UNIVERSITY OF CALIFORNIA

SANTA CRUZ

# CooLSM: Distributed and Cooperative Indexing Across Edge and Cloud Machines

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

**Natasha Mittal**

June 2020

The Thesis of Natasha Mittal
is approved:

Professor Faisal Nawab, Chair

Professor Heiner Litz

Professor Chen Qian

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

## Table of Contents

# List of Figures

# ABSTRACT

## CooLSM: Distributed and Cooperative Indexing Across Edge and Cloud Machines

Natasha Mittal

Emerging edge applications such as the Internet of Things (IoT) and Industry 4.0 require stringent real-time guarantees. This makes it infeasible to rely solely on faraway cloud nodes for these applications. Similarly, edge machines are less capable and reliable than cloud machines. To balance the trade-off of edge and cloud computing, we propose a data infrastructure that spans both edge and cloud machines, that we call edge-cloud computing. We tackle one of the fundamental data management challenges in edge-cloud computing, the problem of data indexing. We propose Cooperative LSM [3] (CooLSM), a distributed Log-Structured Merge Tree that is designed to overcome the unique challenges of edge-cloud indexing such as machine and workload heterogeneity and the communication latency asymmetry between the edge and the cloud. To tackle these challenges, CooLSM deconstructs the structure of LSM into its basic parts. This deconstruction allows a better distribution and placement of resources across edge and cloud devices. For example, append-specific functionality is managed at the edge to ensure appending and serving data in real-time, whereas resource-intensive operations such as compaction are managed at the cloud where more computing resources are available.

# Acknowledgments

I would like to thank my advisor, Professor Faisal Nawab, for introducing me to this research area and for his advice, guidance, careful editing, and patience during the preparation of this thesis.

I am grateful to my parents, my brother, and my friends, for their continuous support and encouragement during my graduate studies at UCSC.

# Section 1: Introduction

## 1.1 Motivation

Cloud computing suffers from a fundamental limit—the speed of light. This restricts applications that require stringent real-time guarantees. Similarly, relying solely on edge computing is not suitable due to the lower capabilities and reliability of edge devices. To overcome this dilemma, we consider an edge-cloud model, where the data infrastructure spans both edge and cloud devices. Ideally, edge devices would handle the real-time aspects of processing, while more resource-intensive processing and long-term storage are performed on cloud machines.

The edge-cloud model has the potential of advancing emerging edge and Internet of Things (IoT) applications such as smart cities and Industry 4.0, by supporting both real-time actions at the edge and more complicated processing at the cloud. For example, consider a surveillance application in a smart city, where thousands of smart cameras are continuously generating labels and information of detected objects. The generated data might be used for real-time actions, such as changing the flow of traffic via signals to autonomous cars when certain labels are detected. This type of processing is time-critical and must be performed at the edge to avoid waiting for sending data to and from the cloud nodes.  Furthermore, the collected data might also be used to generate deeper insights, such as a query to analyze which areas of the city were most busy during the last week. This type of processing is not time-critical and is more resource-intensive, which makes it suitable to be done at the cloud.

## 1.2 Approach

We propose Cooperative LSM [3] (CooLSM), which aims to provide one of the main building blocks of edge-cloud data management systems, which is a distributed data indexing system. Data indexing is the problem of managing storage and access to data. We build on Log-Structured Merge (LSM) trees, which is among the most widely used data indexing technologies. Its main design advantage is that it enables fast ingestion of data. Fast data ingestion is important for edge and IoT applications, which influences our decision to build on LSM trees. However, most LSM-based technologies are designed to reside within a single machine — and the distributed variants are designed for clusters of neighboring machines. This limits the applicability of these solutions to edge-cloud environments, where processing and storage span both edge and cloud nodes across wide-area links.

CooLSM's main design principle is to deconstruct the monolithic structure of LSM trees into smaller basic components. This deconstruction allows distributing and placing components to maximize the potential and utility of edge and cloud nodes. Additionally, the deconstruction allows elastically scaling the resources for each component without affecting the other components.

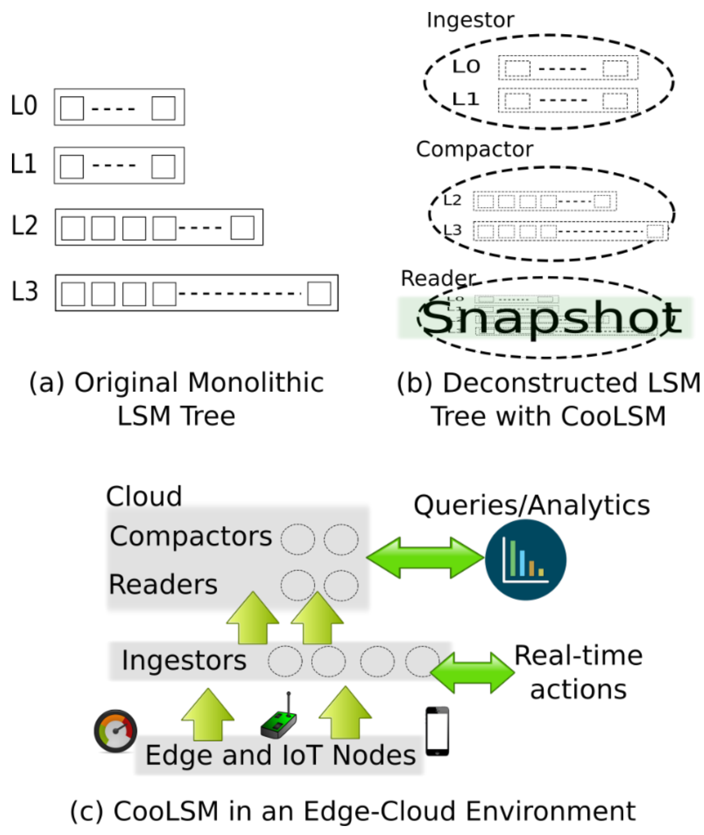

*Figure 1: CooLSM*

Specifically, CooLSM—as shown in Figure 1—breaks down an LSM tree — in Figure 1(a)—into

(1) Ingestors: nodes that receive the data to be added to the index and requests to access the data,

(2) Compactors: nodes that handle the structuring and garbage collection of the index, and

(3) Readers: nodes that maintain recent snapshots of the data to serve efficient analytics and complex read queries.

This deconstruction enables the distribution and placement of nodes, as shown in Figure 1(c), for an edge-cloud environment. Ingestors are placed near sources of data to enable real-time ingestion and actions. Ingestors forward data to compactors and readers at the cloud to enable long-term storage and management of more complex tasks such as garbage collection and large read-only queries.

One of the main challenges that we face is the trade-off between consistency and performance. As we deconstruct the LSM tree and distribute its components across wide-area links, performing read and write operations while maintaining linearizability would require extensive coordination and overhead. Our design is motivated by wanting to support real-time operation, and therefore, we elect to weaken consistency to improve performance. Specifically, we formulate two notions of consistency that relax linearizability to allow us to achieve better performance while having a notion of the consistency guarantees achieved. The first consistency notion we introduce is **Snapshot Linearizability** [4], which aims to model the potential inconsistency resulting from lazily forwarding records to an offline reader. The second consistency notion is the **Linearizable** [4] **+ Concurrent** model, which aims to model the potential inconsistency that results from having more than one Ingestor (i.e., more than one entry point to the LSM tree).

To summarize, CooLSM tackles the following challenges that we address:

1. Breaking LSM into smaller parts that are distributed across machines introduces the need for **safe coordination protocols** between the various components.
2. We want each component type to be **elastic**, for example, the number of compactors can be adjusted dynamically in response to changing workload characteristics without affecting the other components types.

3. Introducing distribution, dedicated readers, and multiple Ingestors bring up interesting consistency challenges that require **new consistency models**. Specifically, we discuss **Snapshot Linearizability** to reason about the correctness of operations on Readers and introduce the **linearizable + concurrent** model to reason about the correctness of systems with multiple Ingestors.

## 1.3 Outline of the Thesis

In the thesis, we begin with presenting background on B+ Trees and LSM trees in Section 2. Then, we propose the CooLSM design in Section 3. Section 4 presents our experimental evaluation. The thesis concludes with related work in Section 5, and a summary in Section 6.

## Section 2: Background

In this section, we will be discussing two database indexing structures in detail: B+ Trees and LSM Trees and then compare them based on the read and write time complexities. We will also briefly talk about different compaction techniques used in LSM trees.

### 2.1 B+ Trees

A Balanced Search Tree or a B-Tree is a tree that searches for any value and will require the same number of nodes to be read from the disk. A B+ Tree is similar to a B-Tree with some additional features, which makes them more efficient. B+ Tree is a popular indexing structure used in databases. It is an m-ary tree data structure comprising of a root node, intermediate or internal nodes, and leaf nodes.

Each tree node contains a sorted list of keys and pointers to the lower level of nodes. These keys and pointers are arranged alternately: $< \text{pointer}, \text{key}, \text{pointer}, \ldots, \text{pointer} >$ with $k$ keys and maximum $n\ (k+1)$ pointers. The leaf nodes contain a next sibling pointer, which points to the next leaf in the tree. This connects all the leaf nodes of a B+ Tree like a linked list, and a linear scan of key values can be done in a single pass of reading from the disk. The insertion and deletion algorithms of B+ Tree are designed so that the tree maintains a fixed height of $\log_{(n/2)}(k)$ where $k$ is the number of keys, and $n$ is the number of pointers in a node.

- **Structure of an internal node of B+ Tree:**
    1. Each internal node is of the form: $< P_1, K_1, P_2, K_2, \dots, K_k, P_{k+1} >$ where $P_i$ is a tree pointer to the lower-level node and $K_i$ is the key and $k + 1 = n$.
    2. Every node is an ordered list of keys, so $K_1 < K_2 < \cdots < K_k$
- **Structure of a leaf node of B+ Tree:**
    1. Each leaf node is of the form:$<< K_1, D_1 >, < K_2, D_2 >, \dots , < K_m, D_m >, < P_{next} >$ where each $< K_i, D_i >$ is a key, and data pointer pair and $P_{next}$ is the next sibling pointer pointing to the next leaf in sequence.
    2. Every leaf node is also an ordered list of keys, so $K_1 < K_2 < \cdots < K_m$.
    3. Since all the leaf nodes are at the same level, so root to leaf path remains constant in a B+ Tree.
- **Properties of B+ Trees**
    1. All internal and leaf nodes in B+ Tree can be at least half full. When a node is full, it is split into two nodes, each containing half of the keys. When keys are deleted, nodes are merged, and the tree structure is re-organized to make sure that each node is at least half full.
    2. B+ Tree grows in an upward direction. This is because all the insertion and deletion of the keys are done at the leaf node, so the splitting and the merging of the nodes to re-structure the tree are done from leaf nodes to the root node in an upward fashion.

3. Since all leaf nodes are at the same level, so the length of root to leaf path is constant in an n-level B+ Tree. This makes it a balanced tree because searching any value 'x' will require the same number of nodes to be read from the disk.
4. The leaf nodes of a B+ Tree are connected in a linked list fashion via next sibling pointers, which makes sequential key searches very efficient.

### 2.1.1 B+ Tree Insertion

**Algorithm 1** B+ Tree Insertion

```
 1: procedure INSERT(record)
 2:     if current leaf is not full then
 3:         add record to node
 4:     else
 5:         split the current leaf
 6:         create new leaf
 7:         move half records to new leaf
 8:         insert smallest <key,address> of new leaf into
    parent
 9:         while parent is full do
10:             add middle key to parent
11:     if root splits then
12:         create new root with one key and two pointers
```

### 2.1.2 B+ Tree Deletion

**Algorithm 2** B+ Tree Deletion

```
 1: procedure DELETE(k)
 2:     begin at root and search down till leaf node containing key k
 3:     find node n containing k
 4:     if n is root then
 5:         if n has more than one key then
 6:             remove k
 7:         else
 8:             remove k
 9:             if child node can lend key then
10:                 borrow key from child
11:                 adjust child links
12:             else
13:                 merge child nodes to create new root
14:     else if n is internal node then
15:         if n has $ceil(k/2)$ keys then
16:             remove k
17:         else
18:             remove k
19:             if sibling can lend key then
20:                 borrow key from sibling
21:                 adjust keys in n and parent node
22:                 adjust child links
23:             else
24:                 merge n with its siblings
25:                 adjust child links
26:     else if n is leaf then
27:         if n has $ceil(k/2)$ keys then
28:             remove k
29:             if k == smallest key of n then
30:                 push new key to upper nodes
31:         else
32:             if sibling can lend key then
33:                 borrow key from sibling
34:                 adjust keys in n and its parent
35:             else
36:                 merge n with its siblings
37:                 adjust keys in parent
```

## 2.2 LSM Tree

Log-Structured Merge (LSM) Trees was introduced by Patrick O'Neil et al. in 1996, and a decade later, Google published its "*BigTable*" [5] paper. BigTable is a distributed storage system used for managing structured data and in which files are organized using LSM trees, and this paper popularized LSM trees as an indexing data structure.

