HLD NoSQL Internals

Contents

[1. Introduction to NoSQL 2](#_Toc184294503)

[2. Designing a Key-Value Store for NoSQL Storage 4](#_Toc184294504)

[3. Key-Value Storage Engine: Iterative Solutions 5](#_Toc184294505)

[Solution 1: File-Based Storage 5](#_Toc184294506)

[Solution 2: Write-Ahead Logging (WAL) 6](#_Toc184294507)

[Solution 3: Write-Ahead Log + Indexing 7](#_Toc184294508)

[Summary Till Now 8](#_Toc184294509)

[Solution 4: Compaction 10](#_Toc184294510)

[Detailed Example of Compaction 11](#_Toc184294511)

[Handling Read/Update Requests During Compaction 13](#_Toc184294512)

[Solution 5: Optimizing Read and Write 15](#_Toc184294513)

[Chunked WAL and Memory-Disk Interaction 17](#_Toc184294514)

[Log-Structured Merge Tree (LSM Tree) 20](#_Toc184294515)

[Dry Run of LSM Tree Operations 23](#_Toc184294516)

[LSM Tree and Related Concepts 26](#_Toc184294517)

[Summary: Storage Engines and the LSM Approach 30](#_Toc184294518)

# Introduction to NoSQL

**1. Overview of Database Components**

* **Database Parts:**

A database is not a single monolithic entity but comprises multiple components:

* + **Execution Engine:** Example: InnoDB (MySQL's execution engine).
  + **Storage Engine:** Responsible for storing and retrieving data efficiently.
  + **SQL Parser:** Interprets SQL queries and transforms them into operations.

**2. Difference Between SQL and NoSQL Databases**

* **Structured Data (SQL):**
  + Uses a **fixed schema** (e.g., tables with predefined columns like ID, name, email).
  + Rows in a table share the same structure and data types.
  + **Advantages:**
    - Predictable storage.
    - Deterministic retrieval.
  + **Drawback:** Wasteful storage in some cases (e.g., padded VARCHAR fields).
* **Unstructured Data (NoSQL):**
  + Does not enforce a fixed schema.
  + Rows can have different attributes (e.g., Row 1: ID, name, email; Row 2: ID, name).
  + Retrieval is challenging due to schema variability.

**3. Data Storage Challenges**

* **SQL Storage on Disk:**
  + Data is stored sequentially.
  + **Example:**  
    Row 1: ID (8 bytes) + Name (20 bytes) → Address: 0 to 28.  
    Row 2: Starts at offset 28.
  + Sequential reading is efficient due to deterministic schema.
* **Variable-Length Fields (e.g., VARCHAR):**
  + Typically have an upper bound.
  + Space is padded for unused portions.
  + **Trade-offs:**
    - Wastes space for unfilled fields.
    - Ensures predictability for reading offsets.
* **NoSQL Storage on Disk:**
  + **Challenge:** Cannot store unstructured data sequentially like SQL due to variable schemas.
  + Reading offsets and determining positions is non-deterministic.

**4. Objective: Designing a NoSQL Storage Engine**

* **Problem Statement:**

How to design a storage engine for NoSQL databases to store and retrieve unstructured data efficiently.

* **Key Considerations for NoSQL Storage:**
  + Schema variability (rows can differ in structure).
  + Efficient data retrieval without sequential access.
  + Storage mechanisms that handle dynamic and diverse data types.

**5. Preliminary Observations**

* **SQL's Strengths in Storage:**
  + Fixed schema simplifies storage and retrieval.
  + Deterministic addressing (sequential reading based on known offsets).
* **NoSQL's Requirements:**
  + Must handle schema-less or flexible schema storage.
  + Cannot rely on fixed offsets or sequential reading.
* **Why a New Engine is Needed:**

NoSQL data structure variability demands a novel approach to storage, different from traditional SQL storage engines.

# Designing a Key-Value Store for NoSQL Storage

**1. Agenda: Building a Simplified Key-Value Store**

* **Goal:**  
  To design a **storage engine** for a **key-value data store** similar to Redis, focusing on **disk persistence** and **update efficiency**.
* **Scope Simplification:**

Instead of a full-fledged NoSQL database, the focus will be on creating a simplified version with two main operations:

* 1. **Get (key):** Retrieve the value associated with a given key.
  2. **Update (key, value):**
     + If the key exists, overwrite its value.
     + If the key does not exist, insert a new key-value pair.

