LSM_TREES

A DEEP DIVE INTO THE DS POWERING MODERN DATA PIPELINES

UC BERKELEY EXTENSION - DSA

A PRESENTATION BY REZA RAJAN

MOTIVATION

I have selected this topic to deepen my understanding into how modern databases are designed to handle the data of today.

Without LSM Trees, believe it or not, our day-to-day lives will look much different.

INTRODUCTION

THE PROBLEM

Traditional storage systems were not designed for such high-frequency workloads.

Random disk I/O was becoming a huge bottleneck.

THE PROBLEM

Engineers needed a way to handle this constant flood of incoming data efficiently, without slowing down reads or overwhelming the storage system.

In comes LSM Trees.

TOPICS OF DISCUSSION

IN THIS DEEP DIVE...

- Design principles
- Implementation
- Scope and Limitations
- Big O Analysis
- Demo

DESIGN PRINCIPLES

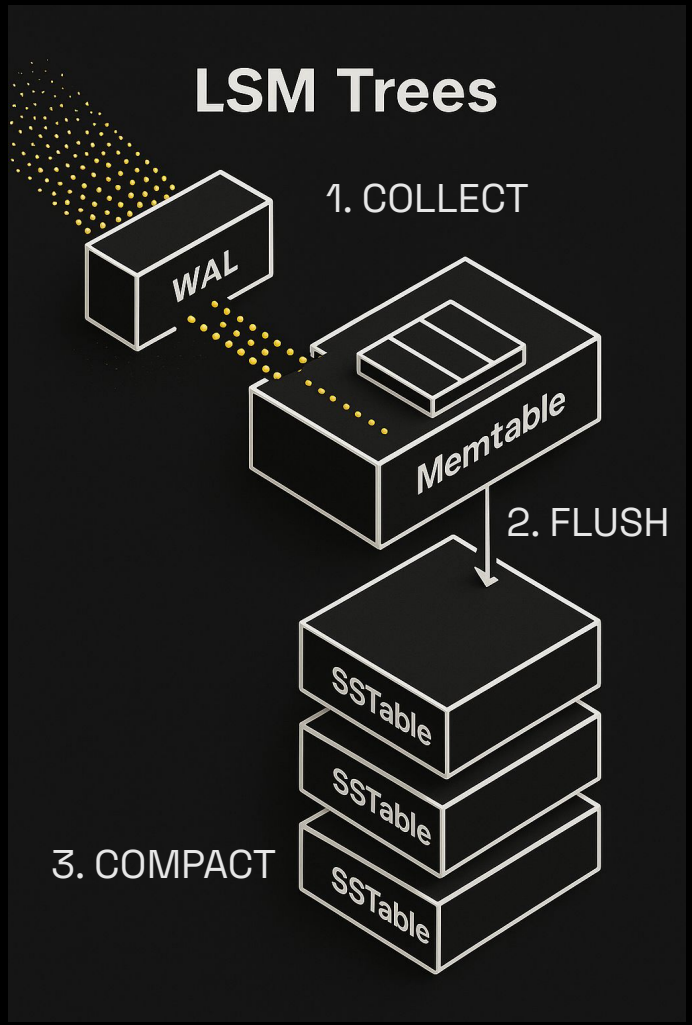
LSM TREES

DEFINITION:
LSM Tree = Log-Structured Merge Tree.

An abstract data structure designed to handle workloads with **high-frequency writes** by buffering updates in memory and periodically merging them into sorted, immutable files on disk.

DESIGN PRINCIPLES

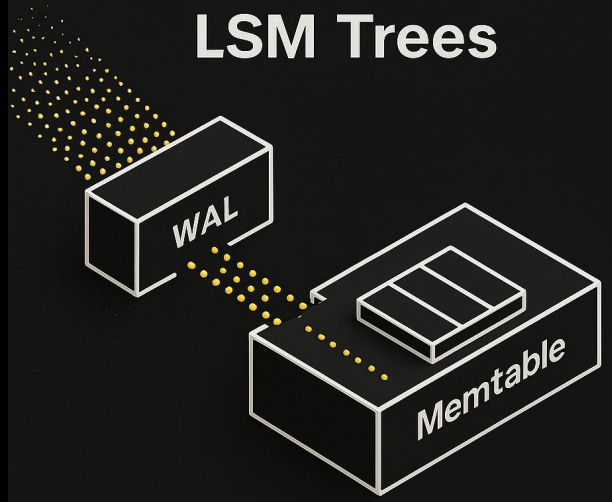
- Leverage fast sequential I/O
- Decouple write layer and read layer
- Utilize high-speed memory, and flush data to disk over time for durability
- Keep data on disk sorted for fast lookups



HOW IT WORKS - PROCESS OVERVIEW

We can broadly materialize these principles into three processes: collect, flush and compact.

LSM Trees



HOW IT WORKS - COLLECTION

In the LSM Tree, all operations are treated as append-only writes—including inserts, updates, and deletes. They are first collected in the **WAL**, and then pushed to a **Memtable**.

Key Concept: WAL and Memtable form the high-throughput "write" layer of the LSM Tree.

WRITE-AHEAD LOG (WAL)

The Write-Ahead Log is an **append-only** log.

Supports sequential I/O for fast operation.

Resides on disk to ensure durability in the event of a crash.

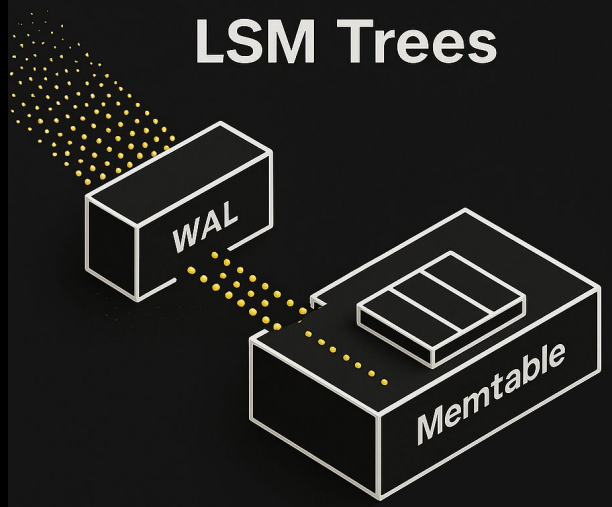
MEMTABLE

In-memory data store, which *may* be sorted.

Reads and writes benefit from **fast in-memory storage.**

Acts as caching layer for recent data.

LSM Trees



HOW IT WORKS - COLLECTION

In the LSM Tree, all operations are treated as append-only writes—including inserts, updates, and deletes. They are first collected in the **WAL**, and then pushed to a **Memtable**.

Key Concept: WAL and Memtable form the high-throughput "write" layer of the LSM Tree.

WRITE-AHEAD LOG (WAL)

The Write-Ahead Log is an **append-only** log.

Supports sequential I/O for fast operation.

