Will need to have set up:

- Docker desktop
- Anaconda or miniconda python
- Database access tool: datagrip or DBeaver
- Vs code for python development (3.10 or higher)
- Github
- AWS account (amazon web services)

Review topics:

- Terminal commands
- Docker and docker compose
    - Basics of docker filers and docker-compose.yaml files
    - Setting up volumes and mapping between host and guest OS

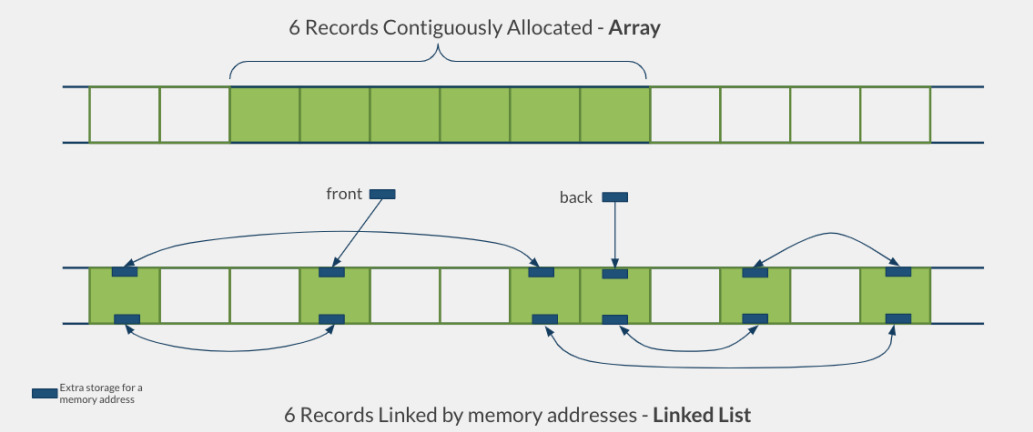
1/8/25

Foundations

- Searching is the most common operation performed by a database system
- SELECT is the most versatile/complex
- Baseline for efficiency is **Linear Search**
    - Starts 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
- Record
    - 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
        - Trivally, store in some sequential order like a list
- Search Key - A value for an attribute from the entity type
    - Could be >= 1 attribute

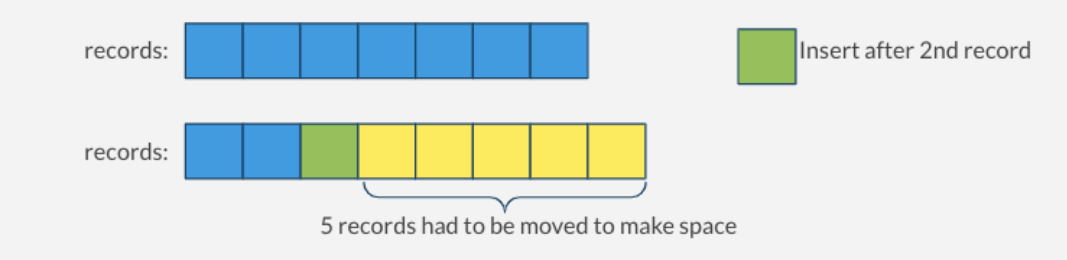
Lists of records

- If each record takes up x bytes of memory, then for n record, we n eed n*x bytes of memory
- Contiguously 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
    - Individual record are linked together in a type of chain using memory addresses
- Contiguous vs linked
-

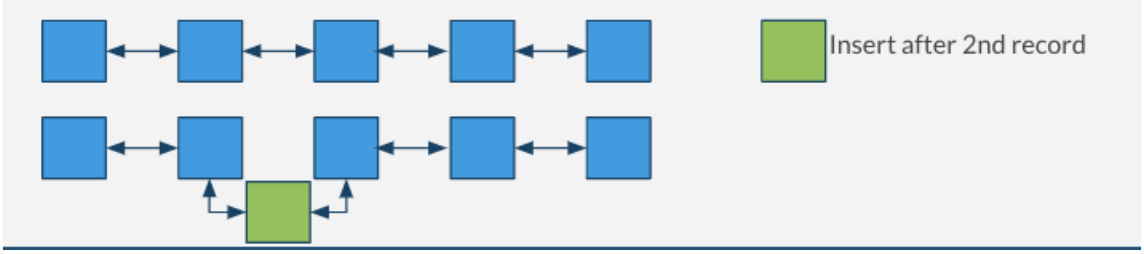


Pros and Cons

- 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
-



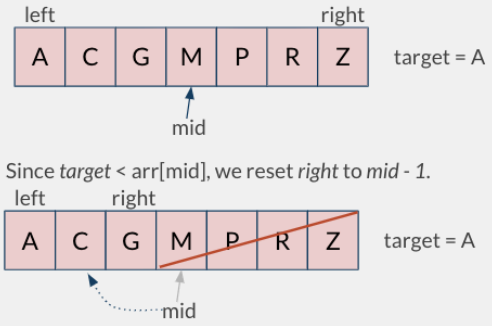
Observations

- 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
-

```
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
```



Time Complexity

- Linear Search
    - Best Case:
        - Target is found at the first element; only one comparison
    - Worst case:
        - Target is not in the array; n comparison
        - Therefore, in the worst case, linear search is O(n) time complexity.
- Binary Search
    - Cant perform on an unsorted array
    - 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.

Database Searching

- Assume data is stored on disk by column id's value
- Searching for a specific id = fast
- What do 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
    -

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

- We need on external data structure to support faster searching by specialVal than a linear scan

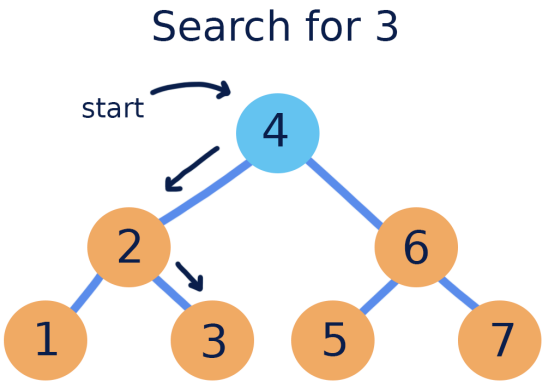
What do we have in our arsenal?

1) An array of tuples specialVal, rowNumber) sorted by specialVal
    a) We could use Binary Search to quickly locate a particular specialVal and find its corresponding row in the table
    b) But, every insert into the table would be like inserting into a sorted array - slow…
2) A linked list of tuples (specialVal, rowNumber) sorted by specialVal
    a) searching for a specialVal would be slow - linear scan required

b) But inserting into the table would theoretically be quick to also add to the list

Something with fast insert and fast search

- Binary search tree
    - A binary tree where every node in the tree left subtree is less than its parent and every node in the right subtree is greater than its parent
    - Needs to be sorted



-

1/13/25

[AVL tree rotation notes]

AVL Trees Notes

AVL trees are self-balancing binary search trees where the height difference between left and right subtrees (the balance factor) of any node is at most 1. This balancing property ensures operations like search, insert, and delete maintain O(log n) time complexity.

