # returns as list [‘val1’, ‘val2’, ‘val3’]
- More String Commands
  - set(), mset(), setex(), msetnx(), setnx()
  - get(), mget(), getex(), getdel()
  - incr(), decr(), incrby(), decrby()
  - strlen(), append()
- List Commands
  - # create list: key = ‘names’
  - # values = [‘mark’, ‘sam’, ‘nick’]
  - redis_client.rpush('names',
    'mark', 'sam', 'nick')
  - # prints [‘mark’, ‘sam’, ‘nick’]
  - print(redis_client.lrange('names', 0, -1))
- More List Commands
  - lpush(), lpop(), lset(), lrem()
  - rpush(), rpop()
  - lrange(), llen(), lpos()
  - Other commands include moving elements between lists, popping from multiple lists at the same time, etc.
- Hash Commands
  - redis_client.hset('user-session:123',
    mapping={'first': 'Sam',
    'last': 'Uelle',

'company': 'Redis',
'age': 30

- })
- # prints:
- #{'name': 'Sam', 'surname': 'Uelle', 'company': 'Redis', 'age': '30'}
- print(redis_client.hgetall('user-session:123'))

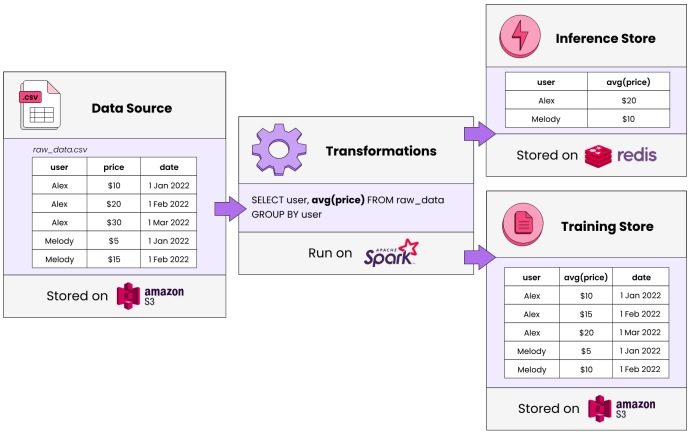
- More Hash Commands
  - hset(), hget(), hgetall()
  - hkeys()
  - hdel(), hexists(), hlen(), hstrlen()
- Redis Pipelines
  - Helps avoid multiple related calls to the server → less network overhead
    - r = redis.Redis(decode_responses=True)
    - pipe = r.pipeline()

    - for i in range(5):
    - pipe.set(f"seat:{i}", f"#{i}")

    - set_5_result = pipe.execute()
    - print(set_5_result) # >>> [True, True, True, True, True]

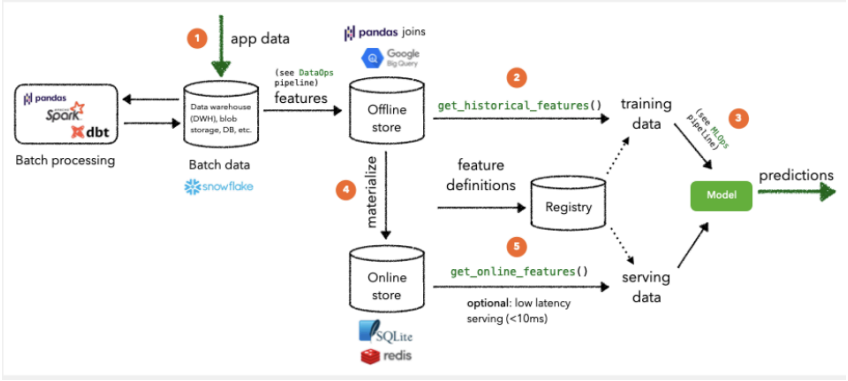
    - pipe = r.pipeline()

    - # "Chain" pipeline commands together.
    - get_3_result = pipe.get("seat:0").get("seat:3").get("seat:4").execute()
    - print(get_3_result) # >>> ['#0', '#3', '#4']
- Redis in ML