Resides on disk to ensure durability in the event of a crash.

MEMTABLE

In-memory data store, which *may* be sorted.

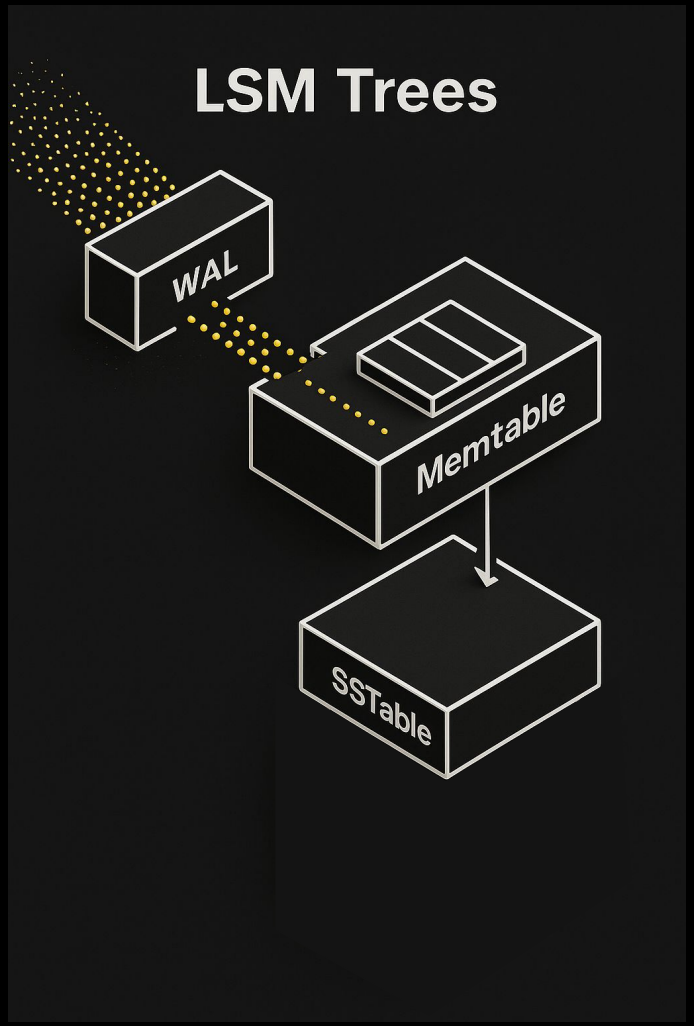
Reads and writes benefit from **fast in-memory storage.**

Acts as caching layer for recent data.

CAVEAT

Memory is a limited resource.

What happens when the Memtable becomes full?



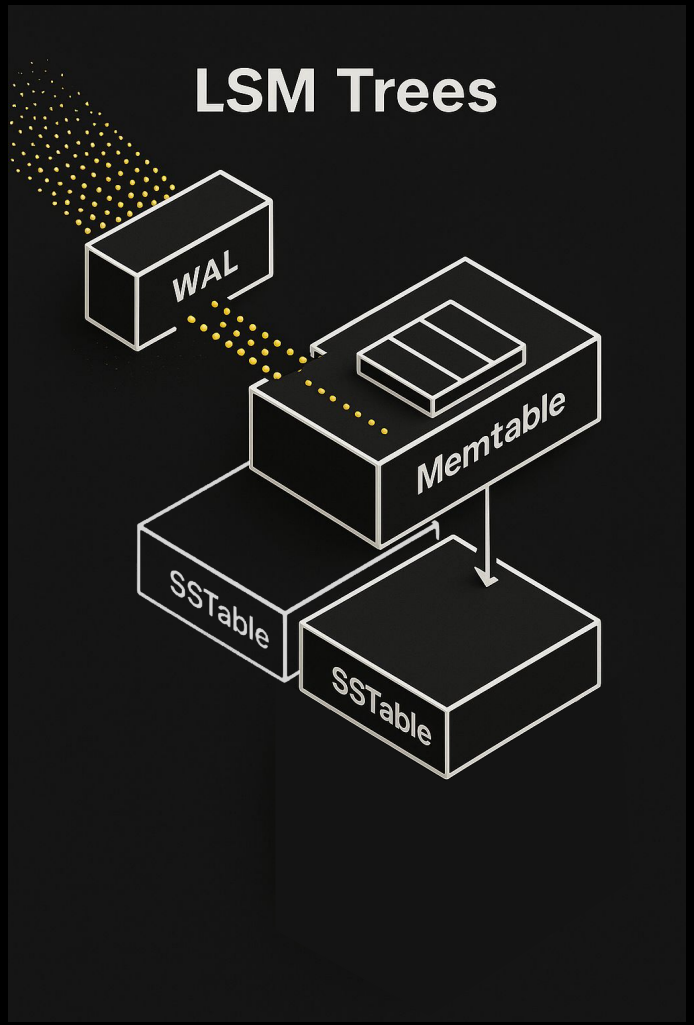
HOW IT WORKS - FLUSHING

Once the Memtable is full, its data is persisted to disk as **SSTables**, and the **Memtable is freed to accept more writes.**

Key Concept: SSTables store sorted, structured data on disk. They support efficient reads and form the "read" layer of the LSM Tree.

SSTABLES

Immutable, sorted files on disk. Provide a structured, queryable format for efficient reads (e.g., Binary Search). Each flush creates a **new** SSTable, skipping in-place updates and leveraging sequential I/O.



HOW IT WORKS - FLUSHING

Once the Memtable is full, its data is persisted to disk as **SSTables**, and the **Memtable is freed to accept more writes.**

Key Concept: SSTables store sorted, structured data on disk. They support efficient reads and form the "read" layer of the LSM Tree.

