1️⃣ Searching

- Searching is the most common operation performed in a database system.
- The `SELECT` statement in SQL is arguably the most versatile and complex.
- **Linear Search (Baseline for Efficiency)**:
  - Start at the beginning of a list and proceed element by element until:
    - The target is found.
    - The last element is reached without finding the target.
  - **Time Complexity**:
    - Best case: The target is found at the first element → **O(1)**.
    - Worst case: The target is not in the array, requiring **O(n)** comparisons.

2️⃣ Basic Database Terminology

- **Record**: A collection of values for attributes of a single entity instance (a row in a table).
- **Collection**: A set of records of the same entity type (a table), typically stored in sequential order.
- **Search Key**: A value for an attribute that can be used for searching. It can consist of one or more attributes.

3️⃣ Lists of Records

- If each record takes `x` bytes, then for `n` records, memory required is `n × x` bytes.
- **Data structures:**
  - **Contiguously Allocated List**: A single chunk of memory is allocated for all records.
  - **Linked List**: Each record has additional space for one or two memory addresses and is linked together.

4️⃣ Contiguous vs. Linked List - Pros & Cons

- **Arrays:**
  - ✅ Fast for **random access**.
  - ❌ Slow for **inserting** elements anywhere except the end.
- **Linked Lists:**
  - ✅ Fast for **inserting** anywhere.
  - ❌ Slow for **random access**.

5️⃣ Binary Search

- **Definition**: A divide-and-conquer algorithm that searches for a target in a **sorted array**.

- - Input: A sorted array of values and a target value.
- - Output: The index of the target or an indication that it was not found.

Algorithm: `python`
CopyEdit

```
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**:
  - Best case: Target is found at the middle index → **O(1)**.
  - Worst case: Requires **O($\log_2$ n)** comparisons.

6️ Database Searching and Indexing

- **Assume data is stored on disk by a specific column (e.g., `id`).**
  - Searching for `id` is fast because it follows the storage order.
  - Searching for another attribute (e.g., `specialVal`) requires a **linear scan** unless indexed.
- **Storage Constraints**:
  - **Cannot store data sorted by multiple attributes simultaneously** (e.g., sorted by `id` and `specialVal`).
  - **Data duplication** is required to store multiple orderings, which is space inefficient.
- **Indexing Strategies**:
  - **Array of Tuples (`specialVal`, `rowNumber`) Sorted by `specialVal`**:
    - ✅ **Supports fast binary search** to locate `specialVal` and find the corresponding row.

        - ❌ **Slow inserts**, since inserting a new row requires maintaining the sorted order.
    - **Linked List of Tuples (`specialVal`, `rowNumber`) Sorted by `specialVal`**:
        - ✅ **Fast insertions** (appending to the list).
        - ❌ **Slow searches**, as it requires a **linear scan**.

7️⃣ Optimized Data Structures for Fast Search and Insert

- **Binary Search Tree (BST)**:
    - A tree where each node satisfies:
        - Left subtree contains values **less than** the node.
        - Right subtree contains values **greater than** the node.
    - **Provides fast insertions and searches** compared to sorted arrays or linked lists.

8️⃣ Summary of Search Complexity in Different Structures

Data Structure	Search Complexity	Insert Complexity	Notes
Unsorted Array	**O(n)**	**O(1)**	Fast inserts, slow searches
Sorted Array	**O(log n)** (Binary Search)	**O(n)**	Fast searches, slow inserts
Unsorted Linked List	**O(n)**	**O(1)**	Fast inserts, slow searches
Sorted Linked List	**O(n)**	**O(n)**	Both slow
Binary Search Tree	**O(log n)**	**O(log n)**	Fast for both (on average)

---

1. Definition of BST
A Binary Search Tree (BST) is a binary tree that satisfies the BST-property:
- For all nodes x and y:
  - If y is in the left subtree of x, then key(y) < key(x).
  - If y is in the right subtree of x, then key(y) > key(x).
- The keys in a BST are assumed to be pairwise distinct.
- Each node has the following attributes:
  - p (pointer to the parent node)
  - left (pointer to the left child)
  - right (pointer to the right child)

- key (stores the key value of the node)

2. BST Traversal
Traversal refers to visiting all nodes in a specific order. The three traversal strategies are:

1. Inorder Traversal (Left → Root → Right)
  - Outputs keys in non-decreasing order.
  - Pseudocode:
```
def inorder_walk(x):
    if x is None:
        return
    inorder_walk(x.left)
    print(x.key)
    inorder_walk(x.right)
```

2. Preorder Traversal (Root → Left → Right)
3. Postorder Traversal (Left → Right → Root)

3. BST Operations
1. Searching for a Key
- Given a key k, search in the subtree rooted at node x:
  - If key[x] == k, return the node.
  - If key[x] < k, search in the right subtree.
  - If key[x] > k, search in the left subtree.
- Algorithm:
```
def bst_search(x, k):
    while x is not None:
        if x.key == k:
            return x
        elif x.key < k:
            x = x.right
        else:
            x = x.left
    return "NOT FOUND"
```

2. Finding Minimum and Maximum
- Minimum: Leftmost node (follow left pointers).
- Maximum: Rightmost node (follow right pointers).
- Algorithms:
```
def bst_minimum(x):
    while x.left is not None:
        x = x.left
    return x.key

def bst_maximum(x):
    while x.right is not None:
```

```
        x = x.right
    return x.key
```

3. Insertion
- Insert a node z with key k by finding the correct position.
- Algorithm:

```
  def bst_insert(root, z):
      y = None
      x = root
      while x is not None:
          y = x
          if z.key < x.key:
              x = x.left
          else:
              x = x.right
      if y is None:
          root = z
      elif z.key < y.key:
          y.left = z
      else:
          y.right = z
```

4. Finding Successor and Predecessor
- Successor: Smallest key greater than k.
  - If x has a right subtree → Find minimum in the right subtree.
  - Otherwise → Traverse up until x is in a left subtree.
- Algorithm:

```
  def bst_successor(x):
      if x.right is not None:
          return bst_minimum(x.right)
      y = x.parent
      while y is not None and x == y.right:
          x = y
          y = y.parent
      return y
```

5. Deletion
- Three cases:
  1. Node has no children → Replace z with nil.
  2. Node has one child → Promote its child.
  3. Node has two children → Replace z with its successor.
- Algorithm:

```
  def bst_delete(root, z):
      if z.left is None:
          transplant(root, z, z.right)
      elif z.right is None:
```