**2. Key Differences from In-Memory Data Structures**

* **Hash Map Analogy:**
  + Similar functionality to a hash map, but a hash map operates entirely **in-memory**.
  + This storage engine must support **disk persistence** for data durability.
* **Challenges of Disk-Based Storage:**
  + **Disk I/O overhead:** Accessing and writing to disk is slower than in-memory operations.
  + **Efficient updates:** Must minimize the overhead of modifying data stored on disk.
  + **Scalability:** Needs to handle large datasets beyond in-memory constraints.

# Key-Value Storage Engine: Iterative Solutions

**Problem Statement**

* Build a **disk-based key-value storage engine**.
* Start with a simple approach and iteratively optimize:
  + Initial focus on **write complexity**.

## Solution 1: File-Based Storage

1. **Description:**
   * Key-value pairs are stored in a plain text file (e.g., CSV or TXT) on disk.
   * Example:
     + In-memory: {1: "shrelock"}
     + Disk file:

|  |
| --- |
| 1, shrelock  2, watson |

1. **Operations Complexity:**
   * **Read (Get):** O(N)
     + Traverse the entire file sequentially until the key is found.
   * **Write (Update):** O(N)
     + Traverse to locate the key, update the value, or append a new row.
2. **Advantages:**
   * Simple implementation.
   * Cheap storage cost.
3. **Disadvantages:**
   * Inefficient for large datasets.
   * Both read and write operations have O(N) complexity, making it unsuitable for scalable systems.

## Solution 2: Write-Ahead Logging (WAL)

1. **Description:**
   * Inspired by SQL's "write-ahead log" used for durability in ACID properties.
   * All updates are appended to the end of the file; no modification of existing rows.
   * The file grows as new updates are logged.
2. **Example:**
   * Initial data:

|  |
| --- |
| 1, A  2, B  3, C |

* + Update 3 to D: Append the new row.

|  |
| --- |
| 1, A  2, B  3, C  3, D |

1. **Operations Complexity:**
   * **Write (Update):** O(1)
     + Direct append to the file.
   * **Read (Get):** Still O(N) because the entire file must be scanned for the latest value of a key.
2. **Advantages:**
   * Write operations are efficient (O(1)).
   * Simple implementation.
3. **Disadvantages:**
   * File grows indefinitely, leading to performance issues for read operations.
   * Requires periodic **compaction** to clean up outdated rows and reduce file size.

## Solution 3: Write-Ahead Log + Indexing

**Adding an Index**:

* Index is a structure in memory mapping keys to their latest data location on disk.
* Example:
  + WAL

|  |  |  |
| --- | --- | --- |
| Location | Key | Value |
| 1001 | 1 | A |
| 1002 | 2 | B |
| 1003 | 3 | C |
| 1004 | 3 | D |

* + Index

|  |  |  |
| --- | --- | --- |
| Key |  | Location |
| 1 | → | 1001 |
| 2 | → | 1002 |
| 3 | → | 1004 |

**Where is the Index Stored?**

* Stored in **memory** for quick access.
* Persistent backup of the index can be stored on disk and recreated if needed.

**Operations**:

* **Read**: (amortized)
  + Index is queried to fetch the disk location directly.
* **Write**:
  + Data is appended to WAL, and the index is updated with the new location.

**Advantages**:

* Efficient read and write operations.
* No need to traverse the file sequentially for reads.

## Summary Till Now

**1. Flat File Approach**

* **Process:**
  + Data is stored sequentially in a flat file.
  + Read and write operations involve traversing the entire file.
* **Complexities:**
  + **Read:** O(N), as the entire file may need to be traversed to find a specific row.
  + **Write:** O(N), as updating requires finding and modifying the existing row.
* **Problems:**
  + Inefficient for large datasets.
  + High latency for read and write operations.

**2. Write-Ahead Logging (WAL)**

* **Process:**
  + Each update is appended to the end of the file (immutable rows).
  + No overwriting; all changes are logged as new entries.
* **Complexities:**
  + **Write:** O(1), as appending is instantaneous.
  + **Read:** O(N), as searching the file for the latest version of a row requires traversal.
* **Problems:**
  + **Duplicate Data:** Each update increases redundancy, leading to wasted space.
  + **Read Inefficiency:** No direct access to specific rows.

