2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00043

CooLSM: Distributed and Cooperative Indexing Across Edge and Cloud Machines

Natasha Mittal
UC Santa Cruz
nmittal1@ucsc.edu

Faisal Nawab
UC Santa Cruz
fnawab@ucsc.edu

***Abstract*—We tackle one of the fundamental data management challenges in edge-cloud computing, the problem of data indexing. We propose *Cooperative LSM* (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 LSM tree [23] 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 and querying is managed at the cloud where more compute resources are available.**

***Index Terms*—data indexing, edge computing, cloud computing**

I. Introduction

Cloud computing is based on data centers that are typically far away from users and data sources. 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. 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.

In this paper, we propose Cooperative LSM (CooLSM), that 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 the storage and access to data. We build on Log-Structured Merge (LSM) trees [19], 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. Specifically, CooLSM—as shown in Figure 1(c)—breaks down a LSM tree—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 a distribution and placement of nodes as shown in Figure 1(d) 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. Each component of CooLSM can be placed on an independent machine and, thus, can be scaled independently of other components. For example, if there are a lot of analytics queries, the Reader components can be scaled to be on more machines than Ingestor and Compactor components.

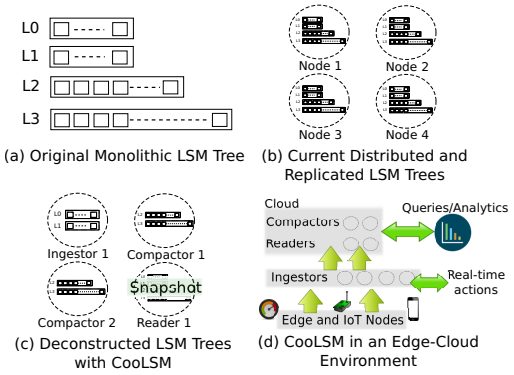


Fig. 1. CooLSM

One of the main challenges that we face in the paper 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 [14] would require extensive coordination and overhead. Our design is motivated by wanting to

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00043

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*, which aims to model the potential inconsistency resulting from lazily forwarding records to an offline reader. The second consistency notion we introduce is the *Linearizable+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.)

In the rest of the paper, we begin with presenting background (Section II). Then, we propose CooLSM design in Section III. Section IV presents our experimental evaluation. The paper concludes with a related work section (Section V) and a summary (Section VI).

II. Background

A. Target Applications

We are motivated to solve the performance challenges of emerging IoT/edge applications [22]. Many of these emerging applications require two types of access: low-latency access for recent data and complex analytics for global data at larger time ranges. This includes examples ranging from smart city and Industry 4.0 to Virtual/Augmented Reality-based mobile games and social networks. For example, a smart city traffic application would employ Vehicle-to-Everything (V2E) technology to allow vehicles to communicate with their surroundings including pedestrians, traffic signals, and other vehicles. Such an application exhibits local low-latency access for operations to coordinate between vehicles and other vehicles and pedestrians in intersections and highways. (A notable example are technologies where hints from vehicles are used for better utilization of highways and faster flow in intersections.) Doing so requires fast local coordination as it involves vehicles writing their data (*e.g.*, which route they are taking after this intersection) and others reading the data of recent vehicles in their intersection. Also, this application exhibits the need for more complex analytics to gain insights from the collected data. For example, city planning and monitoring agencies may want to study the traffic patterns by asking queries about the commuting routes of cars and the average delays they incur.

CooLSM aims to service such applications by providing Ingestors that are distributed at various edge locations for local time-critical actions. Also, CooLSM provides Compactors and Readers at the cloud that are capable of answering more complex queries. Although we present a single example—of smart city traffic—due to space constraints, this pattern or time-critical local action and complex analytics is exhibited in many emerging edge/IoT applications.

B. LSM Trees

LSM-tree [23] is not an in-place data structure, where every update is appended as a new entry rather than updating the old entry. This design targets the sequential I/O nature of the underlying disk which makes LSM-trees ideal for write-intensive workloads.

An LSM-tree [23] is composed of components (also called levels) L_0, L_1, ..., L_k, where each component itself is a B+ tree. L_0 resides in memory while all other components reside on disk. The pages in components are typically called *sstables*. When a component L_i is full, a rolling merge operation is triggered to merge the contents of L_i into L_{i+1}. The rolling-merge process is complex and modern LSM-tree variants use various compaction techniques. Two common compaction techniques are tiering and leveling. Tiering compaction merges all the contents of two levels and writes the newly merged content in the higher level. Leveling only compacts select pages from one level and merge them with sstables they overlap with in the next higher level.

In Figure 1(a), 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 disk. Compaction in L0/L1 is done through tiering, and compaction in the rest of the levels is done through leveling. Following common practice, L0 and L1 have the same size and a constant size ratio of 10 is used for higher levels.

An insert operation is performed via the following steps:

- The key-value pair is appended to an in-memory memtable where writes are buffered.
- If the memtable reaches its threshold size, the memtable entries are sorted based on their keys and the memtable is appended to L0.
- If L0 reaches its threshold size, this triggers a minor compaction between L0 and L1.
- After the minor compaction, if the threshold of L1 is exceeded, a major compaction is triggered. This continues for higher levels.

