FOUNDATIONS

Searching: the most common operation performed by a database system

- SQL SELECT

Linear Search (Baseline for efficiency)

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

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 attribute

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

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

Arrays are **faster** for random access, but **slow** for inserting anywhere but the end

- **If you are inserting into the middle of the array you have to move the chunk afterwards to maek space**

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

- Just have to make space and change memory address linkage

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

Linear Search

Best case: target is found at the first element; only 1 comparison O(1)
Worst case: target is not in the array; n comparisons

- Therefore, in the worst case, linear search is O(n) time complexity.

Database Searching
id: 1, 2, 3, 4, 5, 6, 7, 8
specialVal: 55, 87, 50, 108, 122, 149, 145, 120

Searching for specialVal

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.

Need external data structure to support faster searching by specialVal than a linear scan

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.

Binary Search Tree: a binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent.

Level Order: 4, 2, 6, 1, 3, 5, 7
Root: 4 (start of the tree)

Tree Traversals
Pre Order
Post Order
In Order
Level Order: 1st, 2nd, 3rd, etc. levels listed

- Uses a Que to store the data so you can find the level order
- Python used a **Deque** (double ended queue)

Binary Tree Node
Value: integer
Left: Binary Tree Node
Right: Binary Tree Node

AVL TREES

An **AVL Tree** is a **self-balancing binary search tree (BST)** where the height difference (balance factor) between left and right subtrees is at most **1** for every node. If an insertion causes an imbalance, **rotations** are used to restore balance.

Balance Factor (BF) of a node = Height of Left Subtree−Height of Right Subtree

AVL balance property |h(LST) - h(RST)| <= 1

- **BF = -1, 0, or 1** → Tree is balanced
- **BF < -1 or BF > 1** → Tree is unbalanced

4 Rotations:

Right Rotation (LL Case)

Occurs when a **node is inserted into the left subtree of the left child**.
Fix: Right Rotate (Single Rotation)

Essentially if the h(LST) - h(RST) > 1 for that given node it becomes the node of imbalance

Left Rotation (RR Case)

Occurs when a **node is inserted into the right subtree of the right child**.
Fix: Left Rotate (Single Rotation)

Left-Right Rotation (LR Case)

Occurs when a **node is inserted into the right subtree of the left child**.
Fix: Left Rotate + Right Rotate (Double Rotation)

Right-Left Rotation (RL Case)

Occurs when a **node is inserted into the left subtree of the right child**.
Fix: Right Rotate + Left Rotate (Double Rotation)

AVL Tree Insertion Steps

1. **Insert the node** as in a normal BST.
2. **Update height** of affected nodes.
3. **Calculate balance factor** for each node.
4. **If imbalance occurs**, apply the appropriate rotation:
    - **LL Case** → Right Rotation
    - **RR Case** → Left Rotation
    - **LR Case** → Left Rotation + Right Rotation
    - **RL Case** → Right Rotation + Left Rotation
5. **Tree remains balanced** after rebalancing.

```
class AVLNode:
    def __init__(self, key, value=None):
        self.key = key
        self.value = [value] if value else []  # Store multiple values per key
        self.left = None
        self.right = None
        self.height = 1  # Height of the node, initially 1 (leaf node)
```

```
    def add_value(self, value):
        self.value.append(value)  # Allows adding multiple values per key
```

```
class BinarySearchTreeIndex:
    def __init__(self):
        self.root = None
    def _rotate_right(self, imbalance):
        new_root = imbalance.left
        imbalance.left = new_root.right
        new_root.right = imbalance
        return new_root
    def _rotate_left(self, imbalance):
        new_root = imbalance.right
        imbalance.right = new_root.left
        new_root.left = imbalance
        return new_root
    def _height(self, node):
        if not node:
            return 0

        left_height = self._height(node.left)
        right_height = self._height(node.right)
        if left_height > right_height:
            print("left > right")
            return left_height + 1
        else:
```

```
            return right_height + 1

    def _update_height(self, node):
        node.height = 1 + max(self._height(node.left), self._height(node.right))
    def _check_balance(self, node):
        # if not node:
        #     return 0
        return self._height(node.left) - self._height(node.right)
    def _insert_recursive(self, current, key, value):
        # Perform normal BST insert
        if not current:
            node = AVLNode(key, value)
            return node
        if key < current.key:
            current.left = self._insert_recursive(current.left, key, value)
        elif key > current.key:
            current.right = self._insert_recursive(current.right, key, value)
        else:
            current.add_value(value)  # Handle duplicates by appending values
        # Update the height of the current node
        self._update_height(current)
        # Check balance factor and perform rotations if needed
        balance_factor = self._check_balance(current)
        if abs(balance_factor) > 1:
            print(f'There an IMBALANCE at: {current.key} with {balance_factor}')
        else:
            print(f'Balance factor of {current.key} is {balance_factor}')
        # Left heavy (Right rotation)
        if balance_factor > 1:
            # print(f'There an imbalance at: {current.key} with {balance_factor}')
            if key < current.left.key:
                return self._rotate_right(current)  # Single right rotation
            else:
                current.left = self._rotate_left(current.left)  # Left-Right rotation
                return self._rotate_right(current)
        # Right heavy (Left rotation)
        if balance_factor < -1:
            # print(f'There an imbalance at: {current.key} with {balance_factor}')
            if key > current.right.key:
                return self._rotate_left(current)  # Single left rotation
            else:
                current.right = self._rotate_right(current.right)  # Right-Left
rotation
                return self._rotate_left(current)
        return current
```

```
    def insert(self, key, value):
        self.root = self._insert_recursive(self.root, key, value)
```

```
    def print_tree(self, node, level=0, prefix="Root: "):
        if node is not None:
            print(" " * (level * 4) + prefix + f"({node.key})")
            if node.left or node.right:
                self.print_tree(node.left, level + 1, "L--- ")
```

```
            self.print_tree(node.right, level + 1, "R--- ")
    else:
        print(" " * (level * 4) + prefix + "None")
```