**3. WAL + Index**

* **Process:**
  + **WAL** is used to log updates in the file sequentially.
  + An **index table** is maintained in memory for key-to-address mapping.
  + The index allows direct access to the location of data in the file.
* **Operations:**
  + **Get:**
    - Look up the key in the index to retrieve its disk address.
    - Use the address to fetch data using a disk seek operation.
    - **Time Complexity:** Amortized O(1).
  + **Update:**
    - Append new data to the WAL file.
    - Update the index to point to the new entry.
    - **Time Complexity:** O(1).
* **Benefits:**
  + **Read Efficiency:** Amortized O(1) using the index.
  + **Write Efficiency:** Still O(1) due to appending.
* **Problems:**
  + **Memory Limitation:** Index table size is constrained by available RAM.
    - Large datasets with billions of rows cannot be fully indexed in memory.
  + **Duplicate Data:** Redundancy still exists in the WAL file.

**Key Observations and Challenges**

1. **Memory vs. Disk Size Disparity:**
   * Modern systems often have much smaller RAM compared to disk storage (e.g., 32GB RAM vs. 5TB disk).
   * It’s infeasible to store the **entire index** in memory for large datasets.
2. **Duplicate Data in WAL:**
   * Leads to wasted storage.
   * The index must always point to the latest version of the data, which increases complexity.

## Solution 4: Compaction

**Introduction**

* **Problem**: Write-Ahead Logging (WAL) introduces **duplication of rows** due to frequent updates in write-intensive NoSQL databases.
* **Goal**: Eliminate duplicates in WAL to optimize storage and read efficiency while maintaining data correctness.

**Compaction**

* Compaction is an **asynchronous background process** designed to:
  1. Deduplicate WAL by merging multiple rows of the same key into a single row.
  2. Retain only the latest value for each key.
  3. Periodically clean up redundant data.

**Steps of the Compaction Process**

**1. Periodic Execution**

* Runs on a schedule (e.g., daily or every few days).
* Reduces the computational burden by avoiding continuous execution.

**2. File Chunking**

* WAL files can be large; reading them entirely into memory is impractical.
* The file is broken into **manageable chunks** (e.g., 1,000 rows per chunk).
* Process each chunk sequentially to avoid memory overuse.

**3. Deduplication**

* Deduplication identifies and keeps only the latest row for each key.
* **How Deduplication Works**:
  + For each row in the chunk:
    1. Compare the row’s address with the **latest address** in the **index**.
    2. If the row’s address matches the latest address in the index:
       - Add the row to a **new WAL file**.
    3. If the row’s address does not match:
       - Discard the row.
  + **Key Insight**: The index always contains the address of the most recent row for each key.

**4. Creation of New Files**

* During compaction:
  + A **new WAL file** is created to store the deduplicated rows.
  + A **new index file** is created to store updated key-to-address mappings.

**5. Final Steps**

* Replace the old WAL file and index with the new ones:
  1. Delete the old WAL file.
  2. Delete the old index.
  3. The new WAL file and index become the active files for the database.

## Detailed Example of Compaction

**Input WAL File:**

| **Key** | **Value** | **Address** |
| --- | --- | --- |
| 1 | A | 1000 |
| 1 | B | 2000 |
| 1 | C | 3000 |

**Index:**

| **Key** | **Latest Address** |
| --- | --- |
| 1 | 3000 |

**Process:**

1. Read chunks from the WAL file.
2. For each row in the chunk:
   * Compare the row’s address with the index.
   * Discard rows with addresses not matching the latest address (1000, 2000).
   * Retain rows with matching addresses (3000).

**Output WAL File:**

| **Key** | **Value** | **Address** |
| --- | --- | --- |
| 1 | C | 5000 |

**Updated Index:**

| **Key** | **Latest Address** |
| --- | --- |
| 1 | 5000 |

**Address Change Explanation**

1. **Original Address (3000)**:
   * In the original WAL file, 1C is stored at address 3000.
   * This is the latest version for key 1 as per the current index.
2. **New Address (5000)**:
   * During **compaction**, a **new WAL file** is created.
   * All retained rows, including 1 C, are written into this new WAL file sequentially. The location of the data in this new file determines the **new address**.
   * In this case, when the new WAL file is created, the row 1 C gets written to a new position in the file, which corresponds to the address 5000.
3. **Updating the Index**:
   * After writing 1 C to the new WAL file, the **index** is updated to reflect the new address (5000) for key 1.

**Why Does the Address Change?**

* Compaction creates a new WAL file to store the deduplicated rows, resulting in **new physical locations on disk** for those rows.
* These new locations are reflected as new addresses in the index.

**Advantages of Compaction**

1. Reduces the size of WAL by eliminating duplicates.
2. Ensures faster read operations by simplifying the file structure.
3. Frees up disk space over time.
4. Provides a clean and optimized database state.