III. CooLSM Design

In this section, we present the design of CooLSM.

A. Design Overview and Motivation

Cooperative-LSM (CooLSM) is implemented by deconstructing the monolithic structure of LSM trees to enhance their scalability by utilizing the resources of multiple machines in a more flexible way. CooLSM consists of three components:

- **Ingestor** node receives the write requests. It maintains Levels L0 and L1 of the LSM tree.
- **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 of CooLSM are: (1) Different components can be placed across different machines (this includes Compactor and Reader components that are separate and can be placed on independent 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.
- Increasing the number of readers increases read availability.

Motivation in edge-cloud environments. Although the advantages above are general to any deployment, we are especially motivated by the advantages in 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 compute resources of the cloud. Placing Backups close to data sinks and consumers allows faster and more interactive analytics and read-only queries.

B. Architecture

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

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 a minor compaction is performed with L1. L0 has pages with overlapping key-range while higher levels have pages with non-overlapping key-range. When the number of pages in L1 exceeds the threshold 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 separate levels L2 and L3. When pages are received from an Ingestor, a major compaction of L2 is performed. If the threshold of L2 is exceeded, then L2 is compacted with level L3.

Each backup node represents a snapshot of the state. The backup can receive data from both Ingestors and Compactors and incorporates the data in its state. Clients wishing to read the state of the data can read from Backups, although they might be lagging behind the most recent copies in the Ingestors and Compactors.

In 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.

C. Core CooLSM (One Ingestor and Multiple Compactors)

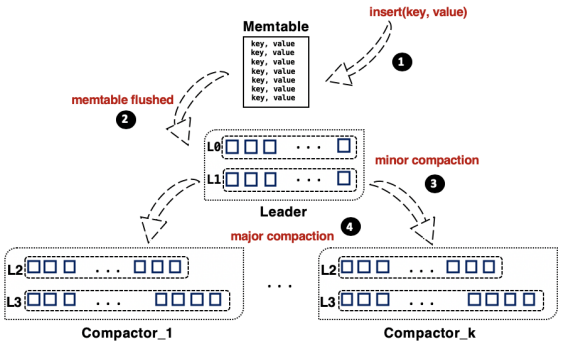


Fig. 2. Core CooLSM (One Ingestor and Multiple Compactors)

This architecture (core CooLSM) 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.

1) Upsert Flow (Figure 2)

The following are 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 table in L0. After the new table is added, the Ingestor checks whether the threshold of the number of tables (L[0].threshold) is exceeded. If it is, then the Ingestor initiates a minor compaction process that compacts all the sstables 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 key. Once all the key-value pairs are sorted, they are divided into ordered sstables, where the size of an sstable is predetermined. At this point, all the pages in L0 and L1 are cleared, and the new merged sstables are inserted to L1. This step is performed atomically.

After finishing 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 forwards 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 divides the sstable into different parts, where each part corresponds to a Compactor's range and then sends each part to its corresponding Compactor.

The Ingestor does not remove the forwarded sstable immediately. Rather, it waits to hear an acknowledgment from the compactor that the sstable is 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 the read path from the Ingestor to the 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 performed, the resulting ordered sstables are used to replace the L2.overlap pages in L2. This step is performed atomically. Then, it checks whether the number of sstables in L2 exceeds the threshold. If it does, the extra pages in L2 are merged with the overlapping pages in L3. Once the major compaction is done, the Compactor notifies the Ingestor.

2) Read Flow

Read operations are sent to the Ingestor. The Ingestor first checks the membtable and if not found, searches in level L0 starting from the most-recent table 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 [9] and fence pointers to speed up the process of looking through the sstables.

3) Safety

In this core architecture with one Ingestor and multiple Compactors, CooLSM guarantees Linearizability [14]. Linearizability is a guarantee that a data operation appears to happen instantaneously at some time between its invocation and return. An execution history $H_{lin.}$ represents a sequence of operations that satisfies the illusion of instantaneous invocations. An implication of linearizability is that if an operation a 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 a in $H_{lin.}$ is between the start and return time. Every past upsert operation is ordered before a in $H_{lin.}$ and every future upsert operation is ordered after a in $H_{lin.}$. 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 $H_{lin.}$ 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 time that is larger than c in $H_{lin.}$ 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 $H_{lin.}$ 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.

D. Adding Backup Components

1) Overview

A backup node maintains a snapshot of the data maintained by the compactors (the backup might also optionally maintain data from the Ingestors as we will discuss later in this section). The most recent key-value pairs might not be present in the backup. However, backups serve an important role to increase 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 two levels only, L2 and L3, since they only receive updates from Compactors. In the case there are more than one partitioned Compactor, the backup receive updates from all of them and integrate the sstables into one structure.

A client wishing to read from CooLSM has the option of reading 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.

2) 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 formulate a weaker consistency guarantee that is achieved when a client reads from a backup. We call this guarantee *snapshot linearizable*. Snapshot linearizability guarantees that for any two consecutive reads, r_1 followed by 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 and the

value corresponds to a past write operation, or (2) r_1 returns the value written by w_1 and r_2 returns the value written by w_2 and $w_1 <_{lin.} w_2$, where $<_{lin.}$ is an ordering of the two writes in the linearizable history $H_{lin.}$ 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 progression of 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 a 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 is applicable to objects within the range of a single Compactor. This is because with more than one Compactor, some Compactor's updates might be delayed in respect to others.