-

-



Module 7 - Document DBs + MongoDB

Document DBs

- A Document Database is a non-relational database that stores data as structured documents, usually in JSON.
- They are designed to be simple, flexible, and scalable.
- What is JSON
    - JSON (JavaScript Object Notation)
        - a lightweight data-interchange format
        - It is easy for humans to read and write.
        - It is easy for machines to parse and generate.
    - JSON is built on two structures:
        - 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 format.
    - Example format

```
{
  "users": [
    {
      "id": 1,
      "name": "Alice",
      "email": "alice@example.com",
      "isActive": true,
      "roles": ["admin", "editor"],
      "profile": {
        "age": 30,
        "city": "Boston",
        "interests": ["hiking", "reading",
      }
    },
    {
      "id": 2,
      "name": "Bob",
      "email": "bob@example.com",
      "isActive": false,
      "roles": ["viewer"],
      "profile": {
        "age": 24,
        "city": "Chicago",
        "interests": ["gaming", "cycling"]
      }
    }
  ]
}
```

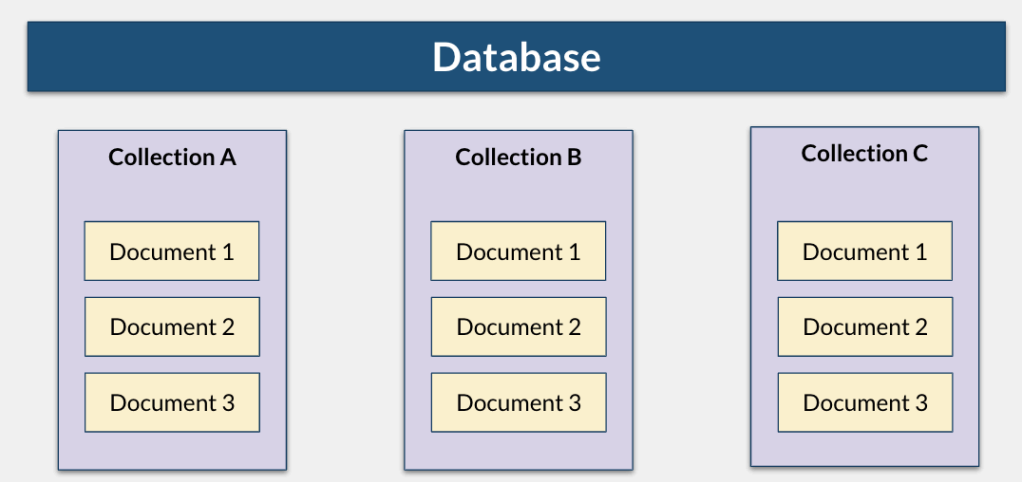
- Binary JSON (BSON)
  - binary-encoded serialization of a JSON-like document structure
  - supports extended types not part of basic JSON (e.g. Date, BinaryData, etc)
  - Lightweight - keep space overhead to a minimum
  - Traversable - designed to be easily traversed, which is vitally important to a document DB
  - Efficient - encoding and decoding must be efficient
  - Supported by many modern programming languages
- 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

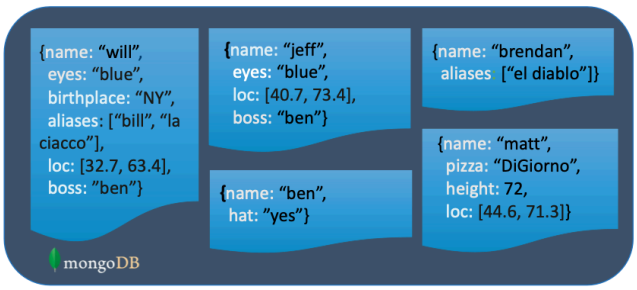
```
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD>
    <TITLE>Hide your heart</TITLE>
    <ARTIST>Bonnie Tyler</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
  </CD>
</CATALOG>
```