```
        transplant(root, z, z.left)
    else:
        y = bst_minimum(z.right)
        if y.parent != z:
            transplant(root, y, y.right)
            y.right = z.right
            y.right.parent = y
        transplant(root, z, y)
        y.left = z.left
        y.left.parent = y
```

4. Efficiency Analysis of BST Operations
- Theorem: For a BST of height h, the following operations run in O(h) time:
  - Search, Minimum, Maximum, Successor, Predecessor, Insert, Delete.
- Average Case: If the BST is balanced, h = O(log n).
- Worst Case: If the BST is a linked list (unbalanced), h = O(n).

5. Height of a Randomly Built BST
- If we insert n distinct keys into an initially empty tree, assuming all n! permutations are equally likely:
  - Average Height: O(log n).
- Mathematical Analysis:
  - If a key x is selected randomly:
    - All keys greater than x go into the right subtree.
    - All keys smaller than x go into the left subtree.
    - Height h(n) is given by:
      h(n) = 1 + max(h(L), h(R))
  - Using Jensen's inequality, it is shown that:
    E[h(n)] = O(log n)

6. Summary
- BST Definition
- Traversal Types (Inorder, Preorder, Postorder)
- Operations
  - Search (O(h))
  - Min/Max (O(h))
  - Insertion (O(h))
  - Successor/Predecessor (O(h))
  - Deletion (O(h))
- Efficiency Analysis
  - Average Case: O(log n)
  - Worst Case: O(n)
- Random BST Height: O(log n)

Extended Knowledge on AVL Trees

1️⃣ Importance of Binary Search Tree (BST) Balancing
- The performance of BSTs is highly dependent on their **shape**, which determines their height.
- **Perfect BST:** Minimizes height, ensuring optimal performance.
- **Degenerate BST:** Maximizes height, effectively becoming a linked list.
- In the worst case, inserting keys in sorted order results in **O(n) lookup time**.

2️⃣ Problems with Degenerate BSTs
- **Time Complexity:** Lookups require **O(n)** time in the worst case.
- **Recursive Lookups:** May consume **O(n) stack memory**, leading to inefficient recursive depth.
- **Tree Construction Cost:** Sequentially inserting **n** keys results in **$\Theta(n^2)$ operations**.
- **Solution:** Maintain **balance** to ensure logarithmic height.

3️⃣ Perfect vs. Complete Binary Trees
- **Perfect Binary Tree:** Every level is **fully filled**.
- **Complete Binary Tree:**
  - All levels except the last are fully occupied.
  - The last level is filled from left to right.
- **Complete BSTs ensure $\Theta(\log n)$ height**, keeping operations efficient.

4️⃣ Challenges in Maintaining Balance
- **Perfect BSTs** cannot be maintained for arbitrary **n**.
- **Complete BSTs** require **O(n) operations** in the worst case to maintain, making them impractical.
- **A compromise is needed**, balancing efficiency and structure.

5️⃣ Defining a "Good" Balance Condition
- **Goals:**
  1. Ensure BST height remains **$\Theta(\log n)$**.
  2. Keep rebalancing operations at **O(log n)** to maintain efficiency.
- **Tradeoff:** Perfect balance is too costly, so we adopt **"nearly balanced"** solutions.

6️⃣ AVL Trees: A Compromise for Balance
- **AVL Property:** Each node maintains a height difference of at most **1** between its left and right subtrees.
- **Key Features:**
  - Ensures **logarithmic lookup, insertion, and deletion** time.
  - Requires rebalancing after insertions and deletions.

7️⃣ AVL Tree Rotations (Rebalancing)
- **Rotations correct imbalances caused by insertions and deletions.**
- **Types of Rotations:**
  1. **LL Rotation (Left-Left):** Right rotation to fix left-heavy imbalance.
  2. **RR Rotation (Right-Right):** Left rotation to fix right-heavy imbalance.
  3. **LR Rotation (Left-Right):** Left rotation followed by right rotation.
  4. **RL Rotation (Right-Left):** Right rotation followed by left rotation.

8️⃣ AVL Tree Insertion
- **Steps for insertion:**
  1. Perform **regular BST insertion**.
  2. Traverse back up to detect imbalances.
  3. Apply **appropriate rotation** (LL, RR, LR, or RL).
- **Example:** If inserting a node causes an imbalance at a parent node:
  - Identify the imbalance type (LL, RR, LR, RL).
  - Apply the appropriate rotation to restore balance.

9️⃣ AVL Tree Deletion
- **Steps for deletion:**
  1. Perform **regular BST deletion**.
  2. Traverse back up to detect imbalances.
  3. **Apply rotations to restore balance** (may require multiple rotations).
  4. Each rotation takes **O(1) time**, but worst-case scenario involves **O(log n) rotations**.

🔟 AVL Tree Height Analysis
- **Key Question:** What is the height of an AVL tree with **n** nodes?
- **Derivation:**
  - The **minimum number of nodes** in an AVL tree of height **h** follows:
    ```
    M(h) = 1 + M(h - 1) + M(h - 2)
    ```
  - This recurrence leads to:
    ```
    M(h) ≥ 2^(h/2)  →  h ≤ 2 log₂(n)
    ```
  - **Conclusion:** AVL tree height is **Θ(log n)**.

📌 Additional Insights from PDFs

Key Points on BST Balancing from ICS 46 Spring 2022 Notes
- **BST Performance Depends on Shape**: The difference between a perfect tree and a degenerate tree is crucial.
- **Worst Case BST Performance**: Sequentially inserting keys leads to O(n) lookup time.

- **Maintaining Perfection is Impossible**: Not all values of **n** allow for a perfect binary tree.
- **Complete Trees as an Alternative**: Complete BSTs maintain **Θ(log n)** height but are impractical due to the **Ω(n) maintenance cost**.
- **AVL Trees Provide a Solution**: By maintaining a **balance factor** of at most 1, AVL trees achieve a **nearly optimal height** with efficient operations.

Insights on AVL Tree Rotations from AVL-Tree-Rotations.pdf
- **Rotations are Critical for Maintaining Balance**: The four types of rotations (LL, RR, LR, RL) ensure AVL trees remain balanced.
- **Constant-Time Rotations**: Each rotation involves only **O(1) pointer changes**, making them efficient.
- **Rotations Preserve BST Order**: Subtrees are rearranged but the **BST property remains intact**.
- **Rotations Are Required During Insertions and Deletions**: If a node's balance factor exceeds **±1**, a **rotation is triggered**.

1️⃣ Introduction to B-Trees
- **B-Trees** were introduced by **R. Bayer and E. McCreight** in 1972 and became the standard for large-file access.
- Used extensively in **file systems** and **databases** for efficient insertion, deletion, and range queries.
- **Advantages of B-Trees:**
  - Always **height-balanced**, keeping **all leaf nodes at the same level**.
  - **High branching factor** reduces tree height, minimizing **disk I/O**.
  - Keeps **related records together** to speed up **range queries**.
  - Guarantees that nodes remain at least **partially full**, optimizing space usage.

2️⃣ Structure of a B-Tree (Order m)
- The **root** is either a **leaf** or has at least **two children**.
- **Internal nodes (except root)** have between **⌈m/2⌉ and m children**.
- **All leaves are at the same level**, ensuring height balance.
- **2-3 Trees** are a special case of **B-Trees with order 3**.
- **Typical B-Tree nodes hold 100+ children**, making them highly efficient for disk storage.

3️⃣ B-Tree Search Algorithm
- **Process:**
  1. Perform a **binary search** within the current node.
  2. If the key is found, return the record.
  3. If not, follow the appropriate child pointer and repeat the process.
  4. If the key is not found in a leaf, return **"not found"**.

- **Example:** Searching for key `47`:
  - Check the **root** node.
  - Follow the correct **branch** based on key comparison.
  - Continue the search at the leaf node where **47 is stored**.

4️⃣ B-Tree Insertion Algorithm
- **Steps:**
  1. Locate the appropriate **leaf node**.
  2. If there is room, insert the key directly.
  3. If the node is **full**, split it:
     - Promote the **middle key** to the parent node.
     - Distribute remaining keys evenly between **two new nodes**.
  4. If the parent node **overflows**, recursively apply the **split operation** up to the root.

