Coordinating Compaction between LSM-tree based Key-Value Stores for Edge Federation

Jeeseob Kim, Honghyeon Yoo, Seungjae Lee, Hongsu Byun*, Sungyong Park*
Department of Computer Science and Engineering, *Sogang University*
Seoul, Republic of Korea
{jeeseob, yhh, ctaaone, byhs, parksy}@sogang.ac.kr

***Abstract*—Edge computing environments increasingly demand real-time data processing, leading to the adoption of log-structured merge-tree based key-value stores (LSM-KVS) for efficient data handling. LSM-KVS periodically runs compaction operations in the background to manage the database. However compaction delays cause write stalls, which lead to degraded throughput of LSM-KVS and system performance on resource-limited edge servers. An edge federation environment, which shares resources and tasks between edge servers, can alleviate the resource limitations. Such environments can leverage compaction offloading where another server performs CPU-intensive compaction operations instead. But coordinating compaction offloading is an important challenge, as the performance of the server performing the compaction can be degraded.**

In this paper, we propose EDGEPILOT. EDGEPILOT is scheduling mechanism of compaction offloading that is designed for LSM-KVS within edge federation. EDGEPILOT schedules where to reallocate compactions among the edge servers. This is achieved by considering the resource and computing power of each server. As a result, the overall resource efficiency and compaction throughput are increased. In addition, EDGEPILOT provides EDGECODE to determine the effectiveness of compaction offloading. EDGECODE is a mathematical modeling based on compaction processing data to approximate inter-server compaction processing times. EDGEPILOT is implemented on the prominent LSM-KVS, RocksDB v8.3.2, and demonstrates notable improvements compared to the conventional RocksDB. The overall write stall duration of the system is reduced by up to 71%, and throughput is increased by 17%.

***Index Terms*—Infrastructure, Edge Computing, Cloud Storage, Key-Value Store, LSM-tree**

I. INTRODUCTION

Edge computing [1] is a computing environment that provides low latency, high mobility, and location-aware capabilities located close to the edge devices that collect data, and is recently used for Internet of Things (IoT) service applications such as smart farms [2], smart factories [3], and health care [4], [5]. Using large amounts of data collected through edge devices such as sensors and mobile devices, IoT applications perform real-time analytics and monitoring [6]. These tasks require data-intensive work capabilities in the edge computing environment.

To perform real-time data processing tasks, it is necessary to store high volumes of data generated by edge devices and manage the state of the data being processed. Therefore, low performance of read and write operations of the database used for data processing adversely affects the entire system, preventing the edge computing environment from providing proper functionalities. Thus, it is appropriate to use a key-value store providing high throughput as a database for data processing [7]. Most of these key-value stores have a log-structured merge-tree structure (LSM-KVS). For example, data processing applications like Apache Flink [8] and Apache Kafka [9] use LSM-KVS as a state management backend.

LSM-KVS has high write performance by storing data in an append-only manner with a simple data structure of key-value pairs. However, to ensure read performance after a certain number of writes, a compaction [10] operation is performed in the background to remove and sort data with duplicate keys. Compaction is compute-intensive [11]–[13] because the data stored in storage must be read and sorted again, and if compaction is delayed, sufficient read performance cannot be guaranteed, causing a write stall [14]. A write stall has a critical impact on LSM-KVS performance by blocking the application's I/O processing until the delayed compaction is completed.

There have been various studies to improve the performance degradation caused by compaction delay in LSM-KVS. First, compaction acceleration through compaction offloading. Accelerators such as GPU [15], [16], FPGA [17], [18], DPU [19], or computational storage drives [20], [21] or separation into compaction-only servers [22]–[24] in distributed and cloud environments have been used to improve compaction throughput. Second, there have been attempts to redesign the architecture of LSM-KVS by utilizing devices such as non-volatile memory [25]–[27], and third, to mitigate write stalls by scheduling compactions on LSM-KVS [12], [28].

However, adding new devices to edge servers is challenging due to their limited capacity and insufficient resources. Thus making it difficult to apply the previously mentioned studies. In edge servers, compaction delays not only cause performance degradation in LSM-KVS due to write stalls but also degrade the performance of co-located applications as the compaction tasks continuously occupy resources. To improve the compaction processing time in these edge servers, we propose compaction offloading among edge servers based on the emerging trend of edge federation, where tasks are performed by resource sharing among heterogeneous edge servers having diverse computational capacities. Yet, this inter-edge server compaction offloading raises several questions. **Q1:**

*Corresponding authors : Sungyong Park and Hongsu Byun.

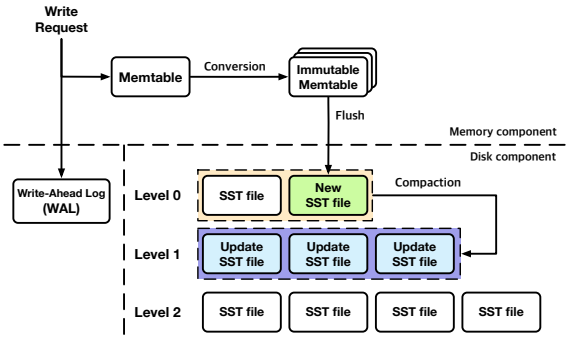


Fig. 1: An architecture of log-structured merge-tree based key-value stores.

Is compaction offloading between edge servers feasible for improving compaction processing time? **Q2:** If it is feasible, how can we determine its viability? **Q3:** Ultimately, to which server should the compaction be offloaded?

To answer the above questions, we propose EDGEPILOT, a resource-efficient and computing power-aware compaction offloading mechanism. It manages compaction placement based on resource and computing power monitoring of edge servers configured in an edge federation. Our approach involves using the EDGECODE, a mathematical model-based method, to predict the processing time of compaction offloading. EDGECODE is a model that predicts processing time for compaction offloading based on server capability. Therefore, when a compaction is triggered, it checks whether the processing time can be reduced by offloading the triggered compaction to another edge server. If so, it determines which server can process the offloaded compaction most efficiently considering both resources.

Our contributions are as follows:

- We propose a resource-efficient and computing power-aware compaction offloading mechanism EDGEPILOT in edge federation.
- We employ the model-based EDGECODE to determine the effectiveness of compaction offloading in EDGEPILOT.
- We model EDGECODE mathematically based on experimental data.
- We implemented EDGEPILOT in RocksDB v8.3.2, with representative results of a 71% decrease in write stall duration and a 17% increase in throughput over the baseline RocksDB.

II. BACKGROUND AND MOTIVATION

A. Edge Computing and Federation

Despite the powerful computing power and relatively low storage costs of the cloud environment [29], it is not suitable for real-time applications and location-based applications due to relatively high latency, network congestion, and data privacy. As a result, edge computing, a decentralized computing model, has emerged. Edge computing effectively addresses these requirements by performing computing in physical proximity to the devices and sensors that generate data.