-
    - XML related tools/technologies
        - Xpath - a syntax for retrieving specific elements from an XML doc
        - Xquery - a query language for interrogating XML documents; the SQL of XML
        - DTD - Document Type Definition - a language for describing the allowed structure of an XML document
        - XSLT - eXtensible Stylesheet Language Transformation - tool to transform XML into other formats, including non-XML formats such as HTML.
- Why document databases?
    - Document databases address the impedance mismatch problem between object persistence in OO systems and how relational DBs 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

Mongo DB

- Started in 2007 after Doubleclick was acquired by Google, and 3 of its veterans realized the limitations of relational databases for serving > 400,000 ads per second
- MongoDB was short for Humongous Database
- MongoDB Atlas released in 2016 → documentdb as a service

- 

- MongoDB Documents
  - No predefined schema for documents is needed
  - Every document in a collection could have different data/schema
  - 

- Relational vs Mongo/Document DB
RDBMS	MongoDB
Database	Database
Table/View	Collection
Row	Document
Column	Field
Index	Index
Join	Embedded Document
Foreign Key	Reference

- MongoDB Features
  - Rich Query Support - robust support for all CRUD ops

    - Indexing - supports primary and secondary indices on document fields
    - Replication - supports replica sets with automatic failover
    - Load balancing built in
- Versions
    - MongoDB Atlas
        - Fully managed MongoDB service in the cloud (DBaaS)
    - MongoDB Enterprise
        - Subscription-based, self-managed version of MongoDB
    - MongoDB Community
        - source-available, free-to-use, self-managed
- Interacting with MongoDB
    - mongosh → MongoDB Shell
        - CLI tool for interacting with a MongoDB instance
    - MongoDB Compass
        - free, open-source GUI to work with a MongoDB database
    - DataGrip and other 3rd Party Tools
    - Every major language has a library to interface with MongoDB
        - PyMongo (Python), Mongoose (JavaScript/node), …
- Commands
    - collection.find ({}, {}) → Find is like SELECT, the spaces are (filters, projections)
    - Use mflix (db)
      db.users.find() → SELECT * FROM users;
    - db.users.find({"name": "Davos Seaworth"}) → SELECT * FROM users WHERE name = “Davos Seaworth”;
    - db.movies.find({rated: {$in:["PG", "PG-13"]}}) → SELECT * FROM movies WHERE rated in ("PG", "PG-13")
    - db.movies.find({"countries": "Mexico", "imdb.rating": {$gte: 7}}) → Return movies which were released in Mexico and have an IMDB rating of at least 7
    - db.movies.find({
      “year”: 2010,
      $or: [
      { "awards.wins": { $gte: 5 } },
      { “genres”: "Drama" }
      ] }) → Return movies from the movies collection which were released in 2010 and either won at least 5 awards or have a genre of Drama
    - db.movies.countDocuments({
      “year”: 2010,
      $or: [
      { "awards.wins": { $gte: 5 } },
      { “genres”: "Drama" }
      ] }) → How many movies from the movies collection were released in 2010 and either won at least 5 awards or have a genre of Drama
    - db.movies.countDocuments({
      “year”: 2010,

$or: [
{ "awards.wins": { $gte: 5 } },
{ “genres”: "Drama" }
]
}, {“name”: 1, “_id”: 0}) → Return the names of all movies from the movies collection that were released in 2010 and either won at least 5 awards or have a genre of Drama

- Comparison Operators
  - $eq → Matches values that are equal to a specified value.
  - $gt → Matches values that are greater than a specified value.
  - $gte → Matches values that are greater than or equal to a specified value.
  - $in → Matches any of the values specified in an array.
  - $lt → Matches values that are less than a specified value.
    $lte → Matches values that are less than or equal to a specified value.
  - $ne → Matches all values that are not equal to a specified value.
  - $nin → Matches none of the values specified in an array.

