DS4300 Large Scale Data Notes

1/6/25

- Using Docker (UPDATE IT!!), Python environment, Datagrip, VSCode, git/github
- Python 3.10 or higher
- All you need for github is to be able to push/pull, don't need to use terminal you can use VSCode GUI

Review:

- shell/command prompt/PowerShell CLI
- Docker and Docker compose
- Python crash course on slideshow, also some video courses on OReilly playlist, under Python section

*only use AI to help with teaching concepts/trying to find something out. Don't use it to do your work for you

- NO LATE HW ALLOWED, same rules as CS3200

1/8/25

Searching:

- Most common operation in database systems
- SELECT statement in SQL is arguably most versatile/common
- Baseline for efficiency is linear search
    - Start at 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
- 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
- Search key: a value for an attribute from the entity type

List of Records:

- 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
    - Python does not have a contiguously allocated list, or an array, built in. Numpy arrays are contiguously allocated lists.
- Linked List:
    - each record needs x bytes + additional space for 1 or 2 memory addresses (for linking)
    - Individual records are linked together in a type of chain using memory addresses

Pros and Cons of different lists:

- 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

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
- For example, you start with the midpoint of the array, and see if the target is more or less than the midpoint. If more (example), then you remove the values less than the midpoint, and then get the midpoint of the new filtered array, and repeat.
- log2(n)+1 maximum comparisons (because you're taking ½, then ½ of the ½, then ½ of the ½ of the ½

Time Complexity:
- Linear Search:
    - Best case: target is found at first element, O(1)
    - Worst case: not in the array, O(n)
- Binary Search:
    - Best case: target is found at *mid*; 1 comparison O(1)
    - Worst case: log2(n), or O(log2(n)) complexity

Back to Database Searching:
- Assume data stored on disk by column id's value
- Searching for a specific id is fast. But what if we want to search for a specific *specialVal*?
    - Only option is a linear scan of that column. Can't use binary search because values aren't in order.
- Can't store data on disk sorted by both id and specialVal (at the same time). Data would have to be duplicated which is space inefficient

**we need an external data structure to support faster searching by *specialVal* than a linear scan

What is this?
It could be…
- 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
    - But, every insert into the table is like inserting into a sorted array. It's slow.
- A linked list of tuples (specialVal, rowNumber) sorted by specialVal
    - Searching for specialVal is slow, but inserting is fast

A Binary Search tree is both fast to search through and fast to insert into

- 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.

Tree Traversals:

- Pre-order
- Post-order
- In order
- Level order ← important for HW

1/27/25

<u>Moving Beyond the Relational Model</u>

Benefits of a Relational Model

- Mostly standard data model and query language
- ACID compliance
    - Atomicity, consistency, isolation, durability
    - A transaction is a unit of work for a database
- Works well with highly structured data
- Can handle large amounts of data
- Well understood, lots of tooling, lots of experience

RDBMS increases efficiency:

- Indexing, directly controlling storage, column oriented storage vs row oriented storage, query optimization, caching/prefetching, materialized views, precompiled stored procedures, data replication and partition

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)
- Helps ensure
    - Data integrity, error recovery, concurrency control, reliable data storage, simplified error handling

ACID Properties

- 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 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:
        - 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
    - Locking helps to promote isolationism within a relational database
- 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