SSTABLES

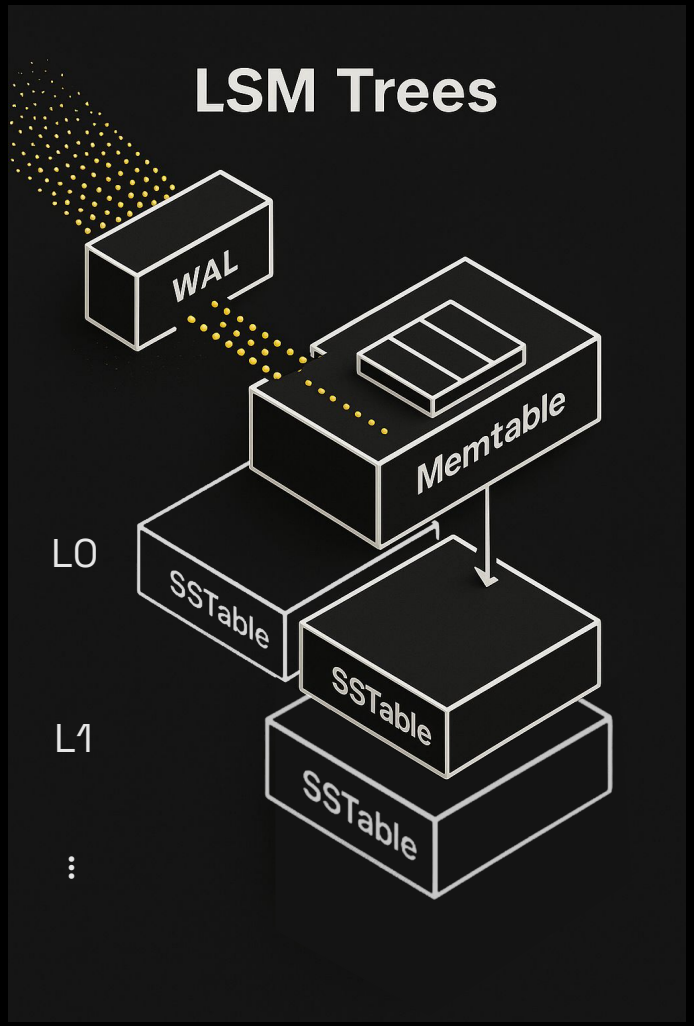
Immutable, sorted files on disk. Provide a structured, queryable format for efficient reads (e.g., Binary Search). Each flush creates a **new** SSTable, skipping in-place updates and leveraging sequential I/O.

CAVEAT

Over time, the number of SSTable files accumulate from repeated flushes.

Scans must traverse all these files to find a key, slowing read operations with each flush of the Memtable.

This problem is called read amplification.



HOW IT WORKS - COMPACTION

Compaction counteracts read-amplification by reducing the sparsity of SSTable files on disk.

Involves organizing SSTables into logical **levels** though merging.

Merging maintains sorted order, prunes outdated and deleted entries, and reduces the total number of SSTable files at each level.

LEVEL 0

The first level.

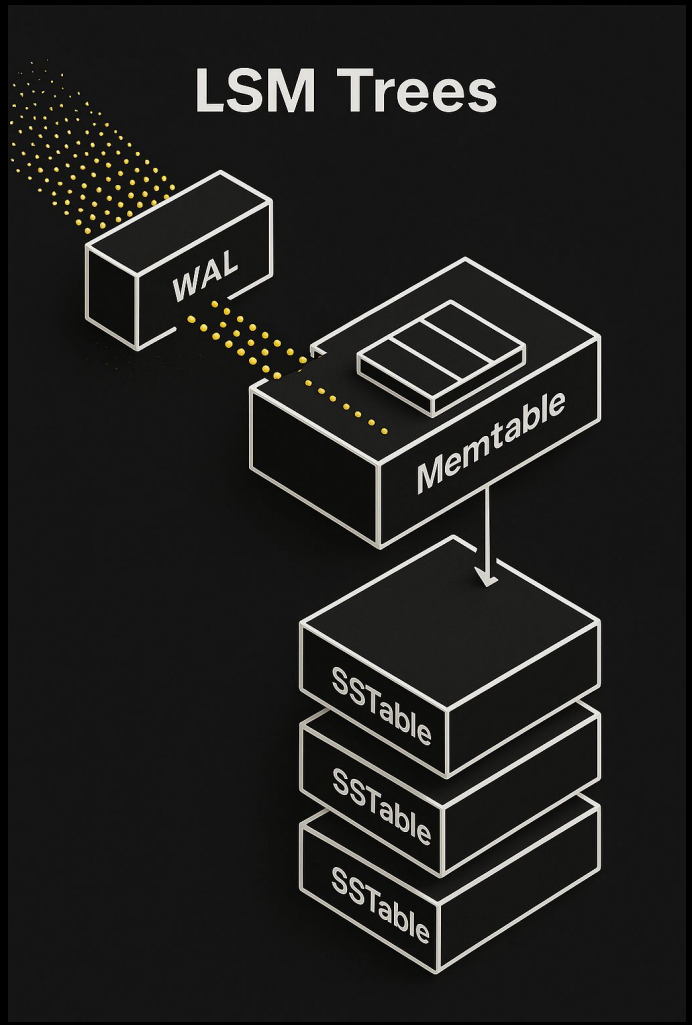
Stores data **directly flushed** from the Memtable.

LEVEL 1

All SSTables from Level 0 are **merged into a single, larger, but more compact** SSTable in Level 1, which is the next level down.

Further Levels

This process happens at each level, reducing the sparsity of SSTables each time.



HOW IT WORKS - SSTABLE FILTERS

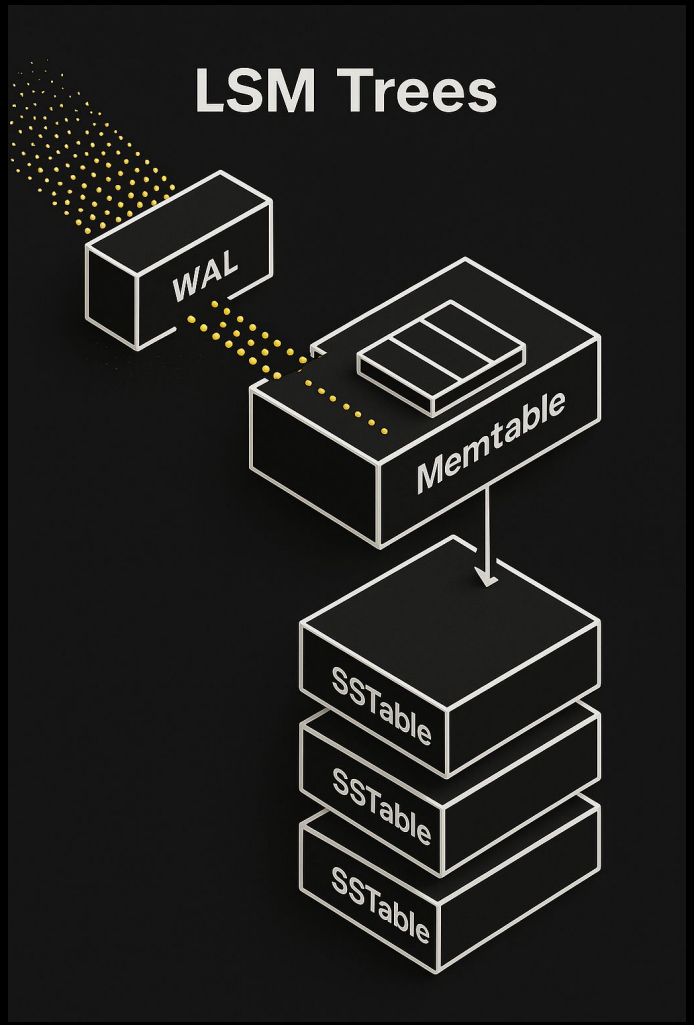
Despite their sorted nature, there is no way to determine if a key exists within an SSTable—or if we have the latest version—without scanning all SSTables until it is found.

Two scenarios:

Range Scans may require scanning across multiple tables to find multiple values.