LSM-tree is an out-of-place update structure, where every update is appended as a new entry (it is like appending logs in a log file) rather than updating the old entry. This design exploits the sequential I/O nature of the underlying disk, making LSM-tree ideal for write-intensive workloads. It also simplifies the recovery process as old data is not modified. However, continuous appending of new entries results in multiple log files, which affects the read performance since the same entry may be stored at multiple locations. So, to improve the read-latency and keep LSM-tree within a memory range, continuous merging/compaction operation is to be carried out in the background.

LSM-tree is composed of components $C_0, C_1, \dots, C_k$, where each component itself is a B+ tree. $C_0$ resides in memory while all other components reside on disk. When a component $C_i$ is full, the rolling merge operation is triggered to merge the contents of $C_i$ into the $C_{i+1}$ component. The rolling-merge process is complex, and modern LSM-tree variants use other simpler and effective compaction techniques.

### 2.2.1 LevelDB [6]

LevelDB is an open-source key-value store implemented by Google after it is BigTable paper. LevelDB presents the implementation of LSM Tree, which is being used as the base implementation for numerous LSM variants.

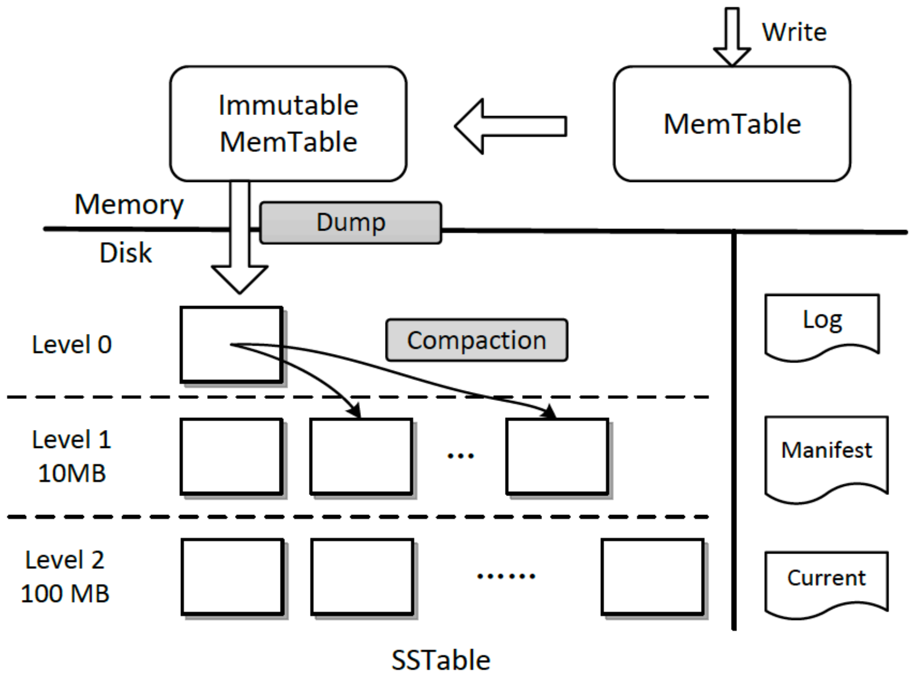

*Figure 2* [7]*: LevelDB Architecture*

- **LSM Components:**
  1. **Memtable:** this is the memory-resident component of the LSM tree where incoming updates are batched.
  2. **Immutable Memtable:** this is the memory-resident component which is created when memtable reaches its capacity
  3. **SSTable (Sorted-Strings Table):** stores a sequence of date entries sorted by their keys
  4. **Level:** it is a sorted sequence of non-overlapping stables

5. **Fence Pointers:** a directory of $< first\ key, last\ key >$ is maintained in memory for each SSTable for every level
6. **Bloom Filter:** associated with each SSTable to check if the key is present in the SSTable without actually reading the SSTable from disk.

- **Upsert Flow:**

1. A $< key, value >$ pair is first inserted into the Memtable, which holds the most recent updates. When the Memtable reaches its threshold capacity, it is transformed into a read-only Immutable Memtable, and correspondingly, a new Memtable is created to receive fresh updates.
2. A separate thread runs, which flushes Immutable Memtable into a disk-resident SSTable, which is organized into a series of levels.
3. Level 0 is the youngest level, which contains SSTables produced by flushing Immutable Memtable. Thus, SSTables in Level 0 has overlapping keys.
4. Higher levels contain non-overlapping SSTables, and each level has a fixed threshold on the number of SSTables it can accommodate. The size of levels grows exponentially with the level number.
5. Eventually, levels will fill up, and in order to maintain LSM within size limits, compaction is done. A separate compaction thread runs in the background, which monitors the levels of LSM.
6. When $Level\ i$ reaches its limit, the compaction thread picks one SSTable from $Level\ i$ and all overlapping SSTables from $Level\ i + 1$ and merge them into a sequence of SSTables, which are then pushed to $Level\ i + 1$.

7. The most recent data resides in lower levels while stale data gradually move to higher levels.

- **Read Flow:**
  1. The key is first searched in Memtable, and if not found, it is searched in Immutable Memtable.
  2. Since Level 0 SSTables have overlapping keys, each SSTable is searched until the key is found.
  3. For higher levels, the appropriate SSTable is first identified using fence pointers. Then the bloom filter is used to confirm if the key is present in the SSTable and if it is, then the SSTable is read from the disk and searched to find the key.

## 2.3 Bloom Filter [8]

Bloom filter, a probabilistic data structure, is used to test set membership. It supports two operations: **test** and **add**.

**Test** is used to check if a given element is a member of the set. If it returns:

- false, then the element is not in the set
- true, then the element is probably present in the set.

Bloom Filter has a false positive rate, which depends on the size of the Bloom filter, the number of hash functions, and the independence of these hash functions.

**Add** operation simply adds the element to the set. Bloom filter does not support delete operation because it introduces false negatives.

The main advantage of Bloom Filter is that it helps in reducing expensive disk lookups. If the item is not in the Bloom filter, then it is assured that we do not need to perform an expensive lookup. If it is present, then only we perform a lookup, but it can fail in some cases due to Bloom Filter's false positive rate.

Algorithm:

Every Bloom filter comprises of two elements:

1. A bit array of size n, where all bits are set to 0 initially. Generally, n is chosen to be much greater than the number of elements to be inserted in the set.
2. A collection of $k$ hash functions $h(x)$. Each hash function $h(x)$ takes a value $v$ and generates a number $i$, such that $i < n$, which maps value $v$ to a position $i$ in the bit array.

- **Add Operation:**

Each hash function $h(x)$ is applied to the new element $v$, which generates a value $i$. This $i$ value generated is used as an index in the bit array, and the corresponding bit position is set to 1. If we use $k$ hash functions, $k$ indices are generated, and correspondingly $k$ positions are set to 1 in the bit array.

- **Test Operation:**

To check if an element $v$ exists in the set, $k$ hash functions are applied on $v$. If at least one of the $k$ indices in the bit array is set to 0, then the given element $v$ is not present in the set, else it is an existing element.

- **Bloom Filter's False Positive Rate:**

We are considering two hash functions $h(x)$ and $g(x)$:

- $\boldsymbol{h(x)}$**:**

1. Compute the binary equivalent of element $x$. Let us call it $b$.
2. Take odd-numbered bits from $b$ to form a number $y$.
3. Take $y$ modulo 11 to generate index $i$ in the bit array.

- $\boldsymbol{g(x)}$**:**

1. Compute the binary equivalent of element $x$. Let us call it $b$.
2. Take even-numbered bits from $b$ to form a number $y$.
3. Take $y$ modulo 11 to generate index $i$ in the bit array.

For $x = 25{,}159{,}585$:

| **x** | **b** | **h(x)** | **g(x)** | **bit array** |
|---|---|---|---|---|
| 25 | 0000011001 | 5 | 2 | 00100100000 |
| 159 | 0010011111 | 0 | 7 | 10100101000 |
| 585 | 1001001001 | 7 | 9 | 10100101010 |

- **Normal case:**

To check for $x = 118, h(118), and\ g(118)$ generates 5 and 3, respectively.

Now,

- bit_array[5] = 1
- bit_array[3] = 0

So, element 118 does not exist in the set.

- **False-positive case:**

To check for $x = 162, h(162), and\ g(162)$ generates 2 and 0, respectively.

Now,

- bit_array[2] = 1
- bit_array[0] = 1

  So, according to Bloom filter element 162 exists in set although it is not present in actual.

To reduce such collisions, we need to increase the size of the bit array and use more hash functions because both these changes will reduce the probability of collision. Applying this analogy to darts and targets, we know that if there are $t\ targets$ and $d\ darts$, then the probability that no dart hits the target is $\boldsymbol{e^{(-\mathrm{d}/\mathrm{t})}}$**.** So, if the number of bits in the bit array is the targets and output of hash functions are the darts, we can compute the probability of collisions.

For a bit array of size 1 billion bits, 5 hash functions and 1 million elements:

- target = 10 ^ 9
- darts = 5 * 10 ^ 8
- probability (no darts hit the target) = $e^{(-\mathrm{d}/\mathrm{t})}$ = $e^{(-0.5)}$ = 0.607

This shows that the density of 0s in the bit array is 0.607. So, the density of 1s is 0.393. Now the probability of false-positive = probability of all 5 hash functions to an index that has 1 in the bit array = $(0.393)^5$ = 0.00937 (which is pretty less).

By tweaking the size of the bit array and number of hash functions, we can further improve the false positive rate of the Bloom Filter.

## 2.4 Comparing B+ Tree with LSM Tree [1]

There are three metrics to compare the performance of indexing data structures: write amplification, read amplification, and space amplification.

- **Write amplification**

  Today, most of the storage systems use flash memories like Solid State Drives (SSDs). The main difference between magnetic hard disk drives (HDDs) and SSDs is the way they handle writes. While HDD writes on empty spaces, SSD always first erases data and then writes new data on Flash storage chips. Flash storage contains blocks and pages. One block contains several pages, and each page comprises of many storage chips. The main challenge is that Flash cells can be deleted block-wise and written page wise. So if any change happens in a page of size 4K bytes, the Read/Modify/Write algorithm of SSD controller, first determines the block to which the page belongs to, then retrieves any data already present in the block, marks the corresponding block for deletion, redistributes the previous data and then lays down the new data in the block. So, flash storage devices have high write amplification, which means that the amount of data written to the storage is much greater than required. Also, flash chips have limited write cycles, so performing multiple writes reduces the life expectancy of flash devices.

- **Read amplification**

  The number of disk reads made to answer a query is called read amplification. For example, if five pages are read to answer a single query, the read amplification is

5. It is defined separately for point and range queries. For range queries, the length of the range matters. So, the read amplification for range queries is the number of rows to be fetched from the disk.

- **Space amplification**

  The ratio of the amount of data on the storage device to the actual size of the database is called space amplification. For example, the size of the database is 10MB, but it takes 100MB on the storage device, then the space amplification is 10.

### 2.4.1 B+ Tree

For a block size of B bytes and assuming that keys, pointers, and records have the same size, each internal node will have $O(B)$ children, and each leaf node will have $O(B)$ records. So, the depth of the B+ Tree will be $O\left(log_B(\frac{N}{B})\right)$, where N is the size of the data.

- **Write Amplification**

If every insertion creates a new leaf block, the write amplification of a B+ Tree is B.

- **Read Amplification**

The number of disk reads per query will be at most $O\left(log_B(\frac{N}{B})\right)$ [depth of the tree].

### 2.4.2 LSM Tree

In an LSM tree, data is organized in the form of levels, and each level is exponentially greater in size than the previous one, i.e., $level_i = k * level_{i-1}$, where k is the growth factor. If we assume that the size of each sstable is B bytes and each level contains just one sstable table, then the number of levels in LSM are: $\Theta\left(logk\left(\frac{N}{B}\right)\right)$ where N is the size of data in the database.