3) Backups with L0 and L1

In the description above, we assume that only Compactors forward data to Backup nodes. It is also possible to make Ingestors forward data to backup nodes. The implication of this change is that it makes the state of the backup more fresh as it does not need to wait for the updates to trickle down to the Compactors. The downside is that this modification leads to the need for more coordination at the backup to make sure that snapshot linearizability is achieved.

E. 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 a deployment of CooLSM with multiple Ingestors and multiple partitioned Compactors but no Backups.

The design and algorithms for Ingestors and Compactors are the same as presented in the core CooLSM design (Section III-C). 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. We discuss these issues in the rest of this section.

1) Consistency Anomalies

To demonstrate the consistency anomalies that are introduced with multiple Ingestors, consider an example with two Ingestors, I_1 and I_2, and one Compactor, C_1. Assume that two independent clients sent upsert requests, where client 1 sent o_1 =upsert(x=1) to I_1 and client 2 sent o_2 =upsert(x=2) to I_2. The two upsert requests happened concurrently, and we assume that there is no specialized time synchronization hardware that would allow accurate time synchronization.

Anomaly 1. Consider the scenario of a read operation, r_1 =read(x), that is issued while both records for o_1 and o_2 are still in their corresponding Ingestors. The read operation would be sent to both Ingestors, since the value of x could be in any one of them. When both Ingestors return both values for o_1 and o_2, the client cannot decide which one of the two values is safe to read while maintaining linearizability.

Anomaly 2. Consider another scenario after one of the Ingestors, I_2, sends the compacted sstables that contains o_2. At this time, o_1 is in I_1 and o_2 is in C_1. If a read operation is issued at this time, it will be sent to both Ingestors first. I_1 would return o_1 and I_2 would forward it to C_1, and finally C_1 would return 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.

Anomaly 3. The final scenario happens when I_1 finally sends o_1 to C_1 for a major compaction. When o_1 is received, 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 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 our relaxation of consistency guarantees to reason about the behavior of CooLSM with multiple Ingestors.

2) Relaxing Consistency: Linearizable+Concurrent

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—which is infeasible without expensive and specialized hardware. However, it is possible to use existing time synchronization technologies that guarantee loose-time synchronization such as NTP [21]. 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 definitely happened before the other) or concurrent (loose-time

[1]If the Ingestors' key ranges are non-overlapping, then we consider the Ingestors to be part of different data partitions, hence different instances of CooLSM.

synchronization cannot decide which operation was first if the timestamps are too close).

Specifically, if loose-time synchronization is deployed, each event e can be timestamped with a timestamp t_e. Each timestamp accuracy is bounded by some threshold δ, where the accurate global timestamp of t_e—denoted t_e^g—is somewhere in the range $t_e - \delta < t_e^g < t_e + \delta$. Using this formula, if the difference between two events timestamps is greater than $2.\delta$, then they can be ordered by their timestamps, *i.e.*, if $t_a - t_b \geq 2.\delta$, then $b <_t a$, 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 $2.\delta$. What remains are data items that are inserted with $2.\delta$ time of each other. Ordering such inserts is infeasible without additional coordination. To overcome the need for additional coordination, we formulate the lack of ordering between concurrent events 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 $t_a - t_b \geq 2.\delta$, then a must be logically ordered after b (Assuming that δ 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 loose-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 one Ingestor that will act as the coordinator of the read. This Ingestor will timestamp the read operation and forward the request to other Ingestors. The read will be served as-of the timestamp given at the coordinator—all nodes will ignore values with higher timestamps.

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 (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 (ts_h) as well as the lowest received ts_c. If $ts_h - ts_c \geq 2.\delta$, then the client knows that all the records in the Compactors are ordered before the record with timestamp ts_h. In such a case, the first phase terminates and there is no need for the second phase to ask the Compactors.

If the record was not found in all Ingestors or $ts_h - ts_c < 2.\delta$, then the client enters the second phase and sends read requests to the corresponding Compactors. 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.

The last modification is to ensure that a compaction process does not garbage collect recent values. The reason for this change is because an ongoing read operation will ignore values with timestamps higher than the read timestamp. If a compaction process garbage collects a value that would have been the most recent value as of a read timestamp, then the response would not be linearizable+concurrent. Therefore, values can be garbage collected only if the new value has a timestamp that is higher than the timestamp of any current or future read operation.

Theorem 1. *The modified CooLSM algorithms for multiple Ingestors guarantee Linearizable+Concurrent consistency.*

Proof. (Due to space constraints, we only include a proof sketch.) To prove by contradiction, consider the case of two operations o_1 with timestamp ts_1, and operation o_2 with timestamp ts_2. Also, assume that $ts_2 - ts_1 \geq 2.\delta$. We now show that for all possible cases, o_2 is logically ordered after o_1:

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

□

F. Consistency-Performance Trade-off Discussion

A main component of the proposed algorithms above is the consistency-performance trade-off that guided our new consistency definitions and designs. In this section, we summarize and discuss these consistency designs and provide an analysis of their trade-offs in comparison to each other.

