# CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure

QIAOLIN YU, Arizona State University and Cornell University, USA
CHANG GUO, Arizona State University, USA
JAY ZHUANG, Independent, USA
VIRAJ THAKKAR, Arizona State University, USA
JIANGUO WANG, Purdue University, USA
ZHICHAO CAO, Arizona State University, USA

Optimizing LSM-based Key-Value Stores (LSM-KVS) for disaggregated storage is essential to achieve better resource utilization, performance, and flexibility. Most of the existing studies focus on offloading the compaction to the storage nodes to mitigate the performance penalties caused by heavy network traffic between computing and storage. However, several critical issues are not addressed including the strong dependency between offloaded compaction and LSM-KVS, resource load-balancing, compaction scheduling, and complex transient errors.

To address the aforementioned issues and limitations, in this paper, we propose **CaaS-LSM**, a novel disaggregated LSM-KVS with a new idea of *Compaction-as-a-Service*. CaaS-LSM brings three key contributions. First, CaaS-LSM decouples the compaction from LSM-KVS and achieves stateless execution to ensure high flexibility and avoid coordination overhead with LSM-KVS. Second, CaaS-LSM introduces a performance- and resource-optimized control plane to guarantee better performance and resource utilization via an adaptive run-time scheduling and management strategy. Third, CaaS-LSM addresses different levels of transient and execution errors via sophisticated error-handling logic. We implement the prototype of CaaS-LSM based on RocksDB and evaluate it with different LSM-based distributed databases (Kvrocks and Nebula). In the storage disaggregated setup, CaaS-LSM achieves up to **8X** throughput improvement and reduces the P99 latency up to **98%** compared with the conventional LSM-KVS, and up to **61%** of improvement compared with state-of-the-art LSM-KVS optimized for disaggregated storage.

CCS Concepts: • **Information systems → Key-value stores**.

Additional Key Words and Phrases: LSM-tree, Key-value store, Disaggregated infrastructure

**ACM Reference Format:**
Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 124 (June 2024), 28 pages. https://doi.org/10.1145/3654927

## 1 INTRODUCTION

Nowadays, due to good write performance and better space efficiency, LSM-based Key-Value Stores (LSM-KVS) (e.g., Big Table [31], Cassandra [5], LevelDB [46], and RocksDB [19]) have become

Authors' addresses: Qiaolin Yu, Arizona State University and Cornell University, USA, qy254@cornell.edu; Chang Guo, Arizona State University, USA, cguo51@asu.edu; Jay Zhuang, Independent, USA, jay.zhuang@yahoo.com; Viraj Thakkar, Arizona State University, USA, viraj.dt@asu.edu; Jianguo Wang, Purdue University, USA, csjgwang@purdue.edu; Zhichao Cao, Arizona State University, USA, zhichao.cao@asu.edu.


ACM 2836-6573/2024/6-ART124
https://doi.org/10.1145/3654927

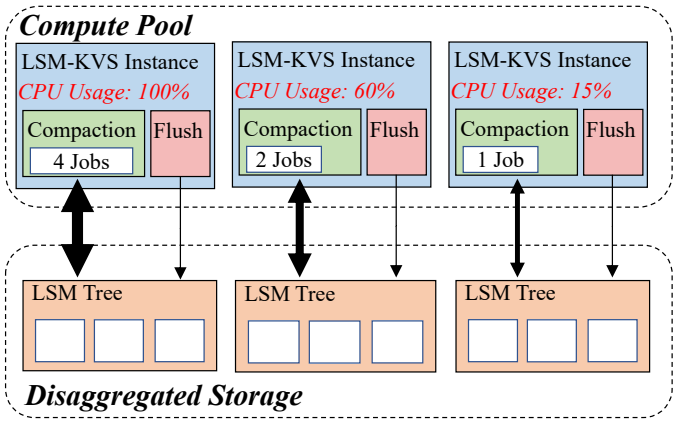



Fig. 1. Challenges in Disaggregated LSM-KVS

the backbone data storage systems for unstructured data and supporting various applications for cloud service providers and big internet companies [29]. Since LSM-KVS was initially designed and widely studied based on monolithic servers and shared-nothing architectures, the LSM-KVS-based distributed applications were developed to use thousands of LSM-KVS instances on hundreds of servers to store sharded data. Each LSM-KVS instance uses the local server's CPU, memory, and storage resources and needs to share these resources with other LSM-KVS instances on the same servers. As the application workload varies and scales up, LSM-KVS instances are facing a number of issues including resource wasting, load balancing difficulties, and low scalability [27, 42].

Therefore, deploying LSM-KVS-based systems and applications to a disaggregated infrastructure (or so-called disaggregated data center) has become attractive and a must in recent years [28, 56, 62, 74, 84, 85]. For example, Meta has built a new version of RocksDB to adapt to the disaggregated Tectonic File System (called Disaggre-RocksDB), achieving availability higher than 99.99999% [41, 42, 61]. In the storage disaggregated infrastructure, LSM-KVS is deployed at the compute node, and all the data is stored at the disaggregated storage. Furthermore, for large cloud service providers or large IT companies, moving data between different data centers is difficult while the LSM data can be accessed from any data center, making cross-datacenter scenarios inevitable [42]. In this case, accessing the data through the network can have longer latency and unpredictable throughput compared with the locally attached SSDs. Since compaction creates most of the I/Os from LSM-KVS (e.g., can be as high as 90% [70]), achieving near- or in-storage compaction is the main focus of related studies, including Disaggre-RocksDB at Meta [42], Nova-LSM, [49], RocksDB-Cloud [63], IS-HBase [28], TerarkDB [3], and Hailstorm [27].

However, existing studies [3, 27, 28, 49, 54, 63] including the state-of-the-art Disaggregated RocksDB [42] only partially address the performance issues caused by compaction, other resource utilization and transient error problems are fully ignored. **First**, the compaction offloading schemes proposed in the related work still tightly couple the compaction execution servers with the LSM-KVS instance. Specifically, each LSM-KVS instance can only send compaction requests to a pre-configured, fixed compaction server. However, since the LSM-KVS instances bound to each compaction server may have different workloads, some compaction servers will be burdened with heavy compaction jobs while others may remain idle. This situation leads to resource wasting and unexpected performance regression. **Second**, in a single LSM-KVS instance, compaction jobs are scheduled based on incoming orders or internal priority. However, in production, tens of KVS

instances are deployed in the same host [22, 29]. The lack of global scheduling causes resource contention between high and low-priority compaction jobs across instances, potentially impacting overall performance. Before compaction offloading schemes were proposed, compaction jobs were executed by each LSM-KVS instance individually and shard-migration was used to balance the overall resource utilization among different servers, without the ability for global scheduling and compaction-level optimizations. Currently, although remote compaction makes global scheduling of compactions feasible, it's still not addressed in current work [42]. **Third**, the existing design overlooks the high overhead associated with preparing compaction jobs for remote execution (it can even be longer than 10 seconds, as detailed in Section 3.2.3), at times making local compaction a preferable choice. The performance and resource tradeoffs between local and in-storage compaction must be considered. **Finally**, the possible storage I/O failures, network transient errors, and offloaded compaction execution failures are ignored in the existing work, which impacts the robustness and reliability of near- or in-storage compaction.

To comprehensively address the aforementioned major issues, we take a holistic approach to build a novel LSM-KVS with Compaction-as-a-Service called **CaaS-LSM** to achieve high-performance, flexible, resource-efficient, and high fault-tolerant LSM-KVS design for disaggregated storage. We propose to decouple the compaction logic from the LSM-KVS instance and execute it as a *lightweight stateless service*. Instead of executing the compaction job locally, LSM-KVS encapsulates the compaction information that is essential for an independent execution including job description, data access authorizations, and control information in the compaction request. The compaction requests are sent to a compaction service execution plane for near- or in-storage execution to avoid creating network traffic between the compute and storage clusters. The execution plane consists of a cluster of Compaction Service Agents (CSA), which are deployed inside the storage nodes or at the compute nodes close to the storage cluster (typically one CSA per node).

More importantly, to improve the overall performance, achieve high resource utilization, and comprehensively handle the various failures, we propose a performance- and resource-optimized control plane in CaaS-LSM. First, the control plane decouples LSM-KVS instances from the execution plane. LSM-KVS cluster or CSA cluster can be scaled in or scaled out without causing changes on other components. Second, to achieve better overall performance, we propose an LSM-specific scheduling algorithm in the control plane to properly arrange the execution order and execution place (i.e., which CSAs should execute the job) of compaction job requests based on priorities. In particular, the scheduling algorithm achieves dynamic allocation of compaction job requests to the execution plane and LSM-KVS local compaction based on the available resources and compaction job intensiveness for a better tradeoff between performance and resource utilization. Finally, to handle various and complex errors and failures, we propose to classify the failures into service execution failure, CSA failure, and execution plane failure and achieve fine-grained error handling.

We implement CaaS-LSM based on RocksDB [19], which is one of the most widely used LSM-KVS. The implementation of our compaction service is merged to RocksDB 7.0 and above versions, and the control plane of CaaS-LSM is open-sourced at GitHub [1]. We evaluated our approach based on different disaggregated storage setups, including in-rack clusters, within the same region, and across different datacenters. In our evaluation using the standard db_bench benchmark for RocksDB [6], compared with legacy LSM-KVS architecture (all LSM-KVS instances are at compute nodes), CaaS-LSM can achieve up to **8X** Operations Per Second (OPS) improvement, reduce up to **98%** of P99 latency, and avoid **99%** of write stalls in cross-datacenter environments. In the in-rack deployment, the OPS of CaaS-LSM surpassed SOTA Disaggre-RocksDB [42] by up to **61%**. We also integrate our proposed control plane with TerarkDB (TerarkDB-CaaS), which achieves up to **42%** of

[1]https://github.com/asu-idi/CaaS-LSM

throughput improvement than that of native TerarkDB [3]. Moreover, to comprehensively evaluate the end-to-end application performance improvement, we integrate CaaS-LSM with Nebula [21] (a distributed graph database) and Kvrocks [1] (a distributed key-value store), which use CaaS-LSM as their storage engines. For Nebula, CaaS-LSM achieves up to **8X** of graph query OPS improvement and **89%** of average latency reduction compared with Nebula with legacy RocksDB. For Kvrocks, CaaS-LSM improves up to **56%** key-value query OPS and reduces the P99 latency up to **76%**.

**Contributions**. Main contributions are summarized as follows:

- We propose the first Compaction-as-a-Service architecture (CaaS-LSM) to achieve lightweight stateless compaction independent of the LSM-KVS instances, which is optimized for disaggregated storage. CaaS-LSM successfully addresses the issues of heavy network I/O penalties and the on-demand nature of compaction with a novel stateless execution.
- Compared with existing studies including Disaggre-RocksDB and TerarkDB, CaaS-LSM achieves better performance, higher resource utilization, and sophisticated failure handling by introducing the novel control plane for CaaS.
- We implement and open-source CaaS-LSM based on RocksDB, and conduct comprehensive evaluation in various deployments and real-world applications, which demonstrate the benefits and tradeoffs.

## 2 BACKGROUND

### 2.1 LSM-based Key-Value Stores

Due to good write performance and better space efficiency, LSM-based Key-Value Stores (LSM-KVS) (e.g., Big Table [31], Cassandra [5], LevelDB [46], and RocksDB [19]) become the backbone data storage systems for unstructured data and supporting various applications in cloud service providers and big internet companies [29]. LSM-KVS batch small random writes (KV-pairs) including Puts, Updates, and Deletes in the memory write buffer, and persist KV-pairs in the write-ahead log (WAL). When the write buffer is full, KV-pairs will be written out as a Sorted String Table (**SST file**) in Level-0 (L0). SST files are organized in multiple levels and periodically merged into new SST files at a higher level during a `Compaction` to eliminate the deleted or invalid KV-pairs [41, 51, 52]. However, compaction introduces extra overhead including read/write amplifications, tuning complexity, resource contention, performance penalties, and scheduling difficulties [7, 73, 81, 86].