- Lauren Examples
  - 1. Create (Insert Operations)
  - MongoDB allows inserting documents into a collection using insertOne() and insertMany().
  - Example:

```
// Insert a single document
db.users.insertOne({ name: "Alice", age: 28, city: "New York" });

// Insert multiple documents
db.users.insertMany([
  { name: "Bob", age: 32, city: "Los Angeles" },
  { name: "Charlie", age: 25, city: "Chicago" }
]);
```

  - 2. Read (Query Operations)
  - To retrieve data from MongoDB, use find() and findOne().
  - Example:

```
// Find one document
db.users.findOne({ name: "Alice" });

// Find multiple documents with a filter
db.users.find({ age: { $gt: 25 } });

// Find all documents and format output
db.users.find().pretty();
```

  - Additional Example:

// Find theaters in Massachusetts
db.theaters.find({"location.address.state": "MA"}, { "_id": 0, "location.address.street1": 1, "location.address.city": 1, "location.address.zipcode":1 });

- 3. Update (Modify Documents)
- Documents can be updated using updateOne(), updateMany(), and replaceOne().

Example:

// Update a single document
db.users.updateOne({ name: "Alice" }, { $set: { age: 29 } });

// Update multiple documents
db.users.updateMany({ city: "Chicago" }, { $set: { state: "IL" } });

// Replace an entire document
db.users.replaceOne({ name: "Charlie" }, { name: "Charlie Brown", age: 26, city: "Chicago" });

- 4. Delete (Remove Documents)
- Documents can be deleted using deleteOne() and deleteMany().
- Example:

// Delete a single document
db.users.deleteOne({ name: "Alice" });

// Delete multiple documents
db.users.deleteMany({ age: { $lt: 30 } });

- 5. Additional Query Operators
- $gt (greater than), $lt (less than), $gte (greater than or equal to), $lte (less than or equal to).
- $in (matches any value in a list), $ne (not equal).
- $or, $and, $not for logical operations.
- Example:

// Find users older than 30 or living in Chicago
db.users.find({ $or: [{ age: { $gt: 30 } }, { city: "Chicago" }] });

- 6. Counting Documents
- To count documents matching a query, use countDocuments().
- Example:

// Count users older than 30
db.users.countDocuments({ age: { $gt: 30 } });

// Count movies in the Comedy genre
db.movies.countDocuments({"genres": {"$in": ["Comedy"]}});

- 7. Projection (Selecting Specific Fields)
- To return specific fields instead of entire documents, use projection.
- Example:

```
// Find all users but only return names
 db.users.find({}, { name: 1, _id: 0 });

// Find movie titles with Rotten Tomatoes rating above 3
 db.movies.find({"tomatoes.viewer.rating": {"$gt": 3}}, {"_id": 0, "title": 1, "viewer_rating": "$tomatoes.viewer.rating"});
```

- 8. Sorting and Limiting Results
- MongoDB allows sorting and limiting query results using sort() and limit().
- Example:

```
// Get top 5 oldest users
 db.users.find().sort({ age: -1 }).limit(5);

// Get the movie with the longest runtime
 db.movies.find({}, {"_id": 0, "title": 1, "genres": 1}).sort({"runtime": -1}).limit(1);
```

- 9. Indexing for Performance Optimization
- Indexes improve query performance. Use createIndex() to add indexes.
- Example:

```
// Create an index on the 'name' field
 db.users.createIndex({ name: 1 });
```

- 10. Aggregation Pipeline
- MongoDB supports aggregation for complex data processing.
- Example:

```
// Group users by city and count them
 db.users.aggregate([
    { $group: { _id: "$city", count: { $sum: 1 } } }
 ]);

// Count theaters per state
 db.theaters.aggregate([
   { "$group": { "_id": "$location.address.state", "count": { "$sum": 1 } } },
   { "$project": { "state": "$_id", "count": 1, "_id": 0 } },
   { "$sort": { "state": 1 } }
]);

// Count movies per year mentioning 'police' in the plot
 db.movies.aggregate([
   { "$match": { "plot": { "$regex": "police", "$options": "i" } } },
```

```
    { "$group": { "_id": "$year", "count": { "$sum": 1 } } },
    { "$project": { "year": "$_id", "_id": 0, "count": 1 } },
    { "$sort": { "year": 1 } }
]);
```

