LSM Tree and SS Tables Explained

What is an LSM Tree?

LSM Tree (Log-Structured Merge Tree) is a write-optimized data structure commonly used in modern databases like LevelDB, RocksDB, and Cassandra. It's designed to handle high write throughput efficiently and is structured as:

- An in-memory balanced binary search tree (like a Red-Black Tree or AVL Tree).
- On-disk immutable sorted files called SSTables.

🔧 How It Works – Step by Step

1.
Start in Memory (The Memtable):
Reads and writes first go to a memory-resident balanced tree, known as the **Memtable**. This structure allows for incredibly fast operations (inserts, updates, and point reads are all $O(\log n)$) because all data access occurs in RAM, avoiding costly disk I/O at this stage.

Memtable State
In-Memory (Fast!)
Example: (Charlie, 110) ← (Bob, 90) ← (Alice, 105)

2.
Write-Ahead Log (WAL) for Durability:
To ensure that no data is lost in the event of a system crash, every single change (insert, update, or delete) is first appended to a sequential **Write-Ahead Log (WAL)** on disk. The WAL is append-only, which makes writing to it very fast. If the system fails, the database can replay the WAL to reconstruct the Memtable's state up to the point of the crash, guaranteeing durability.

WAL State
Disk (Durable!)
Example: PUT (Alice, 100) → PUT (Bob, 90) → PUT (Alice, 105)

3.
Memory Size Limit → Dump to Disk as SSTable:
The Memtable has a configurable size limit. Once it reaches this threshold, it becomes immutable (read-only) and is "flushed" to disk. The contents of this now-frozen Memtable are efficiently converted into a sorted list of key-value pairs (an $O(N)$ operation for a balanced BST traversal) and written sequentially to a new, immutable file on disk called an **SSTable (Sorted String Table)**. After the flush, the Memtable is cleared, and a new, empty Memtable is initialized to handle incoming writes.

SSTable (Disk File)
Sorted, Immutable Data

Alice:	105
Bob:	90
Charlie:	110

4.
Reading Data:
When a read request comes in for a specific key, the LSM tree follows a hierarchical search strategy to find the most recent version of that key. It first checks the active in-memory Memtable. If the key is not found there, it then proceeds to search the on-disk SSTables, always starting from the newest SSTable and moving backward chronologically to older ones. This is because newer SSTables contain more recent data, overriding older versions. As soon as the key is found, the search stops, ensuring the latest value is returned. Each SSTable can be binary searched efficiently due to its sorted nature.

Read Path Hierarchy
Search Order
Memtable → Newest SSTable → Older SSTable 2 → Older SSTable 1
(Stop at first match)

5.
Deletion with Tombstones:
Due to the immutable nature of SSTables, actual data cannot be physically removed from them. Instead, when a key is deleted, a special marker called a "**tombstone**" is written. This tombstone is a new key-value pair where the value indicates a deletion (e.g., a null value or a flag). It is inserted into the Memtable and subsequently flushed to an SSTable just like any other write. During reads, if a tombstone is encountered for a key in a newer SSTable (or the Memtable), it signifies that the key has been logically deleted, and any older, live versions of that key in older SSTables are ignored.

Tombstone Effect
Logical Deletion
SSTable_New: (Key, TOMBSTONE)
SSTable_Old: (Key, Value)
Result: Key is considered deleted (Newer tombstone overrides older value).

6.
Optimizations (Sparse Index, Bloom Filter):
To mitigate the potential slowdown of reads (which might have to check multiple SSTables), LSM trees employ critical optimizations. A **Sparse Index** for each SSTable stores a subset of keys and their disk offsets, allowing for quicker pinpointing of data ranges. A **Bloom Filter** is a probabilistic data structure attached to each SSTable; it can quickly and accurately tell if a key is *definitely not* present in an SSTable, thus allowing the system to skip unnecessary disk I/O.

7.
Compaction:
This is a vital background process that continuously merges multiple SSTables into new, consolidated ones. The primary goals of **compaction** are to reclaim disk space by removing duplicate versions of keys (keeping only the most recent) and physically purging keys that have been marked by tombstones. Compaction also helps maintain read performance by reducing the total number of SSTables that a read operation might need to check. This process happens asynchronously, like garbage collection, to minimize impact on foreground read/write operations.

Compaction Process
Merge & Deduplicate
SSTable A + SSTable B → New SSTable C
(Removes old versions, tombstones)

Alice (SS1):	105
Alice (SS2):	102
↓ Becomes ↓	
Alice:	102

1. The Memtable (In-Memory Component)

The **Memtable** is the heart of an LSM Tree. It's a small, in-memory data structure (often a balanced binary search tree like a Red-Black Tree or AVL Tree) that handles all incoming writes (inserts, updates, deletes).

Every write operation is first recorded in a **Write-Ahead Log (WAL)** on disk for durability, then applied to the Memtable. The WAL ensures that even if the system crashes, the data in the Memtable can be recovered by replaying the log.