1/29/25

Relational databases may not be the solution to all problems…

- Sometimes schemas 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

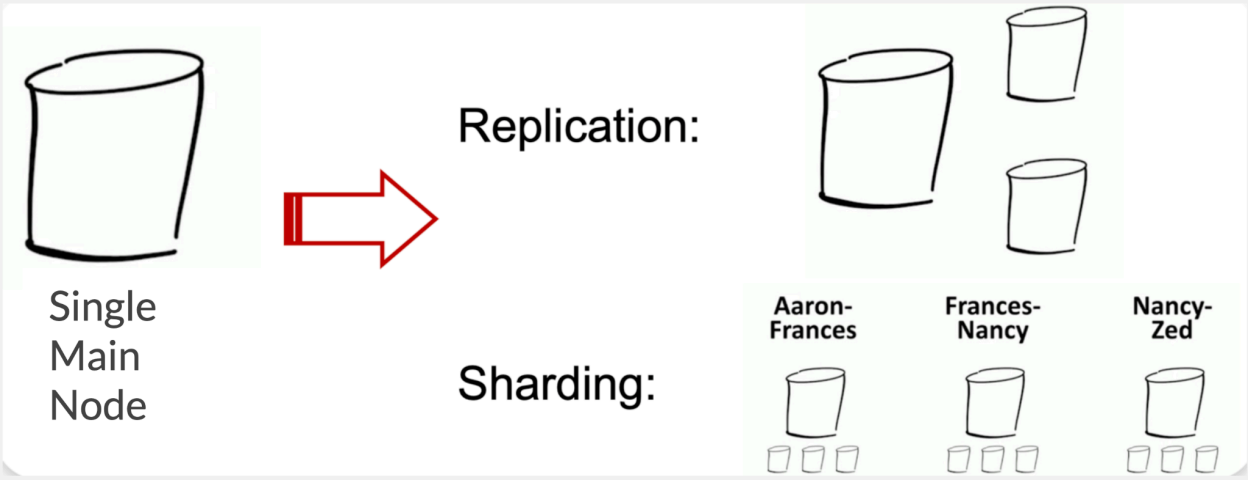
However: there are modern systems that make horizontal scaling less problematic

Essentially: scaling up = more power on one computer, scaling out = more computers

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 Systems:
- Computers operate concurrently
- Computers fail independently
- No shared global clock

Distributed storage: 2 directions
- Replication: creating more instances of a single main node
- Sharding: segmenting the single nodes for multiple categories (like A-G, G-M, M-Z in the alphabet), and each of those nodes having like child nodes



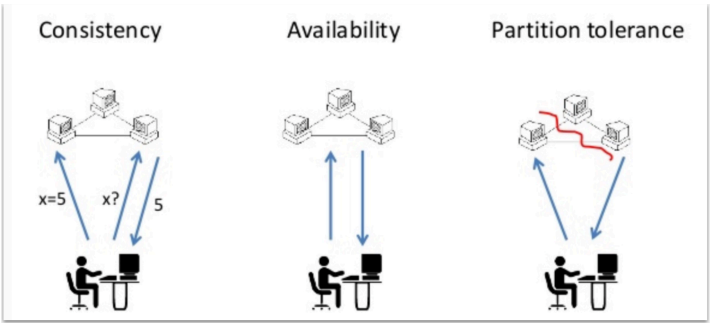
- Distribution is very helpful for geographic distribution. For example, social media is international, but someone in Italy will get their social media from a different computer than someone in the US
- CVN: content delivery networks. Like CloudFlare. Gets data to where your users are

Distributed Data Stores:
- Data is stored on > 1 node, typically replicated
    - 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!

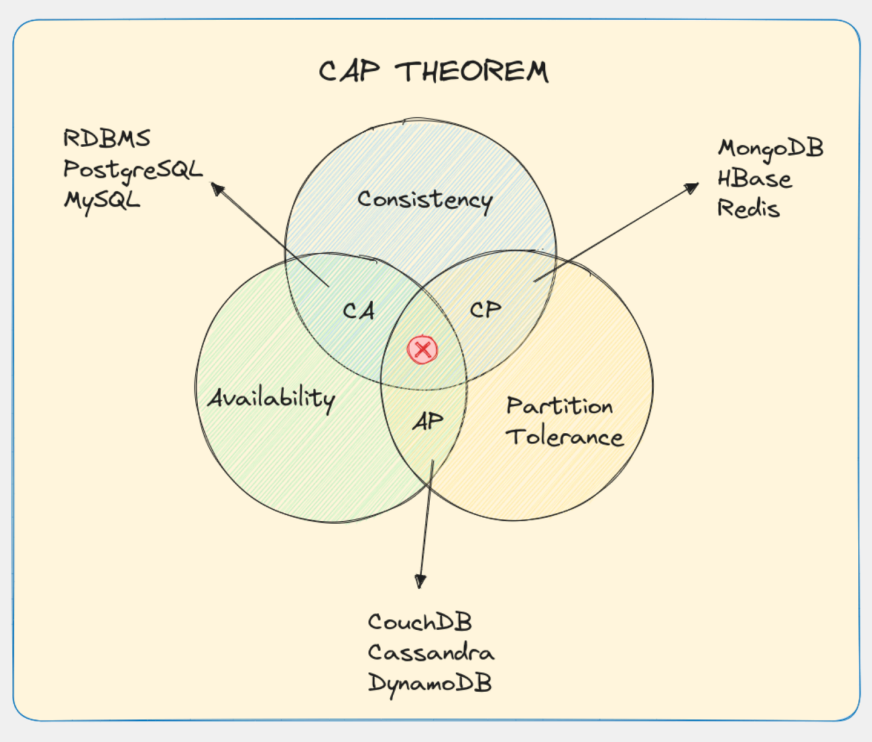
- Network failures, system failures
- Overall system needs to be partition tolerant
    - System can keep running even with network partition