Module 8 - MongoDB and PyMongo

Pymongo

- PyMongo is a Python library for interfacing with MongoDB instances
- Code
    - Setup
        - from pymongo import MongoClient
        - client = MongoClient('mongodb://user_name:pw@localhost:27017')
        - db = client['ds4300']
        - collection = db['myCollection']
    - Inserting a single document
        - post = {
            "author": "Mark",
            "text": "MongoDB is Cool!",
            "tags": ["mongodb", "python"]
          }
        - post_id = collection.insert_one(post).inserted_id
        - print(post_id)
    - Count Documents
        - demodb.collection.count_documents({})

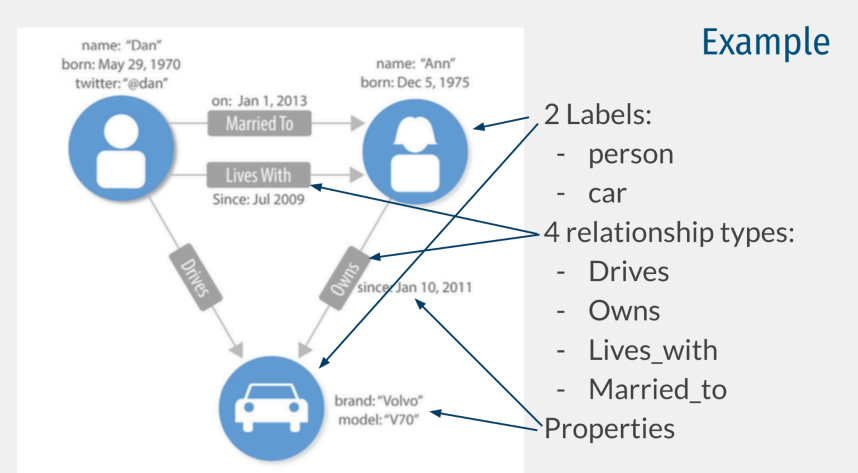
Module 9 - Graph Data Model

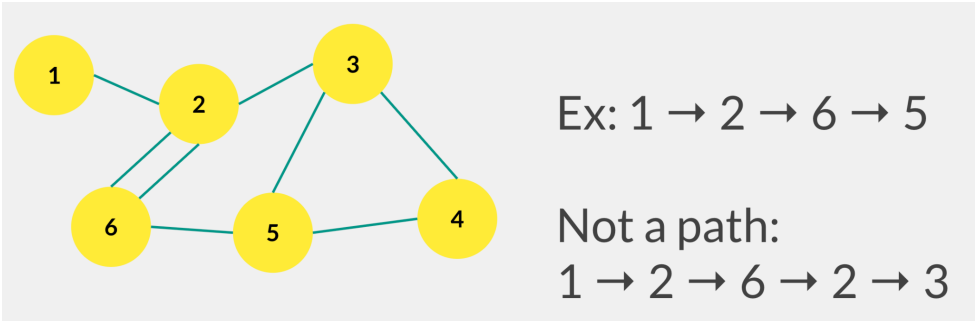
About Graph Database

- Data model based on the graph data structure
- Composed of nodes and edges
    - edges connect nodes
    - each is uniquely identified
    - each node/edge 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
        - yes… things like Instagram,
        - but also… modeling social interactions in fields like psychology and sociology
    - The Web
        - it is just a big graph of "pages" (nodes) connected by hyperlinks (edges)
    - Chemical and biological data