Memory
Memtable
(Balanced Binary Search Tree)
Initial State: Empty

↔

Disk
Write-Ahead Log (WAL)
(Append-Only, Sequential Writes)
Initial State: Empty

2. Writing Data (Inserts & Updates)

When data is written, it first goes to the WAL for durability, then to the Memtable for fast access. Updates are simply new entries with a more recent timestamp.

Write Operations

- **Alice, 100:** User Alice score is 100.
- **Bob, 90:** User Bob score is 90.
- **Charlie, 110:** User Charlie score is 110.
- **Alice, 105:** Alice's score is updated to 105 (new entry).

State After Writes

Memtable (Memory)

Alice:	105
Bob:	90
Charlie:	110

(Simplified – actual BST structure applies ordering)

↓

Write-Ahead Log (Disk)
PUT (Alice, 100)
PUT (Bob, 90)
PUT (Charlie, 110)
PUT (Alice, 105)

3. Flushing to SS Tables (Sorted String Tables)

When the Memtable reaches a predefined size limit, it becomes read-only and is "flushed" to disk. The data is sorted (an efficient $O(N)$ operation for a BST) and written sequentially to a new, immutable file called an **SS Table (Sorted String Table)**. A new, empty Memtable is then started.

Memtable (Memory)

Alice:	105
Bob:	90
Charlie:	110

↓ FLUSH ↓

SS Table (Disk)
SS Table 1
(Sorted & Immutable)

Alice:	105
Bob:	90
Charlie:	110

As more data is written and Memtables are flushed, multiple SS Tables accumulate on disk, each sorted.

Disk
SS Table 1

Alice:	105
Bob:	90
Charlie:	110

Disk
SS Table 2

Alice:	102
David:	85
Eve:	95
Frank:	100

Memory
Memtable (New)
Empty, ready for writes

4. Reading Data (Get Operations)

To retrieve a value, the system always checks the most recent data first.

1. Check the **Memtable** first.

2. If not found, check **SS Tables** on disk, starting from the **newest** SS Table and going backwards to the oldest.

3. The first value found for a key is the correct one, as newer tables override older ones due to immutability.

Example Reads:

Read: Get (Eve)
- Check Memtable: No.
- Check SS Table 2: Find `(Eve, 95)`. **Return 95.** (Fast, only 1 disk check)

Read: Get (Charlie)
- Check Memtable: No.
- Check SS Table 2: No.
- Check SS Table 1: Find `(Charlie, 110)`. **Return 110.** (Slightly slower, 2 disk checks)

Read: Get (Alice)
- Check Memtable: No.
- Check SS Table 2: Find `(Alice, 102)`. **Return 102.** (The value in SS Table 2 is newer than SS Table 1's Alice, 105, so we stop here).

5. Deleting Data (Tombstones)

Since SS Tables are immutable, actual data cannot be removed directly. Instead, a special marker called a **tombstone** is written to indicate a key has been logically deleted. This allows deletions to be propagated consistently across distributed systems and eventually cleaned up during compaction.

Delete Operation: Delete (Bob)

- A new entry `(Bob, TOMBSTONE)` is created.
- This tombstone is written to the WAL and inserted into the Memtable.
- When the Memtable flushes, it becomes part of a new SS Table (e.g., SS Table 3).

SS Table 3 (Disk)

Bob:	TOMBSTONE

(Other new data might be here too)

Read After Delete: Get (Bob)

- System checks Memtable: No.
- System checks SS Table 3: Finds `(Bob, TOMBSTONE)`.
- **Result: Key is considered deleted.** (Stops searching, even if an older `Bob, 90` exists in SS Table 1).

Conceptual Read Flow
Memtable → SS Table 3 → SS Table 2 → SS Table 1
"Bob, TOMBSTONE" found in SS Table 3!
Result: Bob Not Found (Deleted)

6. LSM Tree Optimizations

To improve read performance and reduce unnecessary disk I/O, LSM trees employ several optimization techniques for their SS Tables.

Sparse Index

A **Sparse Index** is a smaller, in-memory or cached index for each SS Table on disk. Instead of indexing every key, it stores only a subset of keys (e.g., every 100th key or keys at fixed byte intervals) along with their exact byte offsets or block numbers within the SS Table file.

When searching for a key, the system can first perform a fast binary search on this sparse index to quickly identify the small block or range within the large SS Table file where the key *might* reside. This significantly reduces the amount of data that needs to be read from disk, as it avoids scanning the entire SS Table. The actual data within that block is then scanned linearly or binary searched.

Sparse Index (Example)
SS Table Index for SS Table X

Adam:	offset 0 (Block 0)
Jordan:	offset 1500 (Block 1)
Quincy:	offset 3000 (Block 2)
Sam:	offset 4500 (Block 3)

(If looking for "Dan", binary search the index to find it's between Adam & Jordan, then scan SS Table from Adam's offset/Block 0.)