Different edge architectures are realized in three main ways, depending on the edge nodes that participate in performing the task. The first is where high performance servers deployed in the edge environment perform the task, the second is where heterogeneous edge nodes with varying computing capability perform the task, and the third is where cloud servers connected to the edge collaborate to perform the task [6].

However, edge servers have limited resources and low computing power compared to servers in the cloud [30] because edge servers have a wide range of capacities, from gateways and routers to micro data center. [31], [32]. Therefore, the second approach, resource sharing among edge servers, is introduced to overcome this limited resource in an environment consisting of heterogeneous edge servers [33], [34]. By coordinating the tasks assigned to an edge server with other edge servers connected to the same hierarchy, this method can efficiently utilize surplus resources and improve performance incidentally. In this paper, we consider an environment consisting of heterogeneous edge servers performing such resource sharing as an **edge federation environment**.

B. Log-Structured Merge-Tree

Basics. The Log-Structured Merge-tree (LSM-tree) is widely used as an engine for key-value stores (KVS). Figure 1 shows the structure of LSM-tree. The LSM-tree consists of memory component and disk component. Within the memory component, memtables buffer write requests. The LSM-tree achieves high write performance adopting append-only manner. Once the memtable reaches a certain threshold size, it becomes an immutable memtable and is then flushed to the disk component at level 0 ($L0$) as a Sorted String Table file (SST file). Since the SST files in $L0$ are stored in an append-only manner. They are not sorted against each other. Therefore, increasing the number of SST files in $L0$ reduces the read performance. Therefore, when the size of $L0$[1] reaches a certain threshold, the compaction operation is performed in the background after merging and sorting and storing in $L1$. Compaction is a compute-intensive operation that guarantees read performance by reordering data stored in append-only manner. Every level has a compaction trigger threshold, just like $L0$.

Write Stalls. As the input workload requests more write operations, the cycle of SST file creation due to flushes shortens, leading to frequent compactions. If the write request ratio is higher than the compaction processing ratio, compaction is delayed and read performance cannot be guaranteed. This causes a write stall that blocks write requests. A write stall occurs in the following three cases: (i) When an Immutable MemTable reaches the write stall threshold. No more MemTable creation or data insertion is allowed. (ii) $L0$ reaches the write stall threshold. Unlike other levels, $L0$ is not sorted. Thus an increase in the size of $L0$ is fatal to read performance. (iii) The pending compaction size reaches the threshold. This is because read performance decreases as the unprocessed and scheduled compaction size increases. When a write stall occurs, I/O is

[1]Size of Level means the number of SST files

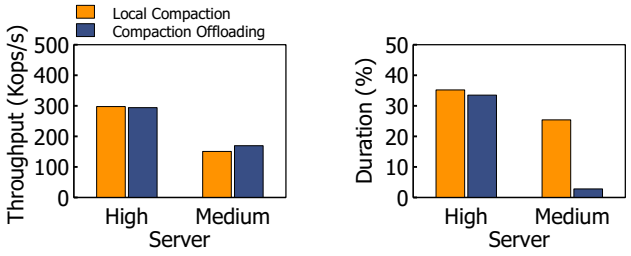
blocked until all delayed flushes/compactions are completed, which is detrimental for both throughput and tail latency. Therefore, compaction has a significant impact on LSM-KVS performance.

C. Related Work

Compaction Offloading with Accelerators. There have been numerous studies focused on offloading compaction to devices like accelerators to improve compaction processing time. For instance, X-Engine [17], [35], proposed for large-scale e-commerce platforms as an OLTP database, utilized FPGA to accelerate compaction and alleviate CPU bottlenecks, thereby enhancing overall throughput. Sun et al. [18] performed a software-hardware co-design to integrate FPGA compaction offloading into LevelDB [36]. LUDA [15] and gLSM [16] offloaded and accelerated compaction using GPUs. DComp [19] leveraged Data Processing Units (DPU) to accelerate compression during compaction, reducing CPU overhead. Lim et al. [20] utilized in-storage processing (ISP) for compaction, reducing data movement needed when using accelerators like FPGA. PStore [37] proposed asynchronous compaction using computational storage drives capable of executing ISP.

Resource Disaggregation for Compaction. Instead of using accelerators for offloading and accelerating compaction, there have also been studies that improve compaction through tiering and resource disaggregation in distributed and cloud environments [22]–[24], [38]–[40]. Nova-LSM [22] utilized Remote Direct Memory Access (RDMA) to separate storage and processing components in cloud environments. It offloaded compaction to the storage component and dynamically configured ranges to increase the parallelism of compaction. dLSM [38] proposed an LSM-based index in a disaggregated memory architecture, where compaction is executed in near-memory nodes to minimize data transfer. ROCKSMASH [23] separated metadata from local storage in the cloud. Compaction is performed in cloud storage, minimizing the compaction overhead on local storage servers. Rockset [39] is an optimization of RocksDB suitable for cloud environments. When multiple instances share cloud storage, remote compaction [40] to an instance other than the one running RocksDB has been proposed.

Limitations of Previous Studies. Despite numerous studies on improving through Compaction Offloading, applying this in edge computing environments presents challenges. Firstly, edge servers can range from gateways and routers to micro data centers. But the weak servers have limited scalability, which makes it difficult to add new devices like accelerators. Secondly, since edge servers do not perform highly like servers in cloud environments, centralized offloading to a specific server can lead to resource overload and potentially degrade performance. Efficient compaction offloading between heterogeneous servers requires detailed consideration of their respective computing power and resources, which poses a significant challenge.



(a) Average Throughput (b) Stall Duration

Fig. 2: Comparison of (a) average throughput and (b) total stall duration for two servers with high and medium computing performance, when performing Local Compaction and Compaction Offloading.

D. Motivation

Feasibility of Compaction Offloading Between Edge Servers. We evaluated how well compaction offloading works between servers that operate LSM-KVS. The experimental settings are described in detail in Section V-A. Figure 2 shows how offloading compaction from medium-tier to high-tier servers affects LSM-KVS performance. The experiment was conducted using the `FillRandom` workload of `db_bench` in RocksDB. *Local Compaction* involves two servers processing data locally, while *Compaction Offloading* requires two servers to collaborate and can offload compaction to each other. In the experiment, the medium-tier server offloaded all compaction to the high-tier server. The high-tier server performed the compaction of both the medium-tier server and its own simultaneously.