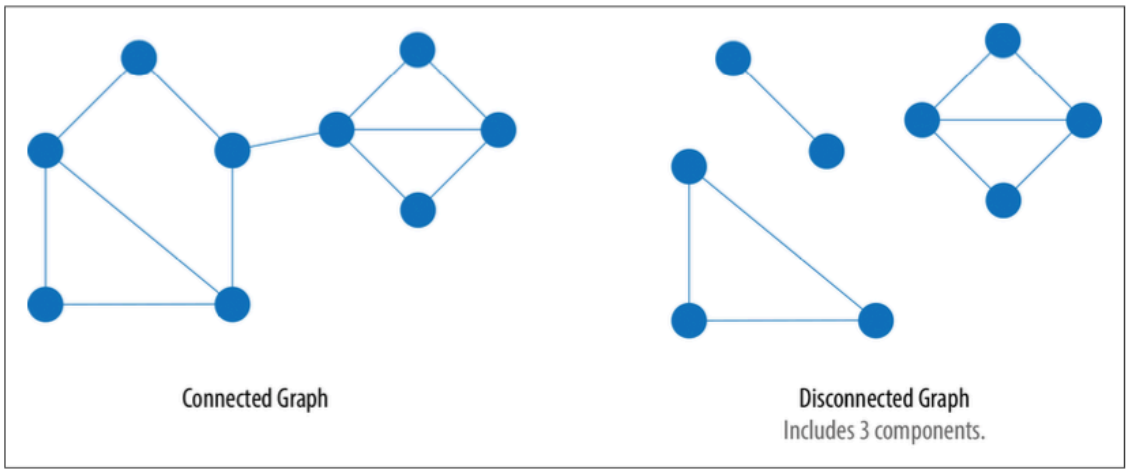
- systems biology, genetics, etc.
- interaction relationships in chemistry

Basics of graphs and graph theory

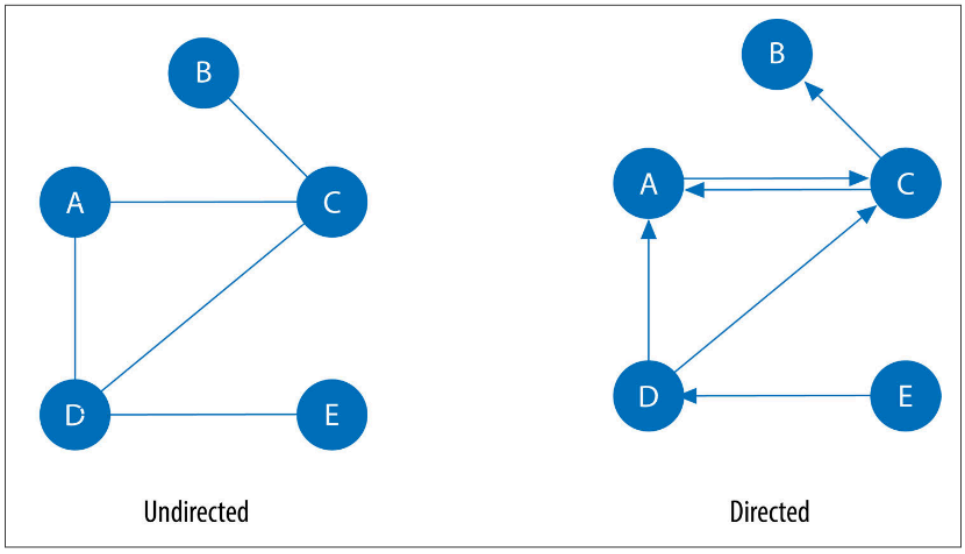
- What is a Graph?
  - Labeled Property Graph
    - Composed of a set of node (vertex) objects and relationship (edge) objects
    - Labels are used to mark a node as part of a group
    - Properties are attributes (think KV pairs) and can exist on nodes and relationships
    - Nodes with no associated relationships are OK. Edges not connected to nodes are not permitted.
    - 

- 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

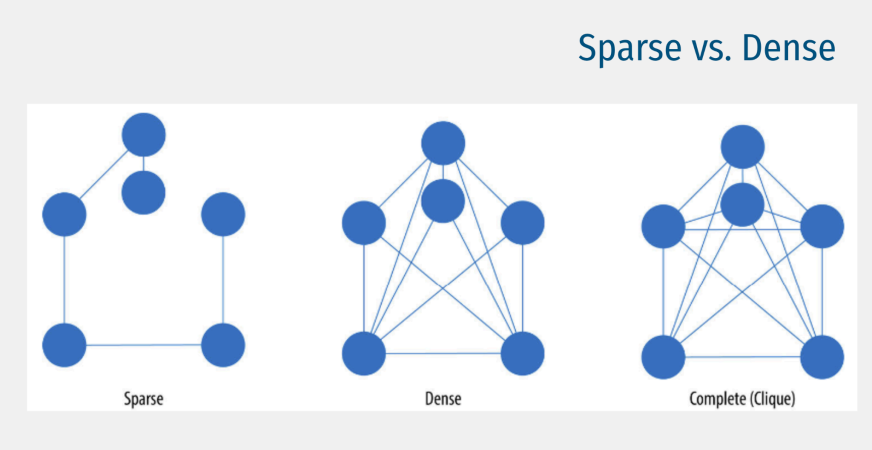
- 

- Weighted (vs. Unweighted) – edge has a weight property (important for some algorithms)
- 