- **Result:** The B-Tree remains balanced, with all leaves at the same depth.

5️⃣ B+ Trees: An Optimized B-Tree Variant
- **B+ Trees** are an improved version of **B-Trees**, commonly used in **databases and file systems**.
- **Key Differences from B-Trees:**
  - **Internal nodes store only keys** (used for navigation, not actual data).
  - **All actual records are stored at leaf nodes**.
  - **Leaf nodes are linked in a doubly linked list**, enabling efficient **range queries**.

6️⃣ B+ Tree Search Algorithm
- **Steps:**
  1. Start at the **root** and perform **binary search** within the node.
  2. Follow the **correct pointer** to the next level.
  3. Continue until reaching the **leaf node**.
  4. If the key is found in the **leaf node**, return the record.

- **Key Difference:** Even if a key is found in an internal node, the search **must continue** to the leaf level.

7️⃣ B+ Tree Insertion Algorithm
- **Steps:**
  1. Find the correct **leaf node** for insertion.
  2. If there is space, insert the key.
  3. If the node is **full**, split it into **two leaf nodes** and **promote** the middle key.
  4. If the parent node becomes full, **split it recursively**, potentially growing the tree height.

- **Effect:** The tree remains balanced, and **all leaf nodes stay at the same depth**.

8️⃣ B+ Tree Deletion Algorithm

- **Steps:**
  1. Locate the **leaf node** containing the key.
  2. If the node is more than **half full**, simply remove the key.
  3. If the node **underflows**, attempt to **borrow a key** from a sibling.
  4. If borrowing is not possible, **merge the underflowing node** with a sibling.
  5. If the parent node underflows, apply **recursive merging** up to the root.

- **Effect:** The B+ Tree maintains **50% minimum storage utilization** while preserving balance.

9️⃣ B+ Tree vs. B-Trees: Advantages
- **B+ Trees excel in range queries** due to their linked leaf nodes.
- **More efficient disk access:** Internal nodes are smaller and contain only keys, improving **search performance**.
- **Guaranteed logarithmic operations:** Search, insertion, and deletion all run in **$\Theta(\log n)$ time**.

🔟 B-Tree and B+ Tree Height Analysis
- **Key Question:** What is the height of a B+ Tree with **n records**?
- **Analysis:**
  - If the tree has **order m**, then the height is approximately **$O(\log_{\mathbf{m}} n)$**.
  - **Typical database trees have $m \approx 100$**, making the tree extremely **shallow** (height rarely exceeds 4 or 5).
  - The tree maintains an **average fill factor of ~75%**, optimizing space usage.

📌 Conclusion:
- **B-Trees and B+ Trees** are the gold standard for **disk-based indexing**.
- **Shallow depth and high branching factors** minimize disk I/O.
- **Efficient for search, insert, delete, and range queries**, making them ideal for **large-scale storage systems**.

1️⃣ Why Use B-Trees?
- **Traditional BSTs have poor cache locality** because nodes are stored separately, often leading to inefficient memory access.
- **B-Trees improve locality** by storing multiple elements in each node, reducing cache misses and disk I/O.
- Originally designed for **disk storage**, but also useful for **in-memory data structures** due to increasing memory latency.

2️⃣ Structure of a B-Tree (Order m)
- A **B-tree of order m** is a search tree where each non-leaf node has up to **m children**.
- **Properties:**
  1. **All paths from root to leaves have the same length** (height-balanced).

2. If a node has **n children**, it contains **n-1 keys**.
3. **All nodes (except root) are at least half full**.
4. Keys in a parent node **separate** values in child nodes.
5. **Root has at least two children**, unless it is a leaf.

- **Example:** An order-5 B-tree (m=5) where leaves store up to 3 records.

3️⃣ B-Tree Lookup Algorithm
- **Steps:**
1. Start at the **root node**.
2. Perform **binary search** on keys in the node.
3. If the key is **found**, return the record.
4. If not found, follow the **appropriate child pointer** and repeat the process.
5. If reaching a **leaf node without finding the key**, return **"not found"**.

- **Efficiency:** Lookups require **$O(\log_m n)$** operations due to the high branching factor.

4️⃣ B-Tree Insertion Algorithm
- **Steps:**
1. Find the correct **leaf node** for insertion.
2. If space is available, insert the key.
3. If the node is **full**, split it:
   - Divide elements **evenly** into two new nodes.
   - Promote the **middle key** to the parent node.
4. If the parent node **overflows**, recursively **split upwards**.
5. If the root splits, create a **new root**, increasing the tree height.

- **Effect:** The B-tree remains balanced, and height grows **slowly**.

5️⃣ B-Tree Deletion Algorithm
- **Steps:**
1. Locate the **leaf node** containing the key.
2. Remove the key from the leaf.
3. If the leaf node has **enough keys**, no further action is needed.
4. If the node **underflows** (falls below the minimum keys required):
   - **Borrow** a key from a sibling if possible.
   - If borrowing is not possible, **merge** the node with a sibling.
5. If merging causes the parent to underflow, repeat the process **up the tree**.
6. If the root has only one child, remove the root and make the child the new root, **reducing tree height**.

- **Effect:** The B-tree remains balanced, and height **may decrease**.

6️⃣ Why B-Trees are Efficient

- **Height remains low:** B-Trees are **logarithmic** in height, typically requiring **O($\log_m$ n) operations**.
- **Optimized for cache & disk:** High **branching factor** reduces the number of memory accesses.
- **Used in databases & file systems** due to their **fast insert, delete, and search operations**.

📌 Conclusion:
- B-Trees **maintain balance automatically**, ensuring efficient operations.
- **Height remains logarithmic**, preventing worst-case O(n) performance.
- **Used in disk storage and in-memory indexing** for optimal efficiency.

———

Relational Databases and Beyond: A Comprehensive Overview

1️⃣ Benefits of the Relational Model
- Standardized Data Model and Query Language (SQL)
- ACID Compliance:
  - Atomicity: Transactions are fully executed or not at all.
  - Consistency: Transactions transition the database from one valid state to another.
  - Isolation: Transactions do not interfere with each other.
  - Durability: Once committed, transactions remain in the database.
- Works well with highly structured data.
- Capable of handling large amounts of data.
- Mature ecosystem with extensive tooling and expertise.

2️⃣ Relational Database Performance Enhancements
- Indexing (primary focus of DS4300).
- Direct control over storage management.
- Row-oriented vs. column-oriented storage.
- Query optimization techniques.
- Caching and prefetching mechanisms.
- Materialized views.
- Precompiled stored procedures.
- Data replication and partitioning strategies.

3️⃣ Transaction Processing in Relational Databases
- A transaction consists of multiple CRUD operations executed as a single logical unit.
- Transaction outcomes:
  - Commit: All operations succeed, making changes permanent.
  - Rollback (Abort): The entire sequence is undone if any part fails.
- Ensures:
  - Data integrity.
  - Error recovery.
  - Concurrency control.

- Reliable storage.
- Simplified error handling.

4️⃣ ACID Properties Explained

- Atomicity: Transactions are all or nothing.
- Consistency: Transactions maintain the integrity constraints of the database.
- Isolation: Concurrent transactions do not interfere with each other.
  - Dirty Read: A transaction reads uncommitted data.
  - Non-Repeatable Read: A repeated read in the same transaction returns different results.
  - Phantom Reads: Another transaction inserts/deletes rows while a transaction is running.
- Durability: Once committed, data changes persist even after failures.

5️⃣ Example: SQL Transaction for Money Transfer