Bloom Filter

A **Bloom Filter** is a space-efficient, probabilistic data structure designed to quickly test whether an element is a member of a set. For LSM trees, a Bloom filter is associated with each SS Table. Before performing a potentially costly disk read to search an SS Table for a key, the system first queries its corresponding Bloom filter.

- If the Bloom filter indicates that a key is **definitely not** in the SS Table, the system can immediately skip reading that entire SS Table file, saving significant disk I/O and speeding up the read operation. This is a guarantee – a false negative is impossible.
- If the Bloom filter says a key **might be** in the SS Table, then the system proceeds to read and search the SS Table. It's important to note that there is a small, configurable chance of a "false positive" (the Bloom filter says the key might be there, but it's not). However, this is a trade-off for its high space efficiency and speed in ruling out non-existent keys.

Bloom filters are critical for reducing "read amplification" in LSM trees, especially when dealing with many SSTables or frequently querying for non-existent data.

Bloom Filter (Concept)
Is 'Zebra' in SS Table 1?
Bloom Filter says: **NO!**
(Avoid expensive disk read for SS Table 1 as 'Zebra' is definitely not present)

7. Compaction: Cleaning Up Disk Space

Over time, as Memtables are flushed and updates/deletions occur, SS Tables accumulate older versions of data and tombstones. This leads to wasted disk space ("space amplification") and potentially slower reads (due to more tables needing to be checked, known as "read amplification"). **Compaction** is a crucial background process that merges multiple existing SS Tables into a new, consolidated set of SS Tables.

During compaction, the system reads data from multiple source SSTables, performs a merge-sort-like operation, and writes the results to new SSTables. Key actions during this process include:

- **Duplicate Resolution:** Only the **most recent version** of each key is kept. Older versions are discarded.
- **Tombstone Elimination:** If a tombstone for a key is encountered, and there are no older, live versions of that key remaining in any of the SSTables being compacted, then both the older versions and the tombstone itself are physically removed from the new SSTable, permanently reclaiming that space.
- **Space Reclamation:** By eliminating duplicates and tombstones, compaction significantly reduces the total disk space consumed by the database.
- **Read Performance Improvement:** Fewer, larger SSTables mean read operations have fewer files to check, improving read performance over time.

Compaction strategies vary (e.g., Leveling vs. Tiering), but the core goal is to balance write amplification (how many times data is rewritten during compaction), read amplification, and space amplification. This process happens asynchronously in the background to minimize impact on foreground user operations.

Example Compaction: Merging SS Table 1 and SS Table 2

SS Table 1 (Older)

Alice:	105
Bob:	90
Charlie:	110

↔

SS Table 2 (Newer)

Alice:	102
David:	85
Eve:	95
Frank:	100

↓ COMPACT ↓

New Compacted SS Table (Disk)
SS Table A

Alice:	102
Bob:	90
Charlie:	110
David:	85
Eve:	95
Frank:	100

(If 'Bob' had a tombstone in a newer table, he'd be removed entirely)

This process reclaims disk space and improves read performance by reducing the number of SS Tables that need to be scanned.

Conclusion

LSM Trees offer a powerful approach to database indexing, excelling in write-intensive environments by leveraging sequential disk I/O. While reads can sometimes be slower due to checking multiple components, optimizations like Bloom filters and sparse indexes significantly mitigate this. Compaction ensures efficient disk space utilization and overall performance.

🆚 Compared to Other Indexes

Index Type	Strengths	Weaknesses
Hash Index	Super fast O(1) lookups for exact keys, ideal for workloads with mostly point lookups.	Cannot efficiently handle range queries (e.g., "find all users with scores between 80 and 90"); the entire dataset must fit within the available RAM, limiting scalability.
B-Tree	Excellent for range queries due to their inherent sorted structure and ability to efficiently store data on disk. Well-suited for transactional workloads where reads are frequent and updates are often in-place.	Can suffer from slower write performance, especially for random writes, as they involve random disk I/O for updates and insertions (potentially requiring node splits and rebalancing on disk), which are slower than sequential writes.
LSM Tree	Highly optimized for fast writes by favoring sequential disk I/O and batching operations. Efficiently handles very large datasets that exceed RAM capacity. Supports range queries due to sorted SS Tables.	Reads can be slower than B-trees in certain scenarios due to the need to potentially check multiple SSTables to find the most recent version of a key ("read amplification"). Requires continuous background compaction to manage disk space and maintain read performance.

🏁 Summary

LSM Trees are ideal when:

- You have high write volume (e.g., logging, sensor data, time-series data).
- You're okay with slightly variable read latency (though optimized by Bloom filters and sparse indexes).
- You need to scale beyond memory and optimize for disk I/O (especially SSDs, where sequential writes are highly performant).

Used in databases and storage engines like:

- LevelDB, RocksDB (Google's open-source key-value stores)
- Apache Cassandra (distributed NoSQL database)
- InfluxDB (time-series database)
- ScyllaDB (high-performance NoSQL database)