- **Write amplification**

In general, the sstable is moved out of each level once, and sstable from $level_i$ is merged repeatedly with the sstable from the previous $level_{i-1}$. On average, each sstable is re-merged back into the same level about $k/2$ times. Hence, the total write amplification is $\Theta\left(k * log_k\left(\frac{N}{B}\right)\right)$.

**Read amplification**

To perform a short-range query in the cold cache case, we must perform a binary search on each of the levels.

For the highest $level_i$, the data size is $O(N)$, so that it performs $O\left(\log\left(\frac{N}{B}\right)\right)$ disk reads.

For the previous $level_{i-1}$, the data size is $O(N/k)$, so that it performs $O\left(\log\left(\frac{N}{kB}\right)\right)$ disk reads.

For $level_{i-2}$, the data size is $O(N/k^2)$, so that it performs $O(\log\left(\frac{N}{k^2B}\right))$ disk reads.

…

For $level_{i-n}$, the data size is $O(N/k^n)$, so that it performs $O\left(\log\left(\frac{N}{k^nB}\right)\right)$ disk reads.

So that the total number of disk reads is:

$$R = O\left(\log\left(\frac{N}{B}\right)\right) + O\left(\log\left(\frac{N}{kB}\right)\right) + O\left(\log\left(\frac{N}{k^2B}\right)\right) + \cdots + O\left(\log\left(\frac{N}{k^nB}\right)\right) + 1$$
$$= O\left(\frac{log^2\left(\frac{N}{B}\right)}{\log k}\right)$$

## 2.5 LSM Compaction Strategies

In the LSM tree, as more updates come in, the number of sstables grows. This affects the performance of LSM tree in two aspects:

1. Space amplification: Old sstables (present in later levels) contain stale data, but they still occupy disk space.
2. Read amplification: The number of sstables to be searched per query are huge, which will affect read performance.

In order to combat these problems, compaction is done to merge multiple sstables into fewer sstables, thereby removing the obsolete data and keeping the overall size of LSM Tree within threshold limits to achieve a good read performance.

A compaction strategy decides which sstables will be compacted and when. There are two main types of compaction strategies:

- **Size Tiered Compaction** [9]**:**

In this technique, sstables are categorized into different buckets based on size, and then compaction runs for each bucket individually.

**Parameters:**

1. min_sstable_size: minimum size of a sstable
2. bucket_high: sstables with a size of about 1.5 times min_sstable_size is put in this bucket
3. bucket_low: sstables with a size of about 0.5 times min_sstable_size is put in this bucket

4. max_thresold: maximum number of sstables allowed in a bucket
5. min_threshold: minimum number of sstables allowed in a bucket

Based on these parameter values, the sstables in LSM Tree are sorted based on size, categorized into different buckets, and compaction is triggered when enough sstables are accumulated in a bucket.

Size Tiered compaction strategy suffers from space amplification problem. When a considerable amount of large sstables accumulate, double the disk space is required as compaction works with a copy of these sstables. Original sstables are removed after the compaction is successful.

- **Leveled Compaction** [10]**:**

In this compaction strategy, sstables are of fixed size, and merging is done level-by-level based on the number of sstables that can be accumulated at a particular level (level threshold). One sstable table is chosen from $level_i$, and multiple sstables that have overlapping key-range with this one sstable are picked from $level_{i+1}$, and they are merged. This process follows from $level_0$ until $level_n$.

Leveled Compaction has high write amplification because the same data is written multiple times during each compaction cycle. For example, one sstable from $level_i$ overlaps with ten sstables on $level_{i+1}$, these 11 sstables are merged (1st write) and then written back to $level_{i+1}$ (2nd write). This happens for every adjacent level.

## 2.6 RocksDB [11]

RocksDB is an LSM-based key-value stored created by Facebook in 2012. It was initially a fork of LevelDB, but since then, a lot of new features have been added, which helps the users tune RocksDB for different workload requirements. RocksDB uses a tiered + leveled compaction strategy. For levels L0 and L1, tiering is used. All sstables of L0 and L1 are merged together and placed at L1, making L0 empty. For levels L1, L2, … Ln partitioned leveling compaction technique is used. Leveled compaction suffers from high write amplification, which affects the write throughput of the system. So, RocksDB introduced Universal Compaction [12] Style (belong to tiering compaction family), which targets systems that require lower write amplification and can trade off read and space amplification.

In Universal Compaction, sstables contain full key-range but avoid overlaps in time-ranges. Compaction is done by merging sstables that have adjacent time-range, and output is a single sstable that does not overlap the time-range of any other sstable. Time-range compaction has lower write amplification than key-range compaction, but it suffers from high space amplification. It might happen that during full compaction, two copies of the database have to be maintained (double size issue).

RocksDB also provides a rate limiter to restrict the I/O bandwidth allowed for internal operations (compaction). The I/O bandwidth of the system can be set to a fixed value or can change it over time in multiplicative-increase, multiplicative-decrease manner. An auto-tuned version of the rate limiter adapts the I/O bandwidth to the amount of internal work left, thereby allocating more bandwidth when there is more pending compaction.

## 2.7 Monolithic LSM Design used in CooLSM

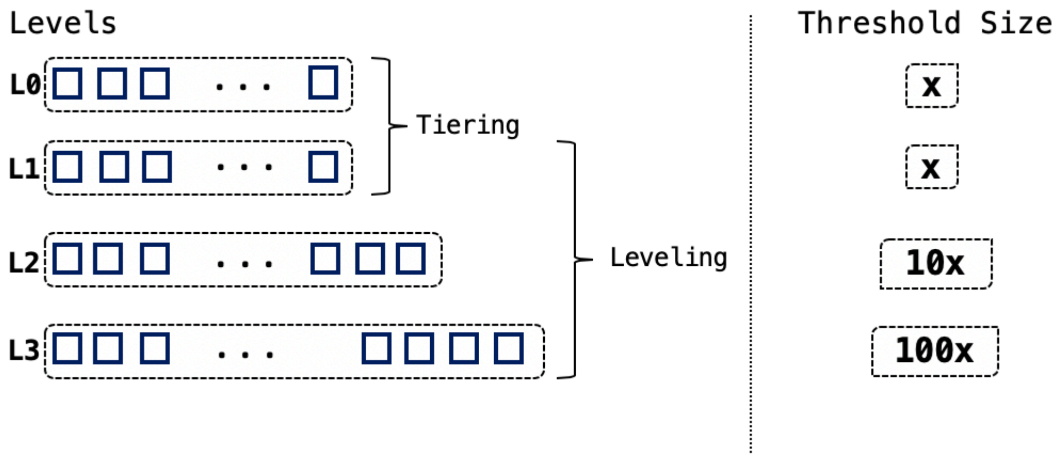

*Figure 3: The model of monolithic LSM that we build upon for CooLSM*

In Figure 3, we show the basic structure of the LSM-tree that we consider. It consists of four levels, L0 in memory and L1 to L3 in the disk. Compaction in L0/L1 is done through tiering, and compaction in the rest of the levels is done through leveling. Since L0 and L1 are using tiering compaction, they are of the same size. A constant size ratio of 10 is used for higher levels.

There are many variants of the compaction operation—both tiering and leveling. We focus on one widely used variant of the compaction strategy that is used by solutions such as RocksDB [11]. In this model, compaction in L0—called minor compaction—uses tiering while compaction in all other subsequent levels—called major compaction—use leveling.

**Algorithm 3** Monolithic LSM insert

```
procedure INSERT(key,value)
    sstable [ ] extraTables = null
    memtable.add(key, value)
    if memtable.size == memtable.threshold then
        sstable newtable = memtable.flush()
        level[0].insert(newtable)
        if level[0].size == level[0].threshold then
            extraTables =
                minorCompaction(level[0],level[1])
            if extraTables.size > 0 then
                lsmMajorCompaction(level[2],
                                   level[3],extraTables)
```

- **Minor Compaction**: when L0 reaches its threshold limit (number of sstables for level L0), all sstables of L0 and L1 are fetched into memory and merged using k-way merging. The merged sstables are then pushed to L1, and L0 is cleared.
- **Major Compaction**: The number of merged sstables created in minor compaction might be greater than L1's threshold. In that case, the extra sstables are then merged with L2. This trigger leveled compaction where sstables present at L2 is fetched, which intersects with the key range of these extra sstables. These sstables are then compacted using a k-way merge process and are written back at the desired location at level L2. The same happens when sstables overflow at any level ($L_i$)—these sstables are merged with overlapping sstables in $L_{i+1}$.

## Section 3: CooLSM Design

In this section, we present the design of CooLSM. We start by an overview and motivation in Section 3.1, followed by the architecture of CooLSM in Section 3.2. We then start presenting the details of CooLSM incrementally. We begin with the core design that consists of one Ingestor and many Compactors (Section 3.3). We then show the algorithms to add Readers to the core design and the notion of Snapshot Linearizability (Section 3.4). Then, we introduce the modifications to the new design to allow multiple overlapping Ingestors and the notion of Linearizable + Concurrent consistency (Section 3.5). The paper concludes with discussions about overlapping Compactors (Section 3.6) and reconfiguration (Section 3.7)

### 3.1 Design Overview and Motivation

Cooperative-LSM (CooLSM) is implemented by deconstructing the monolithic structure of LSM trees to enhance the scalability of LSM trees by utilizing the resources of multiple machines more flexibly. The traditional monolithic structure of LSM Trees lacks flexibility, and the only way to deal with an increased load is to re-partition the data and distribute it across nodes. This is costly and would not be feasible for a dynamic workload.

CooLSM consists of three components:

- **Ingestor** node receives the write requests. It maintains Levels L0 and L1 of the LSM tree and performs minor compaction.

- **Compactor** maintains the rest of the levels (L2 and L3) and is responsible for major compaction.
- **Reader (Backup)** maintains a copy of the entire LSM tree for recovery and read availability.

The advantages that CooLSM provides are two-fold:

1. Different components can be placed across different machines, and
2. There can be more than one instance of each component.

Running more than one instance for each component can enable various performance advantages depending on the type of component:

- Increasing the number of Ingestors enables digesting data faster as multiple ingestors are working in parallel.
- Increasing the number of Compactors enables offloading compaction to more nodes and thus reduces the impact of compaction on other functions and allows scaling to more inserts.
- Backups provide snapshot isolation [13] guarantees for read operations and increasing the number of backups enables to increase fault-tolerance (due to redundancy) as well as read availability.

**Motivation in edge-cloud environments**

Although these advantages are general to any deployment, we are especially motivated by their advantages to edge-cloud systems. CooLSM's design flexibility allows scaling efficiently to the heterogeneous and asymmetric environment of edge-cloud systems. In particular, placing Ingestors close to data sources at the edge allows scaling to the high-velocity data demand of edge and IoT applications. Placing compactors at the cloud enables offloading the compaction overhead away from data sources and direct consumers while leveraging the computing resources of the cloud. Placing Backups close to data sinks and consumers allows faster and more interactive analytics and read-only queries. Across all the various types, the demand on each component might vary from one workload to another and from one time to another. The elastic nature of each component type allows scaling the number of instances of each component to react in real-time to the changes in workload characteristics.

## 3.2 Architecture

The architecture of CooLSM is shown in Figure 1(b), and an example of how it is deployed is shown in Figure 1(c).

In CooLSM, there might be one or more Ingestors. All Ingestors are identical. They receive upsert and read operations for any key in the data range. Each Ingestor maintains a separate levels L0 and L1. L0 contains pages of inserted key-value pairs (that correspond to upsert commands.) Each page represents a batch of inserted keys. Therefore, the keys across pages are not unique or ordered.  However, the keys within a page are ordered before insertion to L0.

Each level i has a threshold denoted L[i].threshold. When the number of pages in L0 exceeds the threshold, then minor compaction is performed with L0. Keys across pages of levels other than L0 are sorted and unique. For example, if L1 has ten pages, then the first page contains the smallest keys, and so on. This is possible because the compaction process orders all keys across the pages before they are inserted to the level.

When the number of pages in L1 exceeds the L[1].threshold, the Ingestor sends the extra pages to one or more of the Compactors.

Like Ingestors, there might be one or more Compactors in a CooLSM instance. Compactors might be partitioned or have overlapping ranges. Each type of compactor scaling (partitioned or overlapping) has its advantages and disadvantages that we discuss later in this section. Each compactor receives pages from Ingestors and maintains a separate levels L2 and L3. When pages are received from an Ingestor, they are compacted to level L2. When the threshold of L2 is exceeded, then major compaction is performed with level L3.

An instance of CooLSM might have Backups. Each backup node represents a snapshot of the state. The backup receives data from compactors and incorporates them in its state. Clients wishing to read the state of the data can read from backups. These reads might miss the most recent updates but represent a consistent snapshot of the state of the data.