Scanning for non-existent data requires scanning all tables before confirming a key does not exist.

Other data structures, like B-Trees, do not suffer from this problem as their data is not stored across multiple files like with LSM Trees.



HOW IT WORKS - SSTABLE FILTERS

Scanning for non-existent data requires scanning **all** tables.

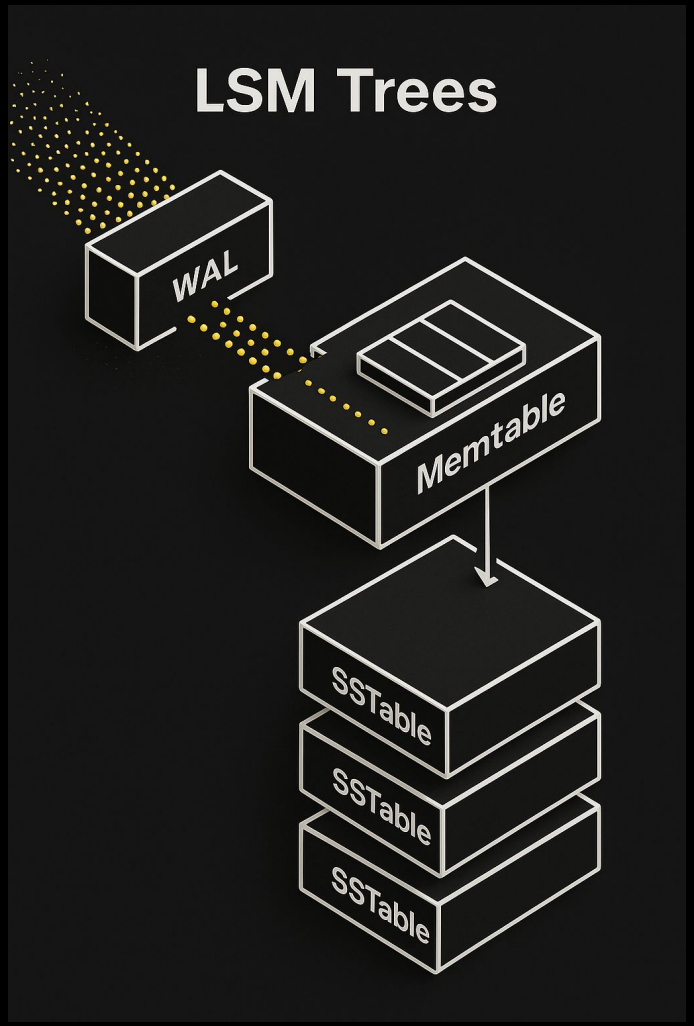
Wouldn't it be nice if we could just tell if a key does not exist in a table? That would make the operation **constant time** instead of **linear**.

We can — by using Bloom Filters!

BLOOM FILTERS

Bloom filters probabilistic data structure.

Quickly indicate whether a key **may exist** in a given SSTable **or definitely does not exist**.



HOW IT WORKS - SSTABLE FILTERS

Range Scans may require scanning across multiple tables.

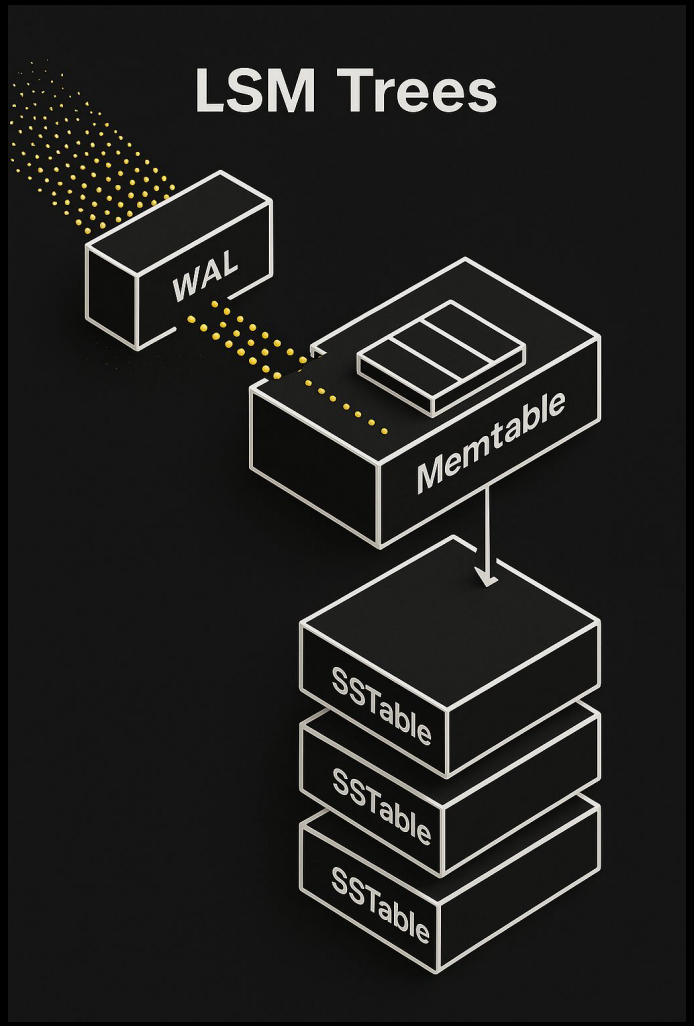
Wouldn't it be nice if we could just seek to a value in an SSTable?

We can (sort of) — by using Sparse Indexes!

SPARSE INDEXES

Sparse indexes point to the **approximate location** of a key within an SSTable.

Enables **fast, targeted access** to keys, drastically reducing search time.



HOW IT WORKS - SSTABLE FILTERS

Together, Bloom Filters and Sparse Indexes help us overcome the inefficiencies with reading across multiple SSTable files.

BLOOM FILTERS

Bloom filters probabilistic data structure.

Quickly indicate whether a key **may exist** in a given SSTable **or definitely does not exist**.

SPARSE INDEXES

Sparse indexes point to the **approximate location** of a key within an SSTable.

Enables **fast, targeted access** to keys, drastically reducing search time.

SCOPE AND LIMITATIONS

SCOPE AND LIMITATIONS

SCOPE

LSM Trees are:

- **Designed for sustained, high-volume writes with durable storage**; not for workloads dominated by point reads or in-memory caching.
- **Built to handle datasets far larger than available RAM**; not for data that fits entirely in memory.

Workloads must justify their requirements for high write throughput and data volume before even considering LSM Trees.

LIMITATIONS

- There is inherent increased disk I/O when performing compactions; schedule compactions wisely.
- Without proper SSTable management policies, read latency grow unbounded as the number of tables increase.
- Consider extra space needed for comaction when selecting storage.

CODE IMPLEMENTATION

IMPLEMENTATION - BACKENDS

WAL

WAL implements the append method.

append accepts a key and value, as well as timestamp, known as a record. It writes the record to disk in a binary format.