### 2.2 Disaggregated Infrastructure

In a disaggregated infrastructure, servers are built with heterogeneous hardware resources and form resource pools, such as computing pools, memory pools, and storage pools in the same data center. It decouples different types of resources and can address the scalability, resource-wasting, and management limitations of conventional monolithic server-based data center designs by adding or removing some resources to create a much more balanced cost-effective system. Resources pools are connected with high-speed networks [45, 78]. Storage disaggregation has been well developed, such as Tectonic [61] at Meta and GFS [47] at Google. Applications at the compute cluster can read/write data at the storage cluster through different interfaces such as block or file interfaces.

Since LSM-KVS is initially designed and widely studied based on monolithic servers and shared-nothing architectures, the LSM-KVS-supported distributed applications will use thousands of LSM-KVS instances on hundreds of servers to store the sharded data [22, 29, 41]. Each LSM-KVS instance uses local CPU, memory, and storage resources and needs to share resources with other LSM-KVS instances on the same server. As the application workload varies and continuously scales up, LSM-KVS instances are facing a number of issues including resource wasting, load balancing difficulties, low scalability, and poor elasticity [27, 42]. Therefore, deploying LSM-KVS-based applications to

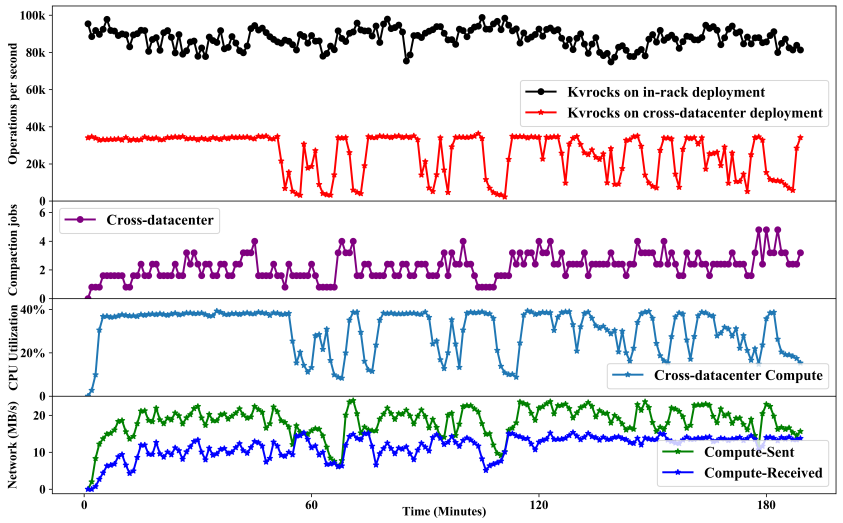


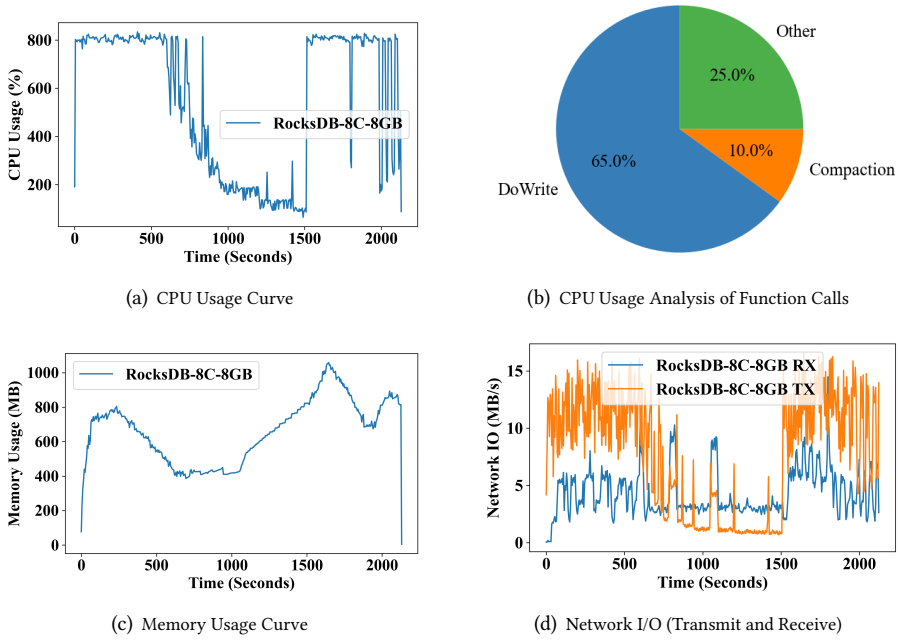

Fig. 2. Write Performance of Single Kvrocks Instance under Disaggregated Infrastructure.

a disaggregated infrastructure has become attractive and a must in recent years [28, 56, 84, 85], illustrated by Meta's disaggregation of RocksDB with the Tectonic File System [42], Nova-LSM for storage disaggregated environments [49], and Rockset's RocksDB-Cloud optimizations [63].

## 3 MOTIVATION AND CHALLENGES

While the emerging trend of disaggregated infrastructure provides promising and beneficial potentials for LSM-KVS, it also generates several fundamental challenges. On one hand, for large cloud service providers or large IT companies, moving data between different data centers is difficult while the LSM data can be accessed from any data center, making cross-datacenter scenarios inevitable [42]. Since the execution of the LSM-KVS is carried out at a compute node, all the needed data is transferred from storage nodes through the network to the compute node. This substantial data traffic incurs explicit performance degradation, attributable to network bandwidth contention and increased latency as corroborated by Dong's studies [41, 42]. The majority of the traffic is caused by compaction. On the other hand, existing LSM-KVS is designed based on monolithic-server and shared-nothing architecture. This lack of synergy impedes the realization of benefits associated with storage sharing, resource elasticity, and enhanced management capabilities that fundamentally block the comprehensive benefits imbued in a disaggregated infrastructure paradigm.

As shown in Figure 2, we run a Kvrocks instance (a RocksDB-based distributed key-value store) with Redis benchmark `Set` queries for over 3 hours at in-rack and cross-datacenter disaggregated deployment (the setup details are presented in Section 5). For the in-rack deployment, we can observe OPS variation (varies from 75K to 100K) caused by the periodic compaction operations. When the network bandwidth drops from 100 MB/s (in-rack) to 10 MB/s (cross-datacenter), the Kvrocks instance shows deeper and longer OPS drops (e.g., from 37K to 1K). We analyze the logs and observe recurrent write stalls in cross-datacenter deployments, primarily attributed to the accumulation of compaction jobs. The concurrent execution of foreground operations and compaction processes contends for the limited network bandwidth, resulting in the buildup of unfinished or waiting compaction tasks. As foreground operations diminish, both CPU utilization and the data transmitted to the storage exhibit corresponding fluctuations. In general, addressing




(a) CPU Usage Curve

(b) CPU Usage Analysis of Function Calls

(c) Memory Usage Curve

(d) Network I/O (Transmit and Receive)

Fig. 3. RocksDB runtime CPU usage, function call analysis, memory utilization, and network I/O (transmit and receive).

the performance issues caused by compaction becomes a must when optimizing LSM-KVS for disaggregated infrastructure.

### 3.1 Bottleneck Analysis

To comprehensively analyze the performance issues caused by the compaction of existing LSM-KVS with storage disaggregation, we conduct a set of experiments on RocksDB. We examine several potential factors, including CPU, memory, network latency, and I/O failures. We use RocksDB's db_bench to generate various workloads (configured as 16 client threads, 100B KV-pair size, 8 background RocksDB threads, 30 million random KV-pair insertion). The RocksDB instance is reserved with 8 CPU cores and 8GB of memory, and all the data is written to a disaggregated storage (here we use HDFS which is similar to Tectonic File System [61] at Meta). We monitor and record CPU, memory, and network utilization as shown in Figure 3. We also use Linux perf to analyze CPU utilization of various functions in RocksDB shown in Figure 3(b).

**CPU Resource Wasting** As shown in Figure 3(b), a vast majority of CPU is occupied by write threads and compaction threads (the occupation classified as "other" in the chart is, in many cases, a result of contention between write and compaction operations). As shown in Figure 3(a), when the CPU utilization rate reaches its full capacity (800%), compaction and foreground operations (e.g., Put) are competing on the CPU resources. It will lead to foreground operation throttling and overall performance regression. On the other hand, there are substantial periods where CPU usage significantly declines (e.g., 200%). This typically happens in two scenarios: 1), write slowdowns or write stalls happen due to an excessive accumulation of Level 0 files. It is usually caused by a large number of waiting compaction jobs. And 2), when the number of compaction jobs is relatively low, the 8 background threads are not fully used. Therefore, decoupling compaction jobs from LSM-KVS

and executing them together with proper scheduling and balancing strategies can achieve better tradeoffs between performance and resource utilization in the disaggregated infrastructure.

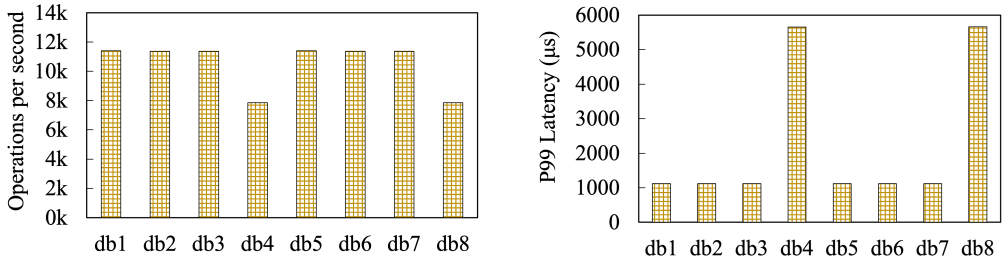
**Network Influence** With disaggregated storage, the I/O throughput and latency are determined by the network conditions. As shown in Figure 2, when the network condition is worse (e.g., changing from in-rack to cross-datacenter deployment), the overall OPS drops significantly. More seriously, compaction is the main cause of storage I/Os leading to heavy network traffic, which results in a deeper OPS drop and a longer OPS recovery time. Therefore, dynamic allocating compaction job requests between LSM-KVS and remote compaction workers can effectively utilize the network resources and avoid contentions.

### 3.2 Limitations of Existing Studies

To mitigate the impact of compaction in disaggregated storage, existing studies propose to separate the compaction process from the LSM-KVS instance and run it remotely [3, 27, 28, 42]. Meta has built disaggregated RocksDB in the Tectonic File System (called Disaggre-RocksDB) [42], which links one or more LSM-KVS instances to a specific remote compaction worker (e.g., deployed near or in the storage cluster). However, this tightly coupled integration can potentially lead to either compaction worker resource wasting or compaction job waiting. Similar issues also exist in the remote compaction design of IS-HBase [28], RocksDB-Cloud [63], and Nova-LSM [49]. TerarkDB [3] implemented remote compaction with Function as a Service (FaaS), which uses a general job scheduling system like Mesos or Kubernetes to schedule the compaction requests. Similarly, Hailstorm [27] monitors cluster instance availability and offloads compaction jobs to idle instances for workload balancing. However, it cannot handle compaction jobs with different priorities and complex transient I/O errors.