```
DELIMITER //

CREATE PROCEDURE transfer(
    IN sender_id INT,
    IN receiver_id INT,
    IN amount DECIMAL(10,2)
)
BEGIN
    DECLARE rollback_message VARCHAR(255) DEFAULT 'Transaction rolled back:
Insufficient funds';
    DECLARE commit_message VARCHAR(255) DEFAULT 'Transaction committed
successfully';

    START TRANSACTION;

    UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;
    UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;

    IF (SELECT balance FROM accounts WHERE account_id = sender_id) < 0 THEN
        ROLLBACK;
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = rollback_message;
    ELSE
        INSERT INTO transactions (account_id, amount, transaction_type) VALUES (sender_id,
-amount, 'WITHDRAWAL');
        INSERT INTO transactions (account_id, amount, transaction_type) VALUES (receiver_id,
amount, 'DEPOSIT');
        COMMIT;
        SELECT commit_message AS 'Result';
    END IF;
END //
DELIMITER ;
```

6️⃣ Challenges with Relational Databases

- Schema evolution: Difficult to handle changing data structures.
- Expensive joins: Large-scale joins degrade performance.
- Semi-structured & unstructured data (JSON, XML).
- Scalability: Horizontal scaling is difficult for relational databases.
- Not ideal for real-time, low-latency applications.

7️⃣ Scaling Relational Databases

- Vertical Scaling (Scaling Up): Add more CPU, RAM, and storage to a single machine.
- Horizontal Scaling (Scaling Out): Distribute the database across multiple machines.
- Distributed computing solutions improve scalability without extreme complexity.

8️⃣ Introduction to Distributed Databases

- Definition: A collection of independent computers that work together as one system.
- Characteristics:
  - Multiple nodes operate concurrently.
  - Nodes fail independently.
  - No shared global clock.

9️⃣ Distributed Data Storage Models

- Distributed databases store data across multiple nodes.
- Data replication: Copies of data exist on multiple nodes for fault tolerance.
- Sharding: Splitting large datasets across multiple nodes.
- Examples:
  - Relational: MySQL, PostgreSQL (support sharding and replication).
  - Newer relational systems: CockroachDB.
  - NoSQL: MongoDB, Cassandra, DynamoDB.
- Challenges in distributed databases:
  - Network failures are inevitable.
  - Systems must be partition-tolerant (i.e., maintain functionality even when network segments fail).

🔟 CAP Theorem (Consistency, Availability, Partition Tolerance)

- It is impossible for a distributed system to provide all three guarantees simultaneously.
  - Consistency (C): Every read gets the latest write or an error.
  - Availability (A): Every request gets a response (but not necessarily the latest data).
  - Partition Tolerance (P): The system continues operating despite network failures.
- Trade-offs in CAP:
  - CA (Consistency + Availability): Strong consistency, but fails under network partitions.
  - CP (Consistency + Partition Tolerance): Always consistent, but might sacrifice availability.
  - AP (Availability + Partition Tolerance): Always available, but may return outdated data.

📌 Conclusion

- Relational databases are powerful but not always ideal.
- Distributed systems trade off CAP properties depending on use case.
- Non-relational databases address limitations of SQL-based systems.
- Scaling decisions (vertical vs. horizontal) depend on requirements and constraints.

———————————————————————————————————————

Data Replication and Consistency in Distributed Systems

1️⃣ Why Replicate Data?
- Scalability & High Throughput: Handles growing data volume and read/write load.
- Fault Tolerance & High Availability: Ensures applications continue working even if machines fail.
- Latency Reduction: Improves response time for geographically distributed users.

2️⃣ Challenges of Distributed Data
- Consistency: Updates must be properly synchronized across replicas.
- Application Complexity: Applications must manage distributed reads/writes.

3️⃣ Scaling Architectures
- Vertical Scaling (Shared Memory): Single powerful machine with hot-swappable components.
- Vertical Scaling (Shared Disk): Multiple machines share a common storage, limited by write contention.
- Horizontal Scaling (Shared Nothing): Each node has independent resources, enabling true distributed systems.

4️⃣ Replication vs. Partitioning
- Replication: Copies of the same data exist on multiple machines.
- Partitioning: Divides data across multiple machines (each node holds only a subset).

5️⃣ Common Replication Strategies
- Single Leader Model:
  - All writes go to the leader node.
  - Leader sends updates to followers.
  - Reads can be served by both leader and followers.
- Multiple Leader Model:
  - Multiple leaders handle writes, requiring conflict resolution.
  - More complex, used in multi-region systems.
- Leaderless Model:
  - No single leader; all nodes accept writes.
  - Requires quorum-based consistency mechanisms.

6️⃣ Leader-Based Replication (Most Common)
- Relational Databases: MySQL, PostgreSQL, Oracle, SQL Server.

- NoSQL Databases: MongoDB, RethinkDB, LinkedIn's Espresso.
- Messaging Systems: Kafka, RabbitMQ.

7️⃣ How Replication Data is Transmitted
- Statement-Based Replication: Sends SQL statements; prone to inconsistencies.
- Write-Ahead Log (WAL) Replication: Logs all changes at byte-level; requires same storage engine.
- Logical (Row-Based) Replication: Captures row-level changes; decouples from storage engine.
- Trigger-Based Replication: Uses database triggers to log changes.

8️⃣ Synchronous vs. Asynchronous Replication
- Synchronous: Leader waits for followers to confirm before committing.
  - ✅ Ensures strong consistency.
  - ❌ Slower performance due to waiting.
- Asynchronous: Leader commits instantly without waiting for followers.
  - ✅ Higher availability & performance.
  - ❌ Risk of temporary inconsistencies.

9️⃣ Handling Leader Failures
- New Leader Selection:
  - Nodes vote based on the most up-to-date data.
  - A controller node may appoint a new leader.
- Challenges:
  - Lost writes if replication was asynchronous.
  - Avoiding split-brain scenarios (multiple active leaders).

🔟 Replication Lag
- The delay between leader updates and follower synchronization.
- Synchronous Replication: Eliminates lag but slows down writes.
- Asynchronous Replication: Increases availability but results in eventual consistency.

📌 Read-After-Write Consistency
- Scenario: A user adds a comment on a post and expects to see it immediately.
- Solutions:
  - Read from leader for recently updated data.
  - Dynamically route requests to leader within a time window.

📌 Monotonic Read Consistency
- Ensures users do not see older data after reading newer data.
- Prevents read anomalies across distributed followers.

📌 Consistent Prefix Reads
- Ensures sequential writes appear in order for all readers.
- Avoids scenarios where data appears out of order due to replication delays.

📌 Conclusion:
- Replication enhances scalability, availability, and fault tolerance.
- Trade-offs exist between consistency and performance.
- Strategies like read-after-write consistency mitigate common issues.

———

Distributed Databases, NoSQL, and Redis

1️⃣ Distributed Databases and ACID - Pessimistic Concurrency
- ACID Transactions: Focus on data safety, ensuring consistency and integrity.
- Pessimistic Concurrency: Assumes conflicts are likely, locks resources until a transaction completes.
  - Example: Write Lock Analogy – borrowing a book from the library prevents others from accessing it.

2️⃣ Optimistic Concurrency
- Transactions do not lock data but instead check for conflicts before committing.
- Works well for low-conflict systems (e.g., backups, analytical databases).
- Implementation: Use timestamps & version numbers to detect conflicts.
- High-conflict systems may prefer a pessimistic locking model.

3️⃣ Introduction to NoSQL
- NoSQL (Not Only SQL) first used in 1998 by Carlo Strozzi.
- Initially developed for handling unstructured, web-based data.
- Often used to describe non-relational databases.

4️⃣ CAP Theorem Review
- A distributed system can only provide two of three guarantees:
  - Consistency (C): All users see the same data instantly.
  - Availability (A): System remains operational despite failures.
  - Partition Tolerance (P): The system works despite network failures.

5️⃣ BASE Model (ACID Alternative)
- Basically Available: Guarantees availability, but responses may be unreliable.
- Soft State: System state may change over time even without input.
- Eventual Consistency: Eventually, all nodes will agree on data values.

6️⃣ Categories of NoSQL Databases

- Key-Value Stores (KV)
- Document Stores
- Column-Family Stores
- Graph Databases

7️⃣ Key-Value Databases (KV Stores)
- Data Model: Simple key = value pairs.
- Optimized for:
  - Speed: Uses in-memory hash tables, typically O(1) lookup.
  - Scalability: Horizontally scalable, supports distributed environments.
  - Simplicity: No complex queries or joins.

8️⃣ Use Cases of KV Stores
- Data Science & ML:
  - Experimentation & EDA Store: Saves intermediate results.
  - Feature Store: Low-latency retrieval for ML training & inference.
  - Model Monitoring: Stores model performance metrics.

- Software Engineering (SWE):
  - Session Management: Fast storage/retrieval of user sessions.
  - User Profiles & Preferences: Quick retrieval of user data.
  - Shopping Cart Data: Persistent across devices and sessions.
  - Caching Layer: Speeds up queries by reducing DB load.

9️⃣ Redis - A Popular KV Database
- Redis (Remote Dictionary Server): Open-source in-memory KV store.
- Supports: Graphs, Spatial, Full-Text Search, Vectors, Time Series.
- Durability Mechanisms:
  - Snapshot-based persistence.
  - Append-only file (AOF) journaling.

🔟 Redis Data Types
- Keys: Strings (can be any binary sequence).
- Values:
  - Strings
  - Lists (Linked Lists)
  - Sets (Unique, Unsorted Elements)
  - Sorted Sets
  - Hashes (Field → Value Pairs)
  - Geospatial Data

📌 Setting Up Redis in Docker
- Pull & run the Redis image.
- Expose port 6379 for connection.

- Security Note: Redis should not be exposed in production.

📌 Connecting to Redis via DataGrip
- File > New > Data Source > Redis
- Enter connection details & test connection.

📌 Redis Commands
- Basic Commands:
  SET user:1 "John Doe"
  GET user:1
  DEL user:1
  EXISTS user:1
  KEYS user*

- Increment & Decrement Counters:
  INCR myCounter
  INCRBY myCounter 10
  DECR myCounter
  DECRBY myCounter 5

📌 Redis Data Structures
1. Hashes (Key-Value Collections)
   - Use Cases: User profiles, session tracking, event logging.
   - Commands:
     HSET bike:1 model "Demios" brand "Ergonom" price 1971
     HGET bike:1 model
     HGETALL bike:1

2. Lists (Linked Lists)
   - Use Cases: Stacks, queues, social media feeds, logging systems.
   - Queue Example:
     LPUSH tasks "Task1"
     LPUSH tasks "Task2"
     RPOP tasks
   - Stack Example:
     LPUSH stack "Item1"
     LPOP stack

3. Sets (Unique Collections)
   - Use Cases: User groups, unique visitor tracking, social networks.
   - Commands:
     SADD students "Alice"
     SADD students "Bob"
     SCARD students
     SISMEMBER students "Alice"

4. JSON Support in Redis
   - Uses JSONPath syntax for navigation.
   - Internally stored in a tree-structure for fast access.

📌 Conclusion:
- NoSQL & Key-Value stores trade off ACID guarantees for scalability & performance.
- Redis is an in-memory KV store with high-speed operations.
- Different data types allow flexible storage of structured & semi-structured data.

Redis-Py: The Standard Python Client for Redis

1️⃣ Redis-Py Overview
- Redis-Py is the standard Python client for Redis and is maintained by the Redis Company.
- GitHub Repository: https://github.com/redis/redis-py
- Installation in Conda Environment (DS4300):
  ```bash
  pip install redis
  ```

2️⃣ Connecting to Redis with Python
- Connection Parameters:
  - host: localhost or 127.0.0.1 (Docker default).
  - port: Default 6379 (unless mapped differently).
  - db: Redis databases 0-15.
  - decode_responses=True: Converts bytes to strings.

- Example Python Code:
  ```python
  import redis