In the rest of this section, we present the algorithms used to perform insert and read operations in CooLSM. We follow an incremental approach in presenting CooLSM design, beginning with the design of CooLSM with a single Ingestor and one or more partitioned Compactors. Later in the section, we introduce backups, overlapping Compactors, and multiple Ingestors.

## 3.3 Core CooLSM (One Ingestor and Multiple Compactors)

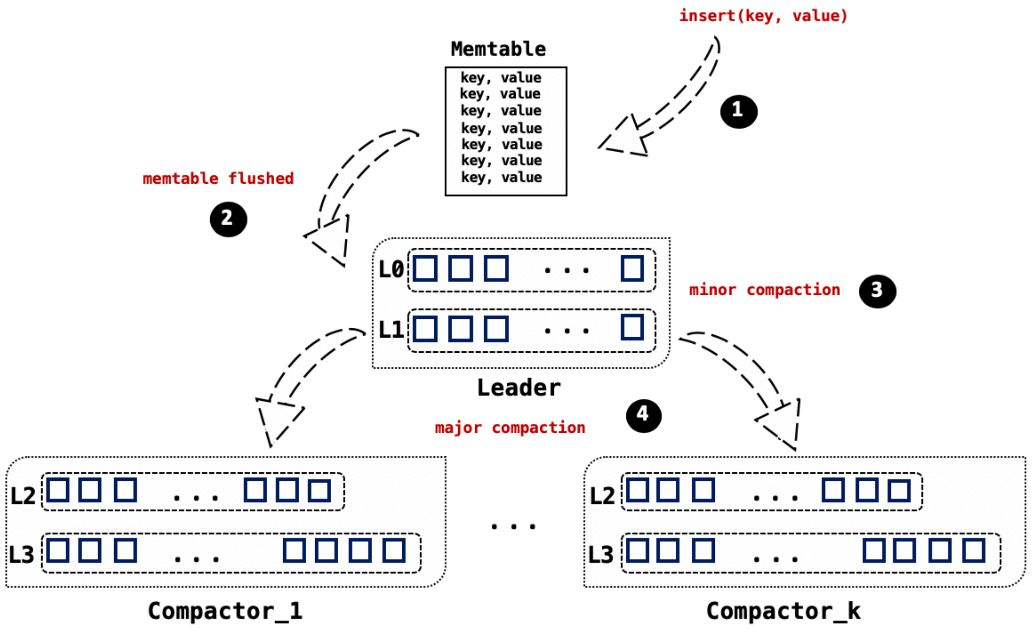

*Figure 4: CooLSM (No Read Server) Write Flow*

We begin describing the details of CooLSM algorithms and protocols using the core CooLSM architecture. This architecture assumes having only one Ingestor and one or more partitioned Compactors. All upsert and read operations are sent to the Ingestor. Each compactor handles a mutually exclusive range of the data. For example, if the range of data items is 100K keys, and there are two compactors, each compactor could be handling 50K keys.

## 3.3.1 Upsert Flow

**Algorithm 4** C-LSM Leader insert

```
1: procedure INSERT(key,value)
2:     sstable [ ] extraTables = null
3:     memtable.add(key, value)
4:     if memtable.size == memtable.threshold then
5:         sstable newtable = memtable.flush()
6:         level[0].insert(newtable)
7:         if level[0].size == level[0].threshold then
8:             extraTables =
9:                 minorCompaction(level[0],level[1])
10:        sendToCompactors(extraTables)
```

**Algorithm 5** C-LSM Leader sendToCompactors

```
procedure SENDTOCOMPACTORS(extraTables)
    map <compactorID, sstable [ ] tables> tableMap =
null
    for $i$ = 1 to size(extraTables) do
        compactorID =
                findCompactorForKeyRange(extraTables[$i$])
        map.put(compactorID, extraTables[$i$])
    for $i$ = 1 to numOfCompactors do
        compactors[$i$].sendTables(map.get($i$))
```

**Algorithm 6** C-LSM Compactor$_i$ majorCompaction

```
procedure MAJORCOMPACTION(sstable [ ] tables)
    lsmMajorCompaction(level[2],level[3], tables)
```

Algorithm 4 shows the steps performed to insert a new key-value pair to the index. First, the Ingestor batches all received upsert operations. Once the batch reaches a threshold, the Ingestor orders the key-value pairs in the batch and adds the batch as a new memtable in L0. After the new memtable is added, the Ingestor checks whether the threshold of the number of memtables (L[0].threshold) is exceeded. If it is, then the Ingestor initiates a minor compaction process that compacts all the memtables in L0 into the sstables in L1.

Minor compaction is a k-way merge operation across all pages in L0 and L1. Specifically, the Ingestor sorts all the key-value pairs in L0 and L1, removing any redundancies by only keeping the most recent key-value pair of each present key. Once all the key-value pairs are sorted, they are divided into ordered sstables, where the size of a sstable is predetermined. At this point, all the pages in L0 and L1 are cleared, and the newly merged sstables are inserted to L1. This step is performed atomically.

After finishing the minor compaction, the Ingestor checks if the threshold L[1].threshold is exceeded. If it is, then the Ingestor picks the extra sstables that exceed the threshold and forward them to the appropriate Compactors. For each sstable, the Ingestor checks if the range of key-value pairs falls within one or more compactors since compactors are partitioned (we discuss compactors with overlapping ranges later in this section.) If it falls within one compactor, then it is forwarded to it. Otherwise, the Ingestor divided the sstable into different parts, where each part corresponds to a Compactor's range and then send each part to its corresponding Compactor.

The Ingestor does not remove the forwarded sstable immediately. Instead, it waits to hear an acknowledgment from the compactor that the sstable was received and merged at the compactor. This is important for the read operation that we present later to ensure that no key-value pairs are temporarily absent from both the Ingestor and Compactors. When a Compactor receives sstables from an Ingestor, it starts a major compaction process. Specifically, the compaction process affects sstables in L2 that overlaps with the range of the received sstable — we will call these pages L2.overlap. A k-way merge operation is performed between the received sstables and the sstables overlapping with them in L2. After the merge is completed, the resulting ordered sstables are used to replace the L2.overlap pages in L2. This step is performed atomically. Once the major compaction is done, the Compactor notifies the Ingestor. Then, it checks whether the number of sstables in L2 exceeds the threshold. If it does, then another major compaction is performed similarly to the compaction to L2. Specifically, the extra pages in L2 are merges with the overlapping pages in L3.

## 3.3.2 Read Flow

**Algorithm 7** C-LSM Leader read

```
procedure READ(key)
    value = search(memtable)
    if value == null then
        value = searchAll(level[0])
        if value == null then
            sstableIndex = bloomfilter(level[1])
            value = search(level[1].sstables[sstableIndex])
            if value == null then
                return readAtCompactors(key)
    return value
```

**Algorithm 8** C-LSM Leader readAtCompactors

```
procedure READATCOMPACTORS(key)
    compactorID = findCompactorOnKey(key)
    return compactors[serverID].compactorRead(key)
```

**Algorithm 9** C-LSM Compactor i compactorRead

```
procedure COMPACTORREAD(key)
    sstableIndex = bloomfilter(level[2])
    value = search(level[2].sstables[sstableIndex])
    if value == null then
        sstableIndex = bloomfilter(level[3])
        value = search(level[3].sstables[sstableIndex])
    return value
```

The algorithm for read operations is shown in Algorithm 7. Read operations are sent to the Ingestor. The Ingestor goes through the memtables, starting from the most recent one to look for the requested key. If it is found, the most recent key-value pair is returned. Otherwise, the Ingestor looks in the sstables in L1. If the key is found, then it is returned. Otherwise, the read request is forwarded to the appropriate compactor based on the compactor key-range.

When a compactor receives a read operation, it looks in its sstables, starting with the corresponding sstable in L2 and then the corresponding sstable in L3. If the key is found, then it is returned to the client. Otherwise, a negative acknowledgment is sent back to the client.

Like many LSM variants, we use bloom filters and fence pointers to speed up the process of looking through the sstables.

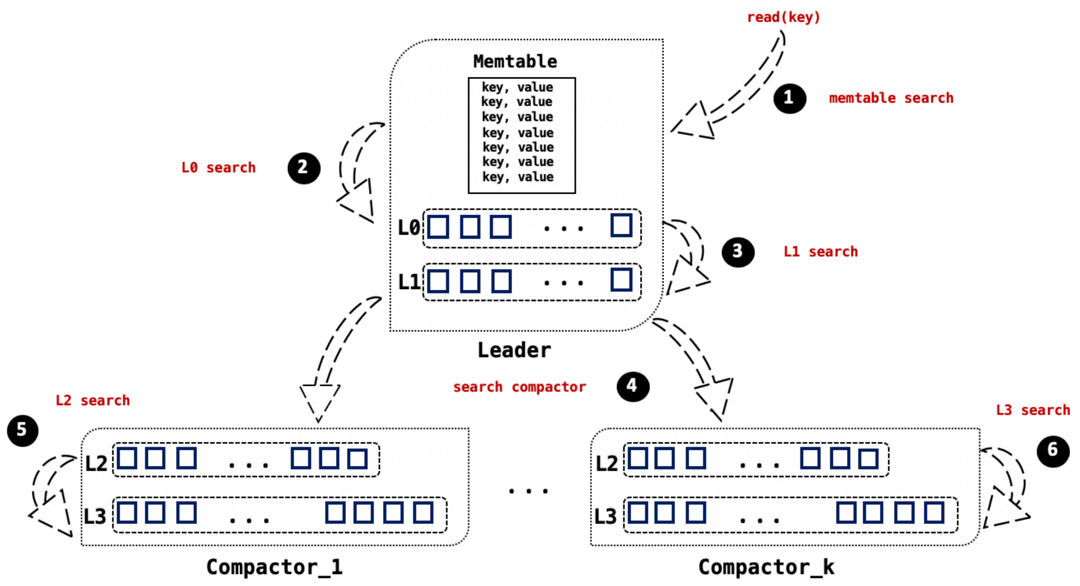

*Figure 5: CooLSM (No Read Server) Read Flow*

### 3.3.3 Safety

In this core architecture with one Ingestor and multiple Compactors, CooLSM guarantees Linearizability [4]. Linearizability is a guarantee that a data operation appears to happen instantaneously at some time between its invocation and return. An execution history $\boldsymbol{H_{lin.}}$ represents a sequence of operations that satisfy the illusion of instantaneous invocations. Linearizability implies that if an operation starts after an operation ***b*** returns, then ***a*** must be logically ordered after ***b***.

The following is a proof sketch of the linearizability of core CooLSM. In core CooLSM, upsert operations are handled one operation at a time. The upsert is appended to a batch, and then an acknowledgment is sent back to the client. The time of an upsert operation in $\boldsymbol{H_{lin.}}$ is between the start and return time. Every past upsert operation is ordered before ***a*** in $\boldsymbol{H_{lin.}}$ and every future upsert operation is ordered after ***a*** in $\boldsymbol{H_{lin.}}$ When a read operation is performed, it is sent to the Ingestor. Assume that the returned value is written by an operation ***c***. It is guaranteed that the time of operation ***c*** in $\boldsymbol{H_{lin.}}$ is less than the return time of the read operation since the upsert is observed. Also, it is guaranteed that there is no other write operation ***d*** with the time that is larger than ***c*** in $\boldsymbol{H_{lin.}}$ and smaller than the start time of the read operation. Otherwise, CooLSM would have observed that read. Therefore, the time of the read operation in $\boldsymbol{H_{lin.}}$ can be set as happening immediately after **c**. With this timing of upsert and read operations, we show that indeed there is an appearance of operations happening instantaneously at some time between their invocation and return.