*3.2.1* ***Inefficient Resource Utilization.*** Most of the existing work uses the static connection between LSM-KVS instances and the remote compaction workers [41, 42, 49]. Due to the unbalanced compaction job intensiveness among different LSM-KVS instances, it often happens that certain compaction workers are extremely busy, while others remain idle. We use Disaggre-RocksDB [42] to demonstrate such resource wasting issue. We run 8 RocksDB instances (8 RocksDB LSM-KVS instances) in 2 compute nodes at different start times, which will trigger compactions from different SST levels at different times. We deployed 2 compaction workers in the same region as the disaggregated storage. Each compaction worker is responsible for the compaction jobs from 4 LSM-KVS instances. Each RocksDB instance receives the queries from 4 db_bench client threads to generate the random write with 16 million KV-pairs (100B size) in total. The results of OPS and P99 latency are shown in Figure 4. In Disaggre-RocksDB, since their compaction jobs are executed until the pre-allocated compaction worker has a free worker thread, 2 of the 8 LSM-KVS instances (db4 and db8) have much lower OPS (OPS decreases 33% from 12K to 8K as shown in Figure 4(a)). Therefore, it is essential to decouple the LSM-KVS instances from remote compaction workers.

*3.2.2* ***Remote Compaction Job Scheduling Issues.*** For a single LSM-KVS instance, compactions are usually scheduled based on the compaction job incoming order (i.e., FIFO) or optimized with priority-based scheduling [26, 33, 60]. However, in production, one host usually deploys tens or even several hundred LSM-KVS instances (e.g., cluster deployment of ZippyDB [22] or Kvrocks [1]). Compaction jobs across different LSM-KVS instances are scheduled independently. Urgent compactions (e.g., L0 or L1 compaction) can be delayed by long-running low-level compactions (e.g., bottom-level compaction). Therefore, ignoring the importance and priority difference of compaction jobs from different LSM-KVS instances can lead to overall application performance degradation.




(a) OPS of multiple db_bench (b) P99 latency of multiple db_bench

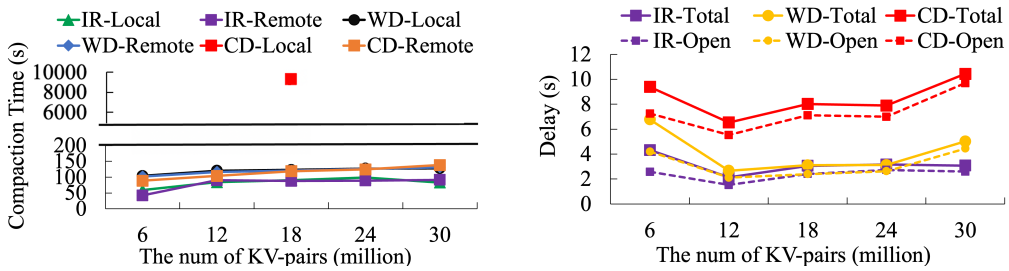
Fig. 4. Performance imbalance in Disaggre-RocksDB.

*3.2.3* ***Balancing Between Local and Remote Compaction.*** Although executing compaction remotely offers numerous benefits, preparing the compaction execution outside the LSM-KVS instance can incur explicit extra overhead, which is overlooked in existing studies. We run db_bench under various network environments on Disaggre-RocksDB [42], ensuring an abundance of CPU and memory. Rocks-Local (execute compaction inside RocksDB instance at compute node) and Disaggre-RocksDB (run compaction remotely at disaggregated storage nodes) are evaluated with different deployments: in-rack Rocks-Local (IR-Local), in-rack Disaggre-RocksDB (IR-Remote), within-datacenter Rocks-Local (WD-Local), within-datacenter Disaggre-RocksDB (WD-Remote), cross-datacenter Rocks-Local (CD-Local), and cross-datacenter Disaggre-RocksDB (CD-Remote). We collect the average time between the start of a compaction job and its completion (Compaction Time), the average time of preparing remote compaction execution (Open DB Delay), and the average elapsed time between the creation of a remote compaction job and the starting of its execution (Total Delay), which includes the Open DB Delay (Open) and network latency.

As shown in Figure 5(a), the Compaction Time of IR-Local, IR-Remote, WD-Local, WD-Remote, and CD-Remote are very similar. However, the situation with CD-Local is different. Owing to the high latency of reading SST files from the disaggregated storage at another datacenter, the compaction process is considerably slower. When inserting 18 million KV-pairs, the Compaction Time of CD-Local is 9,469 s. Figure 5(b) indicates that the total delay (i.e., IR-Total, WD-Total, and CD-Total) caused by remote compaction is dominated by the delay of preparing the remote compaction execution (i.e., IR-Open, WD-Open, and CD-Open). For LSM-KVS instances with a much larger data scale, the compaction worker needs a longer preparation time to initialize a lightweight compaction instance for execution since it needs more time to reconstruct the in-memory data structure from the Manifest file.

Therefore, we need to address the delay overhead of preparing the remote compaction by achieving the balance between remote compaction and local compaction. Remote compaction is not a silver bullet. When LSM-KVS has enough resources (i.e., enough network and CPU resources, and fewer compaction jobs), executing the compaction job locally at the LSM-KVS instance can be more advantageous than executing it remotely. However, existing solutions lack the flexibility and tradeoffs to combine the advantages of remote compaction and local compaction.

*3.2.4* ***Failure Handling.*** When executing compaction remotely at the storage cluster, various transient errors and failures can happen, such as storage I/O failure, transient errors between the compaction worker and LSM-KVS, and failures of the compaction workers. Also, since more software stacks, network stacks, and distributed protocols are involved during the remote I/Os (e.g., local file systems, integrity checks, replications, and network transmissions), the failures can happen frequently at different software levels. Therefore, a comprehensive failure handling is




(a) Overall Compaction Time of Disaggre-RocksDB (b) Total Delay and Open DB Delay of Disaggre-RocksDB

Fig. 5. Compaction time and compaction worker open DB delay evaluation of Disaggre-RocksDB in different deployments of disaggregated storage systems.

essential for LSM-KVS with compaction offloading to guarantee reliability and robustness, which is fully ignored by existing studies.

### 3.3 Challenges

Motivated by the aforementioned analysis, we are aiming to design a novel LSM-KVS for disaggregated storage with high performance, high scalability, better resource efficiency, and high fault tolerance. However, three main challenges need to be addressed:

- How to address the scalability and dependency issues of existing compaction offloading designs, including dynamically adding and removing compaction workers and independent execution of compaction jobs.
- How to design a logically centralized control plane to satisfy monitoring, scheduling, and management requirements for remote compaction execution and to decouple LSM-KVS instances and the compaction workers.
- How to achieve high elasticity, high performance, and high fault tolerance with the control plane, including compaction job scheduling, compaction worker selection, dynamic compaction job distribution between local and remote, and comprehensive error handling.

## 4 CAAS-LSM DESIGN

To resolve the aforementioned issues and limitations of existing studies on optimizing LSM-KVS with disaggregated storage and address the main challenges, we propose a novel LSM-KVS architecture with Compaction-as-a-Service (***CaaS-LSM***). First, To achieve highly flexible and scalable compaction execution, CaaS-LSM encapsulates the compaction logic as a stateless compaction service execution function (i.e., compaction service), which can be executed independently at any host without impacting the running state of LSM-KVS. Second, to improve the scalability and management capability of the compaction service, we propose the **compaction service execution plane**, which is responsible for executing the compaction job requests from different LSM-KVS instances and manages the compaction workers. Third, to address the performance, resource utilization, and failure handling issues, we further propose a **compaction service control plane**, which decouples LSM-KVS from the execution plan, precisely manages the execution of compaction services, and achieves sophisticated compaction job scheduling and failure handling.

### 4.1 System Overview of CaaS-LSM

The overall architecture of CaaS-LSM is shown in Figure 6. LSM-KVS instances are deployed on compute nodes and their data is stored on storage nodes. Ideally, both execution and control planes

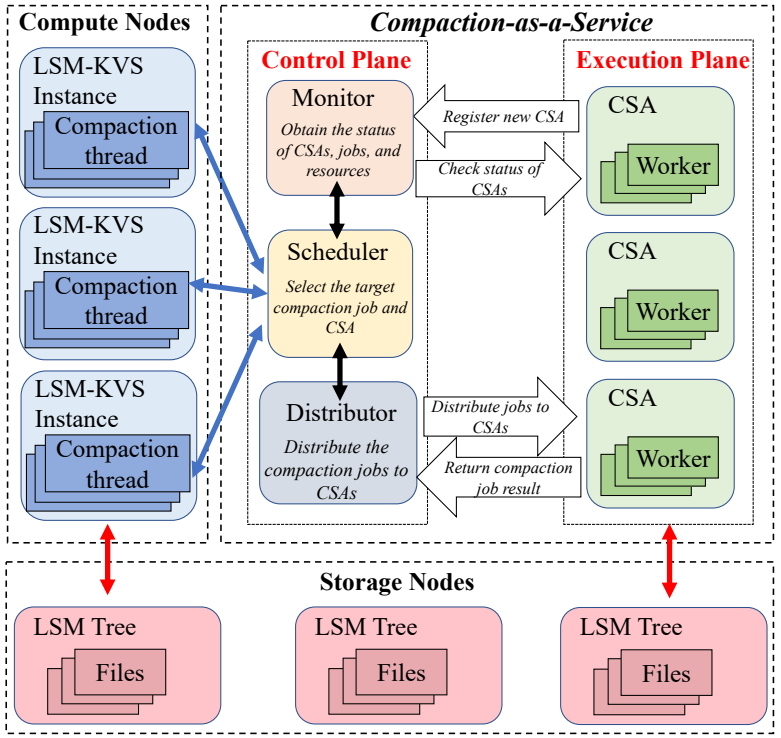



Fig. 6. The overall architecture of CaaS-LSM.

are deployed close to or inside the disaggregated storage cluster to avoid heavy networking overhead between compute and storage nodes during compaction. The compaction service execution plane (discussed in Section 4.2) consists of a cluster of Compaction Service Agents (CSA). If one node is selected for compaction service execution, a CSA daemon will be deployed at this node. When a compaction request is created by LSM-KVS, it carries all the metadata needed to execute the compaction job independently. Therefore, each CSA is stateless, allowing the CSA cluster to scale in or out freely. The compaction service control plane (discussed in Section 4.3) schedules the compaction job requests based on the priority of the compaction job and the resource of the CSAs and LSM-KVS, handles various failures, and achieves high scalability. The communication across LSM-KVS instances, control plane, and execution plane are lightweight metadata and control messages, which utilize small network resources through TCP/IP protocols.

The workflow of using the compaction service in CaaS-LSM is shown as follows: ① When an LSM-KVS initiates a compaction job, it sends the compaction information to the connected control plane; ② Upon receiving a compaction job, the control plane determining the execution place (e.g., local or remote), sequence of the piled-up compaction jobs, and the CSA (if it is executed remotely) based on the proposed scheduling algorithm; ③ If all CSAs are busy, the control plane can reassign the compaction job back to the LSM-KVS instance, allowing it to fall back to local execution (hybrid-scheduling at Section 4.3.2); ④ Once a CSA receives the compaction job request from the control plane, it launches a compaction worker thread to execute the request. The compaction worker reads the target SST files directly from the storage nodes to avoid the traffic between LSM-KVS instances and the storage nodes. Then, the compaction worker executes the compaction

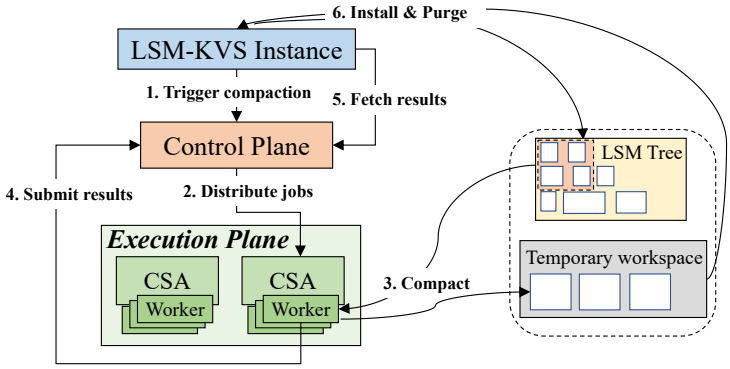



Fig. 7. Architecture of execution plane, and its relationship with the LSM-KVS instances and control plane

and generates the compaction results (i.e., new SST files) at a temporary workplace; ⑤ After the execution, the CSA submits the results to the control plane; ⑥ Upon receiving the successful compaction results, the LSM-KVS instance double-checks the correctness of the execution status and SST file information; ⑦ Subsequently, the LSM-KVS instance renames and installs new SST files by updating the Manifest and in-memory data structure. Once the new SST files are successfully installed, the old input SST files are purged.