It encodes the type of operation in an op code; either PUT or DELETE.

In the event of a crash, this log is replayed to recover the state of the Memtable.

```python
def append(self, key: Key, value: Value | None, ts: Timestamp) -> int:
    """Append a record to WAL.

    Args:
        key: Binary key
        value: Binary value or None for tombstone
        ts: Monotonic timestamp

    Returns:
        WAL sequence number
    """
    if self._fd is None:
        raise RuntimeError("WAL is closed")  # noqa: TRY003

    op_code = OP_DELETE if value is None else OP_PUT
    value_bytes = value if value is not None else b""

    # Build record
    key_len = len(key)
    value_len = len(value_bytes)

    # Pack payload (without CRC)
    payload = struct.pack("<I", MAGIC)
    payload += struct.pack("<Q", key_len)
    payload += key
    payload += struct.pack("<Q", value_len)
    payload += value_bytes
    payload += struct.pack("<Q", ts)
    payload += struct.pack("<B", op_code)

    # Calculate CRC32 of payload
    crc = zlib.crc32(payload)
    record = payload + struct.pack("<I", crc)

    # Write to file
    self._fd.write(record)
    if self.flush_every_write:
        self._fd.flush()
        os.fsync(self._fd.fileno())

    self.sequence += 1
    logger.debug(f"Appended record seq={self.sequence}, key_len={key_len}, ts={ts}")

    return self.sequence
```

IMPLEMENTATION - BACKENDS

MEMTABLE

Memtable implements the put, delete, get and clear methods.

put accepts a record and appends it to the Memtable in **sorted order**.

delete accepts a key to delete and sets its value to None.

get returns a key and its associated value if the key exists; otherwise it returns None.

clear deletes all Memtable entries, freeing it for further writes.

Design Decision

Either sort entries on write, or sort them on flush.

This implementation uses SortedDict to sort entries on write.

```python
class SimpleMemtable:
    """In-memory sorted structure holding recent writes.

    Maintains key-value pairs with timestamps in sorted order.
    Supports efficient range queries and size tracking.

    Invariants:
        - Keys are always maintained in sorted order
        - Most recent value per key (by timestamp) is accessible
        - Size includes approximate overhead of data structures
    """

    def __init__(self):
        """Initialize empty memtable."""
        self._data: SortedDict = SortedDict()
        self._size_bytes: int = 0

    def put(self, key: Key, value: Value, ts: Timestamp) -> None:
        """Insert or update key with value and timestamp."""
        # Track size delta
        if key in self._data:
            old_value, _old_ts = self._data[key]
            old_size = len(key) + (len(old_value) if old_value else 0) + 8
            self._size_bytes -= old_size

        self._data[key] = (value, ts)
        new_size = len(key) + len(value) + 8  # key + value + timestamp
        self._size_bytes += new_size

    def delete(self, key: Key, ts: Timestamp) -> None:
        """Mark key as tombstone with timestamp."""
        # Track size delta
        if key in self._data:
            old_value, _old_ts = self._data[key]
            old_size = len(key) + (len(old_value) if old_value else 0) + 8
            self._size_bytes -= old_size

        self._data[key] = (None, ts)
        new_size = len(key) + 8  # key + timestamp (no value)
        self._size_bytes += new_size

    def get(self, key: Key) -> tuple[Value | None, Timestamp] | None:
        """Return (value_or_none, timestamp) if key found; else None.

        If value_or_none is None, it is a tombstone.
        """
        return self._data.get(key)
```

IMPLEMENTATION - BACKENDS

SSTABLE

SSTable implements the add, may_contain, and get methods.

add is only used when flushing Memtables to disk.

It reads a record from the Memtable, serializes it into a binary format, and then appends it to the SSTable on disk.

add also ensures sorted order.

Since we push the sorting process to the Memtable, this operation runs in **constant time**.

Finally, each add operation creates a Bloom Filter entry, and potentially one for the Sparse Index.

```python
def add(self, key: Key, value: Value | None, ts: Timestamp) -> None:
    """Append record to the writer (must be added in sorted order)."""
    # Verify sorted order
    if self._last_key is not None and key <= self._last_key:
        raise SSTableError("Unsorted keys")  # noqa: TRY003

    # Record current offset
    if self._fd is None:
        msg = "Writer closed"
        raise SSTableError(msg)
    fd = self._fd
    offset = fd.tell()

    # Update metadata
    if self._min_key is None:
        self._min_key = key
    self._max_key = key

    if self._min_ts is None or self._max_ts is None:
        self._min_ts = ts
        self._max_ts = ts
    else:
        self._min_ts = min(self._min_ts, ts)
        self._max_ts = max(self._max_ts, ts)

    # Sample for sparse index
    if self._count % self.index_interval == 0:
        self._index.append((key, offset))

    # Collect for bloom filter
    self._keys_for_bloom.append(key)

    # Write record
    value_bytes = value if value is not None else b""
    key_len = len(key)
    value_len = len(value_bytes)

    fd.write(struct.pack("<Q", key_len))
    fd.write(key)
    fd.write(struct.pack("<Q", value_len))
    fd.write(value_bytes)
    fd.write(struct.pack("<Q", ts))

    self._count += 1
    self._last_key = key
```

IMPLEMENTATION - BACKENDS

SSTABLE

SSTable implements the add, may_contain, and get methods.

may_contain checks the Bloom Filter for a key.

It returns True if a key may exist, and False otherwise.

```python
def may_contain(self, key: Key) -> bool:
    """Use Bloom filter to test potential presence."""
    # Check key range first
    if self._min_key and key < self._min_key:
        return False
    if self._max_key and key > self._max_key:
        return False
    return key in self._bloom
```

IMPLEMENTATION - BACKENDS

SSTABLE

SSTable implements the add, may_contain, and get methods.

get returns the value of an associated key if the Bloom Filter passes, and utilizes the Sparse Index to quicken lookups.

Binary Search

Read operations capitalize on the sorted nature of SSTables, utilizing **binary search** for **logarithmic** lookups.

```python
def get(self, key: Key) -> tuple[Value | None, Timestamp] | None:
    """Return (value_or_none, ts) or None if key definitely not present."""
    if not self.may_contain(key):
        return None

    self._ensure_open()

    # Binary search in sparse index
    offset = self._find_block_offset(key)

    # Scan from offset
    if self._fd is None:
        self._ensure_open()
    assert self._fd is not None
    self._fd.seek(offset)

    while True:
        # Try to read a record
        key_len_bytes = self._fd.read(8)
        if len(key_len_bytes) < 8:
            break  # EOF

        record_key_len = struct.unpack("<Q", key_len_bytes)[0]
        record_key = self._fd.read(record_key_len)

        if len(record_key) < record_key_len:
            break  # Corrupted

        # If we've passed the key, it doesn't exist
        if record_key > key:
            break

        # Read value and timestamp
        value_len = struct.unpack("<Q", self._fd.read(8))[0]
        value_bytes = self._fd.read(value_len)
        ts = struct.unpack("<Q", self._fd.read(8))[0]

        if record_key == key:
            value = value_bytes if value_len > 0 else None
            return (value, ts)

    return None
```

IMPLEMENTATION - BACKENDS

COMPACTOR

The compactor takes in a sequence of SSTables at a given level and merges them into a single SSTable.