## 3.4 Adding Backup Components

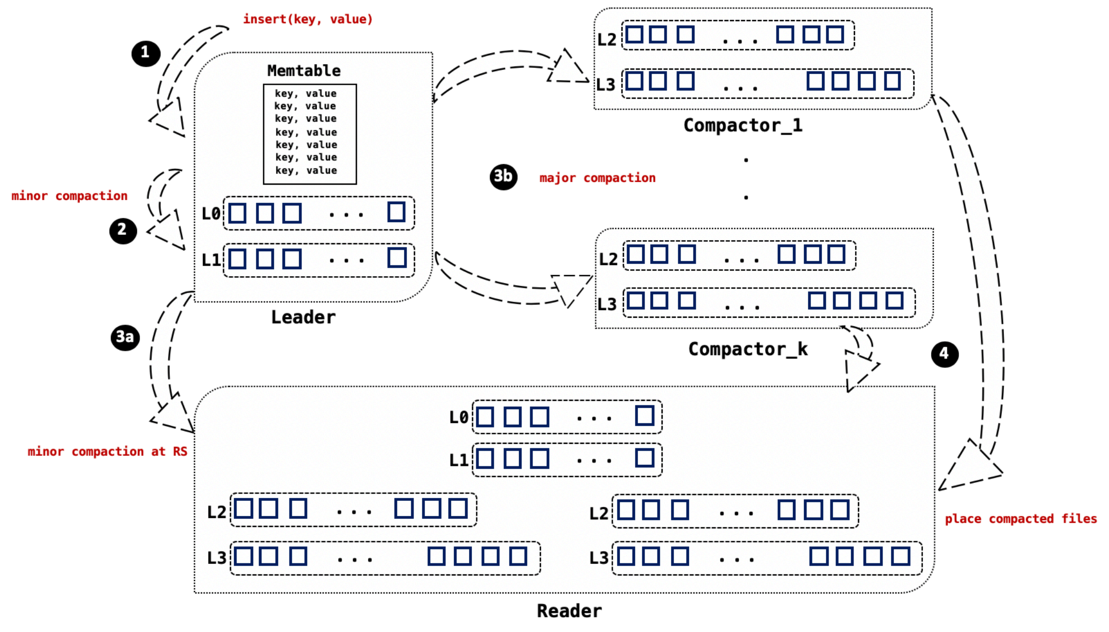

*Figure 6: CooLSM (With Read Server) Write Flow*

A backup node maintains a snapshot of the data maintained by the compactors. The most recent key-value pairs might not be present in the backup. However, backups serve an important role in increasing the read availability of CooLSM. Backups receive updates about the state of the index via the Compactors. Each Compactor, after each major compaction, forwards the newly formed sstables to all the backup nodes. Each backup node uses the newly received information to update its index. Backup nodes have entire LSM tree and they receive updates from both Ingestor and Compactors. In the case there is more than one partitioned Compactor, the backup gets updates from all of them and integrate the sstables into one structure.

A client wishing to read from CooLSM has the option of learning from one of the backup nodes. To do so, a read request is sent to the backup, and the backup serves it by checking the sstables in L2 and then L3, similar to how a Compactor serves a read request. The advantage of reading from the backup node is that the read operation is not affecting the Ingestor and Compactors directly. Therefore, the read operation is not interfering with the ingestion and compaction processes and would not lead to impacting their performance or be impacted by their operation. This is particularly important for large reads that are needed in analytics and complex read-only queries, especially if these operations are interactive, requiring a fast response.

### 3.4.1 Safety (Snapshot Linearizability)

The downside of reading from a backup node is the freshness of the read data. A backup node does not have the most recent data that was not forwarded to it yet. This breaks the safety of reads as they are no longer linearizable. This is a trade-off between read availability and consistency.

Although linearizability is not guaranteed if a client reads from a backup, we use a weaker consistency guarantee that is achieved when a client reads from a backup, called **snapshot linearizable**. Snapshot linearizability guarantees that for any two consecutive reads, $\boldsymbol{r_1}$ followed by $\boldsymbol{r_2}$, that are reading the same object and are served from the same backup node, then either of the following is true: (1) the two read operations return the same value. The value corresponds to a previous write operation, or (2) $\boldsymbol{r_1}$ returns the value written by $\boldsymbol{w_1}$, and $\boldsymbol{r_2}$ returns the value written by $\boldsymbol{w_2}$ and $\boldsymbol{w_1} <_{lin.} \boldsymbol{w_2}$, where $<_{lin.}$ is an ordering of the two writes in the linearizable history $\boldsymbol{H_{lin.}}$ of the main system (the Ingestor and Compactors in our case.)

The notion of snapshot linearizability is useful to applications reading from the backup, such as analytics and read-only queries. The reason is that snapshot linearizability preserves the notion of time in the updates. For example, an application that is interested in observing the patterns in a sequence of readings from an IoT device would still observe the sequence of readings in their original order.

Reading from the CooLSM backup is snapshot linearizable because the writes (sstables) are forwarded from each Compactor in order. Therefore, the state of the backup reflects the state of a compactor as it progresses in time. Note that this guarantee applies to objects within the range of a single Compactor. This is because, with more than one Compactor, some Compactor's updates might be delayed with respect to others.

## 3.5 Multiple Overlapping Ingestors

Now, we consider the case of having more than one Ingestor in a CooLSM deployment where they overlap in the key ranges, they handle[1]. The motivation for having more than one Ingestor is to scale the ingestion process. For simplicity, we consider deployment of CooLSM with multiple Ingestors and Multiple partitioned Compactors but no Backups (we discuss briefly the implications of adding a Backup with Multiple Ingestors later in this section.)

The design and algorithms for Ingestors and Compactors are the same as presented in the core CooLSM design (Section 3.3). The only difference is that there could be more than one Ingestor receiving upsert and read operations. Operations to insert data are treated in the same way as core CooLSM. The operations are batched, inserted to L0, and then get compacted with sstables in L1 when the threshold of L0 is exceeded. When L1's threshold is exceeded, the sstables are sent to the appropriate Compactors.

The implication of having multiple Ingestors is that there could be multiple versions of the same key across different Ingestors. Furthermore, the order of these multiple versions might be forwarded to Compactors in an arbitrary order that does not follow the linearizable order of insertions. Due to these implications, the read process must be modified to read from multiple Ingestors, and the consistency guarantees must be revisited.

### 3.5.1 Consistency Anomalies

To demonstrate the consistency anomalies that are introduced with multiple Ingestors, consider an example with two Ingestors, $\boldsymbol{I_1}$ and $\boldsymbol{I_2}$, and one Compactor, $\boldsymbol{C_1}$. Assume that two independent clients sent upsert requests, where client 1 sent $\boldsymbol{o_1 = upsert(x = 1)}$ to $\boldsymbol{I_1}$ and client 2 sent $\boldsymbol{o_2 = upsert(x = 2)}$ to $\boldsymbol{I_2}$. The two upsert requests happened concurrently, and we assume that there is no accurate time synchronization to learn the relative time ordering of the two operations using timestamps.

Consider the scenario if a read operation, $\boldsymbol{r_1 = read(x)}$, was issued while both records for $\boldsymbol{o_1}$ and $\boldsymbol{o_2}$ are still in their corresponding Ingestors. The read operation would be sent to both Ingestors since the value of $\boldsymbol{x}$ could be in any one of them. When both Ingestors return both values for $\boldsymbol{o_1}$ and $\boldsymbol{o_2}$, the client cannot decide which one of the two values is safe to read while maintaining linearizability.

Consider another scenario after one of the Ingestors, $\boldsymbol{I_2}$, sends the compacted sstables that contains $\boldsymbol{o_2}$. At this time, $\boldsymbol{o_1}$ is in $\boldsymbol{I_1}$, and $\boldsymbol{o_2}$ is in $\boldsymbol{C_1}$. If a read operation is issued at this time, it will be sent to both Ingestors first. $\boldsymbol{I_1}$ would return $\boldsymbol{o_1}$, and $\boldsymbol{I_2}$ would forward it to $\boldsymbol{C_1}$, and finally, $\boldsymbol{C_1}$ would return $\boldsymbol{o_2}$. Although one of the returned records is at an Ingestor and the other is at a Compactor, the client is still unable to decide which one of the two values is safe to read while maintaining linearizability.

The final scenario happens when $\boldsymbol{I_1}$ finally sends $\boldsymbol{o_1}$ to $\boldsymbol{C_1}$ for major compaction. When $\boldsymbol{o_1}$ is received, $\boldsymbol{C_1}$ cannot decide which operation to garbage collect and which operation to keep as it does not know which one is more recent.

The previous scenarios show the anomalies that can occur with two (or more) Ingestors. There are two ways of handling the occurrence of such anomalies: (1) relaxing the consistency guarantees, and (2) Introduce coordination across Ingestors. We elect to use the first approach to avoid the added overhead of Ingestor-to-Ingestor coordination that can be significant in our edge-cloud environment. In the following, we discuss both approaches.

### 3.5.2 Relaxing Consistency: Linearizable + Concurrent

The first approach is to relax consistency for cases when CooLSM is deployed with more than one Ingestor. We observe that the anomalies discussed above are due to concurrent insert operations where a node cannot decide which operation was performed first. Using timestamps is infeasible in the absence of accurate time synchronization across Ingestors. However, it is possible to use existing time synchronization technologies that guarantee loose time synchronization, such as NTP [14]. These protocols can provide bounds on the time difference between the timestamps of events at different machines. These bounds can be used to deduce if two events are either ordered (one of the two events happened before the other) or concurrent (loose-time synchronization cannot decide which operation was first if the timestamps are too close).

Specifically, if loose-time synchronization is deployed, each event $\boldsymbol{e}$ can be timestamped with a timestamp $\boldsymbol{t_e}$. Each timestamp accuracy is bounded by some threshold $\boldsymbol{\delta}$, where the accurate global timestamp of $\boldsymbol{t_e}$ — denoted $\boldsymbol{t_e^g}$ — is somewhere in the range $\boldsymbol{t_e - \delta < t_e^g < t_e + \delta}$. Using this formula, if the difference between two events timestamps is greater than $\boldsymbol{2.\delta}$, then they can be ordered by their timestamps, *i.e.*, if $\boldsymbol{t_a - t_b \geq 2.\delta}$, then $\boldsymbol{b <_t a}$, where $<_t$ is a global time ordering.

Using this global time ordering, we can establish order across the inserted items if the difference in their timestamps is larger than $\boldsymbol{2.\delta}$. What remains are data items that are inserted with $\boldsymbol{2.\delta}$ time of each other. Ordering such inserts is infeasible without additional coordination (which we discuss in the following section.)

To overcome the need for additional coordination, we formulate this in terms of a relaxed consistency guarantee that we call ***Linearizable + Concurrent*** consistency:

**Definition 1.** ***(Linearizable + Concurrent)*** *A history of read and write operations is Linearizable+Concurrent if for any two operations* ***a*** *and* ***b*** *where* $\boldsymbol{t_a - t_b \geq 2.\delta}$*, then* ***a*** *must be logically ordered after* ***b*** *(Assuming that* $\boldsymbol{\delta}$ *is the time synchronization bound.)*

With this new definition of Linearizable + Concurrent consistency, we modify the Ingestor and Compactor algorithms to guarantee it. The first modification is when an upsert request is received. The Ingestor timestamps the inserted key-value pair using the time synchronization service. The second modification is that the Ingestor maintains the timestamp of the most recent record that is sent to Compactors. This becomes useful during the read operation to know whether it is needed to read from the Compactors.

The read operation is modified to consist of two phases, a phase asking the Ingestors and a phase asking the Compactors if necessary.

In the first phase, the read operation is sent to all Ingestors. Each Ingestor responds with the most recent record matching the requested key, as well as the timestamp of the most recent record sent to the Compactors ($\boldsymbol{ts_c}$). The client, then, decides whether it needs to ask Compactors for records. If no records were received from Ingestors, then the client asks the corresponding Compactors. If records were received from Ingestors, then the client might still need to ask Compactors—in case a more recent record was forwarded to Compactors. To decide whether the client needs to ask Compactors, it uses the received record with the highest timestamp ($\boldsymbol{ts_h}$) as well as the lowest received $\boldsymbol{ts_c}$. If $\boldsymbol{ts_h - ts_c \geq 2.\delta}$, then the client knows that all the records in the Compactors are ordered before the record with the timestamp $\boldsymbol{ts_h}$. In such a case, the first phase terminates, and there is no need for the second phase (asking the Compactors.)

If $\boldsymbol{ts_h - ts_c < 2.\delta}$, then the client enters the second phase and sends read requests to the corresponding compactor. Each compactor returns the most recent version of the requested key.

After finishing phase 1 (and phase 2, if it is needed), the client returns the record with the highest timestamp.

THEOREM 1. *The modified CooLSM algorithms for multiple Ingestors guarantees Linearizable + Concurrent consistency.*

PROOF.

