B+ Trees

- Optimized for disk space indexing.
- Minimizes disk access.
- A B+ Tree is an n-way tree with order `M`.
    - `M` = maximum number of keys in each node.
    - Maximum number of children = `M + 1`.

Node structure (for M = 3)

| a | b | c |
< a, >= a & < b
>= b & < c
>= c

- All nodes (except root) must be at least half full.
- Insertions are done at the leaf level.
- Leaves are stored as a doubly linked list.
- Keys in nodes are kept sorted.

Internal Nodes

- Store keys and pointers to children.

Leaf Nodes

- Store keys and actual data.

Example (insert 42, 29, 81):

```
  [42]
  /   \
[29]   [81]
```

In-memory vs Disk-based B+ Trees:

- In-memory: Each node stores key-value pairs.
- Disk-based: Each node stores keys and pointers to children.

Insertion Example (99, 35, 2):

- When node is full:
    - Split the nodes evenly.

    - Create a new parent node.
    - Use the first value in the new node and add it to the parent.

Hash Tables

- Like a Python dictionary.

Components:

- Table size: `M`
- Load factor: `λ = n / m`
    - Ratio of how many values vs slots.
- Keys and values:
    - Example: `k = 10, v = 'cat'`
    - Hash function: `h(k) = k % M`

Good Hash Functions:

- Work independently of table size.
- Spread out values evenly across hash space.
- Keep chains short (ideally < 5).

Buckets:

- Can be Python lists.

Example Hash Table Layout (M = 6):

```
Index | Values
------|--------
0     |
1     | finance, 7.json
      | bank, 15.json
2     |
3     |
4     |
5     | money, 10.json
      | 7.json
```

Inverted File Index:

- Indexing terms to locations in documents.
- Final entries:

(3,1): (finance, 7.json)
(7,4): (money, 10.json)
(3,5): (bank, 15.json)
(7,6): (money, 7.json)

Memory Hierarchy

CPU
Registers
L1 Cache
L2 Cache
RAM
SSD / HDD <-- Slower, lots of storage

- DB systems aim to minimize SSD/HDD access.

KV Pair Storage (e.g. 32 bytes)

- Sorted array of 128 integers = 2048 byte block.
- Binary search on 128 values is faster than one disk access.

B+ Tree Goal:

- Optimize the number of keys per node to reduce disk access.

AVL Trees

Case 4: RR

```
 x
  \
   z
  /
 y
```

- Becomes:

```
 z
 / \
x   y
```

- Insert 3, 2, 1 → Tree becomes unbalanced.
- As soon as unbalanced, rebalance.

Insert 3,2,1,4,5,6,7:

- Rebalance using rotation when needed.
- AVL tree uses max(height) + 1 to rebalance.

AVL Tree Rotations

Imbalance Cases:

1. LL – Insert into left subtree of left child.
2. LR – Insert into left subtree of right child.
3. RL – Insert into right subtree of left child.
4. RR – Insert into right subtree of right child.

Rebalancing

- Case 1 & 4: Single rotation
- Case 2 & 3: Double rotation

Tree Height & Balancing

- Goal: Minimize tree height.
- Minimum height = all levels filled except the last.

AVL Tree

- Balanced Binary Search Tree (BBT).
- Balance factor at each node: `|height(left subtree) - height(right subtree)| ≤ 1`

Example AVL Tree:

- Insert 50 → no imbalance
- Insert 5 → no imbalance
- Insert 7 → imbalance occurs, AVL tree rotates to restore balance.

B+ Tree Insertion

A B+ Tree stores all values in leaf nodes, and only routes (keys) in internal nodes.

Leaf Node Insertion

- Add in sorted order
- If full, split the node:
  - Move first key to left node
  - Move other keys to a new right node
  - Push first key from right node up to the parent

Internal Node Insertion

- If parent is full after a push-up:
  - Split parent, and repeat
- Tree height increases if the root splits

Example with M = 3:

- Max 3 keys per internal node
- Max 3 keys per leaf node
- Max 4 children per internal node

➡ Splitting a leaf node:

Insert → [21, 29, 38, 41]

Split → [21] [29, 38, 41]

Push 29 up

➡ When height increases:

- If the root splits, a new root is created
- Tree gets one level taller

Hash Table Insertion

A hash table maps a key to an index using a hash function.