Figure 2(a) shows the average throughput. The Medium-tier server shows a 10% improvement, while the performance of the high-tier server remains unchanged. With its higher computation power, the high-tier server could handle additional compaction without any remarkable performance degradation. Moreover, the write stall on the medium-tier server decreased by 22% because the high-tier server processed compaction faster, as shown in Figure 2(b). This led to the overall throughput improvement.

Challenges. In an edge federation environment consisting of heterogeneous servers, resource utilization of each server also varies depending on the applications running on individual servers. Delaying compaction on an already overloaded server can lead to reduced performance of LSM-KVS due to write stalls. Additionally, it can also adversely affect the performance of other applications on the same server, as the compaction task can occupy the CPU and cause further degradation. In such cases, offloading compaction to another server can alleviate CPU overload and reduce system performance degradation. However, since compaction is compute-intensive, it may also cause CPU overload on offloaded servers. If the server with lower computing power is chosen for offloading, it might further delay compaction processing. Therefore, the decision of where to offload compaction should be made carefully, considering both resource status and computing power.

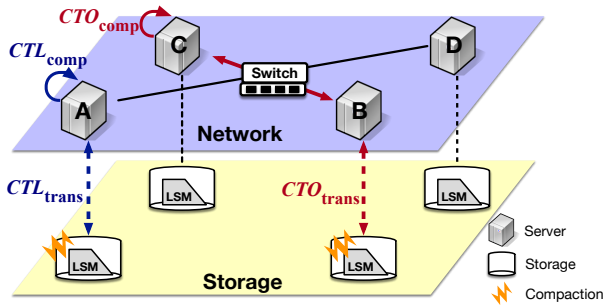
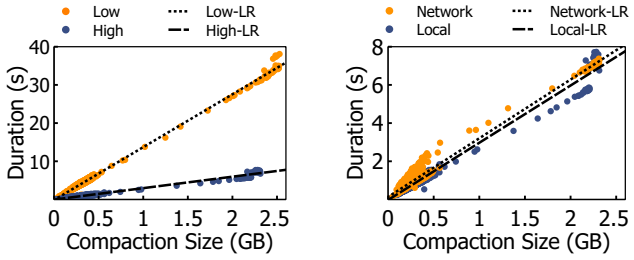


Fig. 3: Describe an example of compaction offloading in Edge Federation. The alphabet of the server indicates its name.



(a) Computing Power (b) Storage

Fig. 4: Comparison of processing times for compactions of different sizes. In the legend, LR stands for a linear regression function. The comparison is made in terms of (a) differences in CPU computing power, and (b) differences in Local/Network Storage.

III. Problem Definition and EdgePilot

A. Problem Definition

Edge federation is proposed to overcome the limitations of servers with constrained resources and computing power in Edge computing. It aims to enhance the efficiency of edge computing by enabling resource sharing and task redistribution among edge servers. The purpose of compaction offloading in edge federation is not only to enhance the resource usage of servers, but also to reduce the occurrence of write stalls by reducing compaction processing. Figure 3 shows an example of edge federation. In an edge federation, the compaction processing time can be set in two ways, with or without offloading, as follows.

- **Compaction Time by Local (CTL):** The duration of LSM-KVS compaction without offloading.
- **Compaction Time by Offloading (CTO):** Time duration compaction offloading from LSM-KVS to other servers in the edge federation.

The CTL consists of the data transfer time of the SST file (CTL_{trans}) and the computation time to perform the merge-sort (CTL_{comp}). In the Figure 3, Server A is an example of performing local compaction. On the other hand, Server B is an example of compaction offloading to Server C. The CTO consists of data transfer time (CTO_{trans}) and computation time (CTO_{comp}). Here, CTO_{trans} is slower than CTL_{trans}, which directly accesses local storage because it goes over the network. The data transfer time difference ($Diff_{\text{trans}}$) is as follows.

$$Diff_{\text{trans}} = CTO_{\text{trans}} - CTL_{\text{trans}} \tag{1}$$

However, CTO_{comp} may be lower than CTL_{comp} if the local server's CPU is overloaded, or if the server performing the offloading has more computing power.

The computation time difference ($Diff_{\text{comp}}$) is as follows.

$$Diff_{\text{comp}} = CTL_{\text{comp}} - CTO_{\text{comp}} \tag{2}$$

Therefore, the cases where compaction offloading is effective are as follows.

$$Diff_{\text{comp}} - Diff_{\text{trans}} > 0 \tag{3}$$

We opportunistically perform compaction offloading among servers within the federation when Equation 3 is satisfied, based on our model. Detailed modeling to meet this criterion is explained in Section III-C.

B. Goals of EdgePilot

We propose EdgePilot to improve the performance of the system by compaction offloading between servers in an edge federation environment. The design goals of EdgePilot are as follows.

Improving Compaction Performance. The main purpose of compaction offloading is to reduce compaction processing time. However, in edge federation, the performance and capacity of each server is heterogeneous, making it difficult to know whether compaction offloading to another server will improve processing time. To solve this EdgePilot uses the logs of LSM-KVS running on each server to collect processing time data by compaction size. When a compaction is triggered on each server, based on the collected data, the server that can process a compaction task of that size faster can be found.

Determining a Compaction Offloading Target Server. Since compaction is a compute-intensive task, it can cause CPU overload and degrade the performance of other co-located applications. If the target server performing the compaction offloading is resource overloaded, it will not only reduce the performance of the target server but also delay the compaction processing time. Therefore, when determining the offloading target server, it is crucial to consider not just the compaction processing time but also the server's resource status. To this end, EdgePilot periodically monitors the resources of each server in a fixed time window. Based on this, we can determine which servers will not be overloaded with resources when performing compaction offloading.

Mathematical Model-based Compaction Processing Time. To determine whether compaction offloading would improve processing time, EdgePilot collects processing time by compaction size from the LSM-KVS running on each server. However, the data collected is limited, so it is not possible to know the processing time for all compaction sizes on all servers. For this purpose, EdgePilot introduces EdgeCode,

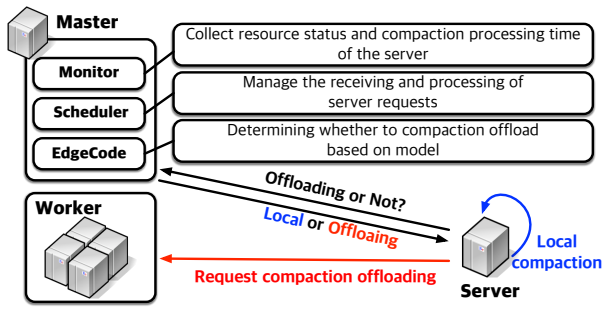


Fig. 5: An Overview of EDGEPILOT.

which leverages the collected data to determine the compaction processing time based on a model.