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

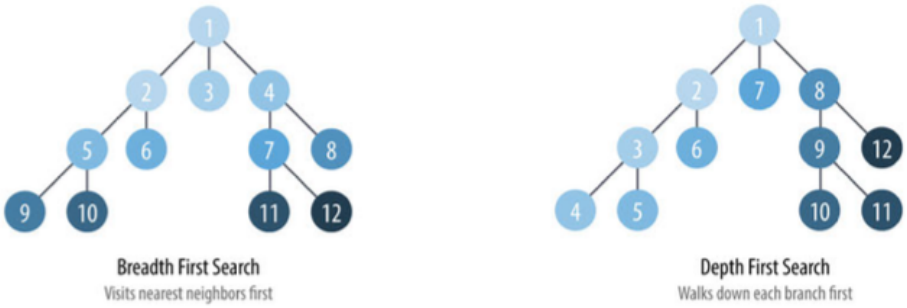
- Acyclic (vs. Cyclic) – Graph contains no cycles
- Sparse vs dense

-



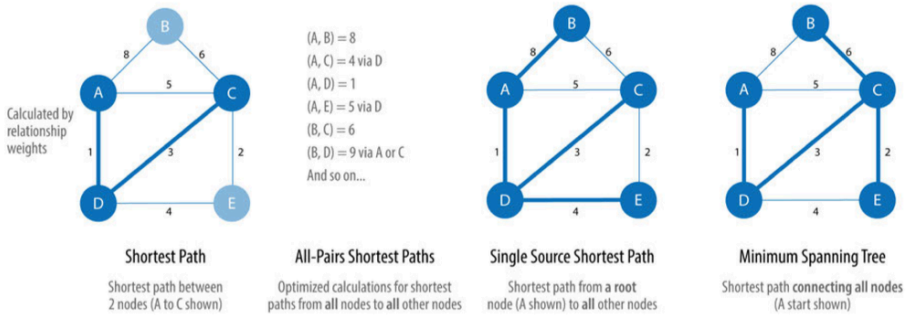
- Types of Graph Algorithms - Pathfinding
  - Pathfinding
    - finding the shortest path between two nodes, if one exists, is probably the most common operation
    - “shortest” means fewest edges or lowest weight
    - Average Shortest Path can be used to monitor efficiency and resiliency of networks.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

-



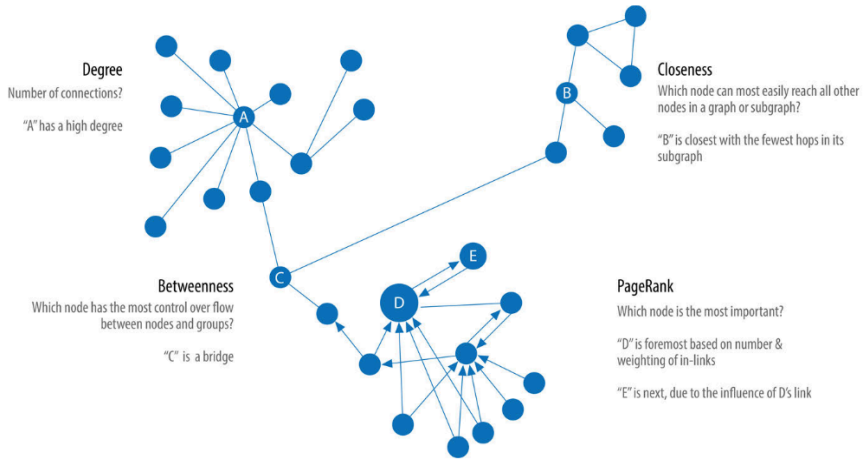
-
  - Shortest Path

-



- Types of Graph Algorithms - Centrality & Community Detection
  - Centrality

- determining which nodes are “more important” in a network compared to other nodes