Core Concepts

1. Balance Factor
- **Definition**: height(right subtree) - height(left subtree)
- **Valid values**: -1, 0, or 1 for an AVL tree
- When the balance factor becomes -2 or 2, rebalancing is required

2. Tree Operations
- **Search**: Same as in a regular binary search tree
- **Insert**: Insert as in a binary search tree, then rebalance if needed
- **Delete**: Delete as in a binary search tree, then rebalance if needed

Insertion Process

1. **Standard BST Insertion**: Insert the new node like in a regular binary search tree
2. **Update Heights**: Recalculate heights of affected nodes along the path from the insertion point up to the root
3. **Check Balance Factors**: Identify nodes with balance factors outside the -1 to 1 range
4. **Rebalancing**: Apply appropriate rotation(s) to restore balance

Rotations

1. Left Rotation
Used when a node has a balance factor of +2 (right-heavy)

```
   A              B
  / \            / \
 X   B    ->    A   Z
    / \        / \
   Y   Z      X   Y
```

2. Right Rotation
Used when a node has a balance factor of -2 (left-heavy)

```
     A              B
```

```
   / \          / \
  B   Z   ->   X   A
 / \          / \
X   Y        Y   Z
```

3. Left-Right Rotation (Double Rotation)
When a node's left child is right-heavy

```
   A            A            Y
  / \          / \          / \
 B   Z   ->   Y   Z   ->   B   A
/ \          / \          /   \
X   Y        B   Q        X     Z
  / \          / \
 P   Q        X   P
```

4. Right-Left Rotation (Double Rotation)
When a node's right child is left-heavy

```
 A            A            Y
/ \          / \          / \
X   B   ->   X   Y   ->   A   B
  / \          / \        /   \
 Y   Z        P   B      X     Z
/ \            / \
P   Q          Q   Z
```

Insertion Rebalancing Cases

1. **Left-Left Case**: A node becomes unbalanced with balance factor -2, and its left child has balance factor -1 or 0
   - **Solution**: Single right rotation

2. **Right-Right Case**: A node becomes unbalanced with balance factor +2, and its right child has balance factor +1 or 0
   - **Solution**: Single left rotation

3. **Left-Right Case**: A node becomes unbalanced with balance factor -2, and its left child has balance factor +1
   - **Solution**: Left rotation on the left child, then right rotation on the unbalanced node

4. **Right-Left Case**: A node becomes unbalanced with balance factor +2, and its right child has balance factor -1
   - **Solution**: Right rotation on the right child, then left rotation on the unbalanced node

Step-by-Step Insertion Algorithm

1. Insert the new node as in a regular BST
2. Start from the newly inserted node and move up to the root:
   - Update the height of each node
   - Calculate the balance factor of each node
   - If the balance factor becomes -2 or +2, identify the case and apply appropriate rotation(s)
3. Continue this process all the way to the root

Example

Let's insert values [10, 20, 30, 40, 50] into an initially empty AVL tree:

1. Insert 10: Tree is balanced

```
10
```

2. Insert 20: Tree is still balanced

```
10
 \
  20
```

3. Insert 30: Tree becomes right-heavy, needs left rotation at root

```
10              20
 \             /  \
  20     ->   10   30
   \
    30
```

4. Insert 40: Tree becomes right-heavy at node 30

```
   20             20
  /  \           /  \
 10   30   ->   10   40
       \             /
        40          30
```

5. Insert 50: Tree becomes right-heavy at node 40, requires left rotation
   ```
     20                20
    / \               / \
   10  40            10  40
      / \     ->        / \
     30  50            30  50
   ```

Implementation Tips

1. Maintain a height field in each node
2. After inserting, trace back from the insertion point to the root
3. Recalculate heights and balance factors at each step
4. Apply rotations when needed
5. Continue this process up to the root

Pseudocode for AVL Tree Insertion

```
function insert(root, key):
    if root is null:
        return new Node(key)

    if key < root.key:
        root.left = insert(root.left, key)
    else if key > root.key:
        root.right = insert(root.right, key)
    else:
        return root  // Duplicate keys not allowed
```

```
    // Update height of this ancestor node
    root.height = 1 + max(height(root.left), height(root.right))

    // Get the balance factor
    balance = getBalance(root)

    // Left-Left Case
    if balance < -1 and key < root.left.key:
        return rightRotate(root)

    // Right-Right Case
    if balance > 1 and key > root.right.key:
        return leftRotate(root)

    // Left-Right Case
    if balance < -1 and key > root.left.key:
        root.left = leftRotate(root.left)
        return rightRotate(root)

    // Right-Left Case
    if balance > 1 and key < root.right.key:
        root.right = rightRotate(root.right)
        return leftRotate(root)

    return root
```

Rotation Functions

```
function rightRotate(y):
```

```
    x = y.left
    T2 = x.right

    // Perform rotation
    x.right = y
    y.left = T2

    // Update heights
    y.height = 1 + max(height(y.left), height(y.right))
    x.height = 1 + max(height(x.left), height(x.right))

    return x

function leftRotate(x):
    y = x.right
    T2 = y.left

    // Perform rotation
    y.left = x
    x.right = T2

    // Update heights
    x.height = 1 + max(height(x.left), height(x.right))
    y.height = 1 + max(height(y.left), height(y.right))

    return y
```

B+ Tree Notes

Key Concepts

- **B+ Tree**: A balanced tree data structure that maintains sorted data for efficient insertions, deletions, and searches
- **Order (m)**: Defines the maximum number of children a node can have (in these examples, m=4)
- **Node Types**:
  - **Leaf Nodes**: Store actual keys and data values
  - **Internal Nodes**: Store keys and pointers to child nodes
  - **Root Node**: The top-level node (can be either a leaf or internal node)

Important Properties

- Leaf nodes are connected in a doubly linked list for efficient range queries
- Internal nodes store keys as guides for traversal but don't store data
- Every node except root must be at least half full
- All leaf nodes appear at the same level (balanced tree)

Insertion Process

1. **Start at the root** and traverse down to the appropriate leaf node
2. **Insert the key** in sorted order within the leaf node
3. **If the node is full** (contains m-1 keys):
   - Split the node into two nodes
   - Redistribute keys between the original and new node
   - For leaf nodes: Copy the smallest key from new node up to parent
   - For internal nodes: Move (not copy) the middle element to parent
4. **If parent becomes full**, repeat the split process upward

Node Splitting Rules

- **For Leaf Nodes**:
  - Original node keeps ⌈m/2⌉ keys (ceiling of m/2)

  - New node gets remaining keys
  - Smallest key in new node is copied up to parent
  - New node is linked into the leaf node chain

- **For Internal Nodes**:
  - Middle key moves up to parent (not copied)
  - Keys less than middle stay in original node
  - Keys greater than middle go to new node

Step-by-Step Example (m=4)

Initial Insertions: 42, 21, 63, 89
- Start with a single leaf node that is also the root: [21, 42, 63, 89]

Insert 35
- Leaf node is full, needs to split
- Original node: [21, 35]
- New node: [42, 63, 89]
- Create new parent (root) node with key 42
- Tree structure becomes:
  ```
  [42]
  /  \
  [21,35]  [42,63,89]
  ```

Insert 10, 27, 96
- These insertions don't cause splits
- Tree becomes:
  ```
  [42]
  ```

/ \
[10,21,27,35] [42,63,89,96]
```

### Insert 30
- Left leaf node is full, needs to split
- Original node: [10, 21]
- New node: [27, 30, 35]
- Copy 27 up to parent
- Tree becomes:
```
[27,42]
/ | \
[10,21] [27,30,35] [42,63,89,96]
```

### Insert 37 (when root is full)
- Causes a leaf node to split, which pushes a new key to the root
- Root becomes full and must split
- Tree grows one level deeper
- Process:
  1. Split the appropriate leaf node
  2. When this causes the root to split:
     - Create new root
     - Move middle element from old root to new root
     - Redistribute remaining keys
  3. Tree becomes a 3-level tree

## Key Differences Between B-Tree and B+ Tree

- In B+ Tree, all data is stored in leaf nodes
```

- Internal nodes only store keys for navigation
- Leaf nodes are linked for sequential access
- Keys may appear in both internal nodes and leaf nodes

1/27/25

Moving beyond the relational model

Benefits of the relational model

- (mostly) standard data model and Query Language
- ACID compliance
    - Atomicity, consistency, isolation, durability
- Works well with highly structured data
- Can handle large amounts of data
- Well understood, lots of tooling, lots of experience

Relationa; database performance

- Many ways that a RDBMS increases efficiency:
    - Indexing (topic we focuses on)
    - Directly controlling storage
    - Column oriented storage vs row storage
    - Query optimization
    - caching/prefetching
    - Materialized views
    - Precompiled stored procedures
    - Data replication and partitioning