While merging, it maintains sorted order, and prunes deleted or outdated records.

```python
def compact(
    self, input_tables: Sequence[SSTableMeta], target_level: int
) -> Sequence[SSTableMeta]:
    """Perform merge compaction and write output SSTables.

    Args:
        input_tables: SSTables to compact
        target_level: Target level for output

    Returns:
        List of produced SSTable metadata
    """
    if not input_tables:
        return []

    logger.info(f"Compacting {len(input_tables)} SSTables to level {target_level}")

    # Open all input SSTables
    readers: list[SSTableReader] = []
    for meta in input_tables:
        reader: SSTableReader = SimpleSSTableReader(meta["data_path"], meta["meta_path"])
        readers.append(reader)

    try:
        # Merge all iterators
        merged = self._merge_iterators(readers)

        # Write to output SSTables
        output_metas = self._write_output(merged, target_level)

        logger.info(f"Compaction produced {len(output_metas)} SSTables")
        return output_metas

    finally:
        # Close all readers
        for reader in readers:
            reader.close()
```

IMPLEMENTATION - BACKENDS

COMPACTOR

This implementation uses **min heaps** under-the-hood to maintain sorted order across SSTables.

Note that due to implementation specifics, the overall time complexity for compaction is **approximately linear - not quasilinear** as you may expect for heaps.

```python
def _merge_iterators(self, readers: Sequence[SSTableReader]) -> Iterator[Record]:
    """Merge multiple sorted iterators, keeping newest by timestamp."""
    # Build heap of (key, -ts, value, reader_idx, iterator)
    heap = []
    iterators = [reader.iter_range(None, None) for reader in readers]

    # Initialize heap
    for idx, it in enumerate(iterators):
        try:
            key, value, ts = next(it)
            heapq.heappush(heap, (key, -ts, value, idx, it))
        except StopIteration:  # noqa: PERF203
            pass

    last_key = None
    last_value = None
    last_ts = None

    while heap:
        key, neg_ts, value, idx, it = heapq.heappop(heap)
        ts = -neg_ts

        # Advance iterator
        try:
            next_key, next_value, next_ts = next(it)
            heapq.heappush(heap, (next_key, -next_ts, next_value, idx, it))
        except StopIteration:
            pass

        # Emit if key changed and we have a value to emit
        if last_key is not None and key != last_key:
            # Check if tombstone should be kept
            assert last_ts is not None
            if last_value is not None or self._should_keep_tombstone(last_ts):
                yield (last_key, last_value, last_ts)
            last_key = None

        # Update last seen (highest timestamp wins)
        if last_key is None or key != last_key:
            last_key = key
            last_value = value
            last_ts = ts
        elif ts > last_ts:
            last_value = value
            last_ts = ts

    # Emit final record
    if (
        last_key is not None
        and (
            last_value is not None
            or (last_ts is not None and self._should_keep_tombstone(last_ts))
        )
    ):
        assert last_ts is not None
        yield (last_key, last_value, last_ts)
```

BIG_O ANALYSIS

BIG_O ANALYSIS

The WAL has two operations: **append** and **replay**.

Append: Add entry to log
Replay: Reads the log from start to end.

WRITE-AHEAD LOG (WAL)

Append operations run in **constant time**.

Replay operations run in **linear time**.

	Best Case	Average Case	Worst Case
Append	O(1)	O(1)	O(1)
Replay	O(n)	O(n)	O(n)

BIG_O ANALYSIS

The Memtable has five operations: **put, delete, get, scan** and **flush**.

MEMTABLE

Implementation uses SortedDict - average **logarithmic** time for all operations.

Flush must pop all elements - **linear** time.

n - total number of records
k - range of records to scan

	Best Case	Average Case	Worst Case
Put	O(1)	O(log n)	O(log n)
Delete	O(1)	O(log n)	O(n)
Get	O(1)	O(log n)	O(n)
Scan	—	O(k log n)	O(n*k)
Flush	—	O(n)	O(n)

BIG_O ANALYSIS

The SSTable has three operations: **point read, range read and compact**.

SSTables

Reads employ **Binary Search**, therefore **logarithmic time.**

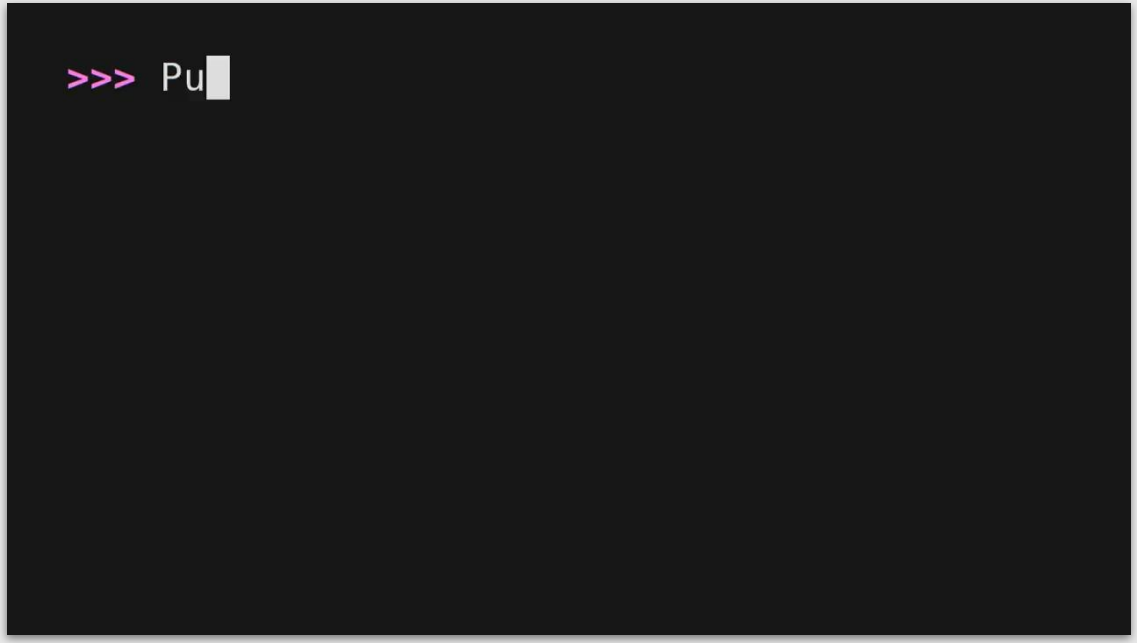
Search against Sparse Indexes + Bloom Filter

n - number of SSTables
k - number of records in an SSTable
b - SSTable block size
l - number of index entries in the sparse index (< k)