## Handling Read/Update Requests During Compaction

**Key Concepts**

1. **Compaction Overview**:
   * Compaction creates new WAL (Write-Ahead Log) and index files to remove duplicates.
   * The old WAL and index files remain valid until the compaction process completes.
   * Challenges arise when read/update requests occur during the compaction process.

**Handling Read Requests During Compaction**

* **Simple Approach**:
  + Use the **current (old) index file** to locate rows in the old WAL file.
  + Since the old files are not modified during compaction, consistency is maintained.
* **No Blockage**:
  + Reads are straightforward as they continue to reference the old files, ensuring no interruption.

**Handling Update Requests During Compaction**

* Updates introduce complexity since new data cannot directly modify the old files during compaction.
* Multiple approaches are used by databases to address this:

**1. Stop-the-World (Maintenance Mode):**

* Pause all update requests until the compaction completes.
* **Pros**: Simple to implement.
* **Cons**: Inefficient and causes downtime (not suitable for high availability).

**2. Queue Update Requests:**

* Log update requests in a **queue** and process them after compaction finishes.
* **Pros**: Prevents inconsistency during compaction.
* **Cons**: Delays update processing.

**3. Temporary File for Updates:**

* Create a **temporary WAL file** to store incoming updates during compaction.
* After compaction, merge this temporary file with the newly compacted WAL file.
* **Pros**: Handles updates concurrently.
* **Cons**: Increases I/O operations and adds complexity.

**4. Partial Compaction (Incremental):**

* Instead of compacting the entire WAL file, compact **chunks** or specific levels of the database.
* Allows updates on unaffected parts of the database while compaction occurs.
* Common techniques include:
  + **Level Compaction**: Compact specific ranges of keys.
  + **Size-Tiered Compaction**: Compact smaller files first.
  + **Data-Tiered Compaction**: Compact files based on data access frequency.
* **Pros**: Reduces the scope of the compaction, making updates feasible elsewhere.
* **Cons**: Requires careful implementation to avoid data inconsistencies.

## Solution 5: Optimizing Read and Write

**Overview**

The lecture introduces a novel approach to optimize read and write operations in a key-value data store by leveraging chunking, memory-resident structures, and efficient disk storage techniques. Key focus areas:

1. Challenges with indexing and memory inefficiency.
2. Chunk-based WAL (Write-Ahead Log) architecture.
3. Efficient read and write strategies using in-memory data structures and sorted storage.

**Core Concepts**

1. **Challenges with Indexes**:
   * Storing all keys in memory for indexing is memory-inefficient.
   * Proposal: Eliminate global indexes and instead chunkify data.
2. **Chunked WAL Files**:
   * Divide the WAL into chunks, each restricted to a fixed size (e.g., **100 MB per chunk**).
   * The newest data is always written to the **topmost WAL chunk**.
   * This structure ensures that the latest values are easily accessible.

**Optimized Memory Management**

1. **In-Memory Write Operations**:
   * Store the most recent WAL chunk in memory.
   * Use a **Tree Map** instead of a HashMap for in-memory storage:
     + **Tree Map Benefits**:
       - Maintains sorted order of keys.
       - Enables faster range queries and binary search capabilities.
2. **Write Process**:
   * Incoming updates are written to the in-memory Tree Map.
   * When the memory limit (e.g., 100 MB) is reached, the data is **flushed to disk** as a sorted chunk.
   * On disk, data is stored in sorted order to allow efficient search operations.
3. **Read Operations**:
   * Check if the key exists in memory:
     + **If key exists in memory**: O(log N) search in Tree Map.
     + **If key is not in memory**: Perform binary search on the sorted chunks on disk.

**Flushing and Persistence**

* **Flushing**:
  + When the in-memory WAL chunk reaches the size limit:
    1. Sort the data.
    2. Flush the sorted data to the disk as a new chunk.
* **Disk Structure**:
  + Disk contains multiple sorted WAL chunks.
  + Sorted storage enables efficient binary search during reads.

**Operation Complexities**

| **Operation** | **Scenario** | **Complexity** |
| --- | --- | --- |
| **Write** | Update key in memory (Tree Map) | O(log N) |
| **Read** | Key present in memory | O(log N) |
|  | Key not in memory (Disk Search) | O(log M) (binary search on sorted chunks) |
| **Flush** | Write memory chunk to disk | Depends on sorting and I/O overhead. |

**Performance Optimizations**

1. **Chunk-Based Storage**:
   * Divide data into manageable chunks to limit memory and disk overhead.
   * Newest data remains in memory, ensuring fast access.
2. **Sorted Storage on Disk**:
   * Sorting during flush ensures that disk-based binary search is possible, improving read efficiency.
3. **In-Memory Tree Map**:
   * Provides sorted order and fast access for recent keys.