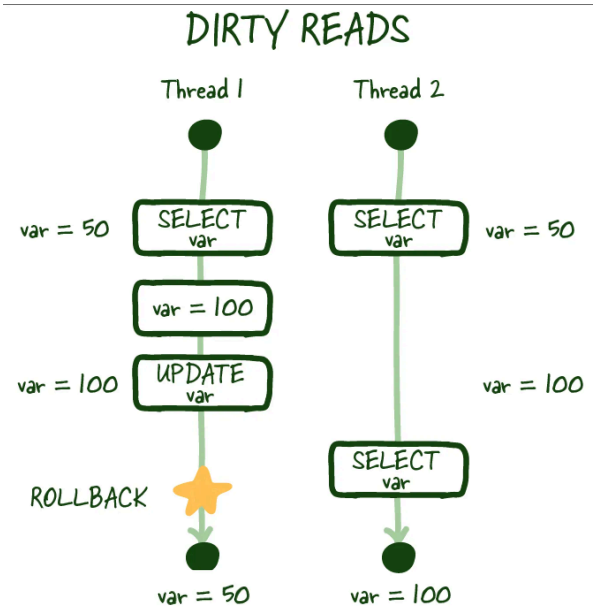
Transaction Processing

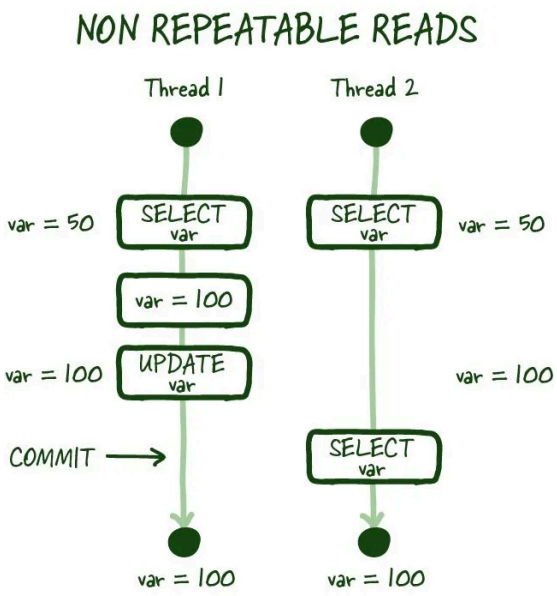
- Transaction
    - A sequence of one or more of the CRUD operations performed as a single, logical unit of work
        - Either the entire sequence succeeds (COMMIT)
        - OR the entire sequence fails (ROLLBACK or ABORT)
- Help ensure

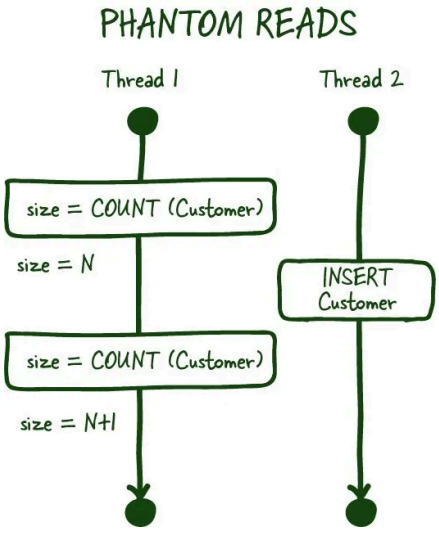
- Data integrity
- Error recovery
- Concurrency control
- Reliable data storage
- Simplified error handling

ACID Properties

- 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
    - 

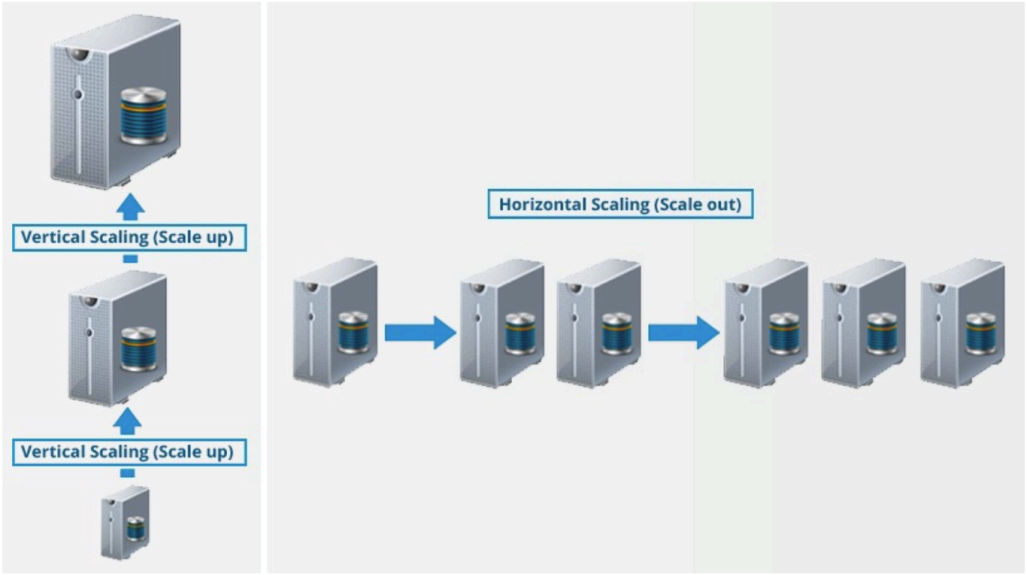
- Durability
    - Once a transaction is completed and committed successfully, its changes are permanent

    - Even in the event of a system failure, committed transactions are preserved

But…

- 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)

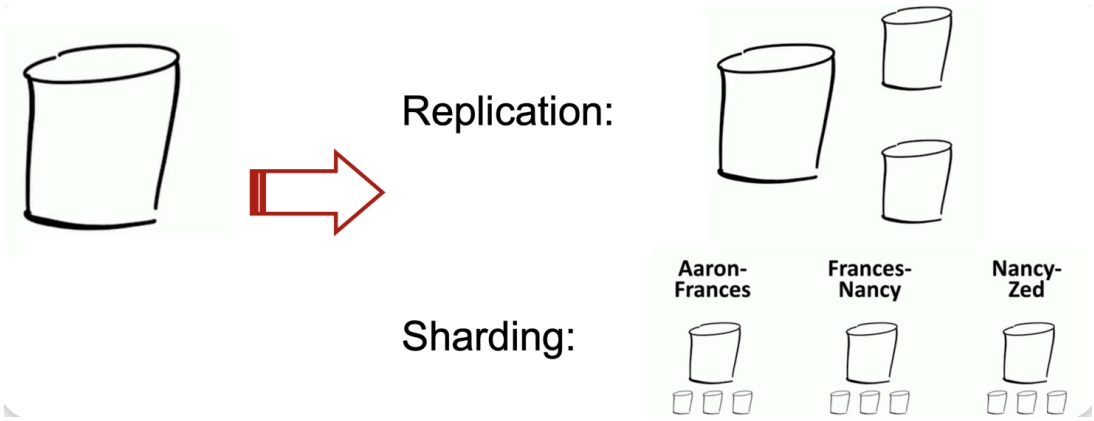
Scalability - Up or Out?