-
  - EX: Social Network Influencers?
- Community Detection
  - evaluate clustering or partitioning of nodes of a graph and tendency to strengthen 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

Module 10 - Docker Compose and Neo4j

Neo4j

- About
  - 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
- Query Language and Plugins
  - Cypher
    - Neo4j’s graph query language created in 2011
    - Goal: SQL-equivalent language for graph databases
    - Provides a visual way of matching patterns and relationships

(nodes)-[:CONNECT_TO]->(otherNodes)

- APOC Plugin
  - Awesome Procedures on Cypher
  - Add-on library that provides hundreds of procedures and functions
- Graph Data Science Plugin
  - provides efficient implementations of common graph algorithms (like the ones we talked about yesterday)

Docker for Neo4j

- Allows you to run Neo4j easily and consistently across any computer or server
- Supports multi-container management
- Process
  - Create docker compose yaml

```
services:
  neo4j:
    container_name: neo4j
    image: neo4j:latest
    ports:
      - 7474:7474
      - 7687:7687
    environment:
      - NEO4J_AUTH=neo4j/${NEO4J_PASSWORD}
      - NEO4J_apoc_export_file_enabled=true
      - NEO4J_apoc_import_file_enabled=true
      - NEO4J_apoc_import_file_use__neo4j__config=true
      - NEO4J_PLUGINS=["apoc", "graph-data-science"]
    volumes:
      - ./neo4j_db/data:/data
      - ./neo4j_db/logs:/logs
      - ./neo4j_db/import:/var/lib/neo4j/import
      - ./neo4j_db/plugins:/plugins
```

  - 1 command can be used to start, stop, or scale a number of services at one time.
  - Provides a consistent method for producing an identical environment (no more “well… it works on my machine!)
  - Interaction is mostly via command line
  - .env files
    - .env files - stores a collection of environment variables
    - good way to keep environment variables for different platforms separate
      - .env.local
      - .env.dev
      - .env.prod
  - Docker Compose commands
    - Major Docker Commands
    - docker compose up
    - docker compose up -d

    - docker compose down
    - docker compose start
    - docker compose stop
    - docker compose build
    - docker compose build --no-cache
- Inserting Data by Creating Nodes
    - CREATE (:User {name: "Alice", birthPlace: "Paris"})
    - CREATE (:User {name: "Bob", birthPlace: "London"})
    - CREATE (:User {name: "Carol", birthPlace: "London"})
    - CREATE (:User {name: "Dave", birthPlace: "London"})
    - CREATE (:User {name: "Eve", birthPlace: "Rome"})
- Adding an edge with no variable names
    - MATCH (alice:User {name:”Alice”})
    - MATCH (bob:User {name: “Bob”})
    - CREATE (alice)-[:KNOWS {since: “2022-12-01”}]->(bob)
- Matching
    - MATCH (usr:User {birthPlace: “London”})
    - RETURN usr.name, usr.birthPlace
- Importing data
    - LOAD CSV WITH HEADERS
    - FROM 'file:///netflix_titles.csv' AS line
    - CREATE(:Movie {
        id: line.show_id,
        title: line.title,
        releaseYear: line.release_year
      })
- Loading CSVs
    - LOAD CSV
    - [WITH HEADERS]
    - FROM 'file:///file_in_import_folder.csv'
    - AS line
    - [FIELDTERMINATOR ',']
    - // do stuffs with 'line'
- Adding Edges
    - LOAD CSV WITH HEADERS
    - FROM 'file:///netflix_titles.csv' AS line
    - MATCH (m:Movie {id: line.show_id})
    - WITH m, split(line.director, ",") as directors_list
    - UNWIND directors_list AS director_name
    - MATCH (p:Person {name: director_name})
    - CREATE (p)-[:DIRECTED]->(m)