## Chunked WAL and Memory-Disk Interaction

**Key Concepts**

1. **Memory and Disk Setup**:
   * **Memory**: Contains the latest WAL chunk stored in a **Tree Map**.
     + All recent operations (reads/writes) are performed here.
   * **Disk**: Contains flushed chunks stored sequentially.
     + Acts as persistent storage for older data.
2. **Data Structure Analogy**:
   * Disk chunks can be thought of as a **linked list**, where each node (chunk) contains sorted data.

### Dry Run of Operations

1. **Data Setup**:
   * In-Memory Chunk:

|  |
| --- |
| Tree Map: [1 -> A, 2 -> B, 3 -> C] |

* + Disk Chunks:
    - Chunk 1: [1 -> D, 2 -> E, 3 -> H]
    - Chunk 2: [4 -> A, 2 -> Y, 3 -> K]

1. **Operation 1: GET(1)**
   * **Process**:
     1. Check the **Tree Map** in memory.
     2. Key 1 is found in the memory chunk.
     3. Return the associated value: A.
   * **Complexity**: O(log N) (Tree Map).
2. **Operation 2: UPDATE(1, V)**
   * **Process**:
     1. Update the value for key 1 in the **Tree Map** in memory.

Tree Map after update: [1 -> V, 2 -> B, 3 -> C]

* + **Complexity**: O(log N).

A diagram of a computer

Description automatically generated

1. **Operation 3: GET(4)**
   * **Process**:
     1. Check the **Tree Map** in memory:
        + Key 4 is **not present**.
     2. Search the disk chunks sequentially:
        + Check Chunk 1: Not present.
        + Check Chunk 2: Key 4 is **found** with value A
        + Return the value: A.
   * **Inefficiency**:
     1. Disk search involves scanning each chunk sequentially, making it slow for large datasets.
   * **Complexity**:
     1. Worst-case: O(M) (M = number of rows across all chunks).

### Challenges of Binary Search on Unstructured Data

**Key Concepts**

1. **Binary Search**:
   * Efficient search technique used on **sorted** and **structured data**.
   * Steps include finding the middle element and narrowing down the search range based on the value comparison.
2. **Limitation with Unstructured Data**:
   * **Unstructured Data Characteristics**:
     + Row lengths vary (e.g., one row is 50 bytes, another 10 bytes).
     + Values can be complex objects, leading to unpredictable storage sizes.
   * **Binary Search Challenges**:
     + Binary search relies on predictable offsets for mid-point calculation.
     + With unstructured data, mid-point calculation becomes infeasible as you cannot accurately determine positions without sequential traversal.
   * Example:

|  |
| --- |
| Disk Rows:  Row 1: 50 bytes  Row 2: 10 bytes  Row 3: 20 bytes |

* + - Finding the mid-point becomes ambiguous since the data is not of uniform size or predictable structure.

1. **Fixed vs Variable Length**:
   * Fixed length allows structured indexing, enabling direct mid-point calculations.
   * Variable length requires scanning or maintaining additional metadata.

**Key Question: Can Binary Search Work on Disk?**

* **Answer**: No, binary search cannot work directly on unstructured data stored on disk due to:
  + Variable row sizes.
  + Lack of deterministic offsets.
  + Unstructured nature of key-value pairs.

## Log-Structured Merge Tree (LSM Tree)

**Key Problem: Optimizing Read and Write Operations on Sorted Data**

1. **Initial Challenge**:
   * Data stored in chunks on disk is sorted.
   * Sequential traversal is inefficient when the required key is not in memory.
   * Binary search cannot be directly applied to unstructured data (variable-sized rows).
2. **Optimization Idea**:
   * Leverage sorted nature of data to enable efficient lookups using a **secondary index**.

**Solution: Log-Structured Merge Tree (LSM Tree)**

**LSM Tree Overview:**

* A data structure used by many NoSQL databases (e.g., Cassandra, HBase).
* Efficient for handling high write throughput while maintaining read efficiency.
* Key Components:
  + **MemTable**: In-memory structure for writes and reads.
  + **SSTables**: On-disk, sorted data files.
  + **Index Table**: Enables efficient chunk lookups using start addresses.