- 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.
- 

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

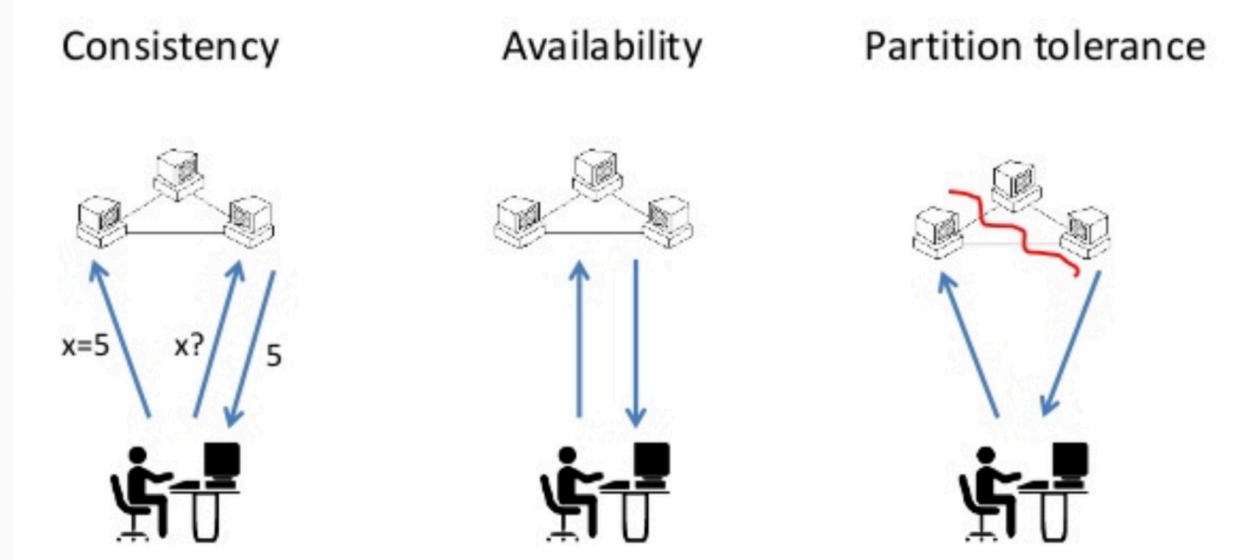
Distributed storage - 2 directions

- 

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 **Partition Tolerant**
        - System can keep running even w/ network partition

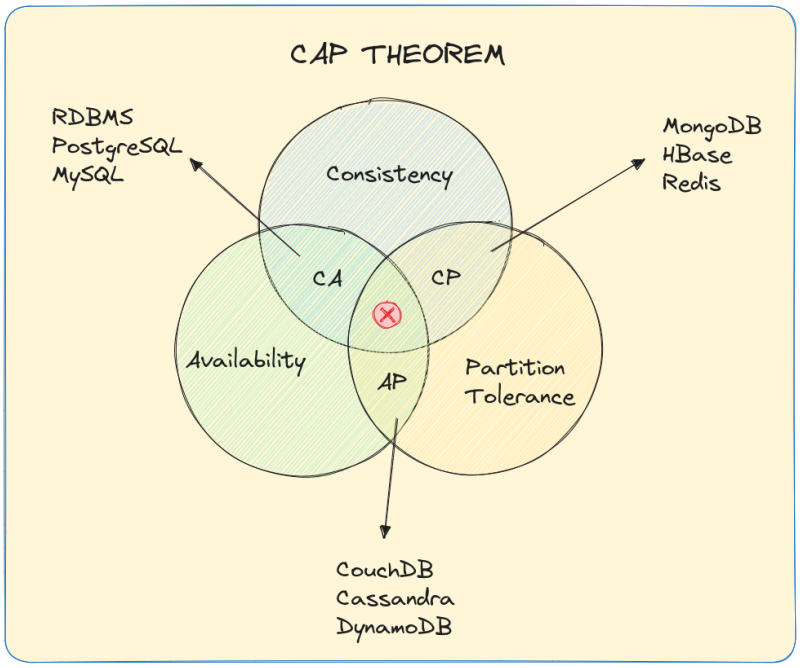
The CAP Theorm



- The CAP Theorem states that 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
    - **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.

CAP Theorom - database view

- Consistency*: Every user of the DB has an identical view of the data at any given instant
    - *note, the definition of consistency in cap is different from that of ACID
- Availability: In the event of a failure, the database 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 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 are responds based on distributed store, but may not be the absolute latest 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: consistency, availability, or tolerance to failure.

2/3/25

noSQL and KV dbs

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
        - IOW, 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.
- But how?
    - Add 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
- High Conflict Systems
    - rolling back and rerunning transactions that encounter a conflict → less efficient
    - So, a locking scheme (pessimistic model) might be preferable

NoSQL

- NoSQL” first used in 1998 by Carlo Strozzi to describe his relational database system that did not use SQL.
- More common, 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.

ACID Alternative for Distrib Systems - **BASE**

- **B**asically **A**vailable
    - 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
- **S**oft State - The state of the system could change over time, even w/o 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
- **E**ventual Consistency - The system will eventually become consistent
    - All writes will eventually stop so all nodes/replicas can be updated

Categories of noSQL DBs

- Document databases
- Graph databases
- Key value databases
- Column oriented

Key value databases

- Key-value stores are designed around:
    - Simplicity
        - the data model is extremely simple
        - comparatively, tables in a RDBMS are very complex.
        - lends itself to simple CRUD ops and API creation
- Key = value
    - 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.

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

- 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

- Redis (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
- From [db-engines.com Ranking of KV Stores]:

Rank			DBMS	Database Model	Score		
Oct 2024	Sep 2024	Oct 2023			Oct 2024	Sep 2024	Oct 2023
1.	1.	1.	Redis	Key-value, Multi-model	149.63	+0.20	-13.33
2.	2.	2.	Amazon DynamoDB	Multi-model	71.85	+1.78	-9.07
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	24.50	-0.47	-9.80
4.	4.	4.	Memcached	Key-value	17.79	+0.95	-3.05
5.	5.	5.	etcd	Key-value	7.17	+0.12	-1.57
6.	↑ 7.	↑ 8.	Aerospike	Multi-model	5.57	+0.41	-0.86
7.	↓ 6.	↓ 6.	Hazelcast	Key-value, Multi-model	5.57	-0.16	-2.60
8.	8.	↓ 7.	Ehcache	Key-value	4.76	-0.03	-1.79

-

2/5/2025

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 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

Some 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
```

Some Basic Commands

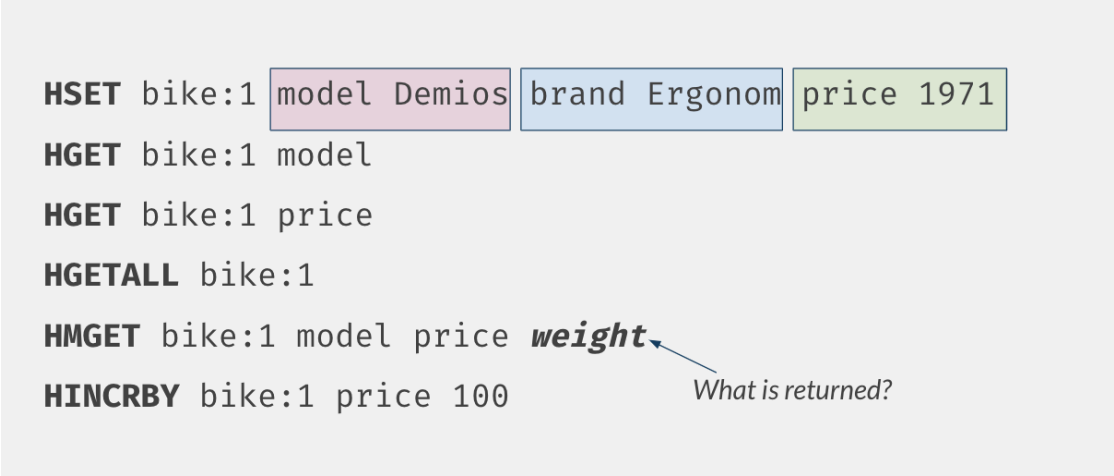
```
- SET someValue 0
  INCR someValue      #increment by 1
  INCRBY someValue 10 #increment by 10
  DECR someValue      #decrement by 1
  DECRBY someValue 5  #decrement by 5
    - INCR parses the value as int and increments (or adds to value)
- SETNX key value
    - only sets value to key if key does not already exist
```

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 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)

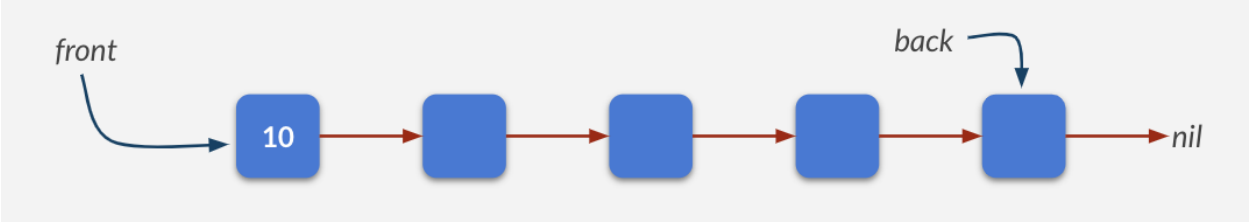
Hash Commands



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

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 one - points to nil/null)
- O(1) to insert new value at front or insert new value at end

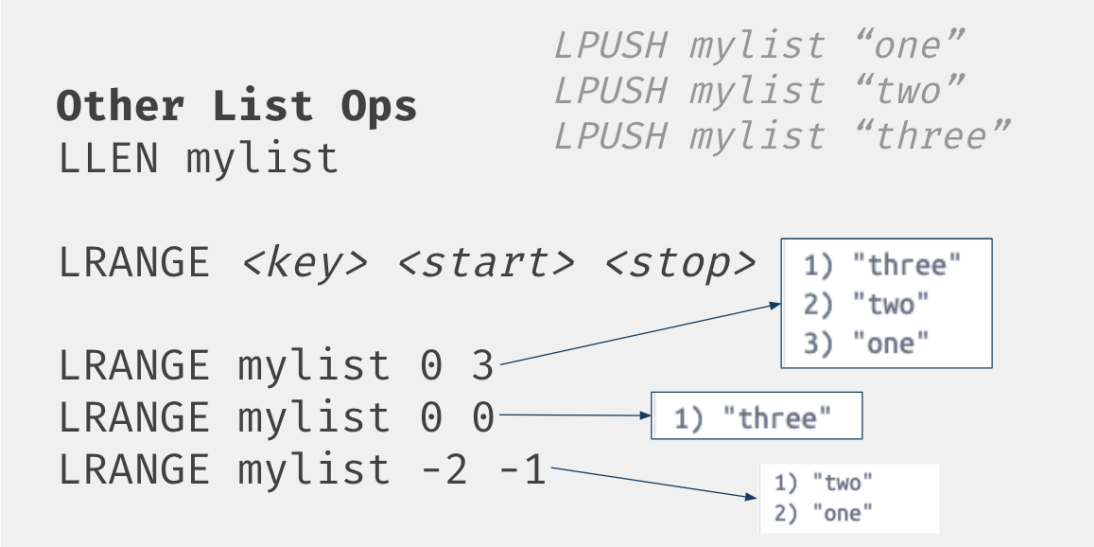
List commands - queue