C. Modeling for EDGECODE

We employ experimental data to model and identify instances that satisfy the Equation 3. Modeling-based **Edge** Compaction **O**ffloading **D**ecision Mak**E**r (EDGECODE) makes the final decision on compaction offloading.

Computation Time Differences. Figure 4(a) shows the computation duration as a function of compaction size when performing a computation on two servers with different computing power (High, Low). On both servers, the computation duration increases linearly as the compaction size increases. Define the functions $High\text{-}LR(size)$ and $Low\text{-}LR(size)$ as a function of compaction size by applying a linear regression to the results from each server. Assuming that the data transfer time is the same for a specific size, the difference in computation time, $Diff_{\text{comp}}$, when offloading compaction from a Low-performance server to a High-performance server based on the size can be expressed with the following function.

$$Diff_{\text{comp}}(size) = Low\text{-}LR(size) - High\text{-}LR(size) \quad (4)$$

Data Transfer Time Differences. Figure 4(b) shows the difference in compaction processing time between storage accesses on a high-performance server. In the legend, 'Local' refers to compaction performed on Local Storage in High-1, while 'Network' denotes compaction performed on High-2's network storage from High-1. Similar to Figure 4(a), linear regression can be applied to define functions $Local\text{-}LR(size)$ and $Network\text{-}LR(size)$. Thus, assuming the computation time is the same for a specific size, the difference in transfer time, $Diff_{\text{trans}}$, based on size can be expressed as the following function.

$$Diff_{\text{trans}}(size) = Network\text{-}LR(size) - Local\text{-}LR(size) \quad (5)$$

Compaction Offloading Decision Maker.

By the Equations 4 and 5, the Equation 3 can be expressed as the following function.

$$Diff_(size) = Diff_{\text{comp}}(size) - Diff_{\text{trans}}(size) \quad (6)$$

Therefore, if $Diff_(size) > 0$, offloading compaction to the target server can reduce the compaction processing time.

Algorithm 1: Monitoring

```
1  Function MonitoringAndUpdatePriorityQueue(priority_queue):
2      while True do
           /* Every 1 second                                  */
3          sleep(1)
4          for this ← array_server[] do
               /* Get Compaction Data and Update              */
5              server.compaction_perf = get_compaction_time(this)
6              update_model(this, server.compaction_perf)
               /* Get Server Status and Update                */
7              server.availalbe_core = get_core_status(this)
8              server.availalbe_NB = get_NB_status(this)
9              priority_queue.update(server, status)
```

Algorithm 2: Comparison for Server Status

```
1  Function CompareServerStatus(serverA, serverB):
2      if (serverA.available_core ≤ 1)
3       or (serverB.available_core ≤ 1) then
4          return serverA.available_core ≥ serverB.available_core
       /* NB: Network Bandwidth                               */
5      if serverA.available_NB ≤ 0) or serverB.available_NB ≤ 0)
         then
6          return serverA.available_NB ≥ serverB.available_NB
7      if serverA.compaction_perf ≠ serverB.compaction_perf then
8          return serverA.compaction_perf ≥
             serverB.compaction_perf
9      return serverA.available_core ≥ serverB.available_core
```

IV. EDGEPILOT:DESIGN AND IMPLEMENTATION

The EDGEPILOT is organized in a master-worker model, here the master is elected from among the servers within the edge federation with the highest computing power and the most resources. Figure 5 shows an overview of the EDGEPILOT. When a compaction is triggered, all servers query the master server about offloading the compaction. If there is a worker server that can process the compaction faster than the querying server, the master server allows querying server to perform compaction offloading to appropriate worker server. Otherwise, the querying servers perform local compaction.

Master is composed of three modules: Monitor, Scheduler, and EDGECODE. Monitor collects the resource (CPU, Memory, Network) status and compaction processing capability of all servers periodically in a time window. Scheduler receives and processes compaction offloading requests from servers. The EDGECODE creates a compaction processing time model based on the data collected from the monitor, and decides whether to offload compaction based on the model.

A. System Modules

Monitor. The monitor in EDGEPILOT collects the compaction processing capability for each server, and monitors the available network bandwidth and CPU utilization. Algorithm 1 shows the algorithm of the Monitor module. Monitor receives compaction data from the edge server every second and updates the model (lines 5-6) and sorts the priority queue based

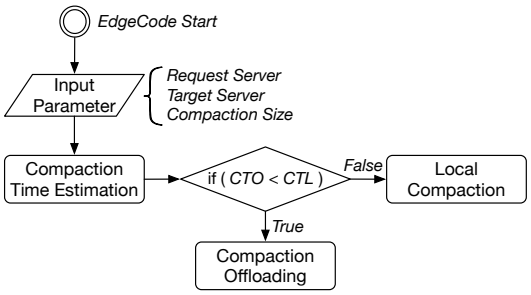


Fig. 6: A Flow Chart of EDGECODE.

on the resource status (lines 7-9). The priority algorithm for sorting the priority queue is performed based on Algorithm 2.

Priority Queue. When a server performs multiple compaction offloading simultaneously, CPU overload and network bottlenecks can cause processing time delays. EDGEPILOT solves these problems based on a resource-aware priority queue. Priority queue sorts the priority of servers based on the status received from Monitor. Algorithm 2 is a compare function algorithm to sort the priority queue. First, it checks if there is at least one available core (line2-4), because if the number of available cores is insufficient, it may cause CPU overload when the compaction is offloaded to the target server. Next, it checks if the available network bandwidth is insufficient (lines 5-6). With the above two factors, it checks whether the compaction can be performed. Then compare the compaction performance, processing time, to determine the priority (lines 7-9).

EDGECODE. Prioritization in the Priority Queue is based on the server's current available resources. However, it is required to ensure that the compaction processing time of the target server received from the Priority Queue is lower than the reqeust server. The EDGECODE uses the size of the compaction to predict the compaction time of the request server and the target server based on a model. The Figure 6 shows the flowchart of the EDGECODE algorithm. EDGECODE receives server information and compaction size as parameters. Based on this, it predicts the compaction time of each server and returns true if the target server's compaction offloading time (CTO) is lower than the local compaction time (CTL), and false otherwise.

B. System Scenario

The Figure 7 shows the system architecture of EDGEPILOT and a specific scenario. The master server's monitor collects the resource status and compaction processing time of all servers periodically during the time window to update the priority queue and model.