	Best Case	Average Case	Worst Case
Point	O(1)	O(log l + b)	O(log l + b)
Range	O(1)	O(log l + b)	O(log l + b)
Compact	—	O(n*k)	O(n*k)

DEMO

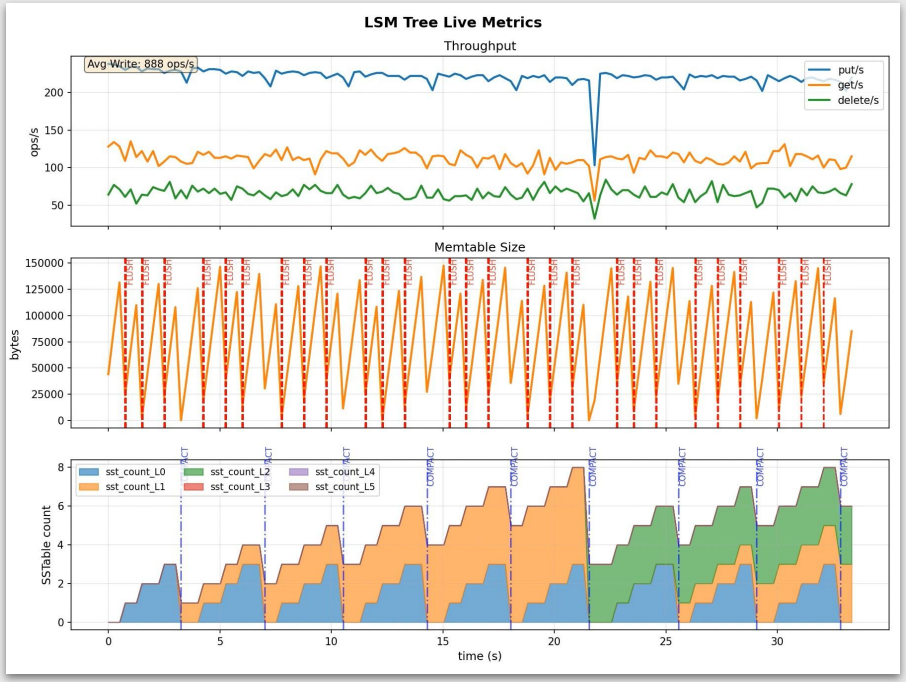
DEMO - CLI



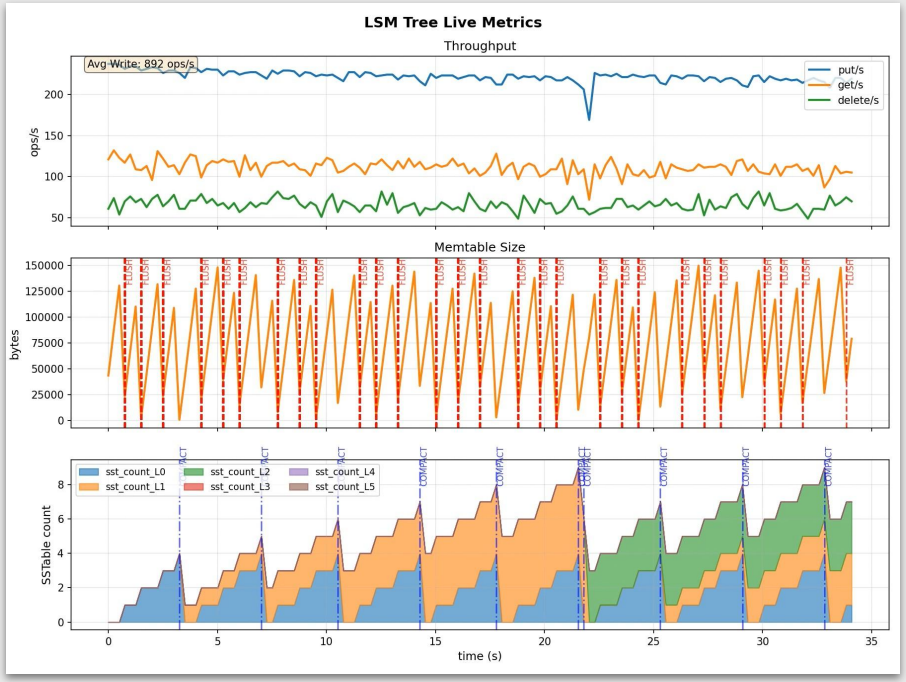
Interacting with the LSM Tree via Python CLI