```
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



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
```

2/10/25

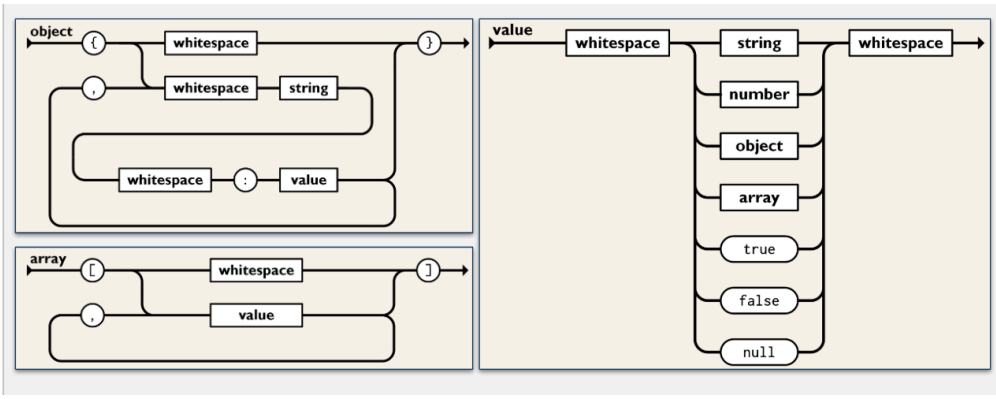
Document database

- 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 structure**s supported by virtually all modern programming languages
    - Thus, JSON makes a great data interchange format.

JSON Syntax



Binary JSON

- BSON → Binary JSON
    - 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

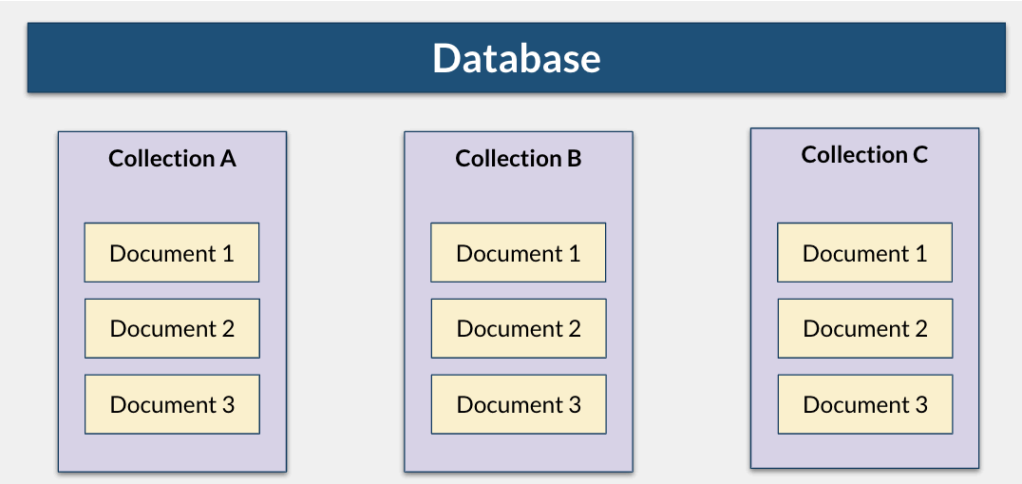
- **Xpath** - a syntax for retrieving specific elements from an XML doc
- **Xquery** - a query language for interrogating XML documents; the SQL of XML
- **DTD** - Document Type Definition - a language for describing the allowed structure of an XML document
- **XSLT** - eXtensible Stylesheet Language 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

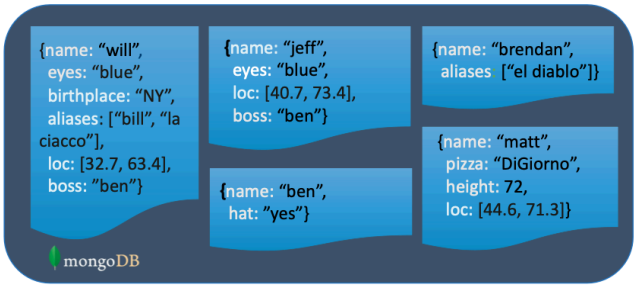
Mongo DB

- 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

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), …

2/12/25

Mongodb + pymongo

- PyMongo is a Python library for interfacing with MongoDB instances

```
- from pymongo import MongoClient
- client = MongoClient(
- 'mongodb://user_name:pw@localhost:27017'
- )
```

Getting a database and collection

```
from pymongo import MongoClient

client = MongoClient(
    'mongodb://user_name:pw@localhost:27017'
)

db = client['ds4300'] # or client.ds4300
collection = db['myCollection'] #or db.myCollection
```

Inserting a Single Document

```
db = client['ds4300']
collection = db['myCollection']

post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
}

post_id = collection.insert_one(post).inserted_id
print(post_id)
```

Find all movies from 2000

```
from bson.json_util import dumps

# Find all movies released in 2000
movies_2000 = db.movies.find({"year": 2000})

# Print results
print(dumps(movies_2000, indent = 2))
```

Jupyter Time

- Activate your DS4300 conda or venv python environment
- Install pymongo with pip install pymongo
- Install Jupyter Lab in you python environment
- pip install jupyterlab
- Download and unzip > this < zip file - contains 2 Jupyter Notebooks
- In terminal, navigate to the folder where you unzipped the files, and run jupyter lab

More Mongo Notes

MongoDB CRUD Operations: Notes, Explanations, and 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 } }
]);
```

This guide provides a detailed reference for MongoDB CRUD operations, including additional features for optimizing queries, data processing, and aggregation pipeline examples based on real-world queries.

2/19/25

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?