  redis_client = redis.Redis(
      host='localhost',
      port=6379,
      db=2,
      decode_responses=True
  )
  ```

3️⃣ Redis Command List & Documentation
- Command Documentation:
  - Redis Command Reference: https://redis.io/docs/latest/commands/
  - Redis-Py Documentation: https://redis-py.readthedocs.io/en/stable/

- Commands are categorized by data structures: strings, lists, hashes, sets, etc.

4 String Commands in Redis-Py

- Basic Set/Get Operations:

```python
redis_client.set('clickCount:/abc', 0)
redis_client.incr('clickCount:/abc')
value = redis_client.get('clickCount:/abc')
print(f'Click count = {value}')
```

- Multiple Key-Value Pairs:

```python
redis_client.mset({'key1': 'val1', 'key2': 'val2', 'key3': 'val3'})
values = redis_client.mget('key1', 'key2', 'key3')
print(values)  # Output: ['val1', 'val2', 'val3']
```

- Additional String Commands:
  set(), mset(), setex(), msetnx(), setnx()
  get(), mget(), getex(), getdel()
  incr(), decr(), incrby(), decrby()
  strlen(), append()

5 List Commands in Redis-Py

- Creating & Retrieving Lists:

```python
redis_client.rpush('names', 'mark', 'sam', 'nick')
print(redis_client.lrange('names', 0, -1))  # Output: ['mark', 'sam', 'nick']
```

- Additional List Commands:
  lpush(), lpop(), lset(), lrem()
  rpush(), rpop()
  lrange(), llen(), lpos()
  Moving elements between lists, popping from multiple lists.

6 Hash Commands in Redis-Py

- Creating & Retrieving Hashes:

```python
redis_client.hset('user-session:123', mapping={
    'first': 'Sam',
    'last': 'Uelle',
    'company': 'Redis',
    'age': 30
```

})
print(redis_client.hgetall('user-session:123'))
```

- Additional Hash Commands:
  hset(), hget(), hgetall()
  hkeys()
  hdel(), hexists(), hlen(), hstrlen()

## 7️⃣ Redis Pipelines (Batch Processing)

- Avoids multiple related calls to the server → reduces network overhead.

- Example:
  ```python
  r = redis.Redis(decode_responses=True)
  pipe = r.pipeline()

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

  set_5_result = pipe.execute()
  print(set_5_result)  # Output: [True, True, True, True, True]

  # Chained commands:
  get_3_result = pipe.get("seat:0").get("seat:3").get("seat:4").execute()
  print(get_3_result)  # Output: ['#0', '#3', '#4']
  ```

8️⃣ Redis in Machine Learning (ML)

- Common Use Cases:
  - Feature stores for real-time ML pipelines.
  - Caching ML model predictions.
  - Storing preprocessed data for fast retrieval.

- Further Reading:
  - FeatureForm: Feature Stores Explained: https://www.featureform.com/post/feature-stores-explained-the-three-common-architectures
  - MadeWithML: Feature Stores: https://madewithml.com/courses/mlops/feature-store/

📌 Conclusion

- Redis-Py enables easy Python interaction with Redis databases.
- Supports key-value storage, lists, hashes, and efficient pipelines.
- Ideal for caching, session management, and ML feature stores.