TABLE I
SUMMARY OF CONSISTENCY DEFINITIONS IN COOLSM (MULTIPLE COMPACTORS ARE ASSUMED IN ALL CASES).

	Without Readers	With Readers
1 Ingestor	Linearizable	Snapshot Linearizable
Multiple Ingestors	Linearizable +Concurrent	Snapshot Linearizable+Concurrent

Table I summarizes the consistency definitions according to the deployment. There are two factors that decides the consistency level: whether there are 1 or more Ingestors, and whether there are Readers. In all these cases, we assume that there are 1 or more Compactors. In the following, we discuss the motivation and justification of each consistency level. Specifically, our goal is to retain as much consistency features while ensuring that no excessive coordination overhead is introduced.

The first case is with 1 Ingestor and no Readers (This corresponds to the core design in Section III-C.) In this case, we observe that although the Ingestor and Compactors are distributed, it is possible to retain strong consistency (linearizability) without additional excessive coordination.

The second case is with 1 Ingestor and with Readers (This corresponds to the design in Section III-D.) The motivation to add Readers is to allow users to query the database without interfering with the on-going operation in Ingestors and Compactors. Doing so is infeasible while maintaining linearizability since it would require Readers to coordinate with Ingestors and Compactors for all read operations. For this reason, we proposed snapshot linearizability that would relax linearizability to enable reading from Readers without coordinating with the rest of the system. Snapshot linearizability, however, retains the consistency feature of reading from a progressive snapshot of data, which is sufficient for the correctness of many analytics workloads.

Analysis: snapshot isolation can be achieved by passively receiving updates from Compactors without having to do any further coordination in response to read requests. This means that the only overhead that is introduced to existing nodes is the overhead of propagating updates from Compactors to corresponding Readers. Because this can be done asynchronously, its overhead is low (we present an evaluation of the overhead of adding Readers in Section IV).

The third case is with multiple Ingestors and no Readers (This corresponds to the design in Section III-E.) Adding Ingestors allows scaling the ingestion of CooLSM and to enable fast Ingestion at multiple locations. However, if we want to maintain linearizability, these Ingestors would need to coordinate which would prevent scaling performance and would make latency high when Ingestor are at distant locations. For this reason, we propose relaxing consistency and propose Linearizable+Concurrent, which allows concurrent ingestion without coordination while retaining the consistency and ordering of operations when they make to Compactors.

Analysis: Linearizable+Concurrent can be achieved by multiple Ingestors that do not coordinate with each other. The only extra coordination overhead is for relatively small meta-information that can be piggybacked in existing communication between Ingestors and Compactors. (Our evaluation section shows experimental results of varying the number of Ingestors.)

The fourth case is a composition of the second and third cases (having Readers and multiple Ingestors). This case is achieved by applying the changes in both Section III-D and III-E. Because the changes needed for each section are on different parts (supporting Readers requires adding Readers and propagating changes from Compactors to Readers and supporting multiple Ingestors requires changing Ingestor-Compactor communication.) Applying both changes would achieve a hybrid of both Snapshot Linearizability and Linearizable+Concurrent that we call *Snapshot Linearizable+Concurrent*. The justification and analysis of combining these two guarantees is the union of their individual justification and analysis as we describe for the cases 2 and 3 above.

Applications. The presented relaxed guarantees aim to allow the deconstruction and scaling of LSM components while minimizing coordination between them. This is motivated by edge/IoT applications with low-latency requirements (such as ones we discuss in Section II-A.) The addition of Readers (case 2 in Table I) is motivated by the need for efficient global analytics in some of these applications. For example, a smart city application, collecting data about traffic, would require functionality to perform large queries across large areas (*e.g.*, what is the number of accidents during the morning commute in a certain highway.) Snapshot Linearizability allows such queries to return consistent results as of a recent time in the past, which is sufficient for such analytics queries. The addition of multiple Ingestors is needed for scenarios where there is a need for scaling data ingestion of local time-critical requests or a need to have ingestors of overlapping data at multiple distant locations (Case 3 in Table I). Linearizable+Concurrent allows different Ingestors to operate without having to coordinate with each other. For example, in the smart city traffic application (Section II-A), the traffic sensors/cameras might be distributed across a large metropolitan area. To achieve fast ingestion and local response, Ingestors must be distributed across the city and be able to ingest/process data immediately without waiting to coordinate with other nodes. Linearizable+Concurrent allows this performance goal of recent data while retaining consistency and order of less-recent data after making it to Compactors. Case 4 (Table I) captures IoT/edge applications in the union of the discussed cases above.

G. 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 straight-forward 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 for 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 are more than one Ingestor and relax consistency for Backups in the same way we did with multiple Ingestors.

H. Fault Tolerance

CooLSM is a distributed system that may face various failure scenarios such as dropped messages and machine failures (that might manifest as timeouts when different nodes are communicating with each other.) To overcome the complexities of handling dropped (and more generally, unordered and delayed) messages, we use a communication framework that guarantees the ordered delivery of messages while handling network message drops, delays, and unordered messages. (We use Google RPC which uses a variant of the TCP protocol.) Handling machine failures is done by employing a recovery mechanism to enable recovering a consistent, recent state of operation after a failure. This includes both the data structure and the meta-information used for coordination with other nodes.