Since CaaS-LSM is a framework to execute the compaction job remotely as a service and retains all the original function modules of LSM-KVS, the CaaS-LSM design is fully compatible with the existing LSM-KVS designs and can be integrated with other LSM-KVS-related optimizations. For example, CaaS-LSM uses the same block cache strategy, filter construction, write buffer management, and compaction policies as Disaggre-RocksDB [42], which maintains the block cache, write buffer, and block-based table builder at compute node locally. CaaS is designed as an alternative way of executing compaction jobs with more flexibility. Even if the compaction service fails to execute a compaction request, there is no corruption or consistency risk since the status of the LSM-KVS will be changed only after a successful compaction job execution. To improve resource utilization, applications can reduce the CPU resource reserved for LSM-KVS local compaction without influencing the overall performance. Different upper-level applications that use CaaS-LSM instances as their storage engines can share the same control plane and execution plane, which better balances the workload and improves the CPU utilization of CSA. Note that, most of the existing works on LSM-KVS compaction optimizations including [32, 33, 40, 48, 60, 67, 71] can be easily integrated with CaaS-LSM as a customized compaction operator.

### 4.2 Compaction Service Execution Plane

We propose a logically centralized compaction service execution plane to manage compaction workers with the help of CSAs and achieve stateless execution for compaction jobs. Though the compaction service execution plane may increase complexity and initial setup cost, it can minimize resource contention and achieve better tradeoffs in elasticity and scalability. The architecture of the execution plane, CSA, and the relationship with the LSM-KVS instances and control plane is shown in Figure 7. One CSA manages a group of compaction workers in a node. The CSA cluster can be horizontally scaled in and out according to demand. CSA receives multiple compaction job requests from different LSM-KVS instances based on the control plane scheduling and then distributes these jobs to the compaction worker for execution. Each compaction worker can only execute one compaction job at one time (single thread). For different compaction jobs, CSA can

allocate different hardware resource capacities (can be pre-defined by user), like memory and I/O bandwidth, to workers to precisely control the compaction execution speed.

To precisely monitor the compaction service execution status and resource utilization which will be used by the LSM-KVS instance and the control plane for scheduling, load balancing, and failure handling, CSA collects the CPU, memory, storage, and network utilization, compaction job status (including SST file properties, execution time, and error information), and execution progress periodically. The monitoring statistics are packed in `CompactionServiceOutput` and return to the control plane and LSM-KVS instance.

To achieve independent and stateless service execution for compaction jobs, CaaS-LSM introduces two sets of information during the compaction preparation stage in the LSM-KVS instance: `ControlMetadata` and `CompactionServiceInput`. We propose to use `ControlMetadata` to carry the compaction job metadata information provided by the LSM-KVS instance, which includes compaction job information: input/output file level, location, size, compaction score, and application priority. The control plane primarily uses `ControlMetadata` to optimize the management of the compaction service, such as scheduling, priority adjustment, and orchestration. On the other hand, the `CompactionServiceInput` is utilized by the compaction worker to execute the compaction job, which encapsulates the self-contained compaction job details for independent execution, including LSM-KVS instance options, statistics, authorizations, and customized operators (e.g., users can customize their own comparators to sort KV-pairs, read filters [17, 39, 66], or compression algorithms [15] during compaction).

`CompactionService::Start` and `WaitForComplete` are the only two interfaces that we added in the LSM-KVS instances such that they can create the compaction job requests and receive the execution results from the execution and control plane. These two interfaces are also flexible and extendable since users can define their own implementations. For example, We can pack two structures `CompactionServiceInput` and `ControlMetadata` in a request within `CompactionService::Start` and send the request to the control plane. Users can directly add or change some arguments to these two data structures. `WaitForComplete` will receive two execution results from compaction service: `Result` and `Status`. `Result` includes the output files path, execution statistics, and the result state (Success, Failure, and UseLocal). For example, if the `Result` state indicates UseLocal rather than Failure, LSM-KVS instance can still use the legacy logic to execute the compaction job locally. `Status` includes the possible error information, like Maincode, Subcode, and Severity. Maincode indicates error's main categories such as Timeout. Subcode shows more detailed information like MutexTimeout, NetworkTimeout, and WaitTimeout. Severity defines the error severity level. For example, the data corruption-related failure is a "UnrecoverableError". If a failure is caused by a certain CSA, it is a "SoftError" and can be solved by rescheduling to another available CSA.

### 4.3 Compaction Service Control Plane

To address the limitations and challenges discussed in Section 3, the introduction of the control plane has become necessary. Although introducing the control plane design may lead to the risk of a single point of failure (addressed in Section 4.4) and the potential performance bottleneck in extreme high request scenarios (as evaluated in Section 5.6), it can achieve better tradeoffs for the scalability of CSAs, resources utilization, and global scheduling of compactions. Therefore, we propose a performance- and resource-optimized control plane to manage the scheduling and execution of compaction services as shown in Figure 6. First, the control plane maintains a list of active CSAs in the execution plane and caches the authorized connection with LSM-KVS instances. Users can dynamically add or remove LSM-KVS instances and CSAs without influencing the compaction service execution to achieve high elasticity. The control plane itself is also a service, which consists

of a number of control plane instances. Each control plane instance is responsible for handling the compaction job requests from a group of LSM-KVS instances.

The control plane consists of three major components: `Monitor`, `Scheduler`, and `Distributor`. They are responsible for monitoring the CSAs' status, determining compaction jobs' priority and execution order, and distributing compaction jobs to the target CSAs respectively. The workflow of executing a compaction service with control plane is as follows: 1) LSM-KVS instance sends a newly created compaction job to the control plane and starts the regular check with the control plane (e.g., every 500ms) to get the job status updates; 2) The **Scheduler** compares the priority of compaction jobs and arranges their execution order; 3) The **Distributor** distributes the compaction job to a CSA with available resources; 4) Upon job completion, the CSA returns the results back to the control plane, and the LSM-KVS instance can fetch these results through regular checks. If all CSAs are busy, the control plane queues compaction jobs to prevent jobs blocking at a CSA.

*4.3.1* ***Monitoring.*** The control plane monitors the status of CSAs, the execution progress of compaction jobs, and resource utilization. When a new CSA is deployed (e.g., dynamically added by users to handle heavy compaction requests or to match the compaction requests from newly added LSM-KVS instances), it will register the CSA to the Monitor. The Monitor will start the regular heartbeat checks for CSA to collect status and resource information such as the thread pool size, current executing task number, and available resources (e.g., memory, storage, and network bandwidth). When a job is distributed to a CSA or CSA returns the compaction results, Monitor will also update the latest status of the CSA.

*4.3.2* ***Scheduling and distribution.*** The Scheduler is the core module of the control plane and has three main scheduling functions to ensure that the most urgent job can be scheduled and executed first: 1) maintaining a priority queue for the waiting compaction job requests to decide priorities, 2) selecting the available and appropriate CSAs for compaction job execution, and 3) dynamically scheduling the compaction jobs between local and remote.

**Compaction Priority Comparison** If more than one compaction jobs are queued in the Scheduler, the Scheduler leverages `ControlMetaData` (e.g., input/output file level, and compaction score) to decide the priority of compaction jobs and arrange their execution order: 1) We first compare the compaction job trigger reasons: manually triggered compaction jobs have higher priority than background scheduled compaction jobs; 2) we then compare the input SST file level of the compaction. A compaction job with a higher input file level (e.g., L0 or L1) has higher priority over the compaction jobs from the lower level (e.g., bottom-level); and 3) compaction with a higher score has higher priority, which is calculated by the compaction policy (e.g., In RocksDB, the compaction picker will generate a score for each compaction job). Furthermore, CaaS-LSM also allows users to add other factors to customize the scheduling algorithm, such as the application priority, compaction job I/O estimation, and compaction trigger types (e.g., space amplification, read amplification, or Time-To-Live compaction).

**CSA Selection and Distribution** After a compaction job with the highest priority is selected to be executed first based on the aforementioned policy, we propose to use the SST file size information included in the `ControlMetaData` to estimate the resource consumption of a certain compaction job. Then, we use the following policy to select an appropriate CSA with sufficient free resources: 1) the Scheduler filters out CSAs where the current CPU or memory utilization rate exceeds 80%; 2) the scheduler selects the CSA with the most available resources from the remaining CSA candidates; 3) the selected CSA is compared with the estimated resources. If the resources of the selected CSA are insufficient for the current compaction job, a message indicating a lack of resources is returned (more details in the next paragraph). Finally, the Distributor distributes the compaction job to the selected CSA for execution. After the execution, the result will be returned to the Distributor by

the CSA. The detailed process is shown in **Algorithm 1**. Note that the algorithm for selecting CSAs can also be easily replaced according to the actual production environment demands.

**Algorithm 1** EnhancedCSAScheduling

```
1:  procedure ScheduleCSA(controlMetaData)
2:      csa ← OptimizedCSASelection(controlMetaData)
3:      return AgentAvailabilityCheck(csa) ? csa : ∅
4:
5:  procedure ProcessTask
6:      loop
7:          if AdvancedTaskQueue.isEmpty() then
8:              AdaptiveWait()
9:          else
10:             task ← AdvancedTaskQueue.dequeue()
11:             controlMetaData ← MetaRetrieval(task)
12:             csa ← ScheduleCSA(controlMetaData)
13:             if csa ≠ ∅ then
14:                 JobDispatch(csa, task)
15:             else
16:                 AdvancedTaskQueue.enqueue(task)
17:                 AdaptiveWait()
```

**Local-Remote Combined Scheduling** Although fully relying on remote compaction service can effectively optimize the overall performance, both the execution plane and control plane introduce extra overhead (e.g., the overhead of preparing and initializing the compaction execution as discussed in section 3.2.3). When a large number of compaction jobs accumulate in the priority queue of the control plane and wait for CSA resources, the compaction job finishing time can be even longer than executing them locally at LSM-KVS instances. However, how to quickly decide the compaction job execution place (i.e., at LSM-KVS local or at the execution plane) and guarantee a better performance is challenging.

We propose a hybrid compaction job scheduling policy to dynamically allocate the compaction jobs between in-LSM-KVS compaction logic and remote compaction service. We include the following information in `ControlMetaData`: 1) the historical execution time of recent compaction jobs (both locally executed and execution plane executed) in a sliding window, and 2) the LSM-KVS thread pool utilization level. To achieve accurate compaction time estimations, we propose the following two estimation equations. The formula (1) calculates the compaction rate using a weighted average of past and recent compaction rates. The formula (2) estimates compaction time based on the estimated compaction rate and total SST file size. Furthermore, $\alpha$ is a weighting factor, typically chosen to be a value close to 1, such as 0.9, to balance the influence of new and old compaction rates.

$$Rate_{\text{new}} = \alpha \times Rate_{\text{old}} + (1 - \alpha) \times \frac{CompactionTime_{\text{actual}}}{\text{SstFileSize}} \tag{1}$$

$$CompactionTime_{\text{estimated}} = \text{SstFileSize} \times Rate_{\text{new}} \tag{2}$$

The control plane first estimates the compaction execution time at local and the time at the execution plane (including queuing time, execution time, and network latency) via the aforementioned two formulas. Then, the Scheduler decides whether the compaction is executed locally or remotely

according to the following rules: 1) If the local thread pool utilization is too high (e.g., more than 95%), execute the job remotely; 2) If a message indicating a lack of CSA resources is received, execute the job locally; 3) If the estimated execution time of the local compaction execution is shorter than remote execution, execute the job in its LSM-KVS instance.