CAP Theorem

- Consistency, availability, partition tolerance



- The CAP Theorem states it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:
    - Consistency: every read receives the most recent write or error thrown
        - (like if you post and someone on instagram in a different country sees your post instantly. Very low chance this happens because instagram is inconsistent)
    - Availability: every request receives a (non-error) response - but no guarantee that the response contains the most recent write
        - Like when social media goes down
    - Partition tolerance: the system can continue to operate despite arbitrary network issues



- Consistency + Availability: always responds with the latest data and every request gets a response, but may not be able to deal with network issues
- Consistency + Partition Tolerance: if system responds with data from a distributed store, it is always the latest, else data request is dropped
- Availability + Partition Tolerance: system always sends response based on distributed store, but may not be the absolute latest data

2/3/25
Distributed DBs and ACID - pessimistic concurrency
ACID transactions:

- Focuses on data safety
- Considered a pessimistic concurrency model because it assumes one transaction has to protect itself from other transactions
- 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 the 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 be ok
- Adds last update timestamp and version number 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

Heavy conflict systems

- Rolling back and rerunning transactions that encounter conflict → less efficient
- So, a locking scheme (pessimistic model) might be preferable

NoSQL:

- When it first started, it was a relational database that did not use SQL
- Modern meaning is "Not Only SQL", but sometimes thought of as non-relational DBs
- Idea originally developed in part as a response to processing unstructured web-based data

CAP Theorem Review:

- Consistency, Availability, Partition Tolerance
- You can have 2 of those 3, but not all 3
- Consistency: every user of the DB has an identical view of the data at any given moment
- Availability: in the event of a failure, the DB system remains operational
- Partition Tolerance: the database can maintain operations in the event of the network's failing between two segments of the distributed system

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 system responds with data from the system, it is always the latest, otherwise the request is dropped
Availability/Partition Tolerance: system always sends a response based on the distributed store, but may not be the absolute latest data.

ACID Alternative for Distributed Systems - BASE:

- 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 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

Categories of NoSQL Databases:
- Document databases (JSON)
- Graph databases (graph data, things with lots of relationships between units of data)
- Key-value databases
- Columnar databases
- Vector databases (LLMs)

Key-Value databases:
- Key-value stores are designed around:
    - *Simplicity*:
        - The data model is extremely simple
        - Comparatively, tables in an 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 an O(1) operation because 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

DS Use Cases:
- EDA/Experimentation Results Store
    - Store intermediate results from data preprocessing and EDA
    - Store experiment or testing (A/B) results without production 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

SWE Use Cases:

- 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 and 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 user
    - Needs to be available across browsers, machines, sessions
- Caching Layer
    - In front of a disk-based database

Redis DB
Remote Directory Server

- 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
- Considered an in-memory database system, but supports the durability of data
- Does not handle complex data. No secondary indexes, only supports lookup by key

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

2/10/25
Document Databases and MongoDB

- A document database is a non-relational database that stores data as structured documents, usually as JSONs
- JSON: JavaScript Object Notation
    - A lightweight data interchange format
    - Easy for humans to read and write
    - Easy for machines to parse and generate
- JSON is built on two structures
    - A collection of name/value pairs (operationalized as an object, record, struct, dictionary, hash table, keyed list, associative array)
    - An ordered list of values. (operationalized as an array, vector, list or sequence)
- Two universal data structures supported by virtually all modern programming languages
    - Thus, JSON makes a great data interchange format