Another concern is ensuring continued operation during machine failures. Indexing technology and databases have leveraged prior work on high-availability and state-machine replication. Specifically, to make a component/node of CooLSM resilient to failures, its state would be replicated to $2f + 1$ nodes, where f is the number of failures to be tolerated. This replication can be made in a consistent and efficient manner using protocols like paxos [17]. For example, a Compactor node would replicate each step of its operation to a set of replicas, such as data received from Ingestors and compaction operations. The Reader Component can be fitted to serve this role of acting like a *backup or a replica* of a state-machine replication instance. For example, a Compactor would broadcast its changes to $2f$ Readers (making the total with the Compactor be $2f+1$ nodes) using a paxos process replicating an ordered log of operation steps. If a failure occurs, one of the Readers can assume the role of the Compactor via a leader election process until the original Compactor recovers. Similarly, this replication and leader election pattern can be used to persist the state of Ingestors.

I. Reconfiguration

To react to changing workloads conditions, we need to perform reconfiguration. We use existing reconfiguration methods used in distributed databases [16] [1] and adapts 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.

IV. Evaluation

We present a performance evaluation of 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 have 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 the 100K key-range, L0 and L1 have 10 sstables, L2 has 100 sstables and L3 has 1000 sstables. For the 300K key-range, L0 and L1 have 10 sstables, 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-cloud experiments, we use five datacenters in Amazon AWS. Virginia datacenter is the cloud where compactors are placed. Ohio, California, Oregon and London are used as edge locations where the Ingestor is placed.

A. 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. We include the performance of running CooLSM as a monolithic system. In this case, an Ingestor and a Compactor are colocated on the same machine and connected in a monolithic design so that network overhead is not incurred. The workload in this experiment is an all-write workload. Figure 3 shows the results of these experiments that are conducted in the Virginia data center.

In Figure 3(a), as the number of compactors increases, the overall write latency of CooLSM reduces (50% reduction from the monolithic case to having three compactors, followed by

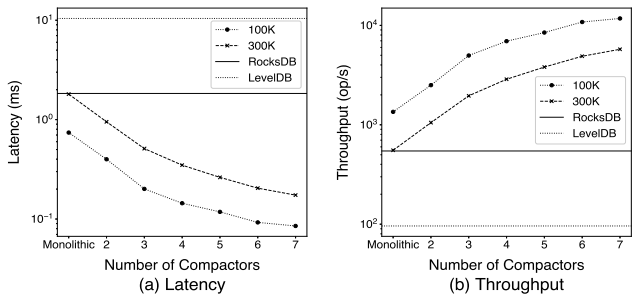


Fig. 3. The write performance of CooLSM while varying the number of compactors

Percentile	Value	Measure	Value
0.99	0.04ms	Average	0.11ms
0.999	1.4ms	Maximum	200ms
0.9999	100ms	latency>50ms	10 ops

TABLE II
LATENCY STATISTICS WITH 1 INGESTOR AND 5 COMPACTORS

25% reduction from three to six compactors and 15% from six to seven compactors). Note that 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 not beneficial. For the 300K key-range, the latency is higher than 100K key-range as the compaction involves more sstables due to a bigger LSM tree.

Detailed latency statistics about one of the cases (with 5 compactors) are shown in Table II. The percentile values show that most requests are fast (99% are below 0.04ms) and that a small fraction of requests take longer than 100ms. These are requests that trigger compaction and thus experience a higher latency. These are about 10 operations which take more than 50ms to complete.

The write throughput of CooLSM in Figure 3(b) increases 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, and as a result, the write throughput is lower than the 100K key-range.

For reference, we also compare CooLSM performance with two widely-used LSM-based systems: LevelDB[2] and RocksDB[3] (we run both with configuration to persist and sync to disk). These two systems offer more complex functionality than our CooLSM system which affects the comparison. Nonetheless, we aim to provide a reference point of existing systems to better interpret the results of CooLSM. The results show that the monolithic case of CooLSM is within milliseconds of the latency of the other two solutions. The positive effect that is experienced with deconstructing CooLSM and adding Compactors is a promising result that such an effect might be observed if applied to other systems such as LevelDB and RocksDB.

[2]https://github.com/google/leveldb
[3]https://rocksdb.org/

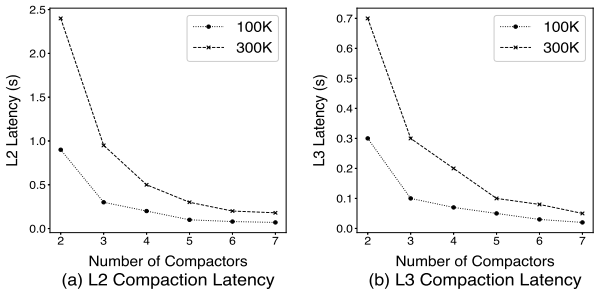


Fig. 4. The compaction latency of CooLSM while varying the number of compactors

Another set of write performance metrics that we analyze is how L2 and L3 compaction latencies varies with the number of compactors. In Figure 4, L2 and L3 compaction time are following a similar pattern for both 100K and 300K key ranges, with 300K taking more time. As the number of Compactors increase, the stress on each Compactor decreases leading to lower compaction latency. The magnitude of the increase is significant, lowering the latency from seconds to a few hundred milliseconds. The L3 compaction latency is lower than the L2 compaction latency. This is because in each cycle of major compaction, most of the incoming sstables are compacted and absorbed at L2 and only a smaller number of sstables overflow and get compacted in L3. This translates to more work in L2 compaction in each major compaction cycle.