B+ TREES

A **B+ Tree** is a self-balancing tree data structure commonly used in databases and file systems. It maintains sorted data and allows for efficient searches, insertions, and deletions. Unlike a B-Tree, a **B+ Tree stores all data in the leaf nodes**, and internal nodes only serve as guides for searching.

Properties of a B+ Tree

- Each node contains **at most `M` children** (determined by the order of the tree).
- Each **internal node (non-leaf)** has **at least ⌈M/2⌉ children** (except the root).
- Each **leaf node** stores between **⌈M/2⌉ and M-1** keys.
- **All leaf nodes are linked** to form a sorted linked list (enables range queries).
- **Root node** can have fewer than ⌈M/2⌉ children.

Insertion Steps

Step 1: Insert the Key

1. Locate the correct **leaf node** where the key should be inserted.
2. Insert the key in **sorted order** within the leaf node.

Step 2: Check for Overflow

- If the leaf **does not exceed the max number of keys** (`M-1` keys), insertion is complete.
- If the leaf **overflows** (i.e., it now has `M` keys), **split the node** and propagate changes **upward**.

Node Splitting & Rebalancing

- **Leaf Node Split**

1. **Split the overfilled leaf node** into **two nodes**:
    - The **first half** contains the smaller keys.
    - The **second half** contains the larger keys.
2. **Promote the first key of the new right node** to the parent internal node.
3. **Adjust parent pointers** accordingly.

Internal Node Split: If an **internal node overflows**, perform the following steps:

1. **Split the internal node** into two nodes:
    - Left node contains ⌈M/2⌉ children.
    - Right node contains the remaining children.
2. **Promote the middle key to the parent node**.
3. If the **root node overflows**, create a **new root**.

Special Cases

- Root Splitting

- If the **root node overflows**, it **splits into two** and the middle key **becomes the new root**.
- This **increases the tree height**.

- Propagation of Splitting

- If an internal node is split and inserted into its parent, the **parent may also overflow**.
- This **recursively propagates** until a new root is created.

Summary of Insertion Process

1. **Find the correct leaf node** where the key should be inserted.
2. **Insert the key in sorted order** within the leaf node.
3. **If the node overflows**, split the node:
    - **Leaf nodes:** Promote the first key of the right node.
    - **Internal nodes:** Promote the middle key.
4. **Adjust parent pointers** and repeat the process **upwards** if necessary.
5. **If the root splits, create a new root** (increases tree height).