Step-by-Step:

1. Hash Function: Compute index `i = h(key)`
2. Insert into bucket at index `i`:
    - If empty, place the key-value pair
    - If not, append to the list at that index (chaining)
3. Chaining is typically implemented as a linked list or Python list

Example:

Hash Function: `h(key) = key % 10`
Insert: `(24:R) → h(24) = 4 → bucket[4].append((24, R))`

Binary Search Tree (BST)

Insertion Example: `23, 17, 20, 43, 31, 50`

```
   23
   / \
 17   43
   \  / \
   20 31 50
```

Traversal Types:

- Preorder
- Inorder
- Postorder
- Level Order

Level Order Traversal:

- Goes level by level.
- Example Output: `23 17 43 20 31 50`
- Use a queue (in Python: `deque` from `collections` module).

Binary Tree Node (Python-style):

```
class BinTreeNode:
    def __init__(self, val):
        self.value = val
        self.left = None
```

```
        self.right = None

root = BinTreeNode(23)
root.left = BinTreeNode(17)
```

Searching in Databases

- Searching is the most common operation in a database system.
- In SQL, the `SELECT` statement is the most versatile/complex.

Baseline for Efficiency: Linear Search

- Start at the beginning of a list and proceed element by element until:
  - You find the target
  - Or you reach the end without finding it

Linear Search

- For `n` values, compare the target with each value individually.
- Best case: 1 comparison (found at start).
- Worst case: `n` comparisons (not found).
- Time complexity: **O(n)**

Key Terms

- **Record**: Collection of values for attributes of a single entity (a row).
- **Collection**: Set of records for the same entity type (a table).
- **Search Key**: Value(s) from one or more attributes.
- **Contiguously Allocated List**: All `n*x` bytes allocated as one chunk in memory.

Memory = Primary Storage (RAM)

Linked List

- Each record uses `x` bytes + memory addresses.
- Records are chained together via pointers.
- Slower for random access—must traverse from start.
- Python doesn't support contiguous arrays natively.
  - Use NumPy for this (it's just C under the hood).

Arrays vs. Linked Lists

Feature	Arrays	Linked Lists

Random Access	Fast	Slow
Random Insertion	Slow	Fast

Binary Search

- Input: sorted array, target value
- Output: index of target or indicator if not found
- Best case: 1 comparison (middle element)
- Worst case: $\log_2(n)$ comparisons (not found)
- Time complexity: **O(log n)**
- Only applicable to sorted, contiguously allocated arrays
- Not suitable for linked lists (can't jump to the middle)

Database Searching Strategy

- Assume data is stored on disk sorted by column `id`
  - Searching for specific `id` is fast
- But what about a different attribute like `specialVal`?
  - Requires a linear scan if not sorted by that attribute

Limitations:

- Can't sort by both `id` and `specialVal` simultaneously → requires data duplication (space inefficient)

Solutions:

- **Sorted Array of Tuples (specialVal, rowNumber)**
  - Binary search is fast
  - But insertion is slow (need to maintain order)
- **Linked List of Tuples (specialVal, rowNumber)**
  - Insertion is fast (append to end)
  - Searching is slow (linear scan)

Pairwise Search Algorithms

1. Find Closest Pair in an Unsorted Array

- **Approach**: Brute-force comparison of each pair.
- **Complexity**: $O(n^2)$ comparisons.

```
def find_min_dist(unsorted_array):
  current_min_val = abs(max(unsorted_array) - min(unsorted_array))
  min_pairs = []
  comparision_counter = 0

  for val in range(len(unsorted_array)):
    target_val = unsorted_array[val]
    for comparison_val in unsorted_array[val+1:]:
      absolute_val = abs(target_val - comparison_val)
      comparision_counter += 1

      if absolute_val < current_min_val:
        current_min_val = absolute_val
        min_pairs = [(target_val, comparison_val)]
      elif absolute_val == current_min_val:
        min_pairs.append((target_val, comparison_val))

  return min_pairs, comparision_counter
```

---

2. Find Closest Pair in a Sorted Array

- **Approach**: Only compare each element with its immediate right neighbor.
- **Complexity**: O(n) comparisons.

```
def sorted_list_search(sorted_array):
  current_min_dist = abs(max(sorted_array) - min(sorted_array))
  min_pairs = []
  comparison_counter = 0

  for ind in range(len(sorted_array) - 1):
    target_value = sorted_array[ind]
    absolute_value = abs(target_value - sorted_array[ind+1])
    comparison_counter += 1

    if absolute_value < current_min_dist:
      current_min_dist = absolute_value
      min_pairs = [(target_value, sorted_array[ind+1])]
    elif absolute_value == current_min_dist:
      min_pairs.append((target_value, sorted_array[ind+1]) )

  return min_pairs, comparison_counter
```

Efficiency Discussion:
The **sorted array** approach is more efficient due to fewer comparisons. In an unsorted array, every number must be compared to every number after it. In a sorted list, the closest neighbor (for minimal difference) must be adjacent.