———
  ```

Redis-Py: The Standard Python Client for Redis

1️⃣ Redis-Py Overview

- Redis-Py is the standard Python client for Redis and is maintained by the Redis Company.
- GitHub Repository: https://github.com/redis/redis-py
- Installation in Conda Environment (DS4300):

```bash
pip install redis
```

2️⃣ Connecting to Redis with Python

- Connection Parameters:
  - host: localhost or 127.0.0.1 (Docker default).
  - port: Default 6379 (unless mapped differently).
  - db: Redis databases 0-15.
  - decode_responses=True: Converts bytes to strings.

- Example Python Code:

```python
import redis

redis_client = redis.Redis(
    host='localhost',
    port=6379,
    db=2,
    decode_responses=True
)
```

3️⃣ Redis Command List & Documentation

- Command Documentation:
  - Redis Command Reference: https://redis.io/docs/latest/commands/
  - Redis-Py Documentation: https://redis-py.readthedocs.io/en/stable/
- Commands are categorized by data structures: strings, lists, hashes, sets, etc.

4️⃣ String Commands in Redis-Py

- Basic Set/Get Operations:

```python
redis_client.set('clickCount:/abc', 0)
redis_client.incr('clickCount:/abc')
value = redis_client.get('clickCount:/abc')
print(f'Click count = {value}')
```

- Multiple Key-Value Pairs:
  ```python
  redis_client.mset({'key1': 'val1', 'key2': 'val2', 'key3': 'val3'})
  values = redis_client.mget('key1', 'key2', 'key3')
  print(values)  # Output: ['val1', 'val2', 'val3']
  ```

- Additional String Commands:
  set(), mset(), setex(), msetnx(), setnx()
  get(), mget(), getex(), getdel()
  incr(), decr(), incrby(), decrby()
  strlen(), append()

5️⃣ List Commands in Redis-Py
- Creating & Retrieving Lists:
  ```python
  redis_client.rpush('names', 'mark', 'sam', 'nick')
  print(redis_client.lrange('names', 0, -1))  # Output: ['mark', 'sam', 'nick']
  ```

- Additional List Commands:
  lpush(), lpop(), lset(), lrem()
  rpush(), rpop()
  lrange(), llen(), lpos()
  Moving elements between lists, popping from multiple lists.

6️⃣ Hash Commands in Redis-Py
- Creating & Retrieving Hashes:
  ```python
  redis_client.hset('user-session:123', mapping={
      'first': 'Sam',
      'last': 'Uelle',
      'company': 'Redis',
      'age': 30
  })
  print(redis_client.hgetall('user-session:123'))
  ```

- Additional Hash Commands:
  hset(), hget(), hgetall()
  hkeys()
  hdel(), hexists(), hlen(), hstrlen()

7️⃣ Redis Pipelines (Batch Processing)

- Avoids multiple related calls to the server → reduces network overhead.

- Example:
  ```python
  r = redis.Redis(decode_responses=True)
  pipe = r.pipeline()

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

  set_5_result = pipe.execute()
  print(set_5_result)  # Output: [True, True, True, True, True]

  # Chained commands:
  get_3_result = pipe.get("seat:0").get("seat:3").get("seat:4").execute()
  print(get_3_result)  # Output: ['#0', '#3', '#4']
  ```

8️⃣ Redis in Machine Learning (ML)
- Common Use Cases:
  - Feature stores for real-time ML pipelines.
  - Caching ML model predictions.
  - Storing preprocessed data for fast retrieval.

- Further Reading:
  - FeatureForm: Feature Stores Explained: https://www.featureform.com/post/feature-stores-explained-the-three-common-architectures
  - MadeWithML: Feature Stores: https://madewithml.com/courses/mlops/feature-store/

📌 Conclusion
- Redis-Py enables easy Python interaction with Redis databases.
- Supports key-value storage, lists, hashes, and efficient pipelines.
- Ideal for caching, session management, and ML feature stores.
———

PyMongo: Python MongoDB Client

1️⃣ What is PyMongo?
- PyMongo is a Python library for interfacing with MongoDB instances.
- Provides an easy-to-use API for CRUD operations on MongoDB.
- Supports database connections, document insertion, queries, and aggregation.

2️⃣ Connecting to MongoDB with PyMongo
- Install PyMongo:

```bash
pip install pymongo
```
- Connect to a MongoDB instance:
```python
from pymongo import MongoClient

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

3️⃣ Getting a Database and Collection
- Retrieve a database & collection:
```python
db = client['ds4300']  # Equivalent to client.ds4300
collection = db['myCollection']  # Equivalent to db.myCollection
```

4️⃣ Inserting a Single Document
- Insert a new document into a collection:
```python
post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
}

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

5️⃣ Finding All Movies from 2000
- Query MongoDB for all movies released in 2000:
```python
from bson.json_util import dumps

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

# Print results in readable JSON format
print(dumps(movies_2000, indent=2))
```

6️⃣ Using PyMongo with Jupyter Lab
- Activate your DS4300 Conda or virtual Python environment.

- Install required dependencies:
  ```bash
  pip install pymongo jupyterlab
  ```
- Download and unzip the MongoDB Jupyter Notebook files:
  [MongoDB Jupyter Notebooks](https://www.dropbox.com/scl/fi/nqlaaxjrgo5ksaw3avb51/25s-ds4300-mongo-ex.zip?rlkey=49rejsqlg9pvdteub2ekg31l6&dl=0)
- Navigate to the folder & start Jupyter Lab:
  ```bash
  jupyter lab
  ```

📌 Conclusion
- PyMongo simplifies MongoDB interactions with Python.
- Supports database connections, document insertion, and querying.
- Jupyter Lab is a useful tool for interactive MongoDB queries.

———

Graph Databases & Graph Data Models

📌 What is a Graph Database?
- A data model based on graphs, consisting of nodes and edges.
- Nodes (Vertices):
  - Represent entities (e.g., people, places, objects).
  - Each node is uniquely identified.
  - Can have properties (e.g., name, occupation).
- Edges (Relationships):
  - Connect nodes to define relationships.
  - Can also contain properties.
- Supports graph-oriented queries, including:
  - Traversals (navigating the graph).
  - Shortest path calculations.
  - Network analysis.

📌 Where Do Graphs Show Up?
- Social Networks:
  - Instagram, Twitter, Facebook.
  - Also used in psychology and sociology to model social interactions.
- The Web:
  - A massive graph of web pages (nodes) connected by hyperlinks (edges).
- Biological & Chemical Data:
  - Genetics, systems biology, chemical reactions.
  - Protein interaction networks, molecular bonding models.

📌 Basics of Graph Theory

- Graph = Nodes + Relationships (Edges).
- Labeled Property Graph (LPG):
  - Nodes and edges can have labels (grouping nodes/edges into categories).
  - Properties (key-value pairs) can exist on nodes and edges.
  - Nodes can exist without relationships (but edges must connect nodes).

📌 Example Graph with Labels & Relationships

- Labels:
  - Person
  - Car
- Relationship Types:
  - Drives
  - Owns
  - Lives_with
  - Married_to

📌 Graph Paths

- A path is an ordered sequence of nodes connected by edges.
- Valid Path Example: 1 → 2 → 6 → 5
- Invalid Path Example: 1 → 2 → 6 → 2 → 3 (node repeats).

📌 Types of Graphs

1️⃣ Connected vs. Disconnected:

- Connected Graph: There exists a path between any two nodes.
- Disconnected Graph: Some nodes are isolated.

2️⃣ Weighted vs. Unweighted:

- Weighted Graph: Edges have weights (e.g., distance, cost, time).
- Unweighted Graph: Edges have no associated weight.

3️⃣ Directed vs. Undirected:

- Directed Graph: Edges have a direction (e.g., follows, transfers money).
- Undirected Graph: Edges do not have a specific direction (e.g., mutual friendship).

4️⃣ Cyclic vs. Acyclic:

- Cyclic Graph: Contains at least one cycle (path returns to the starting node).
- Acyclic Graph: No cycles exist (e.g., a tree).

5️⃣ Sparse vs. Dense Graphs:

- Sparse Graph: Few edges relative to the number of nodes.
- Dense Graph: Many edges relative to the number of nodes.

📌 Tree Structures

- A tree is a special type of graph (a connected, acyclic, undirected graph).
- Used in hierarchical data structures like:
  - File systems
  - XML/JSON document parsing
  - Decision trees

📌 Types of Graph Algorithms

1️⃣ Pathfinding Algorithms

- Finds shortest paths between nodes.
- "Shortest" can mean:
  - Fewest edges.
  - Lowest weight.
- Examples:
  - Dijkstra's Algorithm (single-source shortest path, weighted graphs).
  - A* Algorithm (Dijkstra + heuristic guidance).
  - Minimum Spanning Tree, Cycle Detection, Max Flow/Min Cut.

2️⃣ Centrality & Community Detection

- Centrality: Determines the most important nodes in a network.
  - Example: Social media influencers.
- Community Detection: Identifies clusters and partitions in a graph.
  - Helps understand network structures & relationships.

📌 Famous Graph Algorithms

- Dijkstra's Algorithm:
  - Finds the shortest path from a single source in weighted graphs.
- A* Algorithm:
  - Like Dijkstra, but uses heuristics to optimize traversal.
- PageRank:
  - Measures node importance based on incoming connections.

📌 Neo4j - A Graph Database System

- NoSQL Graph Database for transactional & analytical graph-based queries.
- Key Features:
  - Schema-optional: Schema can be enforced but is not required.
  - Supports Indexing for performance optimization.
  - ACID Compliant (like relational databases).
  - Distributed Computing Support (scales well for large datasets).
- Alternatives: Microsoft CosmosDB, Amazon Neptune.

📌 Conclusion

- Graph databases excel in complex relationship modeling (e.g., social networks, bioinformatics).
- Efficient querying of highly connected data.
- Neo4j & other graph DBs provide powerful graph traversal and analytics tools.

———————————————————————————————————————

📌 **What is Neo4j?**
- A **Graph Database System** supporting **transactional & analytical processing** of graph-based data.
- Part of a relatively new class of **NoSQL databases**.
- **Key Features:**
  - **Schema-optional** (flexibility in defining data models).
  - **ACID-compliant** (ensures data integrity and reliability).
  - **Supports indexing & distributed computing**.
  - **Similar products:** Microsoft CosmosDB, Amazon Neptune.

📌 **Neo4j Query Language & Plugins**

1️⃣ **Cypher Query Language (CQL)**
  - Created in **2011**, designed as an **SQL-equivalent for graphs**.
  - Uses **pattern-matching** to express queries visually.
    ```
    (nodes)-[:CONNECT_TO]->(otherNodes)
    ```

2️⃣ **APOC Plugin (Awesome Procedures on Cypher)**
  - Adds **hundreds of procedures & functions** to enhance Cypher capabilities.

3️⃣ **Graph Data Science Plugin**
  - Provides efficient implementations of **graph algorithms**.

📌 **Neo4j Setup using Docker Compose**
- **Docker Compose:** Manages **multi-container** setups declaratively using `docker-compose.yaml`.
- **Benefits of Docker Compose:**
  - **Environment consistency across machines**.
  - **Simplifies deployment & scaling**.
  - **Easier maintenance & service coordination**.

📌 **Example Docker Compose File for Neo4j**
```yaml
services:
  neo4j:
    container_name: neo4j
    image: neo4j:latest
```