Moving Beyond the Relational Model

Benefits:

- Standard Data Model and Query Language
- ACID Compliance
    - Transaction is a unit of work
    - Atomicity, Consistency, Isolation, Durability
- Works well with highly structured data
- Can handle large amounts of data
- Well understood, lots of tooling, lots of experience

How RDBMS Increase Efficiency:

- Indexing
- Directly controlling storage
- Column oriented storage vs row oriented storage
- Query optimization
- Caching/prefetching
- Materialized views
- Precompiled stored procedures
- Data replication and partitioning

Transaction: A sequence of one or more 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

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 data T2 is writing, they can result in:
    - Dirty Read
    - Non-repeatable Read
    - Phantom Read

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.

Thread 1
1. Select Var (var = 50)
2. Var = 100
3. Update var (var = 100)
4. Might rollback the change

Thread 2
1. Select var (var = 50)
3. Select var (var = 100)

Possibility of first transaction rolling back the change, resulting in the second transaction having read an invalid value

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

Unless you changed the data itself it shouldn’t change for your thread
Getting different values each read because another transaction changed it

Phantom Reads: When a transaction T1 is running and another transaction T2 adds or deletes rows from the set T1 is using

Isolation ensured through locking data

- No one can read or write to say a specific table
- Different levels of locking: read, writes, or both

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

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:

- Vertically: Bigger machine (creates more powerful systems)
- Horizontally: More machines

Distributed System: A collection of independent computers that appear to its users as one computer

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

Distributed Storage

- Replication: Geographic availability
- Sharding: splits database into smaller partitions holding subset of data
    - Reduces load of individual server

Distributed Data Stores:

- Data is stored on > 1 node, typically replicated
    - Each block of data is available on N nodes
- Can be relational or non-relational
    - MySQL
    - CockroachDB
    - Many NoSQL systems support one or both models
- Network Partitioning is Inevitable
    - Network failures and system failures
    - Need to be Partition Tolerant
        - Systems can keep running even w/ network partition

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 Theorem - Database View
Consistency*: Every user of the DB has an identical view of the data at any given instant
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 segment 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 responds based on distributed store, but may not be the absolute latest data

NO SQL AND KV DATABASES

ACID transactions

- Focused on "data safety"
- **Pessimistic Concurrency Model:** assumes one transaction has to protect itself from other transactions
    - Assumes that if something can go wrong, it will
- Conflicts are prevented through 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

- Assumes that conflicts are unlikely
    - Even if there is a conflict, everything will still be OK
- 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.

Low Conflict Systems (backups, analytical DB)

- Read heavy systems
- Conflicts that arise can be handled by rolling back and re-running a transaction that notices a conflict
- Optimistic concurrency works well, allowing for higher concurrency

High Conflict Systems

- Rolling back and re-running transactions that encounter a conflict
    - Less efficient
- A locking scheme (Pessimistic Model) may be preferable

BASE

Basically Available: Guarantees availability of the data, but response can be 'failure/unreliable' because the data is in an inconsistent or changing state

- System appears to work most of the time (prioritizes responsiveness)

Soft State: The state of the system can change over time, even without input. Changes could be the result of eventual consistency. (data may temporarily be inconsistent across nodes)

- 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

BASE is used because: high availability is more important than strict consistency

Categories of NoSQL Databases

Key-Value Databases

- Key = Value
- **Simplicity:**
  - RDBMS are very complex
  - Simple CRUD ops and API creation
- **Speed:**
  - Deployed as in-memory DB
  - Retrieving a value given its key is O(1) op, similar to hash tables
  - No complex queries or joins since they slow things down
- **Scalability:**
  - Horizontal Scaling - Add more nodes
  - Concerned with eventual consistency, the only guarantee is that nodes will eventually converge to the same value

Key Value Database 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 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

Key Value SWE Use Cases

- Storing Session Information: Via a single PUT or POST and retrieved with a single GET very fast
- User Profiles and Preferences: User info obtained with single GET operation…, language, TZ, product or UI interfaces
- Shopping Cart Data: cart data is tied to a 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 used with other models: Graph, Spatial, Full Text Search, Vector, Time Series

Redis supports durability of data by saving snapshots to disk at specific intervals or append-only file which is a journal of changes that can be used for roll-forward if there is a failure

- Can be very fast .. > 100,000 SET operations/second, has a rich collection of commands, and doesn't 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, Sets, Sorted Sets, Hashes, Geospatial Data
    - Values are stored as a linked list in Redis

- Redis provides 16 databased by default
    - Numbered 0-15

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

Initial Basic Commands Redis

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)