Reverse Level Order Traversal of a Binary Tree

Key Concepts:

- Traverse **bottom-up**, level by level.
- Use a **queue** for BFS and then reverse the results.

```
from collections import deque

class BinTreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def reverse_traverse(root):
    """Performs a bottom-up level-order traversal."""
    output = []
    temp_store = deque([root])

    while temp_store:
        current_level = []

        for _ in range(len(temp_store)):
            current_val = temp_store.popleft()
            current_level.append(current_val.value)

            if current_val.left:
                temp_store.append(current_val.left)
            if current_val.right:
                temp_store.append(current_val.right)

        output.append(current_level)

    # Reverse the output for bottom-up display
    reversed_output = []
    for level in reversed(output):
        for val in level:
```

```
        reversed_output.append(val)

    return reversed_output

# Test Tree
def main():
    root = BinTreeNode(40)
    root.left = BinTreeNode(20)
    root.right = BinTreeNode(60)
    root.left.right = BinTreeNode(35)
    root.left.left = BinTreeNode(17)
    root.right.left = BinTreeNode(55)
    root.right.right = BinTreeNode(85)
    root.left.left.left = BinTreeNode(10)
    root.left.left.right = BinTreeNode(19)
    root.left.right.left = BinTreeNode(41)
    root.left.right.right = BinTreeNode(42)
    root.left.left.left.left = BinTreeNode(22)
    root.left.left.left.left.left = BinTreeNode(100)

    print(reverse_traverse(root))  # Should print nodes from bottom level to root
```

The Relational Model

Benefits:

- Mostly standard data model and query language (SQL).
- ACID compliance:
  - **Atomicity**: Entire transaction is treated as a single unit – all or nothing.
  - **Consistency**: Transforms the database from one valid state to another.
  - **Isolation**: Transactions do not interfere with each other.
  - **Durability**: Once committed, changes are permanent.
- Handles highly structured and large volumes of data.
- Well-understood with mature tools and developer expertise.

Transaction:

- A unit of work that may involve one or multiple operations (e.g., SELECT, INSERT).

- Either:
    - Entire sequence **commits** (succeeds), or
    - Entire sequence **rolls back** (fails).

Performance Optimizations in RDBMS:

- Indexing
- Storage control (row vs. column orientation)
- Query optimization
- Caching and prefetching
- Materialized views
- Precompiled stored procedures
- Data replication and partitioning

Transaction Processing:

- CRUD operations (Create, Read, Update, Delete) grouped into logical units.
- Ensures:
    - Data integrity
    - Error recovery
    - Concurrency control
    - Reliable storage
    - Simplified error handling

ACID Properties (Expanded):

- **Atomicity**: Either all operations of the transaction are completed or none are.
- **Consistency**: Ensures the database starts and ends in a valid state.
- **Isolation**:
    - Concurrent transactions appear isolated from each other.
    - Issues:
        - *Dirty Read*: Read uncommitted changes.
        - *Non-repeatable Read*: Re-read yields different data due to commit by another transaction.
        - *Phantom Read*: Data appears or disappears due to changes by another transaction.
- **Durability**: Once committed, changes persist even after system failure.

Limitations of Relational Databases:

- Schema rigidity: not suited for evolving schemas.
- Not ideal for semi-structured/unstructured data (e.g., JSON, XML).
- Expensive JOIN operations on large tables.
- Challenges with horizontal scaling.
- Real-time, low-latency systems may need more performant alternatives.

Scaling Strategy:

- Conventional wisdom: scale **vertically** (bigger machines) first.
  - Simpler, no architecture change.
  - But hits physical and financial limits.
- Eventually: scale **horizontally** (distributed systems).
  - More complex, but increasingly supported by modern systems and cloud-native architectures.