**Components of the LSM Tree**

**1. MemTable (Tree Map):**

* **Definition**:
  + An in-memory **tree map** where keys and values are stored in sorted order.
  + All writes and reads are processed through the **MemTable** first.
* **Features**:
  + Handles **new writes** efficiently.
  + Sorted nature allows quick in-memory searches (O(log N) for a tree map).
* **Behaviour**:
  + When size exceeds a predefined limit (e.g., 100 MB), the data is flushed to disk.
  + Flushing creates an **SSTable** (Sorted String Table).

**2. SSTables (Sorted String Tables):**

* **Definition**:
  + On-disk, immutable chunks of sorted data.
  + Data from the MemTable is flushed to SSTables in sorted order.
* **Features**:
  + Each SSTable represents a chunk with a start address and sorted keys.
  + Immutable nature ensures consistency during reads and writes.
  + SSTables can grow over time and may need **compaction** to merge smaller chunks.
* **Write-Ahead Logging (WAL)**:
  + Ensures durability by logging writes to disk before updating the MemTable.

**3. Index Table:**

* **Definition**:
  + An in-memory index that maps **starting keys** of SSTables to their **start addresses** on disk.
* **Features**:
  + Does not store all keys—only the starting key of each chunk.
  + Enables efficient binary search for locating the appropriate SSTable.
* **Example**:

|  |  |
| --- | --- |
| Index Table: | |
| Key | Start Address |
| ID001 | 100 |
| ID600 | 200 |
| ID900 | 300 |

* **Lookup Process**:
  + For key ID700:
    - Binary search on the index identifies it lies between ID600 and ID900.
    - Jump directly to address 200, avoiding sequential traversal.

### **Operational Workflow**

**Read Operation:**

1. Check the **MemTable**:
   * If the key is found, return the value.
2. If the key is not in the MemTable:
   * Use the **Index Table** to identify the relevant SSTable.
   * Jump to the start address and search within the SSTable.

**Write Operation:**

1. Write to the **MemTable**:
   * Updates are stored in-memory in sorted order.
2. Flush to **SSTable** when MemTable size exceeds the limit.
   * The data is written to disk in sorted chunks.

**Compaction:**

* Periodic merging of SSTables to reduce fragmentation and improve read efficiency.

## Dry Run of LSM Tree Operations

**Overview**

* This dry run illustrates the process of handling **updates** in an LSM Tree architecture.
* Key focus is on **MemTable**, **Flush Mechanism**, and **SSTable Creation**.

**Dry Run Details**

**Step 1: MemTable (Tree Map)**

* **Definition**:
  + An in-memory data structure (tree map) storing updates in **sorted order**.
* **Updates**:
  + Update(1, A): Add key-value pair 1:A to MemTable.

|  |  |
| --- | --- |
| MemTable: | |
| Key | Value |
| 1 | A |

* + Update(2, B): Add key-value pair 2:B to MemTable.

|  |  |
| --- | --- |
| MemTable: | |
| Key | Value |
| 1 | A |
| 2 | B |

* + Update(3, C): Add key-value pair 3:C to MemTable.

|  |  |
| --- | --- |
| MemTable: | |
| Key | Value |
| 1 | A |
| 2 | B |
| 3 | C |

* **Trigger Condition**:
  + MemTable reaches its size limit (e.g., 100 MB).

**Step 2: Flush to SSTable**

* **Definition**:
  + When MemTable is full, its contents are **flushed** to disk as a new SSTable.
* **Sorting Mechanism**:
  + MemTable is a tree map, so it is already **sorted**.
  + Contents are written directly to disk without further processing.
* **SSTable Creation**:
  + A new SSTable is created on disk with the flushed data.
  + SSTable (on disk):

A diagram of a computer

Description automatically generated

* + This is referred to as **SSTable 1**.

**Scenario: Retrieving a Key (Get(4))**

1. **Check MemTable**:
   * **Query**: Is key 4 present in the MemTable?
   * **Outcome**:
     + If found: Fetch value and return.
     + If not found: Proceed to Index Table. (In this example, 4 is not found in MemTable.)
2. **Index Table Lookup**:
   * **Purpose**: Provides a mapping of SSTable starting keys to disk offsets for efficient access.
   * **Structure**:

|  |  |
| --- | --- |
| Index Table | |
| Key | Offset |
| 1 | 1000 (SSTable 1 starts here) |
| 4 | 2000 (SSTable 2 starts here) |

* + **Steps**:
    - Perform **binary search** in the Index Table using the key (4) to locate the appropriate SSTable.