To prove by contradiction, consider the case of two operations $\boldsymbol{o_1}$ with the timestamp $\boldsymbol{ts_1}$, and operation $\boldsymbol{o_2}$ with the timestamp $\boldsymbol{ts_2}$. Also, assume that $\boldsymbol{ts_2 - ts_1 \geq 2.\delta}$. We now show that for all cases, $\boldsymbol{o_2}$ is logically ordered after $\boldsymbol{o_1}$:

- Both operations are writes: Because $\boldsymbol{ts_2 - ts_1 \geq 2.\delta}$, it is guaranteed that when $\boldsymbol{o_2}$ is received at its Ingestor, that $\boldsymbol{o_1}$ has already been received at its Ingestor. Therefore, the state of the index is always going to reflect either the state with $\boldsymbol{o_1}$ only or with both $\boldsymbol{o_1}$ and $\boldsymbol{o_2}$.
- $\boldsymbol{o_1}$ is a write, and $\boldsymbol{o_2}$ is a read: The read operation is guaranteed to be received after $\boldsymbol{o_1}$ is received at its corresponding Ingestor[1]. Otherwise, the difference in the timestamp would not hold. Therefore, the read operation is guaranteed to observe the state that reflects $\boldsymbol{o_1}$.
- Both operations are reads: Any write that is observed by $\boldsymbol{o_1}$ is also observed by $\boldsymbol{o_2}$. This is because $\boldsymbol{o_2}$ is guaranteed to be received at each Ingestor after $\boldsymbol{o_1}$ has been received.

[1] change the design to make the read go through one Ingestor, get timestamped and then forwarded to others. And then only respond with the state as of the read timestamp—this also requires changing the compaction to make it delayed.

- $\boldsymbol{o_1}$ is a read, and $\boldsymbol{o_2}$ is a write: The read is received at each Ingestor before the write. Additionally, the read ignores any writes with timestamps higher than its timestamp. Therefore, it never observes the state of $\boldsymbol{o_2}$.

## 3.6 Overlapping Compactors

Up until now, we assumed non-overlapping Compactors, *i.e.*, each Compactor is responsible for a mutually exclusive set of keys. We anticipate that in most deployments, it is sufficient to have non-overlapping Compactors. The reason is that they would typically be hosted in the cloud where there are enough resources and availability to perform fast reconfiguration—to react to dynamic workload changes. (This is not the case for Ingestors that would be at the edge and handle data in a more time-sensitive manner that makes reconfiguration for Ingestors relatively more disruptive than reconfiguration for Compactors.)

Nonetheless, for completeness, we allow CooLSM to handle multiple overlapping Compactors. Adding this support requires small changes to the original design of read and write operations that we presented in the previous sections. For write operations, when an Ingestor forwards records to Compactors, it chooses one of the overlapping Compactors arbitrarily—potentially using a load balancing strategy—and forwards the records to it. For read operations, an Ingestor forwards the read operation to all Compactors that overlap with the key in the read operation.

A complication that overlapping Compactors introduce is in the case that there are backup nodes. With non-overlapping Compactors, the backup relies on a single source for each key, which makes it straightforward to maintain a progressive history. In the case of multiple Compactors, the Backup must find a way to order overlapping records received from multiple Compactors. In this work, we do not treat this problem as we focus on the case with non-overlapping Compactors and leave the details of the case of overlapping Compactors to future work. However, a possible approach to enable Backup nodes to order operations is to use sequence numbers if there is one Ingestor or use timestamps if there is more than one Ingestor and relax consistency for Backups in the same way we did with multiple Ingestors.

## 3.7 Reconfiguration

To react to changing workloads conditions, we need to perform reconfiguration. We use existing reconfiguration methods used in distributed databases [15] and adapting them to our use case. The design of CooLSM where it is possible to have overlapping Ingestors and Compactors enables elastic reconfiguration via a three-step approach: (1) Expand, where the node with the new configuration is added as an overlapping component, (2) Migrate, where all requests and data are migrated from the node with the old configuration to the node with the new configuration, and (3) Detach, where the node with the old configuration is retired after all the data and requests have migrated to the node with the new configuration.

## Section 4: Evaluation

We present a performance evaluation of the CooLSM in this section. We conducted our experiments on Amazon AWS datacenter in Virginia using t2.xlarge EC2 instances. Each machine runs 64-bit Ubuntu Linux and has four 3.0 GHz Intel Scalable Processors with 16 GB of RAM.

For each experiment, we have used two key ranges: 100K and 300K. For 100K key-range, L0 and L1 have ten sstables, L2 has 100, and L3 has 1000 sstables. For 300K key-range, L2 has 300 sstables, and L3 has 3000 sstables. For write experiments, a batch size of 10K is used, and for read experiments, a batch size of 1K is used.

For edge experiments, we have used five datacenters in Amazon AWS. Virginia datacenter is the cloud where compactors are placed. Ohio, California, Oregon, and London are used as edge nodes where ingestor is placed.

### 4.1 CooLSM Write Performance

In this section, we present a set of experiments to test the write performance of CooLSM by varying the number of compactors. Figure 7 shows the results of these experiments that are conducted in the Virginia data center.

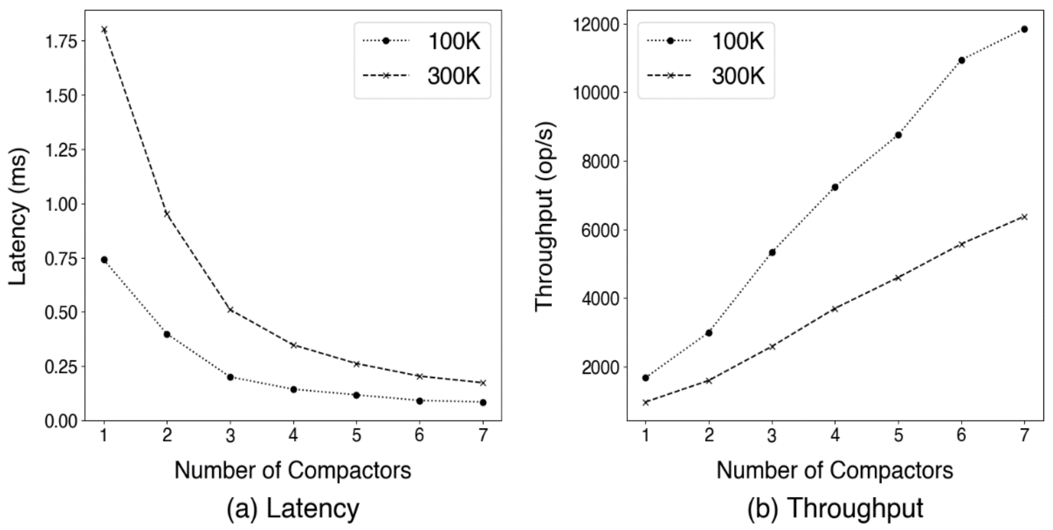

*Figure 7: The write performance metrics of CooLSM with varying the number of compactors*

In Figure 7(a), as the number of compactors increases, the overall write latency of CooLSM reduces (50% reduction till three compactors, followed by 25% reduction till six compactors and 15% after seven compactors) but the reduction in latency is not significant after five compactors. This is because five independent compactors provide enough computation resources to carry out the heavy compaction process (for both 100K and 300K), so introducing more compactors is redundant. For 300K key-range, the latency will be higher than 100K key-range as the compaction involves more sstables due to a bigger LSM tree.

The write throughput of CooLSM in Figure 7(b) increasing nearly linearly with the increase in the number of compactors. Since a bigger LSM tree is being used for 300K key-range, compaction is more resource-intensive due to more sstables. As a result, the write throughput is lower than the 100K key-range.

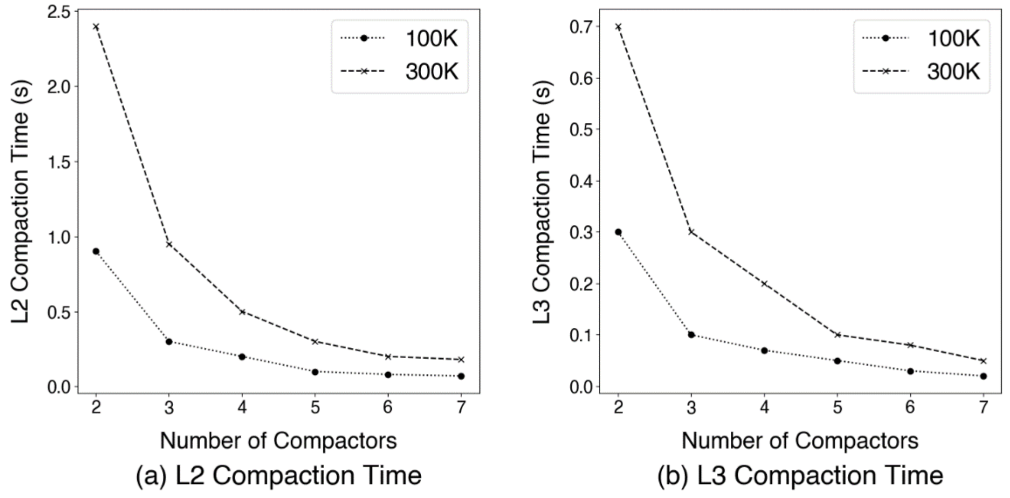

*Figure 8: The compaction metrics of CooLSM with varying the number of compactors*

Another set of write performance metrics that we analyzed is how L2 and L3 compaction time varies with the number of compactors. In Figure 8, L2 and L3 compaction time are following a similar pattern for both 100K and 300K key ranges, with 300K taking more time. To trigger major compaction, a list of extra sstables from level L1 at Ingestor are sent to the appropriate compactor based on the key-range of the sstables. In one cycle of major compaction, one sstable is chosen at random from this list of extra sstables and merged with overlapping sstables at level L2. If, after this merge, L2 reaches its threshold, then L2/L3 compaction is triggered. This repeatedly happens till all extra sstables have been merged in the LSM tree residing on the compactor. As a result, L2 compaction time is more than the L3 compaction time because L2 undergoes compaction more often than L3.

## 4.2 CooLSM Read Performance

In this section, we performed a set of experiments to test the read performance of CooLSM without the backup server. We conducted two sets of experiments, one with two compactors (one compactor means monolithic CooLSM) and the other with five compactors (five compactors provide enough computational resources to conduct the compaction process for both 100K and 300K key ranges). We used mix (both reads and upserts) workload for these experiments, where we auto-tuned the percentage of reads (25%, 50%, and 75%).

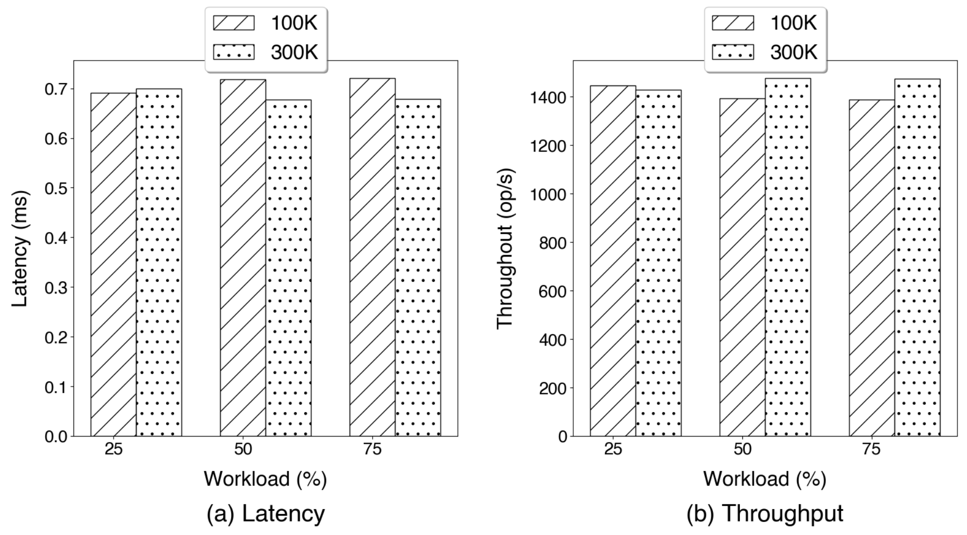

*Figure 9: The read performance of CooLSM (key-range = 100K and 300K) for two compactors with varying the read workload*

Figure 9 shows that CooLSM has a consistent read latency of about 0.7 ms (per read operation) and throughput of 1400 read operations per second for both 100K and 300K key ranges. This shows that using a larger LSM tree is not affecting the read performance because of fence pointers and Bloom filter. Fence pointers help in narrowing down the search to one sstable and bloom filter tests if the key is present in the sstable. Also, varying the number of compactors is not affecting the read latency as the ingestor directs the read request to only one compactor based on the partitioning of the key-range amongst the compactors, as shown in Figure 10.