SET someValue 0
INCR someValue # increments 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 set value to key if key does no 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 has is 2^32-1
    - Practical limit: available system resources (e.g. memory)
- Session information management
- User/Even tracking. (could include TTL)
- Active Session Tracking (all session under one hash key)

Hash Commands

HSET bike:1 model Demois 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 a linked list 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 queuing up a set of tasks to be executed sequentially at a later time

List Commands - Queue Operations

LPUSH bikes:repairs bike:1
LPUSH bikes:repairs bike:2
RPOP bike:repairs
RPOP biles:repairs

Stack Operations

LPUSH bikes:repairs bike:1
LPUSH bikes:repairs bike:2
LPOP bikes:repairs
LPOP bikes:repairs

Other List Operations

LPUSH mylist “one”
LPUSH mylist “two”

LPUSH mylist “three”
LLEN mylist

LRANGE <key> <start> <stop>

LRANGE mylist 0 3 # Returns 1) “three” 2)”two” 3) “one”
LRANGE mylist 0 0 # Returns 1) “three”
LRANGE mylist -2 -1 # Returns 1) “two” 2) “one

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

Set Commands
SADD ds4300 “Mark” # add Mark to the set of 4300
SADD ds4300 “Sam”
SADD cs3200 “Nick”
SADD cs3200 “Sam”
SISMEMBER ds4300 “Mark” # checks to see if Mark is in the set returns 1 if true 0 otherwise
SISMEMBER ds4300 “Nick”
SCARD ds4300 # number of elements in ds4300

More Set Commands
SCARD ds4300
SINTER ds4300 cs3200 # returns the intersection of ds4300 and cs 3200
SDIFF ds4300 cs3200 #returns elements in ds4300 that are not in cs3200
SREM ds4300 “Mark” # removes mark from ds4300
SRANDMEMBER ds4300 #selects one random member of ds4300

Redis + Python Documentation

Set Connection

```
import redis
# Connect to Redis
r = redis.Redis(host='localhost', port=6379, db=0)
# Test connection
print(r.ping())  # True if connected
```

Key Operations

```
r.set('key', 'value')  # Set a key
print(r.get('key'))  # Get a key (returns bytes, use .decode() if needed)

r.exists('key')  # Check if key exists (1 = True, 0 = False)
r.delete('key')  # Delete a key
r.expire('key', 10)  # Set key expiration in seconds
r.ttl('key')  # Get remaining time-to-live (TTL)
r.persist('key')  # Remove expiration
```

String Operations

```
r.set('count', 10)  # Set a numerical value
r.incr('count')  # Increment value (+1)
r.decr('count', 2)  # Decrement by 2
r.append('key', ' more data')  # Append to a string
r.getrange('key', 0, 4)  # Get substring
r.setrange('key', 6, 'new')  # Modify part of the string
```

List Operations

```
r.rpush('list', 'A', 'B', 'C')  # Push values to right
r.lpush('list', 'X')  # Push value to left
r.lrange('list', 0, -1)  # Get all list items
r.lpop('list')  # Remove & get first element
r.rpop('list')  # Remove & get last element
r.llen('list')  # Get list length
```

Set Operations

```
r.sadd('myset', 'A', 'B', 'C')  # Add to set
r.smembers('myset')  # Get all set members
r.sismember('myset', 'A')  # Check if A is in set
r.srem('myset', 'B')  # Remove an element
r.sunion('set1', 'set2')  # Union of sets
r.sinter('set1', 'set2')  # Intersection of sets
```

Hash Operations

```
r.hset('user:1000', 'name', 'Alice')  # Set field in hash
r.hget('user:1000', 'name')  # Get field value
```

```
r.hgetall('user:1000')  # Get all fields & values
r.hdel('user:1000', 'name')  # Delete a field
r.hkeys('user:1000')  # Get all field names
r.hvals('user:1000')  # Get all field values
r.hincrby('user:1000', 'age', 1)  # Increment field value
```

DOCUMENT DATABASES AND MONGODB

Document Database: A non-relational database that stores data as structured documents usually in JSON.

- Designed to be simple, flexible, and scalable.