### 4.4 Fault Tolerance and Error Handling

Considering the various types of failures and errors happening in different software stacks during the compaction service execution, it is challenging to design a comprehensive failure-handling logic to cover all the possible scenarios. Therefore, we propose to classify the failures in three levels (i.e., `job-level`, `CSA-level`, and `Service-level`) and handle them accordingly with fine-grained solutions, which ensures the successful execution of compaction jobs and minimizes the performance penalties propagated to the upper-layer applications.

**Job-level failure**, which can be caused by the I/O, network, or even hardware errors, is indicated by the returned status code. We design three levels of failure seriousness and handle them accordingly: 1) retriable error (e.g., system errors like I/O or network errors), the control plane will retry the execution in the same CSA; 2) solvable error (e.g., insufficient resources or connection timeout), the control plane will reschedule the same job to another CSA; And 3) unrecoverable error (e.g., data corruption or data access authorization denied), the control plane will return the error info back to LSM-KVS instance.

**CSA-level failure**, which can be caused by CSA crashes, host failure, or network partition, is indicated by losing CSA's heartbeat. The control plane will first try to reconnect the CSA. If the CSA cannot be recovered after several attempts, the control plane will remove this CSA from its registration list and reschedule all compaction jobs to other healthy CSAs with the highest priority.

**Service-level failure** means the control plane cannot normally respond to LSM-KVS instance requests due to single-point failure or network connection issues. In this case, LSM-KVS instance will execute the compaction job locally. To make the service reliable, the control plane can be deployed at multiple nodes with consensus algorithms (such as Raft [59] and Paxos [53]). For example, in the Raft algorithm, if the leader node loses connection, followers will elect a new leader to take over the service responsibility.

## 5 IMPLEMENTATION AND EVALUATION

### 5.1 Goals

We evaluate CaaS-LSM to demonstrate the following statements.

- CaaS-LSM has better performance and resource utilization than Legacy LSM-KVS (§5.4.1) and Disaggre-RocksDB (§5.4.2) in CPU-intensive scenarios.
- CaaS-LSM can be applied concurrently with other optimizations of LSM-KVS like TerarkDB, bringing better performance than TerarkDB's native remote compaction (§5.4.4).
- CaaS-LSM will not negatively impact scenarios with minimal or no compaction (§5.5.1). Furthermore, in addition to reducing CPU resource wastage (§5.4.1), CaaS-LSM also mitigates the impact of bad network conditions (§5.5.2).
- The scheduling algorithm in CaaS-LSM is effective and the control plane will not be the bottleneck (§5.6).
- CaaS-LSM can bring performance improvements to real-world distributed applications (e.g., Kvrocks, Nebula) (§5.7).
- CaaS-LSM offers comprehensive failure handling mechanisms and scalability (§5.8).

### 5.2 Implementation

We implement CaaS-LSM based on RocksDB, which is a widely used LSM-KVS (e.g., as storage engines for MyRocks [12], ZippyDB [22], MongoRocks [18], TiKV [14], Nebula [21], and Kvrocks [1]). The implementation of compaction logic encapsulating as a stateless service is production-ready and merged to RocksDB 7.0 and above versions [19]. We use gRPC [9] to achieve the communications between LSM-KVS instances, control plane, and execution plane. The source code of CaaS-LSM is available at GitHub [4].

### 5.3 Evaluation Setup and Methodology

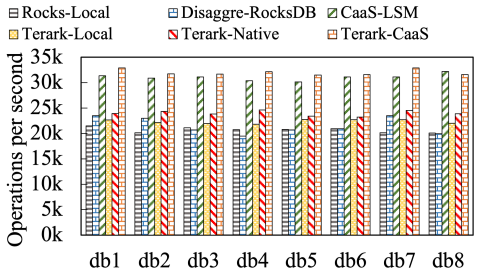
**Experimental Environments** Our experiments are conducted in three different storage disaggregation deployments, which are also discussed in Disaggre-RocksDB [42]: in-rack, within-datacenter, and cross-datacenter. For the in-rack deployment, we use the cluster with all machines in the same rack and connected to a 10Gbps network switch. Each machine is equipped with Intel Xeon Gold 6330 CPU, 256GB memory, and 7TB HDD. We utilize the Google Cloud Platform (GCP) to achieve within-datacenter and cross-datacenter deployments. The compute nodes (E2) are configured with 16 vCPUs, 32GB memory, and 1-16 Gbps network (depending on the deployments and workloads) [11]. The storage nodes are configured with 8 vCPUs, 16GB memory, 1-16 Gbps network, and large storage space. Separately, for the within-datacenter environment, compute and storage nodes are deployed in the same GCP region. For the cross-datacenter environment, compute and storage nodes are deployed in 2 different GCP regions separately. In all three deployments, we configure HDFS as the Disaggregated Storage (`D-Storage`), similar to the Tectonic File System at Meta.

**Deployment** The baseline deployment runs RocksDB instances at compute nodes, executing compaction jobs locally and storing all RocksDB files in D-Storage (called `Rocks-Local`). For CaaS-LSM, we run RocksDB instances at compute nodes and deploy the execution/control plane of CaaS-LSM in the same region as D-Storage. We RocksDB native benchmark db_bench to evaluate the basic performance. Moreover, we integrate CaaS-LSM with two real-world distributed applications Kvrocks [1] and Nebula [21] to further demonstrate the end-to-end performance improvement and the effectiveness of the control plane. Kvrocks is a Redis-compatible distributed KVS using RocksDB as its storage engine and Nebula is a distributed graph database that uses RocksDB to persist all the graph-related data. Based on Nebula v3.2.0 and Kvrocks v2.2.0, we added some changes to make them compatible with RocksDB 7.0, CaaS-LSM, and HDFS libraries. We launch 4 Kvrocks instances on 2 compute nodes (each with 2 Kvrocks instances as master), forming a Kvrocks storage cluster by Kvrocks Controller [10]. We deploy Nebula with 2 RocksDB instances in its storage service.

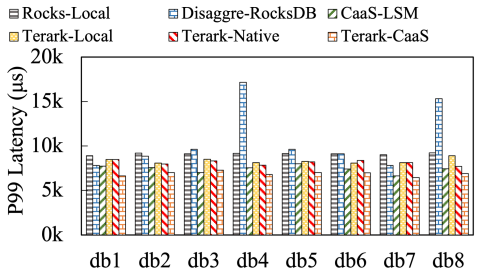
**Database Configuration** For db_bench, we modify the option `max_background_jobs` and the number of client threads to suit each specific case, while other configurations are maintained at their default values (16B key size, 100B value size). For Nebula, we change the option `rocksdb.block.cache` to 1GB. For Kvrocks, we use the default configurations. We use direct I/O in all evaluation to exclude the unpredictable influence of page cache [29].

### 5.4 Comparison with Baselines

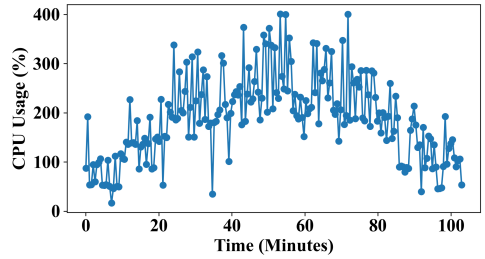
We use the following 5 systems as baselines for performance comparison with CaaS-LSM: 1) fundamental remote compaction schema proposed in Disaggregated RocksDB, called Disaggre-RocksDB [42]; 2) TerarkDB [3] with both local compaction (Terark-Local); 3) TerarkDB with its native remote compaction mode (Terark-Native); 4) To ensure a fair comparison, we also implement the CaaS-LSM design based on TerarkDB, named Terark-CaaS; And 5) we compare CaaS-LSM with Cassandra [5] using YCSB [35]. We use the in-rack deployment and utilize Docker to distribute the machine resources into 8 discrete containers to run RocksDB/TerarkDB instances. For the



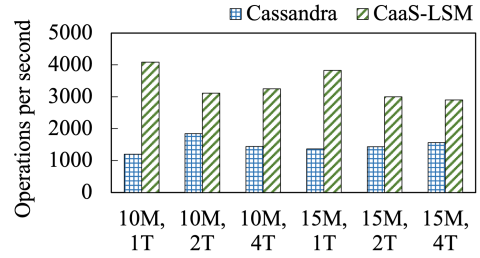

(a) OPS comparison




(b) P99 latency comparison




(c) CPU usage of remote endpoints (CaaS-LSM)




(d) OPS comparison (CaaS-LSM vs. Cassandra) with varying operation counts in millions (M) and write thread counts (T)

Fig. 8. Performance comparison with related work in CPU-intensive scenarios (16 client threads and 8 max_background_jobs). The OPS of CaaS-LSM surpassed Disaggre-RocksDB by up to **61%**, and TerarkDB-CaaS surpassed native TerarkDB up to **42%**. Furthermore, CaaS-LSM achieves up to 240% in OPS compared to Cassandra.

local compaction mode (i.e., Rocks-Local, and Terark-Local), each RocksDB/TerarkDB instance is provisioned with 8 CPU cores and 8GB of memory. For the remote compaction mode (i.e., Disaggre-RocksDB, CaaS-LSM, Terark-Native, and Terark-CaaS), each RocksDB/TerarkDB instance is allocated 6 CPU cores and 6GB memory, and a dedicated container (equipped with 4 cores and 1GB memory) is deployed remotely for compaction execution (1 control plane instance and 2 CSA instances). Systematic tests are executed within these containers using db_bench, configured with 16 client threads and 8 `max_background_jobs`, to evaluate the CPU-intensive scenarios. We use `random_fill` workload to insert 64 million KV-pairs into each RocksDB/TerarkDB instance. We run these RocksDB/TerarkDB instances at different times to trigger compaction jobs from different SST file levels at the same time. All SST files are stored in the in-rack D-Storage.

*5.4.1* ***CaaS-LSM vs. Rocks-Local.*** Compared to the Rocks-Local, CaaS-LSM utilizes 12 fewer CPU cores but much higher performance. Specifically, as shown in Figure 8, CaaS-LSM achieves a 50% of OPS improvement, a 19% reduction in P99 latency, and a 49% decrease in write stall count. It validates that the design of CaaS-LSM can effectively save CPU resources while enhancing performance. Figure 8(c) depicts the CPU usage of the control plane and execution plane. The resource utilization at the remote end is significantly higher compared to the Rocks-Local. This experiment also shows that CaaS-LSM effectively addresses the resource-wasting and scaling issue discussed in Section 3.2 shown in figure 3(a), where the original RocksDB design cannot fully utilize local CPU resources. We also evaluate the performance of CaaS-LSM and Rocks-Local in other cases, with details in Section 5.5.

*5.4.2* ***CaaS-LSM vs. Disaggre-RocksDB.*** As shown in Figure 8, Disaggre-RocksDB experiences resource allocation imbalance since one RocksDB can only send the compaction job to a pre-allocated worker lacking of scheduling for compaction jobs and workers. Even if the resources of a certain compaction worker are fully occupied and other compaction workers are available and waiting for the compaction job requests, the compaction jobs will continue to be queued at the busy worker. Consequently, db4 and db8 exhibit poor P99 latency under such resource contention, as shown in 8(b). When evaluated individually for each instance, the OPS of CaaS-LSM surpasses that of Disaggre-RocksDB, delivering an enhancement of up to 61%. When evaluating the average performance across eight instances, the OPS of CaaS-LSM outperforms Disaggre-RocksDB by 45%.

*5.4.3* ***Rocks-Local vs. Terark-Local.*** According to Figure 8, under the same condition of using local compaction, Terark-Local performs better than Rocks-Local since TerarkDB implements additional optimizations (i.e., new compression scheme, new SST file format, and novel indexing) on top of RocksDB. These optimizations are compatible with the design of CaaS-LSM. Therefore, we implemented the CaaS schema based on TerarkDB, named Terark-CaaS, to facilitate a more equitable comparison with Terark-Native.