A diagram of a computer

Description automatically generated

1. **SSTable Search**:

* Navigate to the SSTable identified by the Index Table.
* Perform a localized search (e.g., binary search or sequential scan) within the SSTable to locate the specific row:

|  |  |
| --- | --- |
| SSTable 2: | |
| Key | Value |
| 4 | D |
| 2 | E |
| 3 | H |

* **Outcome**: Retrieve the value for key 4 (D).

## LSM Tree and Related Concepts

**Overview**

This lecture provides an in-depth explanation of various aspects of the Log-Structured Merge (LSM) tree, covering **durability**, **deletions**, **duplicate keys**, **compaction**, and **optimization using Bloom Filters**. It highlights practical considerations and efficient techniques used in modern databases.

**Key Concepts and Notes**

**1. Durability: Handling Volatile Memory**

* **Problem**: The latest data is stored in the **MemTable** (in-memory) for fast access, but it is **volatile** (lost on a crash or restart).
* **Solution**: Use a **Write-Ahead Log (WAL)** for durability.
  + **WAL** stores a copy of the latest data in **disk**.
  + On a system crash, the **WAL** is replayed to recreate the **MemTable**.
  + Common in databases like MySQL (binlog serves as WAL).
  + **Dual Storage**:
    - MemTable (RAM) → for fast reads and writes.
    - WAL (Disk) → for durability.

**2. Indexes in LSM Trees**

* **Where are indexes stored?**
  + Typically stored **in memory** for faster access.
  + **Recreation**: If lost, indexes can be rebuilt by traversing the SSTables.
* **Durability**: Indexes are not persisted because they can be regenerated from disk data.

**3. Deletion in LSM Trees: Tombstoning**

* **Problem**: Actual deletions in SSTables lead to:
  + **Fragmentation**: Unused spaces in SSTables.
  + Inefficiency: Requires traversing all SSTables to ensure the data is removed.
* **Solution**: **Soft Deletes (Tombstones)**.
  + Mark a key as deleted using a special value (e.g., null or a predefined "tombstone").
  + On a GET operation:
    - If the key has a tombstone, return **not found**.
  + Tombstones are eventually cleaned up during **compaction**.

**4. Duplicate Keys Across SSTables**

* **Why does duplication happen?**
  + Multiple flushes from MemTable to disk lead to the same key being present in multiple SSTables.
* **Example Workflow**:
  + MemTable is flushed to SSTable 1 (keys: 1, 2, 3).
  + Later, updates to the same keys are flushed to SSTable 2 (keys: 1, 2 with new values).
* **Problem**: Redundant storage and inefficiency during lookups.
* **Solution**: **Compaction**.
  + Periodic merging of SSTables into a single table.
  + Removes duplicate keys and keeps the latest version of each key.
  + Updates the index table accordingly.
  + Compaction is a background process, typically run at scheduled intervals.

**5. Bloom Filters for Optimized Key Lookups**

* **Problem**: Searching for a key across multiple SSTables is expensive.
  + Requires scanning each SSTable or using binary search within each SSTable.
* **Solution**: **Bloom Filter**.
  + A **probabilistic data structure** that quickly determines whether a key might exist in an SSTable.
  + Guarantees:
    - **False negatives**: Never (if the Bloom filter says a key is absent, it is guaranteed absent).
    - **False positives**: Possible (if it says present, the key might not actually exist).
  + **How it works**:
    - Maintains a **bit array** and uses multiple **hash functions**.
    - For a given key:
      * Hash functions map it to specific bits in the array.
      * If all corresponding bits are 1, the key might exist.
      * If any bit is 0, the key is absent.
  + **Advantages**:
    - Reduces unnecessary SSTable lookups.
    - Memory-efficient and fast.
  + **Use Case**:
    - Each SSTable has its own Bloom Filter to check if a key might be present before searching.

**6. Compaction in Detail**

* **Purpose**:
  + Reduces the number of SSTables by merging them.
  + Eliminates duplicate keys and tombstones.
* **How it works**:
  + Select multiple SSTables for compaction.
  + Merge them into a single SSTable using techniques like **merge sort**.
  + Remove redundant or obsolete entries (e.g., keys with tombstones).
  + Update the **Index Table** and **Bloom Filters** accordingly.
* **Frequency**: Compaction is a periodic background process.

**Key Takeaways**

1. **Durability**:
   * WAL ensures persistence of data in case of crashes.
2. **Soft Deletes**:
   * Tombstoning avoids fragmentation and simplifies deletion.
3. **Compaction**:
   * Periodic merging resolves inefficiencies like duplicate keys and fragmentation.
4. **Optimization**:
   * Bloom Filters minimize unnecessary disk accesses.
   * Efficient hash-based key existence checks.
5. **Trade-Offs**:
   * Bloom Filters are probabilistic and introduce a small chance of false positives.
   * Compaction is resource-intensive but necessary for long-term efficiency.