```
    ports:
      - 7474:7474  # Web interface
      - 7687:7687  # Bolt protocol for queries
    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
```

📌 **Environment Variables (`.env` Files)**
- Store **passwords & environment settings** separately.
- Example `.env` file:
  ```
  NEO4J_PASSWORD=abc123!!!
  ```

📌 **Useful Docker Commands**
- **Check installation:** `docker --version`
- **Start Neo4j container:** `docker compose up -d`
- **Stop container:** `docker compose stop`
- **Rebuild without cache:** `docker compose build --no-cache`

📌 **Accessing the Neo4j Browser**
- Open http://localhost:7474 in a browser.
- Log in using **neo4j / your_password**.

📌 **Basic Data Insertion in Neo4j (Cypher)**

1️⃣ **Creating Nodes**
```cypher
CREATE (:User {name: "Alice", birthPlace: "Paris"})
CREATE (:User {name: "Bob", birthPlace: "London"})
CREATE (:User {name: "Carol", birthPlace: "London"})
```

2️⃣ **Creating Relationships**

```cypher
MATCH (alice:User {name: "Alice"}), (bob:User {name: "Bob"})
CREATE (alice)-[:KNOWS {since: "2022-12-01"}]->(bob)
```

3️⃣ **Querying Data**
```cypher
MATCH (usr:User {birthPlace: "London"})
RETURN usr.name, usr.birthPlace
```

📌 **Importing Data into Neo4j**

1️⃣ **Download & Prepare Dataset**
- Clone repo: [Graph-Data-Science-with-Neo4j](https://github.com/PacktPublishing/Graph-Data-Science-with-Neo4j)
- Unzip `netflix.zip` and copy `netflix_titles.csv` to:
  ```
  neo4j_db/import/
  ```

2️⃣ **Basic CSV Import (Movies)**
```cypher
LOAD CSV WITH HEADERS FROM 'file:///netflix_titles.csv' AS line
CREATE(:Movie {
    id: line.show_id,
    title: line.title,
    releaseYear: line.release_year
})
```

3️⃣ **Loading CSV with Directors**
```cypher
LOAD CSV WITH HEADERS FROM 'file:///netflix_titles.csv' AS line
WITH split(line.director, ",") AS directors_list
UNWIND directors_list AS director_name
CREATE (:Person {name: trim(director_name)})
```

4️⃣ **Fixing Duplicate Nodes using `MERGE`**
```cypher
MATCH (p:Person) DELETE p
```