In the Figure 7, when compaction is triggered on Server B, a request is sent to master(❶). Requests are inquired into the request queue, and the Scheduler processes the requests sequentially(❷). The Scheduler gets information from Server C, the server with the highest prioirty in the priority queue(❸). The Scheduler asks EDGECODE to return the sever that is expected to perform the compaction faster between

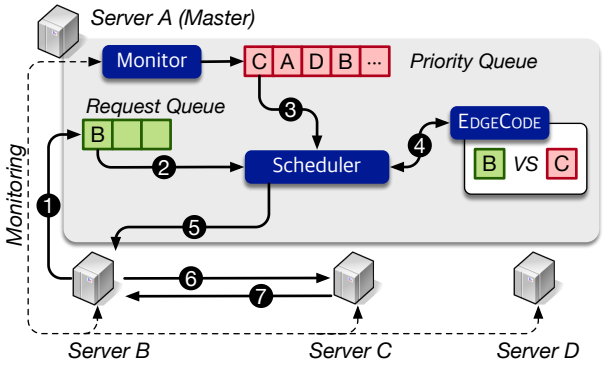


Fig. 7: A System Scenario of EDGEPILOT.

Server B, which requested scheduling, and Server C, which was pulled from the priority queue4). The Scheduler sends the return result of EDGECODE to Server B, where compaction is triggered(❺).

Server B sends a compaction request to Server C because it predicts that Server C will take less time to perform the compaction(❻). Server C performs compaction and sends the result to Server B(❼). This design allows compaction to be performed on the currently most optimized server based on real-time monitoring, which is expected to improve the performance of LSM-KVS.

V. EVALUATION

A. Experimental Setup

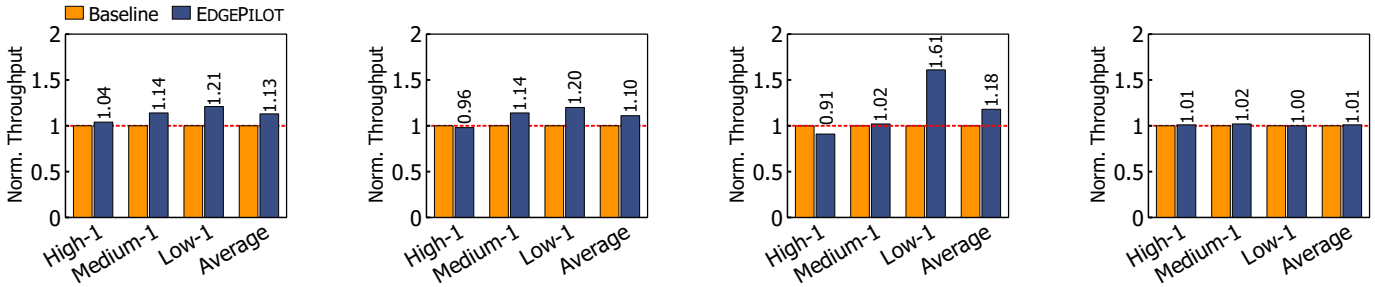
Setup. We composed experimental environment with heterogeneous servers and classify each server into three tiers (high, midieum, low) according to the specifications. And we limited number of cores in order to assume an edge server. The details of experimental specification of servers are shown in Table I. Each server is connected to a 10Gbps network and can access storage on other servers through the network.

We evaluated the performance of the following two systems.

- **Baseline:** We used RocksDB v8.3.2. RocksDB running on each server performs local compaction on local storage.
- **EDGEPILOT:** We implemented EDGEPILOT by modifying the remote compaction [41] of RocksDB v8.3.2. Local compaction is accessed by local storage. Compaction offloading is accessed by other server's storage via 10 Gbps network. EDGECODE uses the result of `FillRandom` performed 50 Mops in advance.

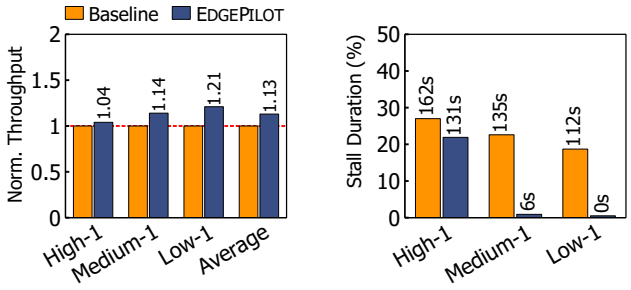
Workload. For evaluation, we ran the following three workloads on `db_bench` [42] for 600 seconds. The key-value values use 16-1000 bytes.

- `FillRandom` : It consists of 100% write.
- `ReadWhileWriting` : It consists of writer(100%) and reader(100%) and performs write and read at the same time.
- `Mixgraph` : It consists of Put(25%), Get(50%), Seek(25%). This is a proposed workload considering the actual workload characteristics of a key-value store [43]. The key value is 16 bytes, and the value value follows the



(a) FillRandom (b) ReadWhileWriting (Write) (c) ReadWhileWriting (Read) (d) Mixgraph (All random)

Fig. 8: Normalized throughput of three different workloads.



(a) FillRandom (b) ReadWhileWriting

Fig. 9: Comparison of stall duration for each server in FillRandom and ReadWhileWriting.

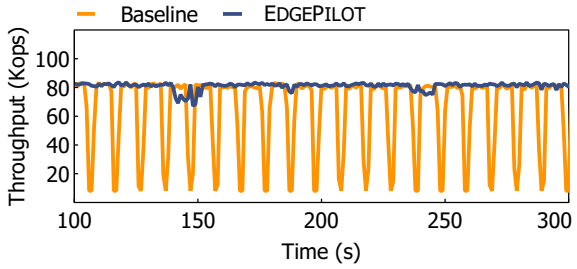


Fig. 10: Throughput of low-1 according to compaction offloading.

Generalized Pareto Distribution [44], which is the default setting of `db_bench` based on 1000 bytes.

B. Performance Analysis

Compaction occurs on all edge servers where LSM-KVS is running. EDGEPILOT offloads a compaction to the server that can process it within the shortest time at the moment. This improves the throughput of LSM-KVS across the entire system. We demonstrate this through an experiment that compares EDGEPILOT with the baseline in a system comprising one server per tier, with a total of three servers.

Figure 8 shows the average throughput over 600 seconds for both baseline and EDGEPILOT. The baseline is set to 1, and the EDGEPILOT is represented as a ratio to the baseline throughput. Figure 8(a) shows result of `FillRandom`. EDGEPILOT has 14% and 21% higher throughput than baseline on medium-1 and low-1, respectively, except for high-1. The high-1 shows negligible performance gain because there are no other servers to leverage through offloading.