Distributed Systems

Definition: A distributed system is "a collection of independent computers that appear to its users as one computer." — Andrew Tannenbaum

Key Characteristics:

- Multiple computers (nodes) operate concurrently
- Each computer can fail independently
- No shared global clock
- Data is distributed across > 1 node and typically replicated (each block is available on multiple nodes)

Distributed Databases:

- Can be relational (e.g., MySQL, PostgreSQL with replication/sharding)
- Or non-relational (e.g., Cassandra, MongoDB)
- Newer systems like CockroachDB aim to combine relational models with distributed guarantees

Resilience to Network Issues:

- Network failures and partitions are inevitable in distributed systems
- Systems must be **Partition Tolerant**: able to continue operation despite network splits

---

CAP Theorem (Brewer's Theorem)

Statement: It is impossible for a distributed system to simultaneously provide all three of:

- **Consistency (C)**: Every read receives the most recent write or an error
- **Availability (A)**: Every request receives a response, without guarantee that it contains the latest write
- **Partition Tolerance (P)**: The system continues to function even if parts of it cannot communicate

What CAP Really Means:

- In the presence of a partition (which *must* be tolerated), you can choose either Consistency or Availability, but not both

CAP Combinations:

Combination	Description
C + A	Always return latest data and respond to all requests – but can't handle partitions reliably
C + P	Ensures data is always consistent – but may refuse requests during partitions (less availability)
A + P	Always responds, even during partitions – but may return stale data

Misinterpretation vs Reality:

- It is *not* saying you must give up one of the three all the time
- Rather: during a **partition**, you must choose between consistency and availability

Real-World Tradeoffs:

- **CP Systems**: HBase, MongoDB (with strong consistency modes)
- **AP Systems**: Cassandra, DynamoDB (eventual consistency for better uptime)
- **CA Systems**: Not achievable in real networks (because network partitions are a fact of life)

NoSQL and Key-Value Databases

Distributed Databases and ACID

Concurrency Models:

Pessimistic Concurrency:

- Prioritizes data safety
- Assumes conflicts are likely; transactions must protect themselves

- Locks data (read and write locks) to prevent concurrent modifications
- **Write Lock Analogy**: Like borrowing a book—no one else can use it until you're done
- Works best for high-conflict environments

Optimistic Concurrency:

- Assumes conflicts are rare
- Does not lock resources during reads/writes
- Relies on timestamps or version numbers
- Conflicts are detected at commit time
- On conflict: roll back and retry transaction
- Ideal for low-conflict, read-heavy systems (e.g., analytics, backups)

NoSQL Databases

Definition:

- Originally coined in 1998 by Carlo Strozzi to describe a non-SQL relational DB
- Now stands for "Not Only SQL"
- Typically refers to non-relational databases
- Developed in response to the need to handle unstructured or semi-structured data at scale

BASE Model (ACID Alternative):

- Designed for distributed systems
- **Basically Available**: System guarantees availability, though responses may indicate temporary failure or inconsistency
- **Soft State**: Data may change over time due to eventual consistency
- **Eventual Consistency**: All nodes will converge to the same value eventually, assuming no new updates

---

Key-Value Stores

Overview:

- Extremely simple data model: Key → Value
- Values can be any type: string, number, JSON, binary object, etc.
- No complex queries, joins, or relationships

Advantages:

- **Simplicity**: Very fast CRUD operations via simple APIs
- **Speed**: Often in-memory, with O(1) retrieval using hash tables
- **Scalability**: Horizontal scaling is straightforward; ideal for distributed environments
- Typically embrace **eventual consistency** for performance and availability

Use Cases:

- **EDA/Experimentation Results Store**: Save intermediate processing steps or A/B test results
- **Feature Store**: Store reusable features for model training/inference with low-latency access
- **Model Monitoring**: Log real-time performance metrics of deployed models
- **Session Storage**: Fast, single GET/PUT access for storing session data
- **User Profiles & Preferences**: Rapidly retrieve user-specific configurations
- **Shopping Cart**: Maintain state across sessions/devices
- **Caching Layer**: Store frequently accessed values to offload disk-based DBs

Redis (Remote Directory Server)

Overview:

- Open-source, in-memory data structure store
- Primary usage: Key-Value storage
- Supports other models: Graph, Time Series, Full Text, Spatial, etc.
- Developed in C++ (2009)

Performance & Persistence:

- Extremely fast (>100,000 SET ops/sec)
- Though in-memory, Redis can persist data via:
  - Snapshots (save to disk at intervals)
  - Append-only files (AOF, journaling system)

Key Structures in Redis:

- **Strings**: Basic values
- **Lists**: Ordered collection (like linked lists)
- **Sets**: Unique unordered elements
- **Sorted Sets**: With ordering based on scores
- **Hashes**: Key → subkey/value

- **Geospatial Data**: Store and query location data

Limitations:

- No secondary indexes
- Lookup is only by key (no search or filtering by value)
- Not designed for handling complex relationships or multi-field filtering

Common Redis Commands & Use Cases:

String Operations

- # Basic SET and GET
- r.set('clickCount:/abc', 0)
- val = r.get('clickCount:/abc')
-
- # Increment the counter
- r.incr('clickCount:/abc')
- ret_val = r.get('clickCount:/abc')
- print(f'click count = {ret_val}')

- `set(key, value)`: Set value for a key
- `get(key)`: Retrieve value
- `incr(key)`: Increment numerical value by 1
- `decr(key)`: Decrement numerical value by 1
- `incrby(key, n)`, `decrby(key, n)`: Increment/decrement by `n`
- `strlen(key)`: Return length of the string
- `append(key, value)`: Append to an existing string

Multi-Key Operations

- # Set multiple keys
- redis_client.mset({'key1': 'val1', 'key2': 'val2', 'key3': 'val3'})
- # Retrieve multiple keys
- print(redis_client.mget('key1', 'key2', 'key3')) # ['val1', 'val2', 'val3']

- `mset()`, `mget()`: Set/get multiple keys at once
- `setnx()`: Set only if key doesn't exist
- `setex()`: Set with expiration time (seconds)
- `msetnx()`: Set multiple keys only if none exist

- `getex()`: Get and optionally update expiration
- `getdel()`: Get and delete key in one step

List Operations

- # Create a list
- redis_client.rpush('names', 'mark', 'sam', 'nick')
- print(redis_client.lrange('names', 0, -1)) # ['mark', 'sam', 'nick']

- `rpush()`, `lpush()`: Push elements to right/left
- `rpop()`, `lpop()`: Remove element from right/left
- `lrange(key, start, stop)`: Get sublist
- `llen(key)`: List length
- `lset(key, index, value)`: Set value at index
- `lrem(key, count, value)`: Remove elements by value
- `lpos(key, value)`: Get index of a value

Hash Operations

- redis_client.hset('user-session:123', mapping={
-     'first': 'Sam',
-     'last': 'Uelle',
-     'company': 'Redis',
-     'age': 30
- })
- print(redis_client.hgetall('user-session:123'))
- # {'first': 'Sam', 'last': 'Uelle', 'company': 'Redis', 'age': '30'}

- `hset()`: Set one or multiple fields
- `hget()`: Get value for a field
- `hgetall()`: Get all fields and values
- `hkeys()`: List of field names
- `hdel()`: Delete fields
- `hexists()`: Check if field exists
- `hlen()`: Number of fields
- `hstrlen()`: Length of a string value in hash

Pipelining

- r = redis.Redis(decode_responses=True)
- pipe = r.pipeline()
-
- # Batch multiple commands
- for i in range(5):
-     pipe.set(f"seat:{i}", f"#{i}")
- set_5_result = pipe.execute()
- print(set_5_result) # [True, True, True, True, True]
-
- pipe = r.pipeline()
- get_3_result = pipe.get("seat:0").get("seat:3").get("seat:4").execute()
- print(get_3_result) # ['#0', '#3', '#4']

- `pipeline()`: Batches multiple commands into a single network call (reduces latency)
- `execute()`: Executes the chained commands

Document Databases

Definition: A **Document Database** is a type of non-relational database that stores data as structured documents, typically in **JSON** (JavaScript Object Notation) format. These databases are designed to be:

- **Simple**: Easy to understand and use
- **Flexible**: Schema-less or dynamic schema
- **Scalable**: Naturally suited for horizontal scaling

Why JSON?

- Lightweight data-interchange format
- Easy for humans to read and write
- Easy for machines to parse and generate
- Supported natively by most modern programming languages

JSON is built on two structures:

1. **Name/Value pairs** (object, dictionary, map, etc.)
2. **Ordered lists of values** (array, list, sequence)

Because of these universal structures, JSON is ideal for transmitting structured data between systems and is a natural fit for document databases.