JSON: JavaScript Object Notation

- Lightweight data-interchange format
- Easy to read and write, and parse and generate
- Built on 2 structures:
    - Collection of name/value pairs: Dictionary, hash table, or keyed list
        - In various languages operationalized as object, record, struct, dictionary, hash table, keyed list, or associative array
    - Ordered list of values: operationalized as an array, list, or sequence
- Universal Data Structure

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)
- Loss of being understood by humans
- Lightweight - keeps space overhead to a minimum
- Traversable - vitally important to document DB
- Efficient - encoding and decoding

eXtensible Markup Language XML (preceded JSON)

Extensible set of tags (structurally similar to html, but it isn't which is static)

XML + CSS -> web pages that separated content and formatting

Xpath, syntax for retrieving specific elements from an XML doc

Xquery, a query language from 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

Relational Database (not fast, and doesn't access data in a specific way)

- The JOINS are very operationally heavy, ACID compliance, etc.
- If you store all of the information together in a denormalized fashion (schema-less) it is more forgiving

Why 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
- The structure of a document is *self-describing*
    - They are well aligned with apps that use JSON/XML as a transport layer
    - One example is the characteristics of JSON can create filters for product searches

MongoDB Structure:

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

The structure is a collection of documents, and the documents itself don't have a predefined schema (JSON)

Relational vs Mongo/Document DB
RDMBS

1. Database
2. Table/View
3. Row
4. Columns
5. Index
6. Join
7. Foreign Key

MongoDB

1. Database
2. Collection
3. Document
4. Field
5. Index
6. Embedded Document
7. Reference

MongoDB Features:

- **Rich Query Support:** Robust support for al CRUD operations
- **Indexing:** Supports primary and secondary indices on document fields
- **Replication:** supports replica sets with automatic failover
- Load balancing built in

Mongo DB Comparison Operators

1. $eq : matches values that are equal to a specified value
2. $gt : greater than
3. $gte: greater than or equal to
4. $in: matches any of the values in array
5. $lt : less than
6. $lte : less than or equal to
7. $ne: matches all values that are not equal to a specified value
8. $nin: not in an array

Mongo DB Commands

The following aren't PyMongo commands

find(...) is like SELECT

- collection.find({ filters } , { projections})

SELECT * FROM users;

- use mflix
- db.users.find()

SELECT * FROM users WHERE name = "Davos Seaworth";

- db.users.find({"name" : "Davos Seaworth"})

SELECT * FROM movies WHERE rated in ("PG", "PG-13")

- db.users.find({rated: {$in: ["PG", "PG-13"] } })

Return movies which were released in Mexico and have an IMDB rating of at least

- db.movies.find({ "countries" : "Mexico" , "imbd.rating" : {$gte: 7} })

How many movies from movies collection were released in 2010 and either won at least 5 awards or have a genre of Drama

- Db.movies.countDocuments({ "year" : 2010, $or : [{ "awards.win" : {$gte :5 }" , {"genres" : "Drama" }]})

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

- Db.movies.find({ “year” : 2010, $or : [{ “awards.win” : {$gte :5 }” , {“genres” : “Drama” }]} , {“name”: 1, “_id”: 0})

GRAPH DATA MODEL

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
    - Supports queries based on graph-oriented operations
        - Traversals
        - Shortest path

Where do graphs show up:

1. Social Networks, also in modeling social interactions in psychology and sociology
2. The Web, big graph of “pages” (nodes) connected by hyperlinks (edges)
3. Chemical and Biological Data: systems biology, genetics, etc. interaction relationships in chemistry

Label Property Graph

- Composed of a set of node (vertex) objects and relationship (edge) objects
- Labels are used to mark a node as a part of a group
- Properties are attributes (think KV pairs) and can exist on nodes and relationships
- Nodes with no associated relationships are okay, edges not coned to nodes are not permitted

Path: An ordered sequence of nodes connected by edges in which no nodes or edges are repeated

- One example of a path is 1, 2, 6, 5 and an example of not a path is 1, 2, 6, 2, 3

Types of Graphs

- **Connected (vs. Disconnected):** There is a pair between any two nodes in the graph
- **Weighted (vs. Unweighted):** Edge has a weight property
- **Directed (vs. Undirected):** Relationships define a start and end node
- **Acyclic (vs. Cyclic):** Graph contains no cycle
- **Sparse vs Dense: Sparse, Dense, Complet (Clique) where every node is connected to each other**

Spanning Tree, subgraph of all nodes but not all relationships and no cycles

Pathfinding

- Finding the shortest path between two nodes
- "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 (Breadth First Search), visits nearest neighbors first
DFS (Depth First Search), walks down each branch first