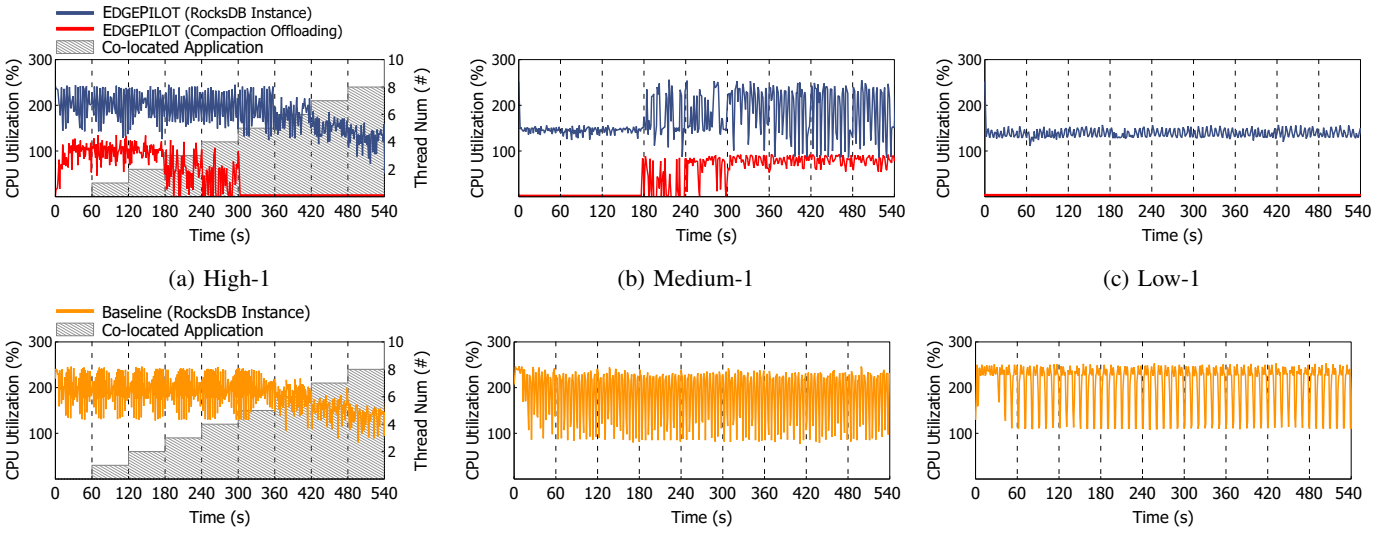
TABLE I: Server specifications.

Server	CPU	Memory	Storage
High-1	Ryzen 9 7950X @ 4.5GHz Enable 8 Cores	32 GB (DDR 5)	Samsung 970 EVO 1 TB
High-2	Ryzen 9 7950X @ 4.5GHz Enable 8 Cores	32 GB (DDR 5)	Samsung 970 EVO 1 TB
Medium-1	Ryzen 5 2600 @ 3.4 GHz, Enable 6 Cores	16 GB (DDR4)	Samsung 970 EVO 512 GB
Medium-2	Ryzen 5 2600 @ 3.4 GHz Enable 6 Cores	16 GB (DDR4)	Samsung 970 EVO 512 GB
Low-1	Ryzen 7 1700 @ 2.0 GHz (downclocked) Enable 4 Cores	8 GB (DDR4)	E-Star SSD 250 GB
Low-2	i7-6700 @ 2.0 GHz (downclocked) Enable 4 Cores	8 GB (DDR4)	Samsung PM981 512 GB

Figures 8(b) and (c) present the average throughput of executing `ReadWhileWriting`. Figure 8(b) shows the throughput for write requests of the `ReadWhileWriting` workload. In terms of write performance, EDGEPILOT achieves 14% and 20% higher throughput than the baseline on medium-1 and low-1, with the exception of high-1. This result is quite similar to Figure 8(a). Figure 8(c) depicts the throughput for read requests. The read performance in the low-1 exhibits a 61% improvement over the baseline, attributed to the efficient execution of compactions which reduces the number of $L0$ SST files required to be searched during read operations. Although EDGEPILOT is designed to reduce write stalls, it indirectly enhances read performance. Conversely, the medium-1 shows read performance similar to the baseline. This is due to the increased workload operation overhead from the reader and writer compared to other workloads. It leads to fewer compaction requests to the high-1 and an accumulation of SST files, which does not increase performance.

Figure 8(d) shows the results of executing `mixgraph`. Since the baseline also exhibits a stall duration of zero for `mixgraph`, there is no significant difference between it and all tiers of servers. This indicates that the overhead from EDGEPILOT's scheduling is almost zero.

Figure 9 shows the total write stall duration. `mixgraph` was excluded because the stall duration was 0 in all tier



(a) High-1 (b) Medium-1 (c) Low-1

(d) High-1 (e) Medium-1 (f) Low-1

Fig. 11: CPU utilization of the RocksDB instance and Compaction offloading process on high, medium, low-tier servers. To simulate a co-located condition with other applications, the CPU consumption (number of threads) of an extra application gradually increases over time.

servers, including baseline. Figure 9(a) shows the medium-1 experiences a significant decrease in stall duration by 99.6%, and the low-1 sees a 100% reduction. The duration of stalls in the high-1 also reduced. This reduction is attributed to the performance of additional computations (offloaded compaction), which slightly extended the processing time of write requests, thereby decreasing the duration of write stalls. Figure 9(b) exhibits a pattern similar to that of Figure 9(a). Decreased stall duration by 99.5% in medium-1, and the low-1's stall duration sees zero.

Figure 10 compares baseline and EDGEPILOT in low-1 for write stall when running `FillRandom`. In the case of baseline, there are periodic write stalls that reduce throughput to about 8-9 kops. In the case of EDGEPILOT, the throughput remains constant. It means EDGEPILOT reduces the occurrence of write stalls by offloading compaction and reducing the execution time of compaction.

C. Performance Analysis with Co-Locate Applications

We ran extra applications and assumed that they were co-located with the `FillRandom` workload. To see how the load placed on the server by the extra application changes, we gradually increased the number of threads that put load on the server. The extra application used the Stress [45] test with the task of computing square roots. The experimental environment consisted of three machines, one for each tier. The extra application was executed on a high-tier server, and the extra application threads was increased by one every minute from 0 to 8 for a total of 9 minutes.

Figure 11(a) shows the CPU utilization of the high-1 as the number of extra threads increases. On the high-1, the RocksDB instance takes up to 250% CPU utilization. Even if the extra application is co-located, if the amount of computation of the extra application is small, offloaded compaction can be performed on the medium-1 or low-1. From 0 to 180 seconds, the high-1 performs the compaction of medium-1 and low-1, and the compaction offloading shows a maximum CPU utilization of 100%. However, after 180 seconds, the compaction task offloaded to high-1 decreases depending on the status of the high-1 CPU resource. After 300 seconds, when the 5 extra application threads start running, high-1 does not perform compaction on other servers. As a result, its compaction offloading process remains 0% until 360 seconds. This is because the monitor in EDGEPILOT checks the core status of the high-1 and only offloads tasks to that server that do not affect the RocksDB instance on the high-1.