*5.4.4* ***Terark-CaaS vs. Terark-Native.*** Terark-Native demonstrates a better resource allocation than the design of Disaggre-RocksDB. However, Terark-Native provides only a rudimentary level of scheduling based on general metrics of CPU, memory, and network conditions. It does not take into account compaction job-specific metrics such as compaction job priority and compaction job SST file levels. As shown in Figure 8(a), the OPS of Terark-CaaS exceeds that of Terark-Native, showing an improvement of up to 42%. In the context of average performance across eight instances, Terark-CaaS demonstrates a 34% higher OPS compared to Terark-Native. This evaluation not only demonstrates that the performance of the CaaS-LSM design is superior to TerarkDB's local compaction mode and native remote compaction mode, but also proves the versatility of the CaaS-LSM design. The design of CaaS-LSM can be applied to other LSM-KVS implementations and is compatible with other optimizations.

*5.4.5* ***CaaS-LSM vs. Cassandra.*** We deploy Cassandra on disaggregated storage to compare with CaaS-LSM, using Cassandra's default configuration with a replication factor of 1. As Figure 8(d) shows, we employ a custom YCSB workload, initially loading 1,000,000 records, followed by inserting 10,000,000/15,000,000 records using 1/2/4 threads. In the 6 cases tested, CaaS-LSM achieves a maximum increase of 240% in OPS compared to Cassandra.

### 5.5 CaaS-LSM vs. Legacy LSM-KVS

In Section 5.4.1, we evaluate CaaS-LSM and legacy LSM-KVS in CPU-intensive scenarios. In this section, we conduct a more detailed comparison between CaaS-LSM and legacy LSM-KVS with different deployments and workloads. The six implementation and deployment combinations are: in-rack Rocks-Local (IR-Local), in-rack CaaS-LSM (IR-CaaS), within-datacenter Rocks-Local (WD-Local), within-datacenter CaaS-LSM (WD-CaaS), cross-datacenter Rocks-Local (CD-Local), and cross-datacenter CaaS-LSM (CD-CaaS).

*5.5.1* ***Evaluation of Scenarios with Minimal or No Compaction.*** We evaluate the performance improvement of CaaS-LSM compared with legacy LSM-KVS. We use different workloads (random read/write vs. sequential read/write) under various network conditions. Sequential write will have much fewer or no compaction jobs compared to random write operations, which is designed to demonstrate that our approach does not lead to performance regression with light compaction demands. We use db_bench, configured with 1 client thread and 4 `max_background_jobs`.

**Random Read Write** To evaluate the performance on read and write mixed queries, we use the database generated by the RandomWrite benchmark as the base (already inserted 18 million key-value pairs) and continue to issue queries with 50% reads and 50% writes. In this experiment, Get queries trigger the data block read (32KB blocks) from SST files. Since the key is randomly distributed in the whole key space, the block cache hit ratio will be low, and reading the data blocks from D-Storage is heavily impacted by the network conditions. Therefore, for the cross-datacenter environment, the RocksDB read OPS is lower in both Rocks-Local and CaaS-LSM. The OPS of IR-Local, IR-CaaS, WD-Local, WD-CaaS, CD-Local, and CD-CaaS are 13130, 13855, 6037, 10071, 141, and 246 respectively. Overall, CaaS-LSM can effectively improve the OPS by 5%, 66%, and 74% at in-rack, within-datacenter, and cross-datacenter deployments respectively due to the lower network bandwidth contention from compaction jobs.

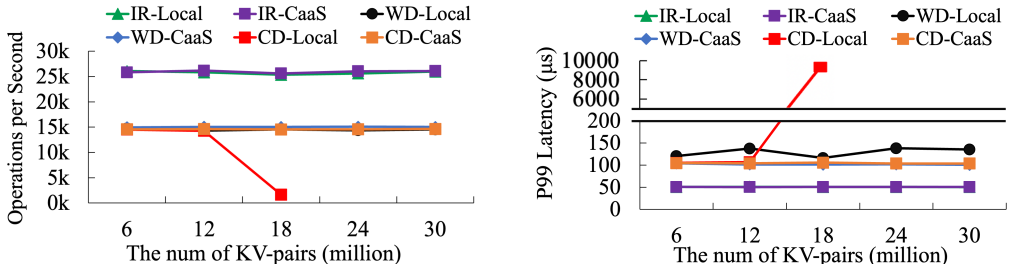
**Sequential Read and Write** Theoretically, the sequential write will only trigger trivial move compactions that do not actually read and write SST files since there is no key-ranges overlap between SST files in different levels. In this test, for the same network condition, the OPS and P99 latency of Rocks-Local and CaaS-LSM do not have an explicit difference if we insert 6 to 30 million KV-pairs sequentially. The OPS is about 340,000 for both IR-Local and IR-CaaS, about 200,000 for WD-Local, WD-CaaS, CD-Local, and CD-CaaS. After sequentially writing the KV-pairs, we apply the sequential read, which is achieved by Scan (SeekRandom with 100 Next queries) in RocksDB. When we issue 6 to 30 million range queries, the OPS is about 180,000 for both IR-Local and IR-CaaS, about 150,000 for both WD-Local and WD-CaaS, and about 16,000 for both CD-Local and CD-CaaS. CaaS-LSM has no performance regression on the sequential read/write queries with minimal compactions.

*5.5.2* ***Evaluation Across Varied Network Scenarios.*** We conduct a performance evaluation under various network conditions, while deliberately disabling the resource scheduling feature of CaaS-LSM and providing ample CPU and memory resources. The intent behind this configuration is to isolate and focus on the impact of the network conditions. In this test, we run db_bench at compute nodes with the `random_fill` workload (random write with uniform distribution to trigger a substantial amount of compaction), which randomly inserts KV-pairs to RocksDB. We adopt the same six deployments as described in Section 5.5.1.

As shown in Figure 9, we insert 6 to 30 million KV-pairs with 1 client thread. The OPS of the six scenarios is shown in Figure 9(a). IR-Local and IR-CaaS achieve the best performance because of the fastest network speed. WD-Local, WD-CaaS, and CD-CaaS show similar OPS and latency. Generally, IR-Local, IR-CaaS, WD-Local, WD-CaaS, CD-CaaS have stable OPS and latency but the performance of CD-Local degrades after inserting 18 million KV-pairs. Compared to CD-Local, when inserting 18 million KV-pairs, CD-CaaS improves OPS by 8$X$ by reducing most of the compaction network traffic between two datacenters. As more KV-pairs are inserted, CD-Local experiences intensive write slowdown or even write stall due to the accumulated compaction jobs and it causes very high P99 latency, which leads to 78$X$ higher P99 latency than CD-CaaS. According to the statistics of RocksDB, when inserting 18 million KV-pairs, the count of write stalls is 6,693,747 in CD-Local, whereas it is 0 in CD-CaaS. In general, CaaS-LSM shows significant performance improvement over legacy LSM-KVS design, especially when the network condition is worse.

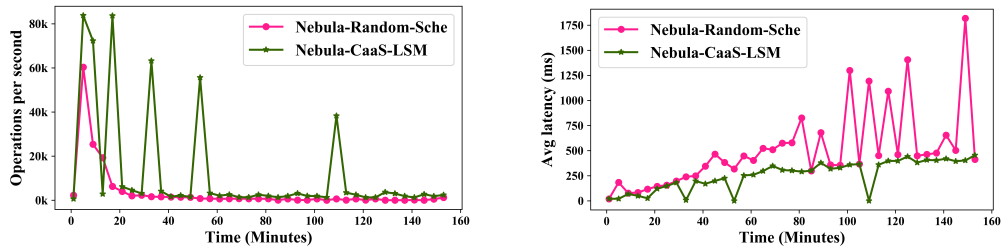
### 5.6 Control Plane Evaluation and Analysis

**Scheduling Algorithm Evaluation** To evaluate the effectiveness of the scheduling algorithms in CaaS-LSM, we run multiple Nebula instances and db_bench instances with different DB sizes in a cross-datacenter deployment to trigger concurrent compactions from applications with different compaction types and priorities. We use 2 compute nodes to run 4 RocksDB instances, inserting 5




(a) OPS of RocksDB (b) P99 latency of RocksDB

Fig. 9. db_bench random write performance under different data scales and deployments. CaaS-LSM shows significant performance improvement over legacy LSM-KVS




(a) OPS of Nebula (b) Latency of Nebula

Fig. 10. Mix Nebula with other DBs to evaluate CaaS-LSM scheduling. Nebula-Random-Sche has a total OPS of 5,669 and an average latency of 526 ms, which are about 86% lower and 6$X$ higher than Nebula-CaaS-LSM respectively.

million key-value pairs via db_bench to the 4 RocksDB instances. Two of the RocksDB instances have 15 million existing key-value pairs and the other 2 are empty when we start the benchmarking. In this way, compaction jobs with different priorities will be triggered concurrently. Concurrently, we run Nebula to import graph data with a scale factor of 10 (12GB) in the third compute node. We give higher compaction priority for RocksDBs in Nebula than that of db_bench. We deploy 3 CSAs at D-Storage and each CSA has one compaction worker thread to simulate the scenario that the execution plane receives intensive compaction requests and limited available compaction service resources. We implement a random scheduling algorithm in CaaS-LSM as the baseline, which randomly selects a CSA to execute the compaction job. CaaS-LSM will schedule compaction jobs based on resources, types, application priority, and SST file levels.

According to the collected results, there is almost no performance difference between the RocksDB instances tested by db_bench because they require fewer compaction resources and the compaction priority is lower than Nebula. As shown in Figure 10(a) and Figure 10(b), Nebula with random scheduling (Nebula-Random-Sche) has a total OPS of 5,669 and an average latency of 526 ms which are about 86% lower and 6$X$ higher than Nebula using CaaS-LSM (Nebula-CaaS-LSM) respectively. Also, Nebula-Random-Sche fails before finishing the benchmarking since too many compaction jobs are waiting for execution and it causes serious write stalls at Nebula. Differently, Nebula-CaaS-LSM achieves about 6$X$ OPS (35,682) and reduces the average latency by 84% (82 ms).

**Control Plane peak throughput**. We evaluate the peak throughput of a single control plane instance by issuing a large number of compaction requests concurrently. The peak OPS of a single

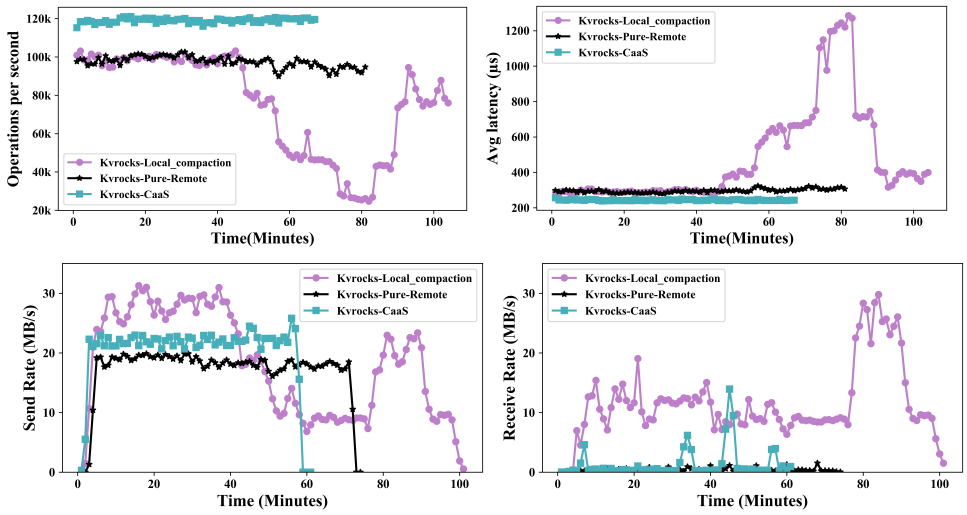
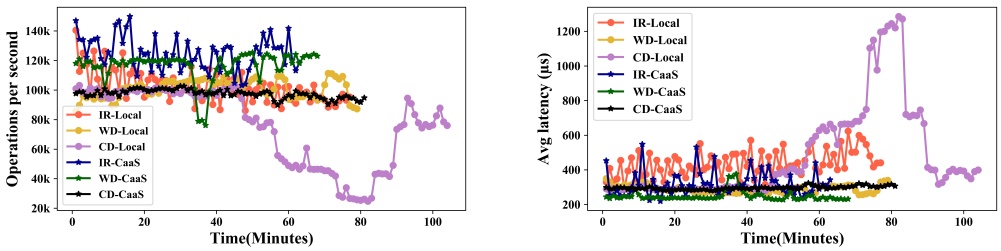



Fig. 11. Kvrocks OPS, latency, and network traffic under cross-datacenter deployment. Kvrocks-CaaS with hybrid-compaction scheduling achieves the highest overall performance compared to Kvrocks-Local_compaction and Kvrocks-Pure-Remote.