BSON (Binary JSON)

- Binary-encoded serialization of JSON-like documents
- Extends JSON to include additional data types:
  - Dates, binary data, timestamps, etc.
- **Lightweight**: Minimal overhead
- **Traversable**: Designed for quick access and traversal
- **Efficient**: Fast to encode/decode
- Commonly used in MongoDB as the internal storage format

XML (eXtensible Markup Language) – A Predecessor to JSON

- Structured like HTML but with a custom, extensible tag set
- Used heavily in early web and enterprise systems
- Often paired with:
  - **XPath**: Syntax for locating elements/attributes
  - **XQuery**: Query language for XML (like SQL for relational)
  - **DTD (Document Type Definition)**: Defines legal structure of an XML document
  - **XSLT**: Tool to transform XML into other formats like HTML

While XML is more verbose than JSON, it is still used in enterprise systems that require rigorous data validation and structure.

Why Document Databases?

- **Object-Oriented Programming Compatibility**:
  - OO systems model data via composition and inheritance
  - Mapping complex objects into relational databases often requires flattening/deconstructing objects
  - Document databases avoid this **impedance mismatch** by storing entire object structures as-is
- **Schema Flexibility**:
  - Different documents in the same collection can have different fields
  - Changes to document structure don't require expensive schema migrations
- **Ideal for Modern Web and Mobile Apps**:
  - JSON/XML used widely as transport formats (e.g., REST APIs)
  - Document structure matches application data models closely

Common Use Cases:

- Content management systems
- Catalog data (e.g., e-commerce products)
- User profiles and session info
- Log data and analytics
- Applications with rapidly evolving schema

Popular Document Databases:

- MongoDB
- Couchbase
- Amazon DocumentDB
- Firebase Realtime Database (JSON tree structure)

MongoDB Overview

Origin and Background:

- Started in 2007 by former DoubleClick engineers after observing limitations of relational databases at massive scale (>400,000 ads/sec)
- The name **MongoDB** comes from "Humongous Database"
- **MongoDB Atlas**: Fully managed cloud-based version (DBaaS), launched in 2016

MongoDB Data Model Mapping vs RDBMS:

RDBMS Term	MongoDB Term
Database	Database
Table/View	Collection
Row	Document
Column	Field

Index	Index
Join	Embedded Document
Foreign Key	Reference

Core MongoDB Features:

- **Rich Query Support**: Full CRUD support
- **Indexing**: Primary and secondary indexes on document fields
- **Replication**: Built-in support for replica sets with automatic failover
- **Load Balancing**: Supported natively

MongoDB Editions:

- **MongoDB Atlas**: Cloud-hosted, fully managed version
- **MongoDB Enterprise**: Self-managed, subscription-based version with enterprise tooling
- **MongoDB Community**: Free, source-available, self-managed edition

MongoDB Tools:

- `mongosh`: Command-line interface (CLI) shell for MongoDB
- **MongoDB Compass**: GUI for visualizing and querying MongoDB
- **DataGrip** and other third-party database clients
- **Language Drivers**:
    - `PyMongo` (Python), `Mongoose` (Node.js), `Motor` (Async Python), etc.

MongoDB with PyMongo – Example Assignment

Setup:

import pymongo

from bson.json_util import dumps

Connect to MongoDB (update URI with your credentials)

```
uri = "<your-uri>"

client = pymongo.MongoClient(uri)

mflixdb = client.mflix  # Use the 'mflix' sample database
```

---

1. Find All Theaters in Massachusetts (MA):

```
# Query to return street, city, and zip of all MA theaters

all_theatres = mflixdb.theatres.find(

   {"location.address.state": "MA"},

   {"location.address.city": 1,

    "location.address.street1": 1,

    "location.address.zipcode": 1,

    "_id": 0}

).limit(5)

print(dumps(all_theatres, indent=2))
```

- `find()` retrieves documents matching the query
- Fields are projected using a second dictionary
- `limit()` restricts results

---

2. Count Theaters Per State (Alphabetical by State Code):

```
theatre_sum = mflixdb.theatres.aggregate([

   {"$group": {"_id": {"state": "$location.address.state"}, "theatre count": {"$sum": 1}}},

   {"$sort": {"_id": 1}},

   {"$limit": 10}
```

])

print(dumps(theatre_sum, indent=2))

- `$group` aggregates by state and counts
- `$sort` sorts by state
- `$limit` caps output

---

3. Count All Comedy Movies:

comedy_movies = mflixdb.movies.count_documents({"genres": "Comedy"})

print(f"There are {comedy_movies} comedy movies.")