## Summary: Storage Engines and the LSM Approach

**Why a Storage Engine?**

* **Purpose**: In NoSQL databases, data is often unstructured and lacks a fixed schema, necessitating a dedicated storage engine to manage and query data efficiently.
* **Objective**: To build a storage engine optimized for **key-value stores** with efficient read and write operations.

**Key Approaches to Building a Storage Engine**

**1. Simple File-Based Storage**

* Data is stored sequentially in a file.
* **Pros**: Easy to implement.
* **Cons**:
  + Worst-case time complexity for all operations (O(n)).
  + Inefficient for large datasets.

**2. Write-Ahead Log (WAL)**

* **Concept**: Data is appended to the end of a log file for faster writes.
* **Pros**:
  + **Fast Write Operations**: Appends are quick.
* **Cons**:
  + **Read Complexity**: Still O(n) due to sequential scanning.
  + **Data Duplication**: Redundant entries may exist.

**3. WAL with an Index**

* **Idea**: Introduce an in-memory index to track key-value pairs and their disk addresses.
* **Pros**:
  + **Fast Read and Write**: Both operations become O(1) for the indexed data.
* **Cons**:
  + **Memory Limitation**: Indexing all data may exceed available memory.

**4. Compaction**

* **Problem**: WAL files can have duplicate data, leading to inefficiency and wasted disk space.
* **Solution**:
  + Periodically combine multiple WAL files into a single, optimized file.
  + Use the in-memory **index** to identify the latest version of each record.
  + Retain only the latest data during compaction.
* **Benefits**:
  + Reduces duplicates.
  + Optimizes storage and query efficiency.

**5. Log-Structured Merge (LSM) Trees**

* **Core Idea**: Store data in smaller, sorted chunks and periodically merge them for optimization.
* **Components**:
  1. **MemTable**:
     + In-memory structure for current data.
     + Data is written here first for faster access.
     + Acts as a **write-back cache**.
     + Flushed to disk when a size threshold (e.g., 100MB) is reached.
  2. **SSTable (Sorted String Table)**:
     + Immutable, sorted chunks of data stored on disk.
     + Created when MemTable is flushed.
  3. **Index**:
     + In-memory metadata pointing to the starting addresses of sorted chunks in SSTables.
     + Enables efficient binary search to locate data.

**How the LSM Approach Works**

**Writing Data**

1. Data is written to the **MemTable**.
2. Once the MemTable reaches its threshold size, it is flushed to disk as a **sorted SSTable**.
3. Periodically, **compaction** combines SSTables to remove duplicates and reduce fragmentation.

**Reading Data**

1. **Step 1**: Check the MemTable for the key.
   * If found, return the value.
2. **Step 2**: If not in MemTable, use the **index** to locate the relevant SSTable.
   * Perform a binary search on the index to get the disk address.
   * Read the value from the corresponding chunk in the SSTable.

**Challenges in LSM Trees**

**1. Duplicate Data in SSTables**

* Problem: Keys can exist in multiple SSTables due to incremental flushing.
* Solution: **Compaction** eliminates duplicates by merging SSTables and keeping only the latest version of each key.

**2. Efficient Key Lookup Across SSTables**

* Problem: Searching multiple SSTables for a key is expensive.
* Solution: **Bloom Filters**:
  + A probabilistic data structure used to check if a key might exist in an SSTable.
  + **Properties**:
    - If the Bloom filter says "No," the key definitely doesn't exist.
    - If it says "Yes," the key might exist, and further verification is needed.
  + **Implementation**:
    - Uses a bit array and multiple hash functions.
    - Efficiently reduces unnecessary reads across SSTables.

**Key Concepts**

**Compaction in LSM**

* Combines multiple SSTables into a single, optimized SSTable.
* Ensures efficient reads by reducing duplicates and reorganizing data.

**Bloom Filters**

* A space-efficient, probabilistic data structure.
* Reduces the cost of searching across SSTables by quickly filtering out non-existent keys.

**Summary of LSM Tree Operations**

| **Operation** | **Process** | **Efficiency** |
| --- | --- | --- |
| **Write** | Append data to MemTable, flush to SSTable when full | Fast (O(1)) |
| **Read** | 1. Check MemTable.  2. Use Index to locate key in SSTable. | Optimized |
| **Delete** | Perform a **soft delete** by adding a tombstone marker. | Space-efficient |