Shortest Path: between 2 nodes
Single Source Shortest Path is the shortest path from a root node (A shown) to all other nodes
Minimum Spanning Tree, shortes path connecting all nodes (A start shown)

Types of Graph Algorithms

Centrality: Determining which nodes are "more important" in a network compared to other nodes

- Example: social network influencers)

Community Detection: Evaluate clustering or partitioning of nodes of a graph and tendency to strengthen or break apart

Centrality More Details

1. Degree is the number of connections
2. Betweenness represents which node has the most control over flow between does and groups (is the node a bridge?)
3. Closeness describes which node can most easily reach all other nodes in a graph or subgraph
4. PageRank, which rank is the most important (D is foremost based on number & weighting of in-links, E is next due to influence of D's link)

Famous Graph Algorithms:

1. Dijkstra's Algorithm, single-source shortest path algorithm for positively weight graphs
2. A* Algorithm, Similar to Dijkstra's with added feature of using a heuristic to guide traversal
3. PageRank, measures the importance of each node within a graph based on the number of incoming relationships and the importance of the does 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
- Supports various types of indexing
- ACID Compliant
- Support distributed computing
- Similar to Microsoft CosmoDB, Amazon Neptune.

Neo4j - Query Language and Plugin

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

- Extend funtionality of cypher (add-on libraries)

Graph Data Science Plugin

- Implementation of graph algorithms

Docker Compose

Support multi-container management

1 command used to start, stop, or scale a number of services at one time

- Services, volumes, networks, etc.

Consistent method for producing an identical environment

Interaction through command line

Virtual private network: sets up infrastructure around your services so they can talk together

- Nothing can talk to them from outside that virtual firewall unless you punch say a port through

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
```

Instead of talking to the local host you talk to the neo4j private network?

- Forward information from host port 7474 to inside the private network

- Never sync files with GitHub env
- Containers are ephemeral meaning it can go up and down so you map data and logs to something else (not inside the container)

.**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

Docker compose up

- Images don't exist starts from images all of the services in the docker compose file

Docker compose up - d (detached)

Docker compose down (turn down and delete)

Docker compose start (turning back on after stopped)

- Containers exist after doing up

Docker compose stop

Docker compose build (create image before tries to spin it up)

Docker compose build --no-cache

- Redownload all layers from scratch if you get stuck in python dependencies

PyMongoDB Documentation

Find Queries

```
db.collection.find({}) // Find all documents
db.collection.findOne({ key: value }) // Find a single document
db.collection.find({ key: "value" }) // Exact match
db.collection.find({ key: { $ne: "value" } }) // Not equal to
db.collection.find({ key: { $in: ["value1", "value2"] } }) // Matches any in array
db.collection.find({ key: { $nin: ["value1", "value2"] } }) // Not in array
```

Comparison

```
db.collection.find({ key: { $gt: 10 } }) // Greater than
db.collection.find({ key: { $gte: 10 } }) // Greater than or equal to
db.collection.find({ key: { $lt: 10 } }) // Less than
db.collection.find({ key: { $lte: 10 } }) // Less than or equal to
db.collection.find({ key: { $eq: "value" } }) // Equal to
```

Logical

```
db.collection.find({ $and: [{ key1: "value1" }, { key2: "value2" }] }) // AND condition
db.collection.find({ $or: [{ key1: "value1" }, { key2: "value2" }] }) // OR condition
db.collection.find({ key: { $not: { $eq: "value" } } }) // NOT condition
db.collection.find({ $nor: [{ key: "value1" }, { key: "value2" }] }) // Neither condition matches
```

Text Searches

```
db.collection.find({ key: /pattern/ }) // Case-sensitive regex search
db.collection.find({ key: /^pattern/ }) // Starts with
db.collection.find({ key: /pattern$/ }) // Ends with
db.collection.find({ key: { $regex: "pattern", $options: "i" } }) // Case-insensitive
```

Arrays

```
db.collection.find({ key: { $all: ["value1", "value2"] } }) // Array contains all values
db.collection.find({ key: { $size: 3 } }) // Array length is 3
db.collection.find({ key: { $elemMatch: { subKey: "value" } } }) // Matches at least one element in array
```

Sorting and Limiting

```
db.collection.find().sort({ key: 1 }) // Ascending order
db.collection.find().sort({ key: -1 }) // Descending order
db.collection.find().limit(5) // Limit results to 5 documents
db.collection.find().skip(5).limit(10) // Skip 5 documents and return 10
```

Aggregation Framework