(a) OPS of Kvrocks

(b) Latency of Kvrocks

Fig. 12. Kvrocks cluster performance at three deployments. Kvrocks-CaaS provides better performance than Kvrocks-Local in all three scenarios.

control plane instance can reach about 12,500. This level of OPS is typically more than sufficient for most scenarios, as they are unlikely to demand over 12,500 compaction jobs per second. For larger-scale applications that may exceed this number, we need to cluster KVS instances into groups and each control plane instance is responsible for handling compaction requests from the assigned group. How to achieve better load balancing for the control plane cluster will be our future work.

### 5.7 Evaluation on Real-World Applications

Evaluating the performance with db_bench may not be able to cover more complex scenarios in real production. To conduct a comprehensive end-to-end evaluation, we integrate Rocks-Local and CaaS-LSM with **Kvrocks** (Kvrocks-Local and Kvrocks-CaaS) and **Nebula** (Nebula-Local and Nebula-CaaS). For Kvrocks, we evaluate it with the Redis benchmarks, and Nebula is evaluated with LDBC graph benchmark [24, 43]. Kvrocks and Nebula are also deployed and tested at three different infrastructure deployments: in-rack, within-datacenter, cross-datacenter.

**Kvrocks-Local vs Kvrocks-CaaS**. To evaluate the effectiveness of the CaaS-LSM design, we first disable the local fallback scheduling of CaaS-LSM and test the Kvrocks cluster in pure compaction service mode under three deployments. For the execution plane of Kvrocks-CaaS, we deploy 1 control plane and 2 CSAs (each with 5 worker threads) instances in the D-Storage. We use Redis benchmark to generate Set queries to the cluster to write the data. Internally, the Set queries are mapped to a set of data and metadata Put operations to RocksDB instances. The benchmark issues 400 million Set queries by 32 threads in total (the whole data set is about 80 GB). We set the KV-pair size as 100 bytes and the key-space length as 10 billion, which is large enough to trigger compactions in RocksDB instances. The initial databases are all empty.

We plot out the OPS and average latency in Figure 12. The performances of in-rack and within-datacenter scenarios are similar since all machines are connected by a high-speed internal network. With better scheduling of compaction jobs in Kvrocks-CaaS, the overall OPS is about 20% better than that of Kvrocks-Local, and the average latency improves by 30%. In the cross-datacenter scenario, according to the log file, Kvrocks-Local experiences compaction jobs piled and a severe write slowdown after intensive compaction starts. In contrast, Kvrocks-CaaS runs smoothly and improves the overall OPS by 28% and P99 latency by 65%.

**Kvrocks-CaaS with hybrid-compaction scheduling in the control plane**. We further conduct evaluation on the Kvrocks-CaaS cluster with a local-remote combined scheduling control plane under cross-datacenter deployment. Based on the scheduling design in Section 4.3.2, the control plane will decide whether the compaction job should be executed locally or at the execution plane of CaaS. We plot out the OPS, average latency, egress network traffic (data sent), and ingress network traffic (data received) in Figure 11. Kvrocks-CaaS with hybrid-compaction scheduling enabled (Kvrocks-CaaS) achieves the highest overall performance (improved by 56% and 22%) and lowest latency (improved P99 by 76% and 30%) compared to Kvrocks-Local_compaction and Kvrocks-Pure-Remote respectively. The network traffic of Kvrocks-CaaS and Kvrocks-Pure-Remote is much lower and stabler than Kvrocks-Local_compaction. Since all compaction jobs are executed by CSAs in the D-Storage, the ingress traffic is the lowest in Kvrocks-Pure-Remote. Differently, due to the I/Os of compactions, Kvrocks-Local_compaction has explicit network traffic spikes. In addition, Kvrocks-CaaS receives more data than Kvrocks-Pure-Remote, indicating that there are some compaction jobs executed locally at the RocksDB instances.

**Nebula-Local vs Nebula-CaaS**. We use the LDBC Social Network Benchmark Data Generator [24, 43] to generate an SF3 scale graph dataset (3.6GB) and import it to Nebula via the Nebula importer. We evaluate Nebula-Local and Nebula-CaaS at in-rack, within-datacenter, and cross-datacenter environments respectively. For the in-rack and within-datacenter cases, Nebula-CaaS achieves about 30% throughput improvement and reduces by about 17% average latency compared to Nebula-Local. For cross-datacenter Nebula-Local, when the experiment proceeds to 10,950 seconds, all operations begin to fail due to serious write stalls, and a total of 52,746,687 operations are executed. Cross-datacenter Nebula-CaaS delivers an 8X OPS improvement and reduces latency by 89% compared to cross-datacenter Nebula-Local.

### 5.8 Failure Handling and Scalability

To evaluate the failure handling and scalability in CaaS-LSM, we inject the compaction worker failure, CSA failures (CSA cluster scale in), execution plane failure, and storage failures during the run-time. Four CSAs are deployed and registered at the control plane. We evaluate the OPS of Kvrocks with Redis benchmark in a write-intensive workload as shown in Figure 13. When compaction starts at one CSA, we manually fail the compaction worker at 100 seconds. The compaction job is rescheduled automatically and it does not cause an explicit OPS drop. Then, we stop one CSA that is executing multiple compaction jobs at 160 seconds (CSA cluster scale in), and

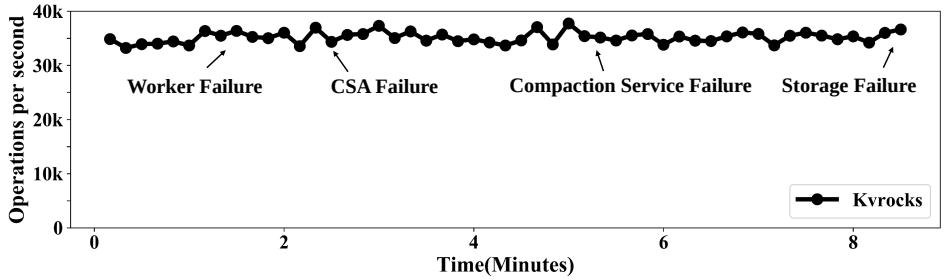



Fig. 13. Evaluate the failure handling and scalability of CaaS-LSM in Kvrocks.

the control plane loses the connection to the CSA. Since the control plane only takes very little time (less than 2 seconds) to identify losing the heartbeat of a certain CSA and re-runs all failed compaction jobs in other CSAs, OPS keeps at the same level and Kvrocks runs without interruptions. This scenario underscores the system's scalability, demonstrating how workload redistribution among available CSAs ensures consistent performance even in a scaled-in operational setting.

We make all the CSAs offline to simulate the execution plane failure at 320 seconds. As shown in Figure 13, upon detecting that all CSAs are offline and all compaction jobs have failed, the control plane directly informs the LSM-KVS instances at Kvrocks to fall back to local compaction. At this point, the local compute resources are relatively idle, which allows for the seamless transition of compaction tasks from remote to local execution without any performance degradation. Consequently, the write OPS is maintained at the same level, corroborating the effectiveness of error and failure handling in the control plane. We finally shut down the HDFS at 520 seconds which is an unrecoverable failure for Kvrocks, the LSM-KVS instance in Kvrocks reports the error and stops running since it cannot apply any I/O operations. In general, the fault tolerance and scalability in CaaS-LSM ensure service reliability and availability with minimal performance penalties.

## 6 RELATED WORK

**Optimizing LSM-KVS and Databases on Disaggregated Infrastructure** In recent years, several existing studies focus on optimizing LSM-KVS on disaggregated infrastructure, including Kemme et al [23], Hailstorm [27], Nova-LSM [49], IS-HBase [28], RocksDB-Cloud [63], TerarkDB [3], Disaggre-RocksDB [42], and dLSM [75, 76]. Most of the existing studies focus on optimizing compaction (e.g., compaction offloading [3, 27, 28, 42, 49]) for disaggregated storage, while dLSM [75, 76] leverages the RDMA-based remote memory to achieve high performance. Compared with existing studies of optimizing LSM-KVS for disaggregated storage, CaaS-LSM achieves better scalability, scheduling, and failure handling. Several other databases are also optimized for disaggregated infrastructure including Aurora [72], Azure SQL Hyperscale [2], Google AlloyDB [8], Facebook HyperDex [44], and Alibaba OceanBase [13].

**LSM-KVS Read/Write Optimizations with Better Tradeoffs.** To serve different workload demands, several existing studies are focusing on achieving better tradeoffs in LSM-KVS, including Dostoevsky [37], LSM-Bush [38], SlimDB [66], PebblesDB [65], and LSM-trie [79]. Those works optimize the architecture of LSM-KVS to improve read or write performance with slight overhead in other perspectives. bLSM [68] combines B-tree with LSM for near-optimal read and scan performance with the tradeoff of higher write latency. VT-Tree [69] avoids unnecessary copy and improves sequential write with the overheads of fragments and expensive space reclamation. MatrixKV [82], HiLSM [55], and SpanDB [32] achieve better tradeoffs with new hardware like Non-Volatile Memory (NVM). Since almost all of the aforementioned tradeoff designs do not influence

the general compaction logic, they can be easily integrated with CaaS-LSM. Moreover, CaaS-LSM can save CPU and memory resources for foreground read/write operations.

**Compaction Design and Policy Optimization** Different compaction policies have been proposed to improve overall write performance and prevent write stalls. bLSM [68] proposes snowshoveling algorithm to increase the effective size and reduce compaction frequency. PCP [87] utilizes parallelism of CPUs and I/O devices for multiple sub-tasks and improves compaction throughput. SILK [26] proposes an I/O scheduler to prioritize low-level compaction, and utilizes spare bandwidth to do high-level compactions. TRIAD [25] and PinK [50] keep hotkeys in memory and only flush cold keys to reduce background compaction workload. Inspired by these works, CaaS-LSM achieves compaction scheduling based on SST file levels, the compaction score, and other factors. And those proposed optimizations are fully compatible with CaaS-LSM.

**Key-Value Separation in LSM-KVS** Key-value separation can effectively reduce the compaction overhead in large-value cases. WiscKey [57] first proposes key-value separation to avoid value copy and mitigate I/O amplification of compaction. RocksDB implements BlobDB [16] and stores large values in dedicated blob files. HashKV [30] uses hash-based data grouping to map values to specific storage zones, improving update and garbage collection efficiency. Parallax [80] classifies values into small, medium, and large groups and places them in different logs to reduce garbage collection overheads. In disaggregated infrastructure, key-value separation can improve the overall performance due to the reduced I/Os between compute and storage. Moreover, in-storage garbage collection can further reduce I/Os, and the control plane of CaaS-LSM with minor modifications can be potentially used to manage garbage collection jobs and achieve better scheduling and execution.