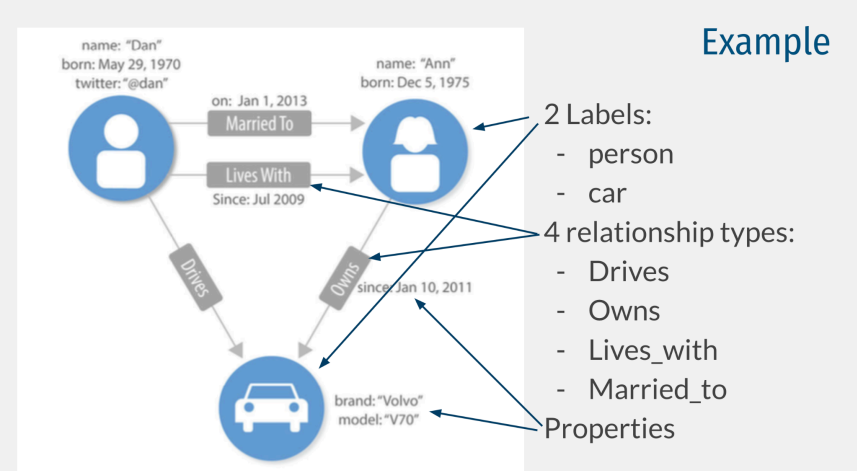
- Social Networks
    - yes… things like Instagram,
    - but also… modeling social interactions in fields like psychology and sociology
- The Web
    - it is just a big graph of “pages” (nodes) connected by hyperlinks (edges)
- Chemical and biological data
    - systems biology, genetics, etc.
    - interaction relationships in chemistry

Basics of Graphs and Graph Theory

What is a Graph?

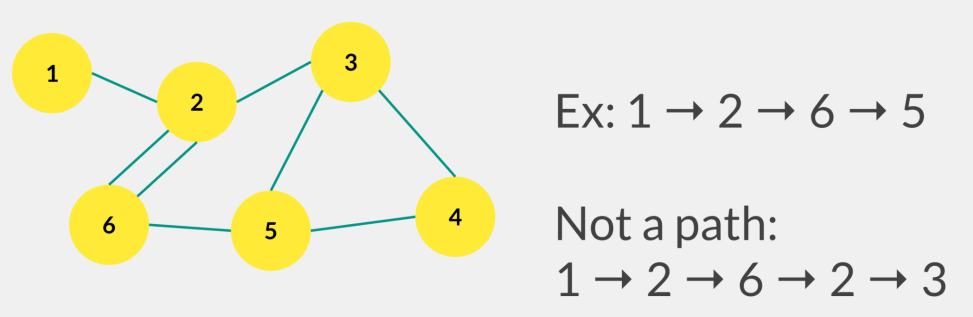
- Labeled Property Graph
    - Composed of a set of node (vertex) objects and relationship (edge) objects
    - Labels are used to mark a node as part of a group
    - Properties are attributes (think KV pairs) and can exist on nodes and relationships

- Nodes with no associated relationships are OK. Edges not connected to nodes are not permitted.



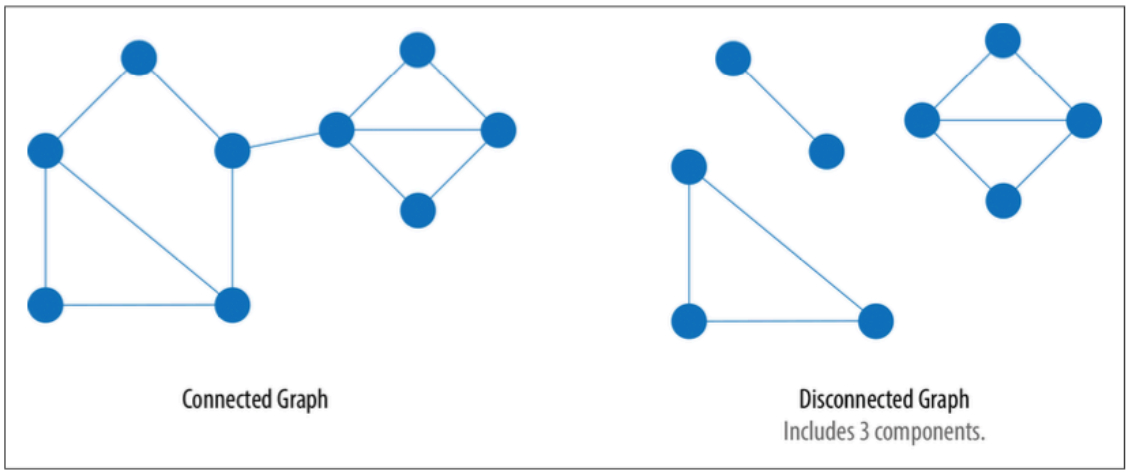
Paths

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



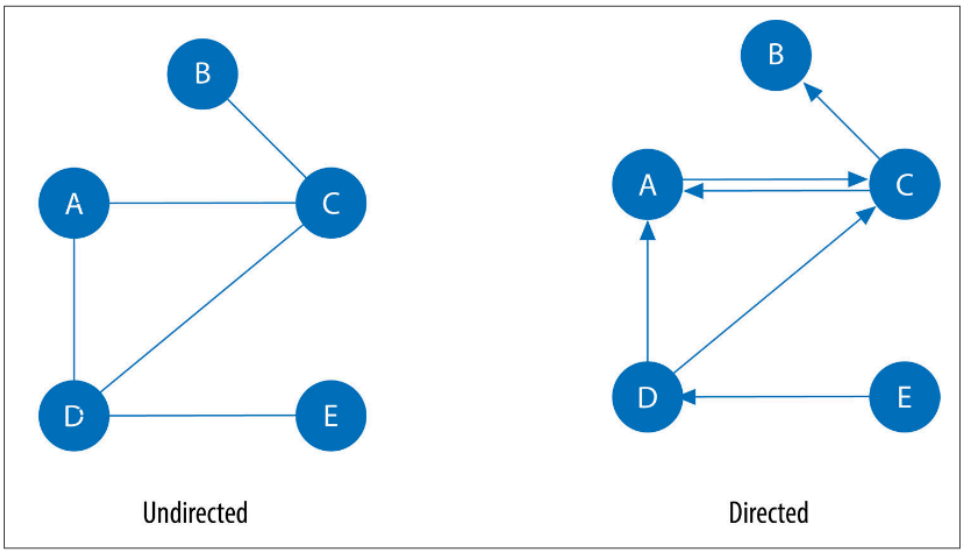
Flavors of Graphs

- **Connected (vs. Disconnected)** – there is a path between any two nodes in the graph