```
LOAD CSV WITH HEADERS FROM 'file:///netflix_titles.csv' AS line
WITH split(line.director, ",") AS directors_list
UNWIND directors_list AS director_name
MERGE (:Person {name: director_name})
```

5️⃣ **Adding Relationships (Directors → Movies)**
```cypher
LOAD CSV WITH HEADERS FROM 'file:///netflix_titles.csv' AS line
MATCH (m:Movie {id: line.show_id})
WITH m, split(line.director, ",") AS directors_list
UNWIND directors_list AS director_name
MATCH (p:Person {name: director_name})
CREATE (p)-[:DIRECTED]->(m)
```

📌 **Verifying Data Import**
```cypher
MATCH (m:Movie {title: "Ray"})<-[:DIRECTED]-(p:Person)
RETURN m, p
```

📌 **Conclusion**
- **Neo4j is a powerful NoSQL graph database** for handling highly connected data.
- **Cypher Query Language simplifies graph traversal & relationship queries**.
- **Ideal for applications like social networks, fraud detection, and recommendation systems**.

———

AWS Introduction - Knowledge Extracted

Amazon Web Services Overview
- AWS is the leading cloud platform with over 200+ services.
- It is globally available through a network of regions and availability zones.
- Uses a pay-as-you-go pricing model, which can be cheaper than renting physical servers.

History of AWS
- 2006: AWS launched with S3 (Simple Storage Service) and EC2 (Elastic Compute Cloud).
- 2010: Expanded services to include:
  - SimpleDB
  - Elastic Block Store (EBS)
  - Relational Database Service (RDS)
  - DynamoDB
  - CloudWatch
  - Simple Workflow
  - CloudFront

- Availability Zones
- AWS initially ran competitions with big prizes to encourage early adoption.
- Continuous innovation has led to over 200+ services for operations, development, analytics, and more.

AWS Cloud Service Models

1️⃣ IaaS (Infrastructure as a Service)
- Provides the fundamental building blocks for IT infrastructure.
- Examples: EC2, S3, RDS, VPC.

2️⃣ PaaS (Platform as a Service)
- Removes the need for infrastructure management.
- Focus on deploying applications without handling hardware/software maintenance.
- Examples: AWS Lambda, Elastic Beanstalk.

3️⃣ SaaS (Software as a Service)
- Fully managed applications provided by vendors.
- Examples: Gmail, Dropbox, AWS Workspaces.

(Further reference: AWS IaaS, AWS PaaS, AWS SaaS)

AWS Shared Responsibility Model
AWS Responsibilities (Security OF the cloud)
- Physical infrastructure security (Data centers, access control, power, HVAC).
- Hypervisor & Host OS management (Virtualization layer).
- Maintenance of managed services (Software patching, updates).

Customer Responsibilities (Security IN the cloud)
- Data classification, encryption, and access policies.
- Identity and Access Management (IAM) (Users, roles, policies).
- Network security configuration (VPC security).
- Compliance & governance policies.

AWS Global Infrastructure
- Regions: Distinct geographical areas (e.g., us-east-1, us-west-1).
- Availability Zones (AZs): Multiple isolated data centers within a region.
- Edge Locations: Content Delivery Network (CDN) points for faster access.

AWS Core Services

1️⃣ Compute Services
- EC2 (Elastic Compute Cloud): Virtual Machines (VMs).
- ECS (Elastic Container Service): Managed container orchestration.
- EKS (Elastic Kubernetes Service): Managed Kubernetes service.
- Fargate: Serverless container execution.

- Lambda: Event-driven serverless computing.

2️⃣ Storage Services
- S3 (Simple Storage Service): Object storage, scalable with multiple storage classes.
- EBS (Elastic Block Store): Persistent high-performance block storage.
- EFS (Elastic File System): Serverless managed file storage.
- Amazon File Cache: High-speed caching for distributed data.
- AWS Backup: Fully managed backup service.

3️⃣ Database Services
- Relational Databases: Amazon RDS, Amazon Aurora.
- NoSQL Databases:
  - DynamoDB (Key-Value Store)
  - DocumentDB (MongoDB-Compatible)
  - ElastiCache (In-memory caching)
  - Neptune (Graph database).

AWS Analytics & Machine Learning

1️⃣ Analytics Services
- Athena: Query structured data in S3 using SQL.
- EMR (Elastic MapReduce): Managed Hadoop/Spark environment.
- Glue: ETL (Extract, Transform, Load) service.
- Redshift: Data warehousing.
- Kinesis: Real-time data streaming.
- QuickSight: Cloud-native BI tool.

2️⃣ Machine Learning & AI Services
- SageMaker: Fully managed ML platform with Jupyter Notebooks.
- Pre-trained AI models:
  - Comprehend (NLP)
  - Rekognition (Image/Video Analysis)
  - Textract (Text Extraction)
  - Translate (Machine Translation).

AWS Free Tier (12 Months)
- EC2: 750 hours/month (limited instance types).
- S3: 5GB storage (20K GETs, 2K PUTs).
- RDS: 750 hours/month (within specific limits).

EC2 - Elastic Compute Cloud
- Scalable virtual machines (General Purpose, Compute-Optimized, GPU, etc.).
- Pay-as-you-go pricing for cost efficiency.
- Supports both prebuilt and custom AMIs.

EC2 Features
- Elasticity: Scale up/down easily.
- Integration: Works with S3, RDS, IAM.
- Lifecycle:
  1. Launch an instance.
  2. Start/Stop to pause usage without losing data.
  3. Terminate to delete permanently.
  4. Reboot to restart.

EC2 Storage Options
1. Instance Store: High-speed temporary storage (lost on reboot).
2. EBS (Elastic Block Store): Persistent block storage for EC2.
3. EFS (Elastic File System): Shared file storage.
4. S3: Long-term object storage.

Common EC2 Use Cases
- Web hosting.
- Big data processing.
- Machine Learning (GPU instances).
- Disaster recovery.

AWS Lambda - Serverless Computing
- Runs code without provisioning servers.
- Triggered by events (e.g., S3 uploads, API Gateway calls).
- Auto-scales dynamically.
- Pay only for execution time.

Lambda Features
- Supports Python, Java, Node.js, and more.
- Integrates with AWS services (S3, DynamoDB, API Gateway).
- Scales automatically based on load.

Lambda Execution Flow
1. Upload Code (via AWS Console, CLI, or SDK).
2. Configure Triggers (API requests, file uploads, etc.).
3. AWS Lambda executes the code on-demand.

Example Lambda Function (Python)
```
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': json.dumps("Hello from AWS Lambda!")
    }
```

Conclusion
- AWS provides scalable cloud solutions across computing, storage, and databases.
- EC2 is ideal for traditional VMs, while Lambda enables serverless computing.
- The Shared Responsibility Model ensures security between AWS and customers.
- Global infrastructure ensures high availability & low latency.

Amazon EC2 & Lambda (Notes)

1. Amazon EC2 (Elastic Cloud Compute)
   - EC2 offers on-demand, resizable compute capacity in the cloud. It is essentially a virtual machine (VM) that you can configure with different OS options, hardware capacities, and storage.
   - Key features:
     • Elasticity: Scale instances up or down programmatically.
     • Broad OS support (Linux, Windows, etc.).
     • AMI (Amazon Machine Image): Use standard AMIs or create custom ones.
   - EC2 Lifecycle:
     • Launch: Start an instance with a chosen configuration.
     • Start/Stop: Suspend usage without deleting the instance.
     • Terminate: Permanently delete an instance.
     • Reboot: Restart an instance without losing data on the root volume.
   - Storage Options:
     • Instance Store: High-speed, ephemeral storage that exists only for the lifetime of the instance.
     • EBS (Elastic Block Storage): Persistent block-level storage.
     • EFS (Elastic File System): A shared, scalable file system that multiple instances can mount.
     • S3: Object storage useful for storing large data sets, backups, and static files.
   - Common Use Cases:
     • Web hosting (website or application server).
     • Data processing (any processing task on a VM).
     • Machine learning (GPU instances for training).
     • Disaster recovery (backups and quick failovers).

2. Ubuntu VM Commands (on EC2)
   - Default user is often “ubuntu.”
   - Use “sudo” to run privileged commands.
   - apt is the package manager. Example:
     sudo apt update; sudo apt upgrade

3. Installing MiniConda on EC2
   - Download and install via:
     curl -O https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
     bash ./Miniconda3-latest-Linux-x86_64.sh

4. Installing & Using Streamlit

- After re-logging in, confirm pip is available (pip --version).
- Install libraries:
  pip install streamlit scikit-learn
- Create a directory (e.g., “web”) for your app.

5. Basic Streamlit App
- Example (test.py):

```
import streamlit as st

def main():
    st.title("Welcome to my Streamlit App")
    st.write("## Data Sets")
    st.write("""
        - data set 01
        - data set 02
        - data set 03
    """)
    st.write("\n")
    st.write("## Goodbye!")

if __name__ == "__main__":
    main()
```

- Run it:
  streamlit run test.py
- Make sure port 8501 (Streamlit’s default) is open in your EC2 security group.

6. AWS Lambda
- Serverless compute platform that runs code in response to events (like S3 uploads or API requests).
- You pay only for the actual runtime of the function.
- Features:
  • Event-driven triggers (S3, SNS, API Gateway, etc.).
  • Multiple runtimes (Python, Node.js, etc.).
  • Automatic scaling based on demand.
  • Deep integration with other AWS services.

7. How AWS Lambda Works
- You write and upload your code in the AWS Management Console (or via CLI).
- Configure an event source (e.g., S3 file upload).
- When the event occurs, your Lambda function executes automatically.

8. Creating & Testing a Lambda Function
- Create a function in the AWS Console (choose runtime, permissions).
- Edit and deploy your code.
- Use test events in the console to confirm functionality.

9. Additional Clarifications
- Security Groups:
  • EC2 requires you to explicitly open ports for inbound traffic.
  • Lambda typically uses event triggers (no ongoing inbound server connections).
- EC2 vs. Lambda:
  • EC2 is suitable for persistent workloads or full control over the server environment.
  • Lambda is ideal for short-lived, event-driven tasks with minimal operational overhead.
- Python Usage:
  • On EC2, install packages freely using pip or conda.
  • On Lambda, package dependencies with your code or use Lambda layers.