DEMO - BENCHMARKS

SYNCHRONOUS VS ASYNCHRONOUS COMPACTION



Synchronous Compaction

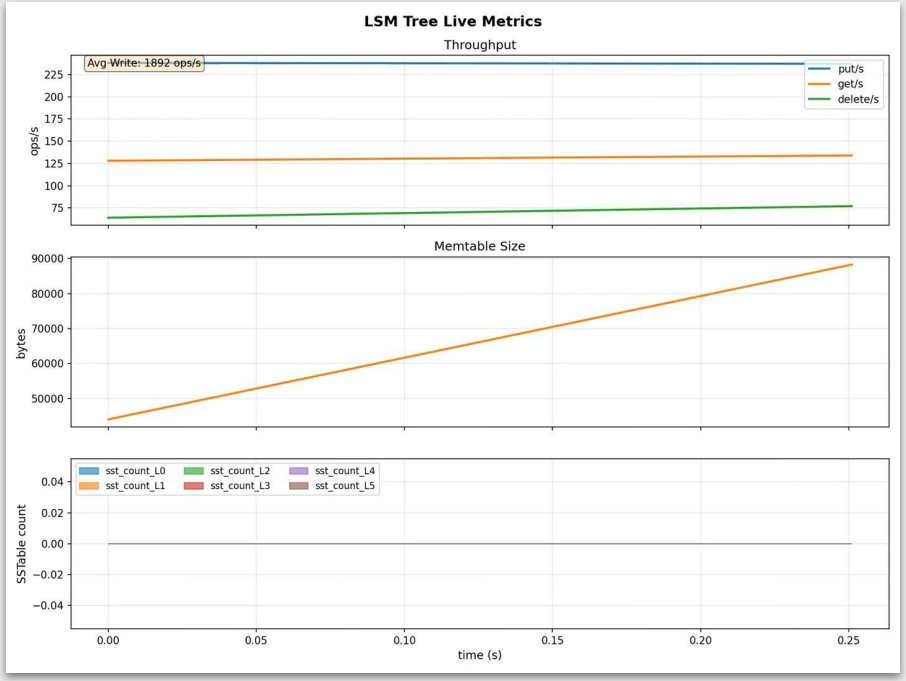


Asynchronous Compaction

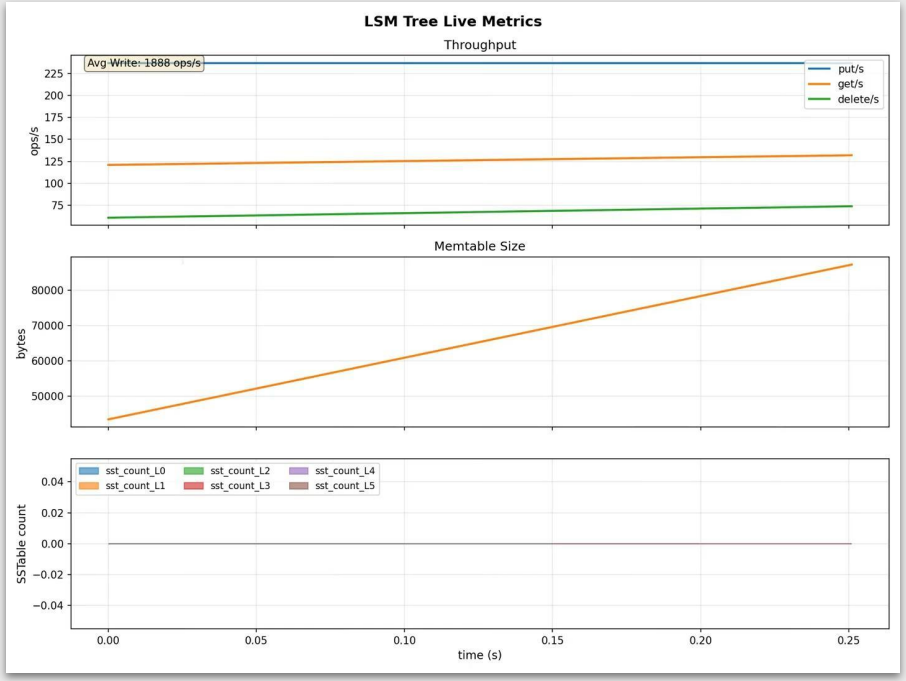
Memtable size reduced to force frequent flushes and compactions

DEMO - BENCHMARKS

SYNCHRONOUS VS ASYNCHRONOUS COMPACTION



Synchronous Compaction

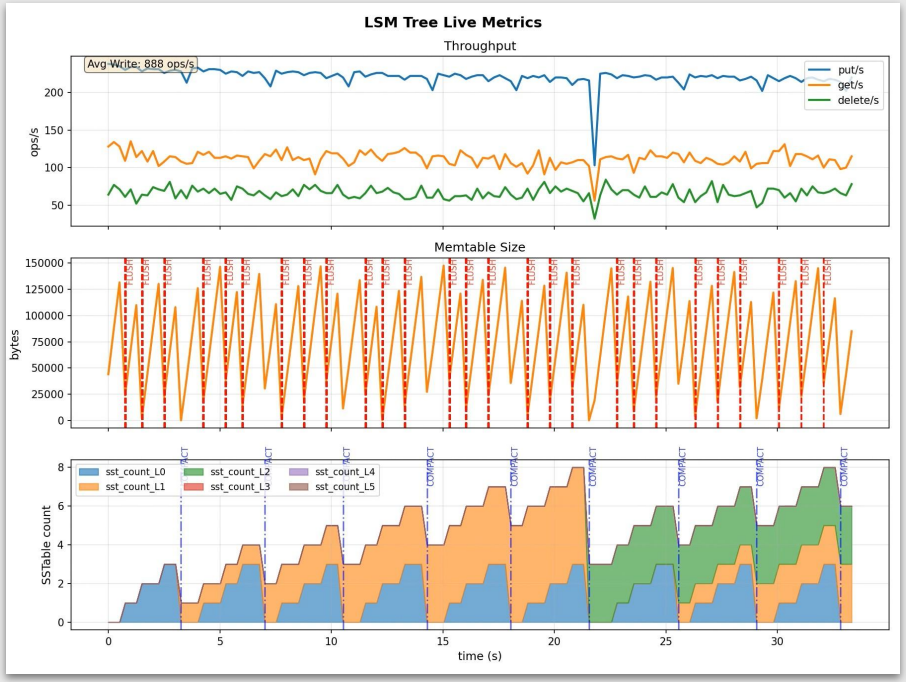


Asynchronous Compaction

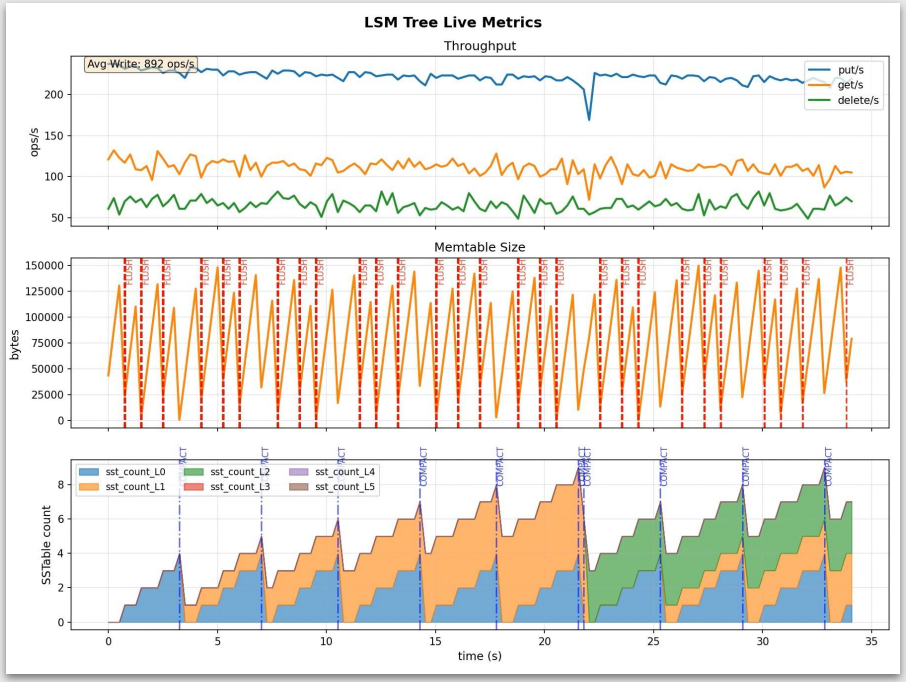
Memtable size reduced to force frequent flushes and compactions

DEMO - BENCHMARKS

SYNCHRONOUS VS ASYNCHRONOUS COMPACTION



Synchronous Compaction



Asynchronous Compaction

Memtable size reduced to force frequent flushes and compactions

IMPROVEMENTS

1. Block Cache

Introduce an **LRU-based cache** for frequently accessed blocks or index pages. This could drastically reduce SSTable read latency.

2. Concurrency

Apply **MVCC** to allow non-blocking reads during ongoing writes, flushes, or compactions.

3. Compaction Policies

Add adaptive policies (size-tiered, hybrid, or leveled with configurable thresholds) to fit different performance targets.

THANK YOU!