```
db.collection.aggregate([
  { $match: { key: "value" } }, // Filter documents
  { $group: { _id: "$category", count: { $sum: 1 } } }, // Grouping
  { $sort: { count: -1 } } // Sorting
])
```

Ds 4300-mongo-db-examples-students.ipynb

Counting Documents

```
numCustomers = demodb.customers.count_documents({})
numOrders = demodb.orders.count_documents({})

# The key (_id) attribute is automatically returned unless you explicitly say to remove it.
(SQL) SELECT name, rating FROM customers
(PyMongo) data = demodb.customers.find({}, {"name":1, "rating":1}) #removing id

# For every customer, return all fields except _id and address.
 data = demodb.customers.find({}, {"_id": 0, "address": 0})

Equivalent to SQL LIKE operator
# SELECT name, rating FROM customers WHERE name LIKE 'T%'
# Regular Expression Explanation:
 # ^ - match beginning of line
 # T - match literal character T (at the beginning of the line in this case)
 # . - match any single character except newline
# * - match zero or more occurrences of the previous character (the . in this case)
data = demodb.customers.find({"name": {"$regex": "^T.*"}}, {"_id": 0, "name": 1, "rating":1})


Sorting and Limiting
# SELECT name, rating FROM customers ORDER BY rating LIMIT 2
# no document filters with { }
# attribute filters with {"_id": 0, "name": 1, "rating":1}
# can't reorder attributes
    - data = demodb.customers.find( { }, {"_id": 0, "name": 1, "rating":1} ).sort("rating").limit(2)

# Same as above, but sorting in DESC order
# SELECT name, rating FROM customers ORDER BY rating DESC LIMIT 2
 data = demodb.customers.find( { }, {"_id": 0, "name": 1, "rating":1} ).sort("rating", -1).limit(2)

# Providing 2 sort keys...  # -1 is desc 1 is asc
 data = demodb.customers.find( { }, {"_id": 0, "name": 1, "rating":1} ).sort({"rating": -1, "name":
1}).limit(2)

# How many Users are there in the mflix database? How many movies? q1 =
mflixdb.users.count_documents({})

# Which movies have a rating of “TV-G”? Only return the Title and Year.  q2 =
mflixdb.movies.find({"rated": "TV-G"}, {"title": 1, "year": 1, "_id": 0})
```

```
# Which movies have a runtime of less than 20 minutes?  Only return the title and runtime of each movie.
q3 = mflixdb.movies.find({"runtime": {'$lt': 20}}, {"title":1, "runtime":1, "_id":0})


# How many theaters are in MN or MA?  q4 = mflixdb.theaters.count_documents({'location.address.state' : {"$in": ["MN", "MA"]}})
print(q4)


# Give the names of all movies that have no comments yet. Make sure the names are in alphabetical order.
q5 = mflixdb.movies.find({"num_mflix_comments" : {"$eq": 0 }}, {"title":1, "_id":0} ).sort("title")

commented_movies = mflixdb.comments.distinct('movie_id')
movies_no_comments = mflixdb.movies.find(     {'_id': {"$nin": commented_movies}}, {'title':1 , '_id':0}).sort('title',1)

# Return a list of movie titles and all actors from any movie with a title that contains the word 'Four'.  # Sort the list by title.
q6 = mflixdb.movies.find({"title": {"$regex": ".*Four.*", '$options': 'i'}}, {"_id": 0, "title": 1, 'cast':1})

movies_with_four = mflixdb.movies.find(     {'title': {'$regex': 'Four', '$options': 'i'}},      {'title':1, 'cast':1, '_id':0} ).sort('title',1)



Ds-4300-mongodb-aggregation.ipynb
$match is essentially the find argument
c = mflixdb.movies.aggregate([     {"$match": {"year": {"$lte": 1920}}}, ])

Math and project
Where clause and project
c = mflixdb.movies.aggregate([     {"$match": {"year": {"$lte": 1920}}},     {"$project": {"_id":0, "title": 1, "cast": 1}}, ])

c = mflixdb.movies.aggregate([     {"$match": {"year": {"$lte": 1920}}},     {"$sort": {"title": 1}}, {"$limit": 5}, #reserve top 5 documents after sorting      {"$project": {"_id":0, "title": 1, "cast": 1}}, ])

c = mflixdb.movies.aggregate([     {"$match": {"year": {"$lte": 1920}}},     {"$sort": {"imdb.rating": -1}},     {"$limit": 5},     {"$unwind": "$cast"}, # takes it out of the embedded json     {"$project": {"_id":0, "title": 1, "cast": 1, "rating": "$imdb.rating"}}, ])
```

Grouping