BSON → binary JSON

- Binary encoded serialization of a JSON-like document structure
- Supports extended types not part of basic JSON (ex: 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 (you can define tags)

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

MongoDB

- No predefined schema for documents is needed
    - Each document has a schema, but MongoDB does not assign a particular file to a collection based on its schema
- Every document in a collection could have different data/schema

RDBMS to Mongo
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 operations
- Indexing: supports primary and secondary indices on document fields
- Replication: supports replica sets with automatic failover
- Load balancing built in

MongoDB Atlas, MongoDB Enterprise, MongoDB Community

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
- PyMongo (Python)
- Mongoose (JS/node)

2/19/25
Practical B
- Dealing with using redis as a vector database for building a retrieval augmented generation system for a self-hosted LLM. all the pieces will be in the system (so no need to make an LLM model from scratch), it's more about getting the data and parsing it through the system.
- Most likely due the end of the week after spring break

Introduction to the Graph Data Model
Neo4j has a ton of built in graph algorithms
- 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*

Examples:

- Social networks: things like Instagram but also modeling social interactions in fields like psychology
- The web: big graph of pages (nodes) connected by hyperlinks (edges)
- Chemical and biological data: systems biology, genetics, etc., interaction relationships in chemistry

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 <u>not</u> connected to node are <u>not permitted</u>

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

- Some debate over the path length measured by counting nodes or edges

Connected (or disconnected): there is a path between any two nodes in the graph

- If a graph is disconnected, then there are multiple components in the graph network. A connected network just has everything connected basically.

Weighted (or unweighted): edge has a weight property (important for some algorithms)
Directed (or undirected): relationships (edges) define a start and end node
Acyclic (or cyclic): graph contains no cycles

Sparse vs Dense

- Sparse graphs are better stored in adjacency list
- Dense graphs are better stored in an adjacency matrix
- Directed graph you need entire matrix
- Undirected graph you only need half the matrix along the diagonal
- If every node is connected to every other node, it is a complete graph (<u>clique</u>)

Trees

- <u>Rooted tree</u>: root node and no cycles

- Binary tree: up to two child nodes and no cycles
    - Ternary tree is up to three child nodes
- Spanning tree: subgraph of all nodes but not all relationships and no cycles. Minimum number of edges to get from one node to another node in the graph

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

- Breadth first search: visits nearest neighbors first
- Depth first search: walks down each branch first

Shortest path

- Shortest path: Calculated by relationship weights
- 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 to all other nodes
- Minimum spanning tree: shortest path connecting all nodes

Centrality:

- Determining which nodes are "more important" in a network compared to other nodes
- How connected a specific node is to other nodes in a graph network
- Ex: social network influencers?

Community detection:

- Evaluate clustering or partitioning of nodes of a graph and tendency to strengthen or break

Dijkstra's Algorithm:

- Single source shortest path algorithm 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

2/24/25

- LLMs take in training data like class notes. These class notes have general embeddings, which contain chunks that contain file names, pdf pages, vectors, etc.
- This flows into a Redis Stack
- A user might come along and say “what is SQL?”. The LLM would generate an embedding for the question, which gives us a vector based on the question. The embedding for that question then is sent to Redis, and we ask it to give us back the 3, 5, 10 chunks that are most similar to the vectorized version of our question. This is through a similarity search (think cosine similarity). Redis Stack takes care of doing this for us.
- We then have the 10 most similar chunks from our documents. When we ask the question to the LLM, we send back in the 10 most similar chunks as context. In the program we construct a prompt to send to the model. Ex: “you are a competent human. Use the context to answer the question. If the context doesn’t include an answer to the question, say ‘i dont know’”. You literally just construct a really long python string that captures what the LLM is supposed to say.
- The LLM will take the context and use it to construct the language version of the output. You get an answer that is contextual to the class notes
- This whole system is called Retrieval Augmented Generation (RAG)
    - We are still using an LLM but we are retrieving a set of contextual elements from our class notes, and augmenting what’s in the model with that context. The model is already pre-set, but we are augmenting it.