Compaction is the main source of performance stress to the system, both in terms of computation and I/O. Therefore, the latency of compaction reflects the stress on the whole system and decreasing enables better scalability and performance stability. Our distributed design enables some degree of isolation between the compaction overhead and the performance of the other components of the system. However, since compactors communicate and coordinate with others, compaction overhead still has an effect on the rest of the system.

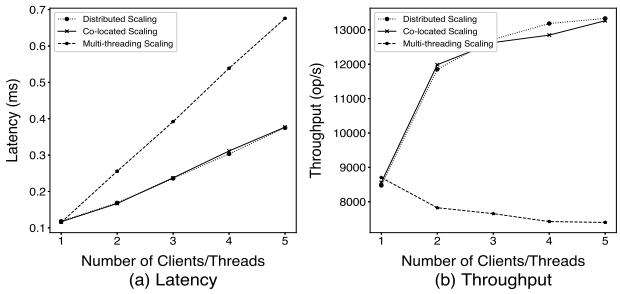


Fig. 5. CooLSM performance while increasing the number of clients

B. Scalability

In this set of experiments (Figure 5), we increase the number of clients in three ways: (1) Distributed scaling, where each client is placed on a separate machine colocated with an independent Ingestor, (2) Colocated scaling, where all clients are placed on the same machine and each client has its own independent Ingestor that is also in the machine, (3) multithreading scaling, where all clients are threads of the same client program and they all share the same Ingestor. The distributed and colocated scaling cases enable increasing performance. This increase is more significant from 1 to 2 clients. Multithreading scaling, on the other hand, does not scale with more clients. This indicates that one client is capable of stressing one Ingestor. This is partly due to the independence of the Ingestor that allows fast Ingestion, and thus a client would be able to rapidly issue operations back-to-back. This experiment also validates that scaling the number of Ingestors has the potential of increasing performance and overcoming stressed resources at the Ingestor.

C. CooLSM Read Performance

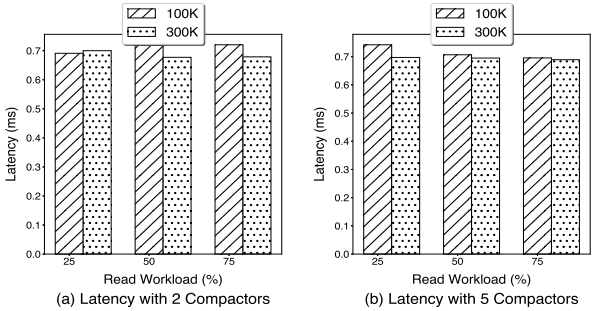


Fig. 6. The read latency of CooLSM (key-range = 100K and 300K) for two and five compactors while varying the read percentage

In this section, we perform a set of experiments to test the read performance of CooLSM without the backup server. We conduct two sets of experiments, one with two compactors (one compactor corresponds to the case of a 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 a mixed (both reads and upserts) workload for these experiments where we vary the percentage of reads (25%, 50% and 75%).

Figure 6 shows that CooLSM has a consistent read latency of about 0.7 ms (per read operation) for both 100K and 300K key ranges. This shows that using a larger LSM tree is not affecting the read performance. This is because of the use of fence pointers and Bloom filter. Fence pointers help in narrowing down the search to one sstable and bloom filters test 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 across the compactors.

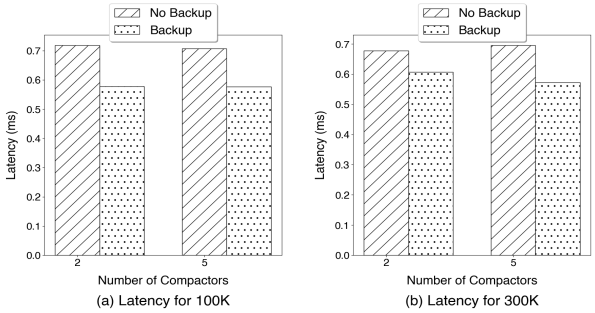


Fig. 7. The read performance of CooLSM (key-range = 100K and 300K) for two and five compactors with and without backup server

Introducing a backup server improves the read latency of CooLSM to about 0.6 ms (per read operation) as shown in Figure 7. This is because the read request is directly sent to the backup server instead of forwarding the request to compactors via the Ingestor, which reduces the network time. We emphasize that the main advantage of adding backup nodes is to isolate the service of read operations from upsert operations and to increase read availability. The lower read latency—though not significant—is an added benefit.