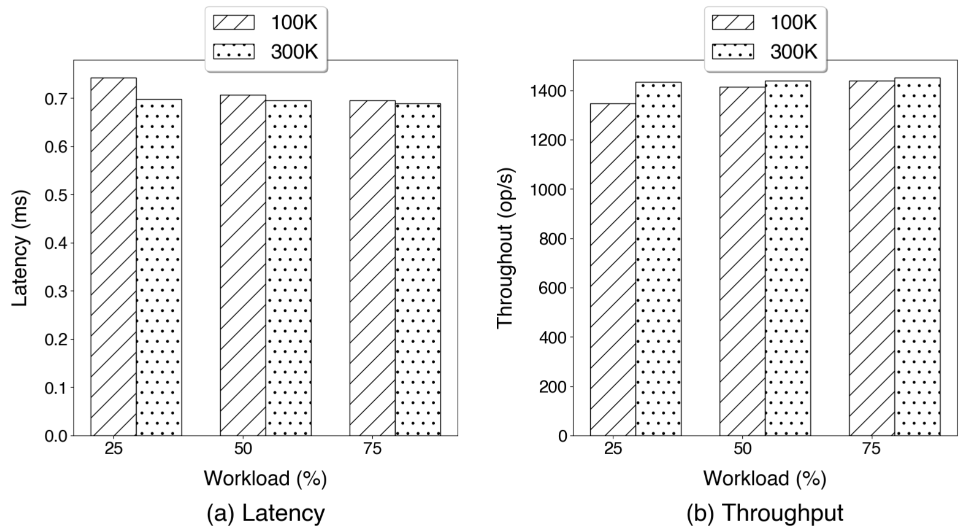

*Figure 10: The read performance of CooLSM (key-range = 100K and 300K) for five compactors with varying the read workload*

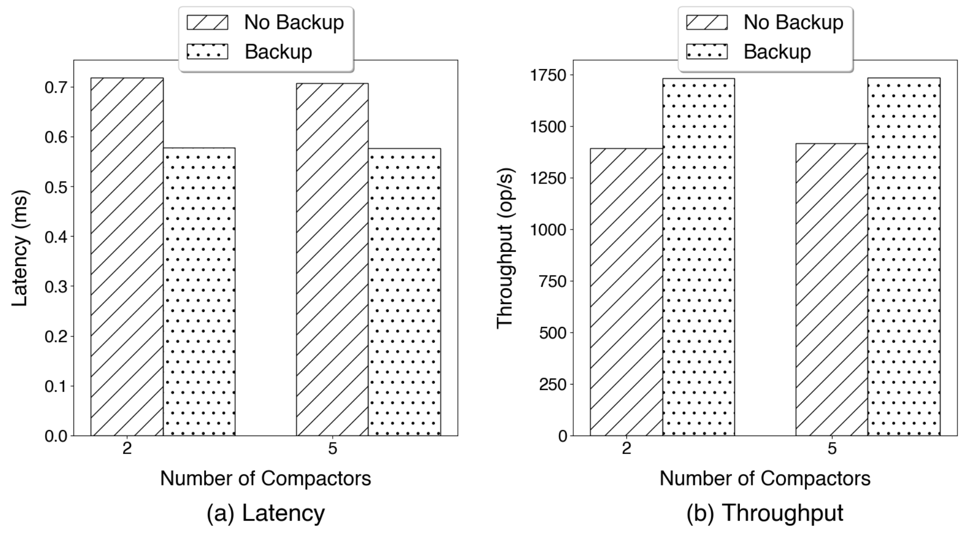

*Figure 11: The read performance of CooLSM (key-range = 100K) for two and five compactors with and without backup server*

Introducing a backup server improves the read latency of CooLSM to about 0.6 ms (per read operation) and read throughput to 1600 operations per second, as shown in Figures 11 and 12. This is because now the read request is directly sent to the backup server instead of forwarding the request to compactors via ingestor, which reduces the network time.

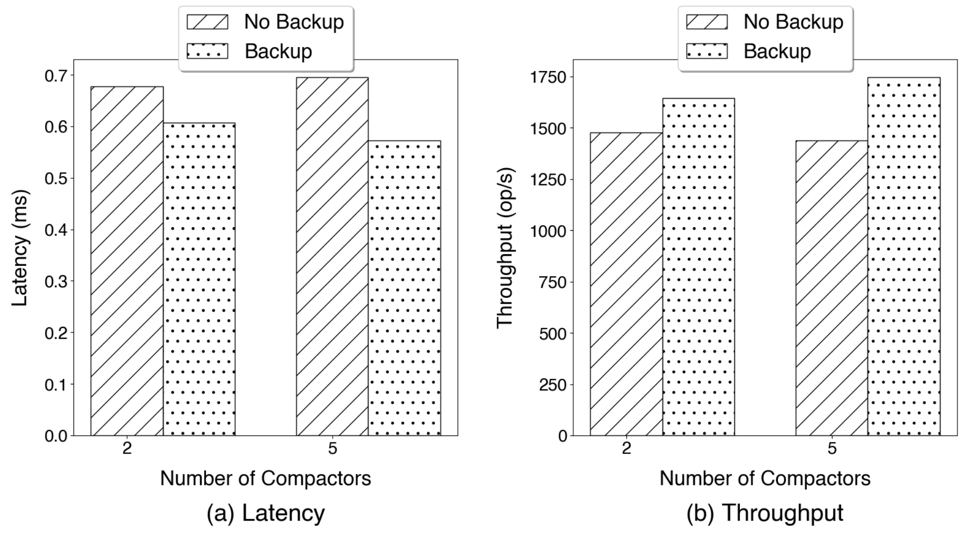

*Figure 12: The read performance of CooLSM (key-range = 300K) for two and five compactors with and without backup server*

## 4.3 Multiple Overlapping Ingestors

In this section, we test the performance of CooLSM with Multiple Overlapping Ingestors. Each ingestor independently generates the write requests, and when level L1 is filled up, major compaction is triggered on compactors. Since compactors are single-threaded, compaction requests are handled one-by-one, thereby queuing concurrent requests.

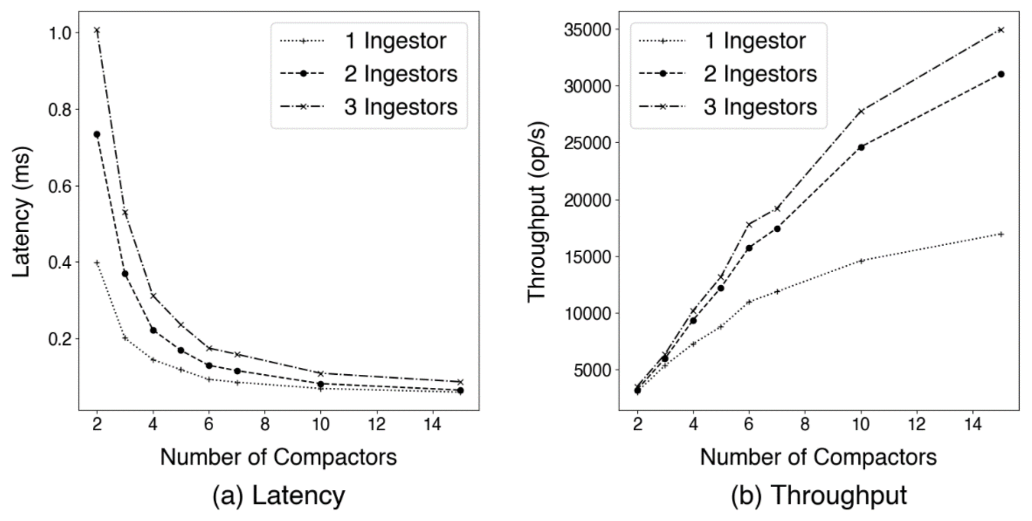

*Figure 13: The write performance of Multiple Overlapping Ingestors (key-range = 100K) with varying the number of compactors*

In Figure 13(a), as the number of ingestors increases, the latency of Multi-Ingestor CooLSM increases. This is because of the increased contention for compaction on the compactors. The ingestors make parallel compaction requests, but since each compactor is single-threaded, one request is handled at a time, while other concurrent compaction requests are queued, which increases the waiting time for these requests.

There is a significant throughout improvement as we move from one ingestor to two ingestors in Figure 13(b). This means that one ingestor is not enough to saturate the given number of compactors, so, using two ingestors improves the throughput significantly. Whereas, there is not much improvement in throughput for three ingestors because two ingestors are enough to saturate the given number of compactors.

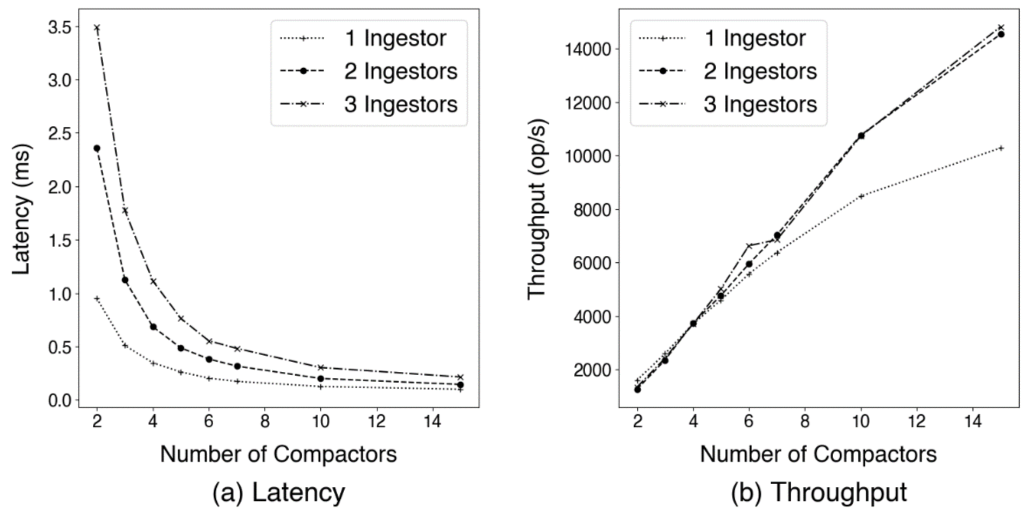

*Figure 14: The write performance of Multiple Overlapping Ingestors (key-range = 300K) with varying the number of compactors*

The same set of experiments is conducted for 300K key-range. Figure 14 shows a similar write performance like the 100K key-range in Figure 13. But there is a difference in the range of latency and throughput values. Since a bigger LSM tree is being used, the latency is higher, while the throughput is lower. Also, there is no improvement in throughput as we go from two intgestors to three ingestors because the LSM tree is three times in size, which already makes the compaction heavy and keeps the compactors busy. Thus, increasing the ingestors is not affecting the throughput because compactors are already saturated.

## 4.4 Edge Performance

In this section, we performed experiments to test the write performance of Edge-Cloud CooLSM. The cloud (comprising of five compactors) is placed at Virginia data center, and the edge (or ingestor) is placed at different locations, namely Virginia, Ohio, California, Oregon, and London. These locations are chosen based on their distance from the cloud datacenter, with Ohio being next to Virginia on East Coast, California, and Oregon on West Coast and London in Europe.

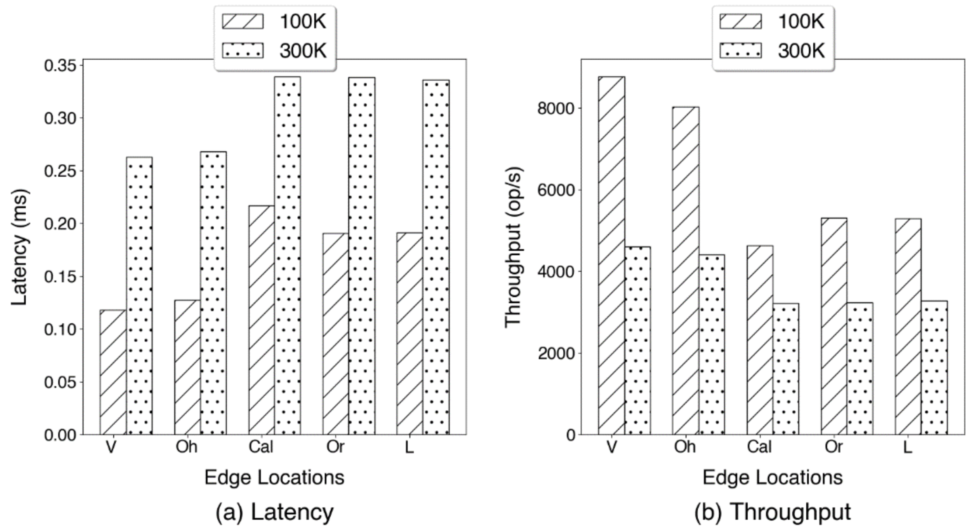

*Figure 15: The write performance of Edge-Cloud CooLSM (key-range = 100K and 300K) with cloud at Virginia and edge node at five different locations*

In Figure 15(a), it can be seen that the write latency is lowest for Virginia because the edge (or ingestor) is local to the cloud datacenter. Since Ohio is close to Virginia, the write latency is approximately equal to that for Virginia. Latency is much higher for edge nodes at California, Oregon, and London, as all these locations are about 3000 miles away from Virginia (cloud) datacenter. As seen in Figure 15(a), the latency pattern is similar for the 300K key-range.