However, after 360 seconds, the CPU utilization of the RocksDB instance decreases from 180% to less than 100%. This is because the CPU consumption of the extra application thread, which is a co-located application, was overloading the server. Since EDGEPILOT is a solution to reduce the load on the server by offloading compaction, it has the limitation that it cannot solve the performance drop that occurs when the co-located application contends for resources with the LSM-KVS instance except for compaction processing.

Figure 11(b) shows the CPU utilization of the medium-1. Until 180 seconds, it offloads all compactions that occur while the co-located application on high-1 has 0-2 threads to high-1. After 180 seconds, if the number of load threads on high-1 increases to 3, medium-1 will either perform the compaction on its own LSM-KVS or perform the compaction on low-1 on medium-1. After 300 seconds, when the number of threads reaches 5, the medium-1 performs all the low-1's compactions.

Figure 11(c) shows the CPU utilization of the low-1. The low-1 can improve its throughput sufficiently by offloading compactions to the medium-1 because it does not generate

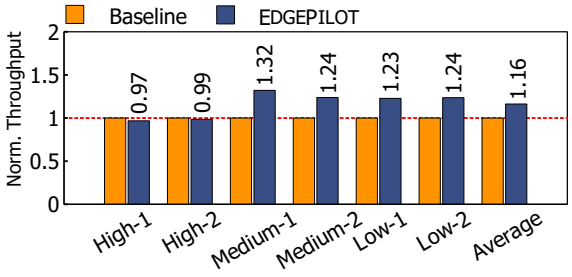


Fig. 12: Normalized throughput of the servers.

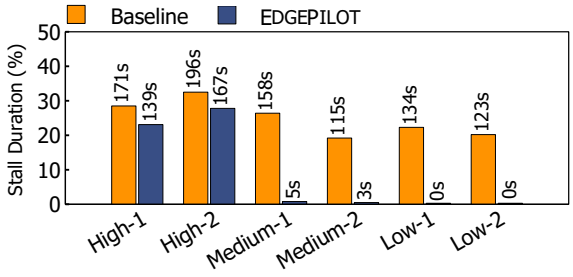


Fig. 13: Normalized stall duration of the servers.

writing stalls. Therefore, the compaction is offloaded to the server that is predicted to have a shorter compaction execution time among the high-1 and medium-1. The CPU utilization of the RocksDB instance on the low-1 remains low and stable because it does not perform compactions. Also, the CPU utilization of the low-1's compaction offloading remains 0% because it does not perform compactions of other servers.

Figure 11(d) shows a similar pattern to Figure 11(a). It is because EDGEPILOT is designed to offload compaction tasks to available cores based on current CPU utilization. This approach ensures that even when compaction tasks are offloaded to the high-1, they do not adversely affect the overall CPU utilization. This indicates a strategic management of computational resources to maintain performance efficiency. Figure 11(e) and (f) show the results for the same workload as in Figure 11(b) and (c). The baseline has fluctuating CPU utilization due to continuously triggered compactions. Such variability in CPU utilization has the potential to impact the performance of co-located applications, which can be particularly detrimental for servers with limited capacity. In contrast, when utilizing EDGEPILOT, although the CPU utilization varies depending on the co-located application, servers with weaker capacity exhibit stable CPU utilization due to the ability to offload compactions to a greater number of servers.

D. Scalability Analysis

As the number of servers in the network increases, the distribution of each node's workload must also be considered. In an edge federation environment, each server's LSM-KVS typically operates alongside other applications and is constrained to minimize the impact on the co-located applications. Therefore, offloading only the compactions generated by a particular server, or overloading a particular server with too many offloaded compactions, can negatively impact these limits or reduce the overall performance of the system. These issues are exacerbated as the number of servers in the edge increases, reducing the scalability of the edge environment. We ran the `FillRandom` workload to benchmark on all servers in Table I by increasing the number of servers per tier in EDGEPILOT.

Figure 12 shows the ratio of the average throughput of LSM-KVS in the experiment with EDGEPILOT to the baseline experiment with set to 1. The servers in each tier had higher percentages of throughput change as in Figure 8(a). This improvement is attributed to increased shared resources, facilitating more efficient scheduling and compaction offloading and enhancing performance. The high-tier servers experienced an average throughput drop of up to 3.25%, which is not significantly different from the drop of the high-1 in Figure 9(a). This indicates that EDGEPILOT distributes the compaction generated across the entire system, with no high-tier servers being overly assigned compaction tasks. From this, we observe that as the number of servers increases, the compaction jobs are properly distributed and have minimal impact on the average throughput of LSM-KVS running on those servers. Furthermore, the average throughput improvement of up to 32.09% and down to 22.80% occurred on the medium and low-tier servers. Therefore, EDGEPILOT was leveraged by all medium and low-tier servers even as the number of servers increased.

Figure 13 shows the writing stall duration of each server. If the writing stall duration decreases while the throughput of the LSM-KVS increases, it means that the compaction is processed at a faster rate. Therefore, in this case, the decrease in writing stall duration can be considered as an indicator of the compaction optimization performed by EDGEPILOT. Since both the medium and low-tier servers are less than or equal to Figure 9, this indicates that the medium and low-tier servers are evenly optimizing compaction performance with EDGEPILOT. At this time, the average throughput increase of LSM-KVS on the medium-tier servers are higher than the medium-1 in Figure 9(a). The stall duration of medium-1 and medium-2 decreased to 97.10% and 97.36%. Therefore, both servers optimize performance equally through EDGEPILOT at the same rate.

VI. CONCLUSION

We proposed EDGEPILOT, a scheduling mechanism to offload compaction for LSM-KVS in an edge cluster. EDGEPILOT solves the write stall problem in LSM-tree by offloading compaction to a server that can perform compaction faster in the edge cluster environment. To achieve this, we devised a resource-aware scheduling model with Priority Queue and EDGECODE. Based on this, EDGEPILOT finds a target server for offloading compaction with minimal computation. Compared to RocksDB, EDGEPILOT reduces system-wide write stall duration by up to 71% and improves throughput by 17% in write-intensive workloads.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386).

REFERENCES

[1] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[2] M. A. Zamora-Izquierdo, J. Santa, J. A. Martínez, V. Martínez, and A. F. Skarmeta, "Smart farming iot platform based on edge and cloud computing," *Biosystems Engineering*, vol. 177, pp. 4–17, 2019. Intelligent Systems for Environmental Applications.

[3] S. Trinks and C. Felden, "Edge computing architecture to support real time analytic applications : A state-of-the-art within the application area of smart factory and industry 4.0," in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 2930–2939, 2018.