Backup nodes can be also used for availability and fault-tolerance (Section III-H). We performed an experiment where a scenario of five Compactors are replicating their updates to two backup nodes that act as replicas. This means that each Compactor can tolerate a failure. The overhead of this added communication to the Backup nodes increases the latency from 0.11ms to 0.17ms.

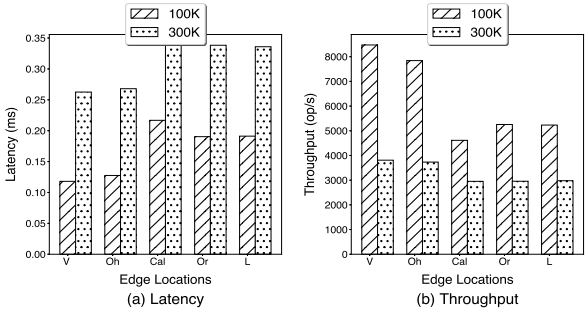


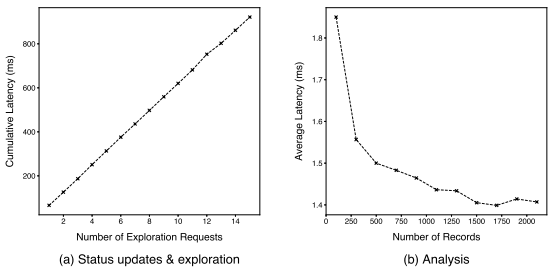
Fig. 8. The write performance of Edge-Cloud CooLSM (key-range = 100K and 300K) with cloud at Virginia and edge node at five different locations (Virginia, Ohio, Oregon, California, London)

D. Edge Performance

In this section, we perform experiments to test the write performance of Edge-Cloud CooLSM. The cloud (comprising of five compactors) is placed at Virginia 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 to the cloud datacenter, with Ohio being next to Virginia on the East Coast, California and

Real-Time WorkLoad		
Client Location	Ingestor Location	Latency(ms)
In cloud	In cloud	0.5584
California	California	0.8393
California	In cloud	122.485

TABLE III
PERFORMANCE OF REAL-TIME ACTIONS



(a) Status updates & exploration (b) Analysis

Fig. 9. Performance of the smart traffic benchmark.

Oregon on the West Coast and London in Europe.

Figure 8(a) shows the latency of write operations on the Ingestor at the edge. In all cases, the latency is between 0.1ms and 0.35ms. The reason for this low latency is that the write is served from the close-by edge Ingestor which masks the wide-area latency to the Compactors in the cloud. Although the Ingestor at the edge masks the wide-area latency to the cloud, this wide-area latency still impacts the performance of the system. This impact is due to the additional asynchronous communication and coordination overhead that affects the write latency indirectly as the wide-area latency increases, especially for compaction operations. For instance, the write latency is lowest for Virginia because the edge (Ingestor) is local to the cloud datacenter. The latency is also low in Ohio as it is close to Virginia. As the distance from the edge to the cloud increases, the latency also increases.

Figure 8(b) shows the throughput for the Edge-Cloud experiment. Throughput results mimic the observations we made for the latency numbers.

E. Real-Time and Mixed Workloads

In this section, we perform experiments to test the performance of CooLSM while emulating the workload patterns of real-life applications that we target (Section II-A). Specifically, we design a benchmark of smart traffic application based on the description we provide in Section II-A. In this application, each car is a client, and cars are distributed across a large metropolitan area. Each car has a record in the database that represents its information and current location. Continuously, the following operations are taking place in the benchmark:
(1) real-time action: this is a Vehicle-to-Everything (V2X) task where cars in an intersection update their status to be read by the other cars in the intersection. This action is emulated as a sequence of a write to a data item x (that represents the car or the intersection) followed immediately by a read by other nearby vehicles and/or pedestrians. Table III shows the results of running this operation. We test with three configurations varying the location of the client and the Ingestor. The benefit of CooLSM is that we can deploy the Ingestor close to the client (in California in this case) while the rest of the system is in the cloud (in Virginia in this case). The first row is a reference result when all operation is happening at the cloud. This represents the best-case performance. The second row represents the CooLSM case where the Ingestor is placed close to users. In this case, the latency is 0.84ms which is around 0.3ms higher than the best-case performance but still low enough to support the stringent latency requirements for edge and IoT applications. The last row represents the traditional case, when the data system is in the cloud faraway from users. In this case, the real-time action takes two lengthy round-trips to the cloud (each round-trip is around 60ms), one to write the update and another for other vehicles to read it. This shows how placing the Ingestor at the edge is crucial to overcome the high edge-cloud latency.
(2) Status updates and exploration: this is an exploration task where a moving vehicle continuously performs two steps: (i) write an update about its location and other information, and (ii) read the information of vehicles that are now in its vicinity. Step 1 is always a single write that is served at the nearby Ingestor. Step 2 depends on the number of vehicles nearby and system configuration. Some of these reads might be served from the Ingestor if the updates were recent and others would be served from the Compactors at the cloud. This task shows an example of a non-time-critical operation that may incur wide-area latency. Figure 9(a) shows the cumulative latency of each update/exploration request sequence as a function of the number of read requests (explorations) in step 2. As the number of explorations increase, the latency increases as well as a multiple of the number of needed round-trip messages to the cloud. In some cases, these reads can be batched, but in many applications, these reads need to be interactive, where the keys of future reads depend on the current read request, which is the case emulated in this experiment.
(3) Analytics: this is an analytics task, where an analyst is issuing queries to read the state of cars in a region of the city. These queries are served from a Backup node that is placed close to the analyst. The aim of this experiment to show that analytics reads can be fast by placing a Backup close to the analyst. Figure 9(b) shows the average latency of a read operation in a query while varying the number of read operations in the query. For small queries (less than 1000 records), the average read latency is between 1.45 and 1.85ms. As the query size becomes bigger, the average read latency becomes close to 1.4ms. The reason for this is that the associated overheads of initiating the query and making the connection to the backup node are amortized by the large number of read requests.