- `count_documents()` counts matching documents

---

4. Find the Movie with the Longest Runtime:

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

print(dumps(runtime, indent=2))

- Sort by `runtime` descending
- Return only top result

---

5. Post-2010 Movies with Rotten Tomatoes Rating ≥ 3:

viewer_rating = {"$match": {"year": {"$gt": 2010}, "tomatoes.viewer.rating": {"$gte": 3}}}

project = {"$project": {"_id": 0, "title": 1, "viewerrating": "$tomatoes.viewer.rating"}}

sort = {"$sort": {"viewerrating": -1}}

limit = {"$limit": 10}

agg = mflixdb.movies.aggregate([viewer_rating, project, sort, limit])

print(dumps(agg, indent=2))

- Use `$match`, `$project`, `$sort`, and `$limit` in an aggregation pipeline

---

6. Police-Related Movies Count by Year:

match = {"$match": {"plot": {"$regex": "police"}}}

group = {"$group": {"_id": "$year", "movie_count": {"$sum": 1}}}

sort = {"$sort": {"_id": 1}}

limit = {"$limit": 10}

agg = mflixdb.movies.aggregate([match, group, sort, limit])

print(dumps(agg, indent=2))

- Uses regex in `$match` to filter by keyword in `plot`

---

7. Average IMDb Votes Per Year (1970–2000):

match = {"$match": {"year": {"$gte": 1970, "$lte": 2000}}}

group = {"$group": {"_id": "$year", "avg_votes": {"$avg": "$imdb.votes"}}}

sort = {"$sort": {"_id": 1}}

limit = {"$limit": 10}

agg = mflixdb.movies.aggregate([match, group, sort, limit]) print(dumps(agg, indent=2))

---

8. List of Distinct Movie Languages:

match = {"$match": {"languages": {"$exists": True}}}

unwind = {"$unwind": "$languages"}

group = {"$group": {"_id": "$languages"}}

sort = {"$sort": {"_id": 1}}

agg = mflixdb.movies.aggregate([match, unwind, group, sort])

languages = [doc["_id"] for doc in agg]

print(languages)

- `$unwind` flattens arrays so each language becomes a separate document
- `$group` extracts unique values

Absolutely! Here's a **MongoDB Query Cheat Sheet** with explanations and examples using **PyMongo**. This is especially helpful for quickly building and understanding queries for projects, assignments, or exams.

MongoDB Query Cheat Sheet (PyMongo Edition)

> Assumes `db` is your database and `collection` is your target collection:
> `collection = db.my_collection`

🔍 Basic Find Queries

Goal	PyMongo Syntax	Notes

Find all documents	`collection.find()`	Returns a cursor to all docs
Find one document	`collection.find_one()`	Returns the first matched doc
Filter by field	`collection.find({"field": "value"})`	Exact match
Use comparison	`collection.find({"age": {"$gt": 25}})`	`$gt`, `$lt`, `$gte`, `$lte`, `$ne`
Use multiple filters	`collection.find({"age": {"$gt": 25}, "status": "active"})`	AND logic
OR logic	`collection.find({"$or": [{"status": "active"}, {"age": {"$lt": 25}}]})`	Use `$or`
IN match	`collection.find({"status": {"$in": ["active", "pending"]}})`	Match any of list

Projections (Choosing fields)

collection.find({}, {"name": 1, "age": 1, "_id": 0})

- `1` = include field, `0` = exclude field
- Always explicitly exclude `_id` if not needed

Aggregations (Advanced Queries)

MongoDB uses an aggregation pipeline: `collection.aggregate([...])`

Example: Count documents per category

```
pipeline = [
    {"$group": {"_id": "$category", "count": {"$sum": 1}}},
    {"$sort": {"count": -1}}
]
collection.aggregate(pipeline)
```

Example: Filter, then project fields

```
pipeline = [
    {"$match": {"year": {"$gt": 2010}}},
    {"$project": {"title": 1, "rating": "$ratings.viewer", "_id": 0}}
]
collection.aggregate(pipeline)
```

Common Aggregation Stages

Stage	Purpose
`$match`	Filter documents
`$project`	Choose/rename fields
`$group`	Aggregate by `_id`, use `$sum`, `$avg`, etc.