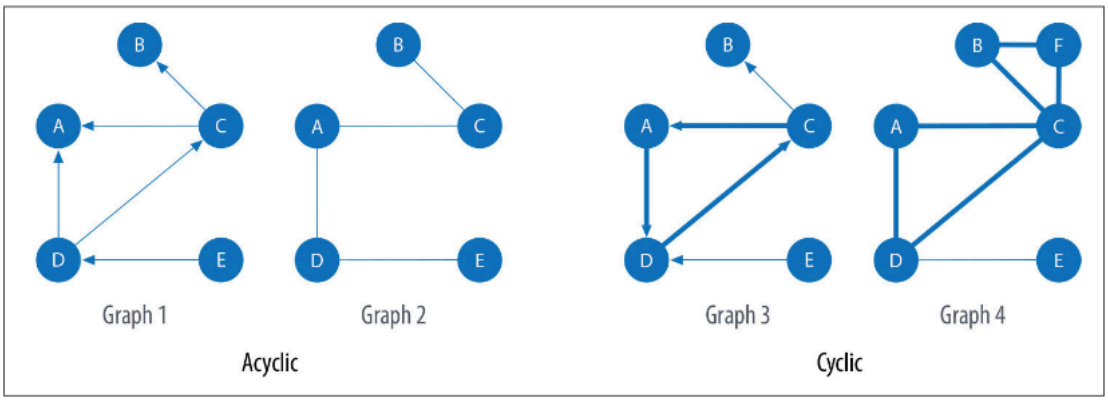


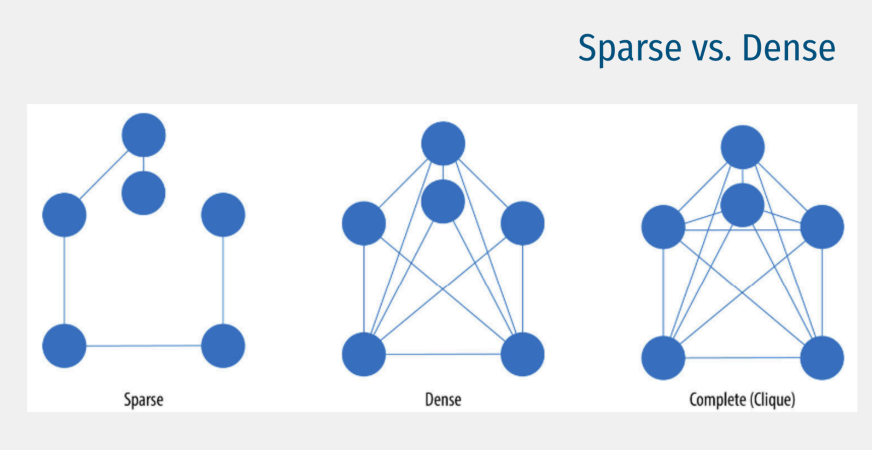
- **Weighted (vs. Unweighted)** – edge has a weight property (important for some algorithms)

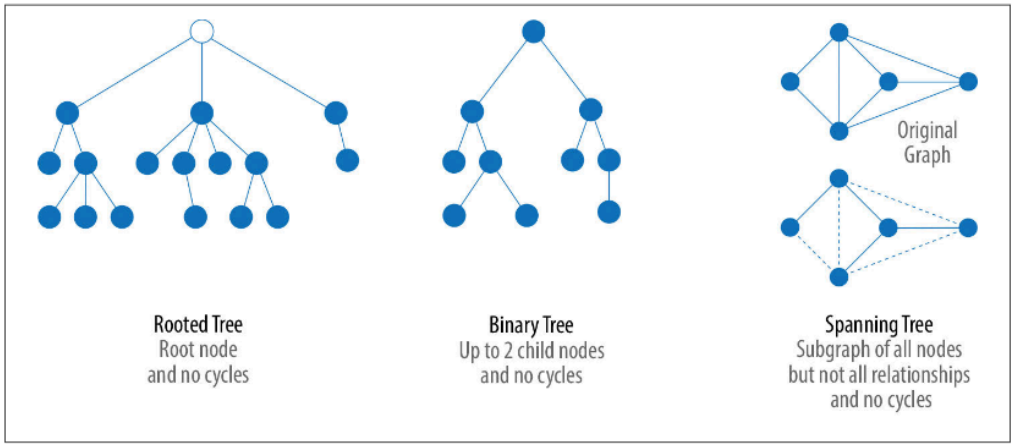
- **Directed (vs. Undirected)** – relationships (edges) define a start and end node



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

- 

- Sparse vs dense
- 

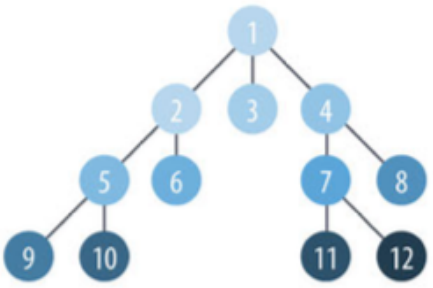
- Trees
- 

Types of Graph Algorithms - Pathfinding

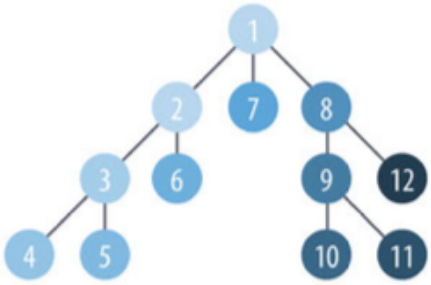
- Pathfinding
    - finding the shortest path between two nodes, if one exists, is probably the most common operation

- “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

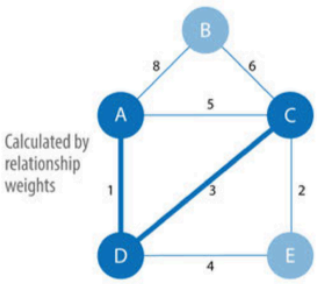


Breadth First Search
Visits nearest neighbors first



Depth First Search
Walks down each branch first

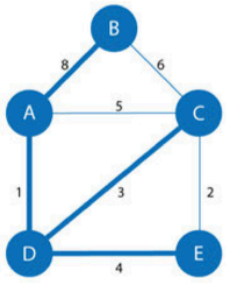
Shortest Path



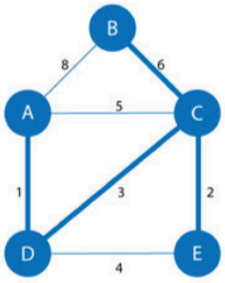
Shortest Path
Shortest path between 2 nodes (A to C shown)

(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...

All-Pairs Shortest Paths
Optimized calculations for shortest paths from **all** nodes to **all** other nodes



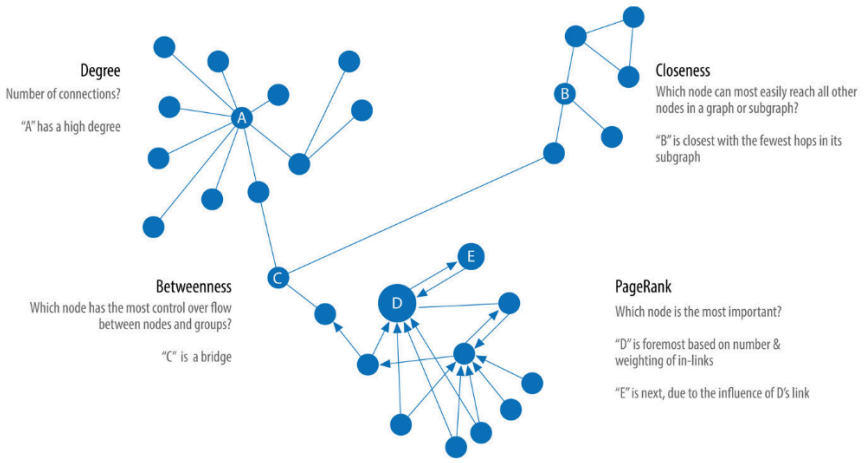
Single Source Shortest Path
Shortest path from a **root** node (A shown) to **all** other nodes



Minimum Spanning Tree
Shortest path **connecting all nodes** (A start shown)

Types of Graph Algorithms - Centrality & Community Detection

- 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 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

Create docler compose yaml

Create env file

Enter what is on the slides into them\

Delete existing neo4jcontainer

Docker compose up

General idea of an llm

1. Generate embeddings
    a. Chunk:
        i. File name, pdf page, vector
    b. Chunk stored in redis stack
2. User comes along and asks a question
    a. Generate an embedding for the question
        i. You get a vector that is based on a question
3. Send the vector to redis to get back the chunks that are most similar to the vectorized version of the question
    a. Can decide on most similar chunks that you want back from redis
4. Send back the k most similar chunks as context (mistral model)
    a. You construct a prompt (text based)
5. You get a contextual answer to your class notes (which is the database for this example)
6. System is called Rtrical Augmented Generation (RAG)