```
# What is the average IMDB rating of all movies by year? sort the data by year.  c = mflixdb.movies.aggregate([     {"$group": {"_id": {"release year": "$year"}, "Avg Rating": {"$avg": "$imdb.rating"}}},     {"$sort" : {"_id": 1}}      ])

# What is the average IMDB rating of all movies by year? sort the data by avg rating in decreasing order.  c = mflixdb.movies.aggregate([     {"$group": {"_id": {"release year": "$year"}, "Avg Rating": {"$avg": "$imdb.rating"}}},     {"$sort" : {"Avg Rating": -1, "_id": 1}}      ])
```

Lookup

Equivalent to simple join? Not as efficient as joins in a relational db

```
data = demodb.customers.aggregate([     {         "$lookup": {             "from": "orders",  "localField": "custid",  "foreignField": "custid",             "as": "orders"         }     },     {"$project": {"_id": 0, "address": 0}} ])
```

Reformatting Queries reconstructs state for each dictionary

```
match = {"$match": {"year": {"$lte": 1920}}} limit = {"$limit": 5} project = {"$project": {"_id":0, "title": 1, "cast": 1, "rating": "$imdb.rating"}}  agg = mflixdb.movies.aggregate([match, limit, project])
```

Assignment 03

Give the street, city, and zipcode of all theaters in Massachusetts.

```
q1 = mflixdb.theaters.find(
{'location.address.state': 'MA'},
{'location.address.street1':1, 'location.address.city':1, 'location.address.zipcode':1, '_id':0})
```

How many theaters are there in each state? Order the output in alphabetical order by 2-character state code

```
q2 = mflixdb.theaters.aggregate([     {'$group': {'_id': {'state':'$location.address.state'}, 'count': {'$sum': 1}}},  # Count theaters per state     {'$sort': {'_id': 1}}   ])
```

How many movies are in the Comedy genre?

```
q3 = mflixdb.movies.count_documents({'genres': 'Comedy'})
```

What movies has the longest run time? Give the movie’s title and genre(s)

```
q4 = mflixdb.movies.find(     {},      {'title':1, 'genres':1, '_id':0}).sort('runtime',-1).limit(1)
```

Which movies released after 2010 have a Rotten Tomatoes viewer rating of 3 or higher? Give the title of the movies along with their Rotten Tomatoes viewer rating score. The viewer rating score should become a top-level attribute of the returned documents. Return the matching movies in descending order by viewer rating.

```
match = {'$match': {'year': {'$gt' : 2010} , 'tomatoes.viewer.rating': {'$gte': 3}}}
project = {'$project': {'_id':0, 'title': 1, 'viewer_rating': '$tomatoes.viewer.rating'}}
sort = {'$sort': {'viewer_rating': -1}}

q5 = mflixdb.movies.aggregate([match,project, sort])
```

How many movies released each year have a plot that contains some type of police activity (i.e., plot contains the word "police")? The returned data should be in ascending order by year.

```
group = {'$group': {'_id': {'release_year': '$year'} , 'movie_count': {'$sum':1}}}
match = {'$match': {'plot': {'$regex': '.*police.*', '$options': 'i'}}}
sort = {'$sort': {'_id.release_year':1}}
project = {'$project': {'_id':0, 'year_release': '$_id.release_year', 'movie_count':1}}

q6 = mflixdb.movies.aggregate([match, group, sort, project])
```

What is the average number of imdb votes per year for movies released between 1970 and 2000 (inclusive)? Make sure the results are order by year.

```
match = {'$match': {'year': {'$gte': 1970, '$lte': 2000}}}
group = {'$group': {'_id': '$year', 'average_num_imdb_votes': {'$avg': '$imdb.votes'}}}
project = {'$project': {'release_year': '$_id', '_id':0, 'average_num_imdb_votes': 1, '_id': 0}}
sort = {'$sort': {'release_year': 1}}

q7 = mflixdb.movies.aggregate([match, group, project, sort])
```

What distinct movie languages are represented in the database? You only need to provide the list of languages.

```
q8 = mflixdb.movies.distinct('languages')
```