`$sort`	Sort results
`$limit` / `$skip`	Pagination
`$unwind`	Flatten array fields

Update Operations

Task	Syntax
Update one field	`collection.update_one({"name": "John"}, {"$set": {"age": 30}})`
Update multiple	`collection.update_many({"status": "active"}, {"$set": {"verified": True}})`
Increment a value	`{"$inc": {"counter": 1}}`
Add to set (no dupes)	`{"$addToSet": {"tags": "new"}}`
Push to array	`{"$push": {"tags": "new"}}`

Delete Operations

Task	Syntax
Delete one	`collection.delete_one({"name": "John"})`
Delete many	`collection.delete_many({"status": "inactive"})`

Count Documents

collection.count_documents({"status": "active"})

Indexes

collection.create_index("field")

collection.create_index([("field1", pymongo.ASCENDING), ("field2", pymongo.DESCENDING)])

Text Search

Ensure text index first:

collection.create_index([("title", "text")])

collection.find({"$text": {"$search": "adventure"}})

Regex Search

collection.find({"title": {"$regex": ".*police.*", "$options": "i"}})

Example Setup with PyMongo

import pymongo

from bson.json_util import dumps

```
client = pymongo.MongoClient("<your_uri>")

db = client["mflix"]

collection = db["movies"]


# Find movies after 2010 with rating >= 3

pipeline = [

   {"$match": {"year": {"$gt": 2010}, "tomatoes.viewer.rating": {"$gte": 3}}},

   {"$project": {"title": 1, "viewer_rating": "$tomatoes.viewer.rating", "_id": 0}},

   {"$sort": {"viewer_rating": -1}},

   {"$limit": 10}

]

results = collection.aggregate(pipeline)

print(dumps(results, indent=2))
```

Graph Databases

- Based on the **graph data structure**: composed of **nodes (vertices)** and **edges (relationships)**
- Both nodes and edges can store **properties** as key-value pairs
- Great for modeling **highly interconnected data**: e.g., social networks, knowledge graphs, recommendation engines

Labeled Property Graph Model

- **Nodes**: entities (e.g. Person, Product, Location)
- **Edges**: relationships (e.g. FRIEND_OF, PURCHASED, LOCATED_IN)
- **Labels**: used to group nodes (like "User", "Product", etc.)
- **Properties**: stored on both nodes and edges
- Rules:
  - Nodes **can exist without edges**
  - Edges **must connect two nodes**

Graph Features & Concepts

- **Path**: sequence of connected nodes (no repeats)
- **Connected vs Disconnected**: every node reachable from any other?
- **Directed vs Undirected**: edges have a direction (start → end)
- **Weighted vs Unweighted**: edges carry a cost/weight
- **Acyclic vs Cyclic**: cycles allowed or not?

Types of Graph Queries & Algorithms

- **Pathfinding**:
    - *Find shortest path*: by hops or edge weights
    - *Minimum Spanning Tree*, *Cycle Detection*, *Flow Algorithms*
- **Centrality**:
    - Find influential or "important" nodes
    - Used in social graphs (e.g., influencer detection)
- **Community Detection**:
    - Identify clusters/groups within the network
    - Helps understand structure or hidden groupings

Famous Graph Algorithms

Algorithm	Purpose
Dijkstra’s	Shortest path (positive weights)
A*	Shortest path with heuristic
PageRank	Node importance (used by Google)

Neo4j and Other Graph DBs

Neo4j:

- Leading graph database system
- Schema-optional (schema can be used, but not required)
- Uses **Cypher** query language
- **ACID compliant**, supports **distributed computing**

- Indexing support for faster lookups
- Great for both **OLTP** (transactions) and **OLAP** (analysis) on graph data

Other Graph DBs:

- **Amazon Neptune**
- **Microsoft CosmosDB**
- **TigerGraph**, **ArangoDB**, **OrientDB**

Docker Compose

Docker Compose is a tool used to define and manage **multi-container Docker applications** using a declarative YAML file. It's great for creating reproducible development and deployment environments.

`docker-compose.yml`

- **Declarative** configuration for services, volumes, networks
- Defines how multiple containers interact
- Can be version-controlled for consistency across teams/environments

`.env` Files

- Store **environment-specific variables**
- Useful for switching between **dev**, **test**, and **prod** environments

Common Docker Compose Commands

Command	Description
`docker --version`	Shows Docker CLI version
`docker compose up`	Builds, creates, and runs containers
`docker compose up -d`	Same as above, but runs in background (detached)

`docker compose down`	Stops and removes containers, networks, volumes
`docker compose start`	Starts containers that were previously stopped
`docker compose stop`	Stops running containers (but doesn’t remove them)
`docker compose build`	Builds images defined in the compose file
`docker compose build --no-cache`	Forces full rebuild without using cache