**Filters and Compression Optimizations** When building the SST files during compaction, constructing filter blocks and using compression can improve read performance and space efficiency respectively. Several related studies focus on optimizing filter memory and CPU utilization and improve false positive rate [34], including SlimDB [66], Chucky [39], Blocked bloom filters [64], Ribbon filters [17], SuRF [83], Rosetta [58], and REncoder [77]. Monkey [36] optimizes bloom filter allocation policy for a given RAM budget, shifting memory from larger level filters to lower level. Mapped SplinterDB [34] replaces filters with maplets that decouple compaction from data, realizing lazy compaction but aggressive filter merging. Meanwhile, effective compression algorithms can significantly save I/O bandwidth. RocksDB integrates multiple compression algorithms, such as Snappy, Zlib, Bzip2, LZ4, and Zstd [15], and TerarkDB also designs highly compressed TerarkZipTable [20]. The aforementioned filter and compression optimizations can be implemented as a customized compaction operator and executed at CSAs in CaaS-LSM with high flexibilities.

## 7 CONCLUSION AND FUTURE WORK

In this paper, we discussed the trend of disaggregated infrastructure and the challenges for LSM-based key-value stores. To address the performance, network, scheduling, and fault tolerance issues of compaction, we first proposed to disaggregate compaction from the KVS instance as a stateless independent service. Importantly, we proposed CaaS-LSM, which includes a compaction service execution plane and an LSM-KVS performance- and resource-optimized control plane. Our design achieves significant OPS improvement, latency reduction, and network cost savings. In the future, we plan to make the compaction service customizable and pluggable to support more applications.

## ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers for their valuable feedback. We thank all the members of ASU-IDI Lab for providing useful comments. This work was partially funded by the Arizona State University startup fund. Jianguo Wang acknowledges the support of the National Science Foundation under Grant Number 2337806.

## REFERENCES


[1] [n. d.]. Apache. Kvrocks. https://github.com/apache/incubator-kvrocks. Accessed 10 Jan, 2023.

[2] [n. d.]. Azure SQL Database. Hyperscale service tier. https://learn.microsoft.com/enus/azure/azure-sql/database/service-tier-hyperscale?view=azuresql,2023.. Accessed 10 Jan, 2023.

[3] [n. d.]. ByteDance. TerarkDB. https://github.com/bytedance/terarkdb. Accessed 10 Jan, 2023.

[4] [n. d.]. CaaS-LSM. https://github.com/asu-idi/CaaS-LSM.

[5] [n. d.]. Cassandra on RocksDB at Instagram. https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram.

[6] [n. d.]. db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Accessed 10 Jan, 2023.

[7] [n. d.]. GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction | USENIX. https://www.usenix.org/conference/fast19/presentation/yao

[8] [n. d.]. Google Cloud. AlloyDB for PostgreSQL Under the Hood: Intelligent, Databaseaware Storage. https://cloud.google.com/blog/products/databases/alloydb-forpostgresql-intelligent-scalable-storage,2022.. Accessed 10 Jan, 2023.

[9] [n. d.]. gRPC. https://grpc.io/. Accessed 10 Jan, 2023.

[10] [n. d.]. Kvrocks controller. https://github.com/KvrocksLabs/kvrocks_controller. Accessed 10 Jan, 2023.

[11] [n. d.]. Machine families resource and comparison guide. https://cloud.google.com/compute/docs/machine-resource. Accessed 10 Jan, 2023.

[12] [n. d.]. Meta. MyRocks. http://myrocks.io/. Accessed 10 Jan, 2023.

[13] [n. d.]. OCEANBASE. https://www.oceanbase.com/. Accessed 10 Jan, 2023.

[14] [n. d.]. PingCAP. TIKV. https://tikv.org/. Accessed 10 Jan, 2023.

[15] [n. d.]. RocksDB Compression. https://github.com/facebook/rocksdb/wiki/Compression. Accessed 10 Jan, 2024.

[16] [n. d.]. RocksDB Integrated BlobDB. https://rocksdb.org/blog/2021/05/26/integrated-blob-db.html. Accessed 10 Jan, 2024.

[17] [n. d.]. RocksDB Ribbon Filter. https://rocksdb.org/blog/2021/12/29/ribbon-filter.html. Accessed 10 Jan, 2024.

[18] [n. d.]. RocksDB Storage Engine Module for MongoDB. https://github.com/mongodb-partners/mongo-rocks. Accessed 10 Jan, 2023.

[19] [n. d.]. RocksDB. https://github.com/facebook/rocksdb. Accessed 10 Jan, 2023.

[20] [n. d.]. TerarkZipTable Compression. TerarkDB. https://bytedance.larkoffice.com/docs/doccnZmYFqHBm06BbvYgjsHHcKc. Accessed 10 Jan, 2024.

[21] [n. d.]. Vesoft Inc. Nebula. https://github.com/vesoft-inc/nebula. Accessed 10 Jan, 2023.

[22] [n. d.]. ZippyDB: a modern, distributed key-value data store. https://www.youtube.com/watch?v=DfiN7pG0D0k. Accessed 10 Jan, 2023.

[23] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* (Mar 2015), 850–861. https://doi.org/10.14778/2757807.2757810

[24] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, et al. 2020. The LDBC social network benchmark. *arXiv preprint arXiv:2001.02299* (2020).

[25] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 363–375.

[26] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.

[27] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 301–316.

[28] Zhichao Cao, Huibing Dong, Yixun Wei, Shiyong Liu, and David HC Du. 2022. IS-HBase: An In-Storage Computing Optimized HBase with I/O Offloading and Self-Adaptive Caching in Compute-Storage Disaggregated Infrastructure. *ACM Transactions on Storage (TOS)* 18, 2 (2022), 1–42.

[29] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[30] HelenH.W. Chan, Yongkun Li, PatrickP.C. Lee, and Yinlong Xu. 2018. HashKV: enabling efficient updates in KV storage via hashing. *USENIX Annual Technical Conference,USENIX Annual Technical Conference* (Jul 2018).

[31] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[32] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast,Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 17–32.

[33] Lidong Chen, Yinliang Yue, Haobo Wang, and Jianhua Wu. 2018. A priority and fairness mixed compaction scheduling mechanism for LSM-tree based kv-stores. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 89–105.

[34] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[35] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[36] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. https://doi.org/10.1145/3035918.3064054

[37] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data*. https://doi.org/10.1145/3183713.3196927

[38] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush amp; the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data*. https://doi.org/10.1145/3299869.3319903

[39] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data*. https://doi.org/10.1145/3448016.3457273

[40] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[41] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 33–49.

[42] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proc. ACM Manag. Data* 1, 2, Article 192 (jun 2023), 24 pages. https://doi.org/10.1145/3589772

[43] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[44] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 25–36.

[45] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 249–264.

[46] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB.

[47] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.

[48] Yanpeng Hu, Li Zhu, Lei Jia, and Chundong Wang. 2023. AisLSM: Revolutionizing the Compaction with Asynchronous I/Os for LSM-tree. *arXiv preprint arXiv:2307.16693* (2023).

[49] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the 2021 International Conference on Management of Data*. 749–763.

[50] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 173–187.

[51] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving performance of flash based {Key-Value} stores using storage class memory as a volatile memory extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 821–837.

[52] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2022. Power-optimized deployment of key-value stores using storage class memory. *ACM Transactions on Storage (TOS)* 18, 2 (2022), 1–26.

[53] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[54] Jianchuan Li, Peiquan Jin, Yuanjin Lin, Ming Zhao, Yi Wang, and Kuankuan Guo. 2021. Elastic and stable compaction for LSM-tree: A FaaS-based approach on terarkdb. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3906–3915.

[55] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. 2020. HiLSM. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*. https://doi.org/10.1145/3387902.3392621

[56] Rui Lin, Yuxin Cheng, Marilet De Andrade, Lena Wosinska, and Jiajia Chen. 2020. Disaggregated data centers: Challenges and trade-offs. *IEEE Communications Magazine* 58, 2 (2020), 20–26.

[57] Lanyue Lu, ThanumalayanSankaranarayana Pillai, AndreaC. Arpaci-Dusseau, and RemziH. Arpaci-Dusseau. 2016. WiscKey: separating keys from values in SSD-conscious storage. *File and Storage Technologies,File and Storage Technologies* (Feb 2016).

[58] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/3318464.3389731

[59] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 305–319.

[60] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325.

[61] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 217–231.

[62] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*.

[63] Hieu Pham. [n. d.]. Remote Compactions in RocksDB-Cloud. https://rockset.com/blog/remote-compactions-in-rocksdb-cloud/. Accessed 10 Jan, 2023.

[64] Felix Putze, Peter Sanders, and Johannes Singler. 2007. *Cache-, Hash- and Space-Efficient Bloom Filters*. 108–121. https://doi.org/10.1007/978-3-540-72845-0_9

[65] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB. In *Proceedings of the 26th Symposium on Operating Systems Principles*. https://doi.org/10.1145/3132747.3132765

[66] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB. *Proceedings of the VLDB Endowment* (Aug 2017), 2037–2048. https://doi.org/10.14778/3151106.3151108

[67] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2022. Constructing and Analyzing the LSM Compaction Design Space (Updated Version). *arXiv preprint arXiv:2202.04522* (2022).

[68] Russell Sears and Raghu Ramakrishnan. 2012. bLSM. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/2213836.2213862

[69] Pradeep Shetty, RichardP. Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. *File and Storage Technologies,File and Storage Technologies* (Feb 2013).

[70] Hui Sun, Shangshang Dai, and Jianzhong Huang. 2020. Cascaded write amplification of LSM-tree-based key-value stores underlying solid-state disks. *Microprocessors and Microsystems* 78 (2020), 103217.

[71] Hui Sun, Bendong Lou, Chao Zhao, Deyan Kong, Chaowei Zhang, Jianzhong Huang, Yinliang Yue, and Xiao Qin. 2023. An Asynchronous Compaction Acceleration Scheme for Near-Data Processing-enabled LSM-Tree-based KV Stores. *ACM Transactions on Embedded Computing Systems* (2023).

[72] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[73] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for {Log-Structured} {Key-Value} Store on Persistent Memory. 773–788. https://www.usenix.org/conference/atc22/presentation/wang-jing

[74] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Companion of the International Conference on Management of Data (SIGMOD)*. 37–44.

[75] Ruihong Wang, Chuqing Gao, Jianguo Wang, Prishita Kadam, M. Tamer Ozsu, and Walid G. Aref. 2024. Optimizing LSM-based Indexes for Disaggregated Memory. *VLDB Journal (VLDBJ)* (2024).

[76] Ruihong Wang, Jianguo Wang, Prishita Kadam, M Tamer Özsu, and Walid G Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2835–2849.

[77] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2036–2049.

[78] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. 2018. Joins: Meeting latency slo with integrated control for networked storage. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 194–200.

[79] Xingbo Wu, Yuehai Xu, Zehui Shao, and Song Jiang. 2015. LSM-trie: an LSM-tree-based ultra-large key-value store for small data. *USENIX Annual Technical Conference,USENIX Annual Technical Conference* (Jul 2015).

[80] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax. In *Proceedings of the ACM Symposium on Cloud Computing*. https://doi.org/10.1145/3472883.3487012

[81] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Transactions on Storage* 13, 4 (Nov. 2017), 1–28. https://doi.org/10.1145/3139922

[82] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 17–31.

[83] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data*. https://doi.org/10.1145/3183713.3196931

[84] Qizhen Zhang, Yifan Cai, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Rethinking data management systems for disaggregated data centers. In *Conference on Innovative Data Systems Research*.

[85] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing data-intensive systems in disaggregated data centers with teleport. In *Proceedings of the 2022 International Conference on Management of Data*. 1345–1359.

[86] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 225–237.

[87] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. 2014. Pipelined Compaction for the LSM-Tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. https://doi.org/10.1109/ipdps.2014.85

Received October 2023; revised January 2024; accepted February 2024.