V. RELATED WORK

HBase [1] and BigTable [11] use size-tiered compaction, which suffers from space amplification. LevelDB [2] and

RocksDB [4] use leveled compaction, which suffers from high write amplification. RocksDB introduced Universal Compaction [5] in which sstables can overlap in key-range but avoid overlap in time-ranges. Time-range compaction has lower write amplification than key-range compaction, but it suffers from high space amplification. Other solutions propose reducing write amplification of LSM by changing the granularity of compaction [20], [24]. Others leverage data skewness and flush cold data while keeping hot data in memory [7], [25]. Monkey [12] and Dostoevsky [13] provide a detailed mathematical analysis of tuning LSM trees hyperparameters to improve its performance.

WiscKey [18] and HashKV [10] store key-value pairs in an append-only log while the LSM tree contains only keys to reduce the memory footprint of LSM-trees. Atlas [15] proposes to write the values onto a different set of servers and replace the values originally in the KV pairs with references (or pointers) to their respective locations in those servers.

SILK [8] uses the notion of balancing between the client (read-write requests) and internal (compaction) operations of LSM via I/O bandwidth allocation. To this end, a monitoring tool is used to make the allocations. RocksDB provides an auto-tuned rate limiter [3], which adapts the I/O bandwidth to the amount of internal work left, thereby allocating more bandwidth when there is more pending compaction.

We are motivated by the goal of offloading the compaction process to separate servers to mask the overhead of compaction from the rest of the system. Ahmad and Kemme [6] first introduced this concept by adding dedicated compaction servers to HBase, which performed compaction on behalf of region servers. 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.

CooLSM extends the idea of using separate servers for compaction beyond what was previously proposed [6]. Specifically, instead of only adding a replica to perform compaction (where the original server is still a monolithic LSM tree), we deconstruct the whole LSM tree structure into three basic components. Then, we study the implications of mixing and matching between different variants of these basic components as well as scaling each component individually.

VI. Conclusion

We propose CooLSM, a distributed edge-cloud indexing system. CooLSM's main innovation is the deconstruction of LSM trees to allow better flexibility and scaling. In this process we tackle challenges in the design and reasoning about correctness as well as explore opportunities in allowing scalable and elastic behavior of individual components. Our evaluation results show that deconstructing the LSM tree has performance benefits in terms of scaling ingestion, compaction, and increasing read availability. Furthermore, we show that it enables a more scalable solution for edge-cloud systems where data spans edge and cloud resources across wide-area links.

VII. Acknowledgements

This research is supported in part by the NSF under grant CNS-1815212.

References

[1] HBase. https://hbase.apache.org/.
[2] LevelDB. https://github.com/google/leveldb.
[3] Rate Limiter. https://github.com/facebook/rocksdb/wiki/rate-limiter.
[4] RocksDB: http://rocksdb.org/.
[5] Universal Compaction. https://github.com/facebook/rocksdb/wiki/universal-compaction.
[6] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *Proc. VLDB Endow.*, 8(8):850861, Apr. 2015.
[7] 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 *USENIX ATC*, pages 363–375, 2017.
[8] 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 *USENIX ATC*, page 753766, 2019.
[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422426, July 1970.
[10] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In *USENIX ATC*, pages 1007–1019, 2018.
[11] 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 *USENIX OSDI*, 2006.
[12] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *ACM SIGMOD*, pages 79–94, 2017.
[13] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *ACM SIGMOD*, pages 505–520, 2018.
[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463492, July 1990.
[15] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *MSST*, pages 1–14. IEEE Computer Society, 2015.
[16] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):3540, Apr. 2010.
[17] L. Lamport. The part-time parliament. *ACM Trans. Computer Systems*, 16(2):133–169, May 1998.
[18] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1), Mar. 2017.
[19] C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.
[20] F. Mei, Q. Cao, H. Jiang, and J. Li. Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. In *ACM SoCC*, pages 477–489.
[21] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.
[22] F. Nawab. Wedgechain: A trusted edge-cloud store with asynchronous (lazy) trust. In *IEEE Int. Conf. Data Engineering (ICDE)*, 2021.
[23] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351385, June 1996.
[24] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *SOSP*, 2017.
[25] H. Yoon et al. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *SoCC*, pages 162–173. ACM, 2018.