Ollama pull llama3.2:latest

Ollama pull mistral:latest

Pip install pymupdf

Make sure redis stack is running

Navigate to source folder

Around line 106, tell the computer what model you are using

3/12/25

AWS Intro

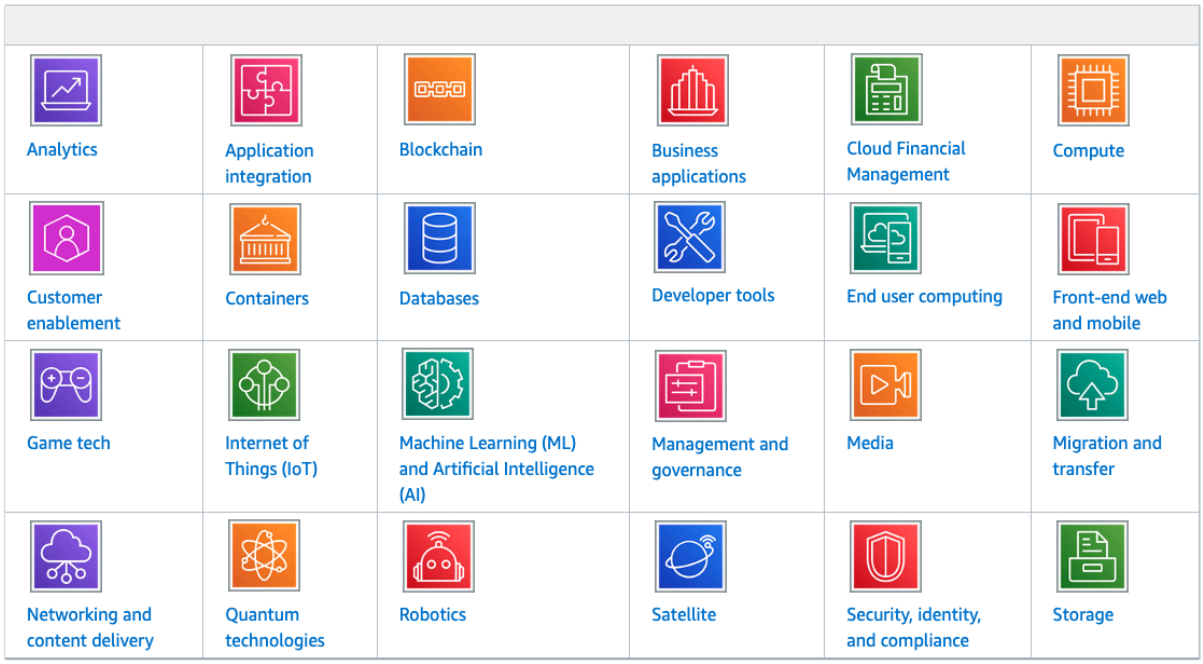
Amazon Web Services

- 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
- Based on a pay-as-you-use cost model.
- Theoretically cheaper than renting rackspace/servers in a data center… Theoretically.

History of AWS

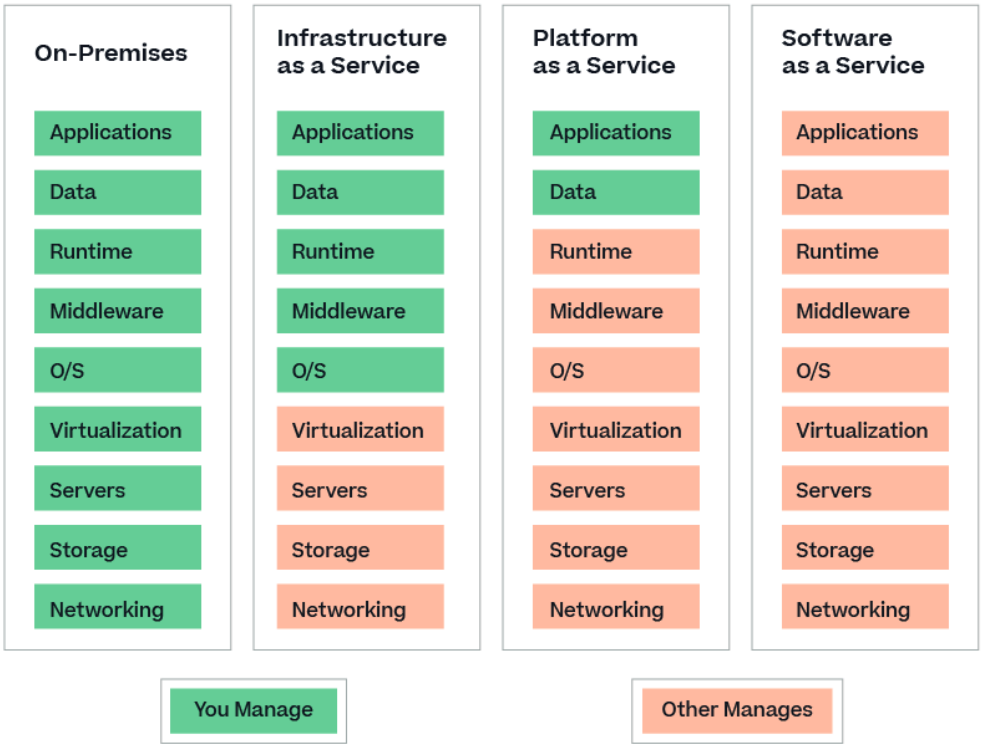
- Originally launched in 2006 with only 2 services: S3 & EC2.
- By 2010, services had expanded to include SimpleDB, Elastic Block Store, Relational Database Service, DynamoDB, CloudWatch, Simple Workflow, CloudFront, Availability Zones, and others.
- Amazon had competitions with big prizes to spur the adoption of AWS in its early days
- They've continuously innovated, always introducing new services for ops, dev, analytics, etc… (200+ services now)

AWS Service Categories



Cloud Models

- IaaS (more) - Infrastructure as a Service
    - Contains the basic services that are needed to build an IT infrastructure
- PaaS (more) - Platform as a Service
    - Remove the need for having to manage infrastructure
- You can get right to deploying your app
- SaaS (more) - Software as a Service
    - 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)

The Shared Responsibility Model - Client

- 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 & IAM
    - properly configure IAM users, roles, and policies.
    - enforce the Principle of Least Privilege
- Manage self-hosted Apps and associated OSs
- Ensure network 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 equiv to isolated data centers
- Edge Locations
    - locations for CDN and other types of caching services
    - allows content to be closer to end user.

Compute Services

- VM-based:
    - EC2 & EC2 Spot - Elastic Cloud Compute
- Container-based:
    - ECS - Elastic Container Service
    - ECR - Elastic Container Registry
    - EKS - Elastic Kubernetes Service
- Fargate - Serverless container service
- Serverless: AWS Lambda

Storage Services

- 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 MemoryDB, Amazon ElastiCache
- Document - Amazon DocumentDB (Compat with MongoDB)
- Graph - Amazon Neptune

Analytics Services

- Amazon Athena - Analyze petabyte scale data where 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

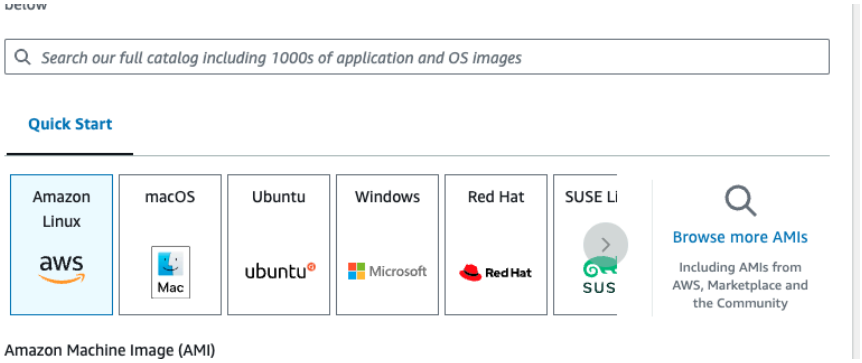
3/13/25

Amazon EC2 & Lambda

EC2

- EC2 → Elastic Cloud Compute
- Scalable Virtual Computing in the Cloud
- Many (Many!!) instance types available
- Pay-as-you-go model for pricing
- Multiple different Operating Systems

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.
    - AMI = amazon machine image
    - 

    - Amazon Machine Image (AMI)

EC2 Lifecycle

- **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?

- **Instance Store**: Temporary, high-speed storage tied to the instance lifecycle
- **EFS** (Elastic File System) Support - Shared file storage
- **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