[4] e. a. Abdellatif, "Edge computing for smart health: Context-aware approaches, opportunities, and challenges," *IEEE Network*, vol. 33, no. 3, pp. 196–203, 2019.

[5] "IoT statistics and facts." https://comparitech.com/internet-providers/iot-statistics, 2023.

[6] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge computing for the internet of things: A case study," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1275–1284, 2018.

[7] H. Gupta and U. Ramachandran, "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access," in *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, DEBS '18, (New York, NY, USA), p. 148–159, Association for Computing Machinery, 2018.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[9] K. M. M. Thein, "Apache kafka: Next generation distributed messaging system," *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.

[10] Facebook, "Compaction." https://github.com/facebook/rocksdb/wiki/Compaction, 2023.

[11] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *27th ACM Symposium on Operating Systems Principles*, pp. 447–461, 2019.

[12] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing latency spikes in Log-Structured merge Key-Value stores," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 753–766, USENIX Association, July 2019.

[13] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "Triad: Creating synergies between memory, disk and log in log structured key-value stores," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 363–375, 2017.

[14] Facebook, "Write Stalls." https://github.com/facebook/rocksdb/wiki/Write-Stalls, 2021.

[15] P. Xu, J. Wan, P. Huang, X. Yang, C. Tang, F. Wu, and C. Xie, "Luda: Boost lsm key value store compactions with gpus," 2020.

[16] H. Sun, J. Xu, X. Jiang, G. Chen, Y. Yue, and X. Qin, "Glsm: Using gpgpu to accelerate compactions in lsm-tree-based key-value stores," *ACM Trans. Storage*, nov 2023. Just Accepted.

[17] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, Z. Huang, and J. Sun, "Fpga-accelerated compactions for lsm-based key-value store," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, FAST'20, (USA), p. 225–238, USENIX Association, 2020.

[18] X. Sun, J. Yu, Z. Zhou, and C. J. Xue, "Fpga-based compaction engine for accelerating lsm-tree key-value stores," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1261–1272, 2020.

[19] C. Ding, J. Zhou, J. Wan, Y. Xiong, S. Li, S. Chen, H. Liu, L. Tang, L. Zhan, K. Lu, *et al.*, "Dcomp: Efficient offload of lsm-tree compaction with data processing units," in *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 233–243, 2023.

[20] M. Lim, J. Jung, and D. Shin, "Lsm-tree compaction acceleration using in-storage processing," in *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pp. 1–3, 2021.

[21] H. Sun, B. Lou, C. Zhao, D. Kong, C. Zhang, J. Huang, Y. Yue, and X. Qin, "An asynchronous compaction acceleration scheme for near-data processing-enabled lsm-tree-based kv stores," *ACM Trans. Embed. Comput. Syst.*, sep 2023. Just Accepted.

[22] H. Huang and S. Ghandeharizadeh, "Nova-lsm: A distributed, component-based lsm-tree key-value store," in *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, p. 749–763, Association for Computing Machinery, 2021.

[23] P. Xu, N. Zhao, J. Wan, W. Liu, S. Chen, Y. Zhou, H. Albahar, H. Liu, L. Tang, and C. Xie, "Building a fast and efficient lsm-tree store by integrating local storage with cloud storage," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 125–134, 2021.

[24] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–14, 2015.

[25] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "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)*, pp. 17–31, USENIX Association, July 2020.

[26] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *Proceedings of the 2018 USENIX Annual Technical Conference(ATC 18)*, pp. 993–1005, 2018.

[27] C. Ding, T. Yao, H. Jiang, Q. Cui, L. Tang, Y. Zhang, J. Wan, and Z. Tan, "Trianglekv: Reducing write stalls and write amplification in lsm-tree based kv stores with triangle container in nvm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4339–4352, 2022.

[28] J. Yu, S. H. Noh, Y. ri Choi, and C. J. Xue, "ADOC: Automatically harmonizing dataflow between components in Log-Structured Key-Value stores for improved performance," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, (Santa Clara, CA), pp. 65–80, USENIX Association, Feb. 2023.

[29] M. S. Aslanpour, S. S. Gill, and A. N. Toosi, "Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research," *Internet of Things*, vol. 12, p. 100273, 2020.

[30] I. Sittón-Candanedo, R. S. Alonso, J. M. Corchado, S. Rodríguez-González, and R. Casado-Vara, "A review of edge computing reference architectures and a new global edge proposal," *Future Generation Computer Systems*, vol. 99, pp. 278–294, 2019.

[31] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019.

[32] Z. Zhang, W. Zhang, and F.-H. Tseng, "Satellite mobile edge computing: Improving qos of high-speed satellite-terrestrial networks using edge computing techniques," *IEEE Network*, vol. 33, no. 1, pp. 70–76, 2019.

[33] F. Bonomi, R. A. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big Data and Internet of Things*, 2014.

[34] T. Nishio, R. Shinkuma, T. Takahashi, and N. B. Mandayam, "Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud," MobileCloud '13, (New York, NY, USA), p. 19–26, Association for Computing Machinery, 2013.

[35] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li, "X-engine: An optimized storage engine for large-scale e-commerce transaction processing," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 651–665, 2019.

[36] Google, "Leveldb." https://github.com/google/leveldb, 2017.

[37] H. Sun, B. Lou, C. Zhao, D. Kong, C. Zhang, J. Huang, Y. Yue, and X. Qin, "An asynchronous compaction acceleration scheme for near-data processing-enabled lsm-tree-based kv stores," *ACM Trans. Embed. Comput. Syst.*, sep 2023. Just Accepted.

[38] R. Wang, J. Wang, P. Kadam, M. Tamer Özsu, and W. G. Aref, "dlsm: An lsm-based index for memory disaggregation," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 2835–2849, 2023.

[39] Rockset, "Rocksdb-cloud: A key-value store for cloud applications." https://github.com/rockset/rocksdb-cloud, 2020.

[40] Rockset, "Remote Compactions in RocksDB-Cloud." https://rockset.com/blog/remote-compactions-in-rocksdb-cloud/., 2021.
[41] Facebook, "Remote Compaction." https://github.com/facebook/rocksdb/-wiki/Remote-Compaction-%28Experimental%29, 2021.
[42] Facebook, "db_bench." https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2023.
[43] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pp. 209–223, 2020.
[44] J. R. Hosking and J. R. Wallis, "Parameter and quantile estimation for the generalized pareto distribution," *Technometrics*, vol. 29, no. 3, pp. 339–349, 1987.
[45] "Stress - tool to impose load on and stress test a computer system." https://manpages.ubuntu.com/manpages/jammy/en/man1/stress.1.html.