Figure 15(b) shows the throughput for the Edge-Cloud experiment. Virginia edge node has the highest throughput, followed by Ohio and then lower throughput for other far off edge nodes.

## Section 5: Related Work

In LSM-based key-value stores, to maintain the LSM tree structure and avoid continuous building up of log files, compaction is a vital maintenance operation.

Over continued periods of upsert and delete operations, the same key spreads across multiple files and numerous deleted objects have not been reclaimed. Compaction helps in merging the same valued keys by discarding stale entries and deleting tombstones. Thus, over the long run, compaction helps in maintaining read latency of LSM tree as now fewer files need to be searched for a read request. However, merging these files is a heavy operation as many files need to be moved to-and-fro from the disk to the main memory, and this interferes with the CPU processing time. As a result, the actual workload gets affected and eventually read and write performance degrades.

In LSM trees, there are three main areas of research:

1. **Modifying LSM data structure or Compaction** improves the write throughput of LSM-based KV stores.
2. **Balancing Client workload and Compaction** defers or slows down compaction during peak client traffic, so that it does not compete for system resources.
3. **Offloading Compaction:** additional compaction servers are added whose task is to carry out compaction operation only, while the main server can handle the client workload.

## 5.1 Modifying LSM data structure or Compaction

### 5.1.1 PebblesDB [16]

PebblesDB is a key-value store built using Fragmented Log-Structured Merge Trees (FLSM), that combines two data structures: skip-list and LSM trees. Classic LSM Trees use Leveling as their compaction technique, which suffers from high write amplification because data is re-written multiple times during compaction. According to this paper, the main insight is that the base LSM maintains an invariant that each level has sstables with disjoint key ranges, and this is the root cause of write amplification. FLSM discards this invariant, thereby allowing at each level, multiple sstables to have overlapping key ranges. These sstables are organized into guards.

Each guard is associated with multiple sstables, which are arranged in a sorted sequence. Between two adjacent guards $G_i$ with key $K_i$ and $G_{i+1}$with key $K_{i+1}$, $G_i$ will be associated with a set of sstables having a key-range of $[K_i, K_{i+1})$. For compaction in FLSM, sstables belonging to a given guard at a level $L_i$ are merged and then fragmented/partitioned so that each child guard on the next level $L_{i+1}$ receives new sstables that fits into its key-range.

The FLSM data structure has two benefits. Firstly, it reduces the write amplification because now data is not re-written during compaction; rather, newly compacted sstables are simply added to the correct guard in the next level. Secondly, since guards are non-overlapping, sstables associated with one guard do not interfere with other guards and their descendants. So, now different guards can undergo compaction at the same time in parallel.

The drawback is that FLSM does not solve the problem of the compaction process interfering with the read/write workload of the system. The last two levels in FLSM still use classic leveling technique, which is again I/O intensive that interferes with CPU utilization.

**5.1.2 SifrDB [2]**

This paper classifies LSM-based KV stores into two models of data structures: MS-Tree, which is Classic Leveled Compaction, and MS-Forest, which is Size-Tiered Compaction. LevelDB is a variant of MS-Tree compaction, called split tree where multiple non-overlapping sstables exist in a level and compaction from level $L_i$ to $L_{i+1}$ is done by merging one sstable from $L_i$ and overlapping sstables from $L_{i+1}$ (10 sstables if exponential growth factor between levels is 10) and then these newly merged sstables are written back to level $L_{i+1}$.

SifrDB uses a novel compaction technique called Split-Forest, which is a variant of MS-Forest. At each level in LSM tree, each tree/sstable comprises of multiple non-overlapped and fix-sized subtrees whose key-range are recorded in a separate global index to give an essence of a logical tree. Compaction is done at the granularity of sub-trees with overlapped key-ranges in order to eliminate unnecessary data re-writing done in leveled compaction. SifrDB performs a variant of MS-forest compaction, so it inherits low write amplification. Using split storing mechanisms within each tree/sstable, SifrDB also exploits the advantages of LevelDB implementation.

In this work, a new compaction technique is implemented, which improves the read, write, and space requirements of LSM-based KV stores. However, this technique cannot guarantee that once compaction is triggered, it will not compete with the actual workload for CPU, memory, and I/O resources.

### 5.1.3 Mutant [17]

Mutant is a new storage layer for LSM-based KV stores implemented by extending RocksDB. Mutant provides a cost-effective key-value data store by dynamically keeping frequently accessed key-value pairs on fast storage like SSD and less-frequently accessed pairs on cheaper storage like HDDs.

Mutant exploits three main properties: (1) in modern workloads, the access patterns have strong temporal locality and popularity of records fade over time, (2) data that arrives in succession is grouped or batched into the same sstable, so each sstable also has a frequency pattern (3) each sstable is a portable unit that contains a subset of database records and hence can be migrated independently across various storage media. Mutant combines these properties and continuously migrates infrequently accessed sstables (cold sstables) to slower and cheaper storage devices.

Mutant explores a different area of research where it focuses on underlying storage media used for LSM-based key-value stores and how data access patterns can be used to reduce the storage costs.

## 5.2 Balancing Client workload and Compaction

### 5.2.1 SILK [18]

This paper defines two kinds of operations in an LSM tree: client operations, which are read and write requests made to the database, and internal operations, which are flushing and compaction. The main issue in LSM-based KVs is the interference between LSM's internal operations and client operations.

SILK, an LSM-based key-value store, has a monitoring tool that constantly monitors client traffic, and based on that, and it dynamically changes the I/O bandwidth. If there is a peak of client requests, less bandwidth is given to internal operations, and high bandwidth is allocated to compactions to catch up during low-load periods.

SILK categories internal operations based on priority. The highest priority is given to flushing so that the memory component of LSM always has space to absorb incoming updates. The second priority is given to L0/L1 compaction, which ensures that L0 does not reach its capacity. The last priority is given to compactions below the L1 level because they maintain the LSM tree structure but do not directly affect the client workload.

SILK introduced this notion of balancing between client and internal operations of LSM via I/O bandwidth allocation. Nevertheless, it still did not solve the problem of the compaction process interfering with the system workload. At times, when the client traffic is at its peak, and the underlying LSM tree also cannot further delay the compaction process, in such a scenario, SILK will not be able to allocate less I/O bandwidth to internal operations.

## 5.3 Offloading Compaction

### 5.3.1 Ahmad and Kemme [19]

In this paper, they added two new components to the current HBase [20] architecture (uses HDFS): a centralized compaction manager and a set of region servers called dedicated compaction servers. These compaction servers perform compaction on behalf of region servers. Each server in HBase hosts a datanode to access the underlying HDFS layer.

When a region server pushes data, it is written as a new store file to HDFS. Since compaction servers are region servers, they can directly access this data via HDFS. So, when a region server is about to trigger compaction, the dedicated compaction server will do the compaction instead and write back the compacted data to the HDFS itself, which can then be accessed by the region server. It is the compaction manager that maps region servers to compaction servers.

In this work, compaction is offloaded to separate servers. However, in order to do so, it uses the HDFS for transferring data. CooLSM deconstructs the LSM tree itself to offload the compaction, instead of using some filesystem.

## Section 6: Conclusion

In this thesis, we proposed Cooperative LSM (CooLSM), which is a distributed Log-Structured Merge Tree, designed to overcome the challenges of data indexing in Edge-Cloud computing. The main design principle of CooLSM is to deconstruct the monolithic LSM tree into smaller components, to support the real-time operations in an Edge-Cloud setting. Each component dynamically adjusts to the changing workload characteristics without affecting other component types. The main challenge we faced is the trade-off between consistency and performance. This is because components are distributed across wide-area links, so performing read and write operations while maintaining linearizability would require extensive coordination and overhead. So, we formulated two notions of consistency that relax linearizability and allow us to achieve better performance while having a notion of consistency guarantees. The first consistency notion we formulated is the **Linearizable + Concurrent** model, which aimed to model the potential inconsistency that resulted from having multiple ingestors. The second consistency notion we used is Snapshot Linearizability, which models the potential inconsistency resulting from lazily forwarding records to an offline reader. We conducted experiments to test the performance of different variants of CooLSM. The evaluation showed that CooLSM gives consistent read and write throughput guarantee for real-time applications, even during the heavy compaction process.

# Bibliography

[1] A. i. L. Trees, "Deep Dive TiKV," [Online]. Available: https://tikv.github.io/deep-dive-tikv/key-value-engine/B-Tree-vs-Log-Structured-Merge-Tree.html. [Accessed 06 06 2020].

[2] F. Mei, Q. Cao, H. Jiang and J. Li, "SifrDB: {A} Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter," in *Proceedings of the {ACM} Symposium on Cloud Computing, SoCC 2018*, Carlsbad, CA, USA, 2018.

[3] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Inf.,* vol. 33, p. 351–385, Jun 1996.

[4] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.,* vol. 12, p. 463–492, July 1990.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, Seattle, WA, WA: USENIX Association, 2006, p. 15.

[6] J. Dean and S. Ghemawat, "LevelDB," [Online]. Available: https: //github.com/google/leveldb. [Accessed Jan 2019].

[7] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang and J. Cong, "An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD," in *Proceedings of the Ninth European Conference on Computer Systems*, Amsterdam, The Netherlands: Association for Computing Machinery, 2014.

[8] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Commun. ACM,* vol. 13, pp. 422-426, July 1970.

[9] ScyllaDB, "Tiered Compaction," 17 Jan 2018. [Online]. Available: https://www.scylladb.com/2018/01/17/compaction-series-space-amplification/.

[10] ScyllaDB, "Leveled Compaction," 31 Jan 2018. [Online]. Available: https://www.scylladb.com/2018/01/31/compaction-series-leveled-compaction/.

[11] Facebook, "RocksDB: A Persistent Key-value Store for Fast Storage Environments," [Online]. Available: https://rocksdb. org. [Accessed Mar 2020].

[12] Facebook, "facebook/rocksdb/universal-compaction," [Online]. Available: https://github.com/facebook/rocksdb/wiki/Universal-Compaction. [Accessed 06 June 2020].

[13] H. Litz, B. Braun and D. Cheriton, "EXCITE-VM: Extending the Virtual Memory System to Support Snapshot Isolation Transactions," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, Haifa, Israel, 2016.

[14] D. L. Mills, "RFC0958: Network Time Protocol (NTP)," RFC Editor, USA, 1985.

[15] D. Bell and J. Grimson, "Distributed Database Systems," Addison-Wesley Longman Publishing Co., Inc., USA, 1992.

[16] P. Raju, R. Kadekodi, V. Chidambaram and I. Abraham, PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees, Shanghai: Association for Computing Machinery, 2017, p. 497–514.

[17] H. Yoon, J. Yang, S. F. Kristjansson, S. E. Sigurdarson, Y. Vigfusson and A. Gavrilovska, "Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores," in *Proceedings of the {ACM} Symposium on Cloud Computing, SoCC 2018*, Carslbad, CA, USA, 2018.

[18] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, Renton, WA, USA, USENIX Association, 2019, p. 753–766.

[19] M. Y. Ahmad and B. Kemme, "Compaction Management in Distributed Key-Value Datastores," *Proc. VLDB Endow.,* vol. 8, p. 850–861, Apr 2015.

[20] "Apache HBase," [Online]. Available: http://hbase.apache.org/.

[21] P. Hunt, M. Konar, F. P. Junqueira and B. Reed, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, Boston, MA, USA, 2010.

[22] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta and P. Konka, "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores," in *2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17)*, Santa Clara, California: {USENIX} Association, 2017.

[23] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.,* vol. 44, pp. 35-40, April 2010.

[24] P. Garefalakis, P. Papadopoulos and K. Magoutis, "ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees," in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, 2014, pp. 211-220.

[25] F. P. Junqueira, B. C. Reed and M. Serafini, "Zab: High-Performance Broadcast for Primary-Backup Systems," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, USA, 2011.