MetaHive: A Cache-Optimized Metadata Management for Heterogeneous Key-Value Stores

Alireza Heidari
alireza.heidarikhazaei@huawei.com
Huawei Cloud

Amirhossein Ahmadi
amirhossein.ahmadi@huawei.com
Huawei Cloud

Zefeng Zhi
zefeng.zhi@huawei.com
Huawei Cloud

Wei Zhang
wei.zhang6@huawei.com
Huawei Cloud

ABSTRACT

Cloud key-value (KV) stores provide businesses with a cost-effective and adaptive alternative to traditional on-premise data management solutions. KV stores frequently consist of heterogeneous clusters, characterized by varying hardware specifications of the deployment nodes, with each node potentially running a distinct version of the KV store software. This heterogeneity is accompanied by the diverse metadata that they need to manage. In this study, we introduce MetaHive, a cache-optimized approach to managing metadata in heterogeneous KV store clusters. MetaHive disaggregates the original data from its associated metadata to promote independence between them, while maintaining their interconnection during usage. This makes the metadata opaque from the downstream processes and the other KV stores in the cluster. MetaHive also ensures that the KV and metadata entries are stored in the vicinity of each other in memory and storage. This allows MetaHive to optimally utilize the caching mechanism without extra storage read overhead for metadata retrieval. We deploy MetaHive to ensure data integrity in RocksDB and demonstrate its rapid data validation with minimal effect on performance.

VLDB Workshop Reference Format:
Alireza Heidari, Amirhossein Ahmadi, Zefeng Zhi, and Wei Zhang. MetaHive: A Cache-Optimized Metadata Management for Heterogeneous Key-Value Stores. VLDB 2024 Workshop: Cloud Databases (CloudDB).

1 INTRODUCTION

Context. The global cloud computing market was valued at *USD 602.31 billion* in 2023 and is projected to grow at a *rate of 24.8%* from 2024 to 2030 [2]. Cloud computing provides businesses with a cost-effective and adaptive alternative to traditional on-premise data management solutions. Key-value (KV) stores represent a substantial portion of the market, compromising *over 20%* of the Cloud Databases sector [3].

A KV store is a form of non-relational database that stores data in pairs of key values, each key acting as a unique identifier and projecting it to the corresponding value [19]. The flexibility of this model allows for the storage of a diverse range of data, from simple objects to intricate compound objects. One of the primary benefits of KV stores is their high partitionability, which allows horizontal scaling that exceeds the capabilities of other database models [25]. In the past decade, *over 20 KV stores* have been developed, including LevelDB [1], RocksDB [8], and DuckDB [17]. These databases are widely used by cloud providers, handling billions of data points across various cloud platforms. [1]

Key-value metadata [19] is essential in various situations in cloud databases, where it requires storing and utilizing additional information related to the specified key and value in downstream application processes. An example is the *ETL (Extract, Transform, and Load) process* [11], where data from various sources are combined into a centralized repository known as a data warehouse. Metadata is heavily relied upon in this process [20], as it facilitates the application of business rules to clean, organize, and prepare raw data for storage, data analysis, and machine learning (ML) applications. Another significant example in the Cloud DB context is *verifying data integrity*. Cloud KV stores often form a distributed KV cluster due to the substantial volume of data and processing involved [21]. As these clusters evolve, they encompass a diverse range of nodes with varying hardware capabilities, each running a distinct version of the database software code, resulting in heterogeneous KV clusters. In such environments, verifying the accuracy of the data is essential, as the data frequently migrate between various nodes, and the hardware or software characteristics of each node may introduce errors in the KV store entries.

Motivational Example. Consider a cluster of three nodes N_1, N_2, and N_3 within a banking system, interconnected via communication protocols and APIs. However, these nodes are not identical; the cluster is heterogeneous where N_1 and N_3 share the same setup, while N_2 is an older setup with more error-prone devices. In this scenario, the cloud owner opts to prevent error injection by implementing a data integrity system (e.g., checksum) on N_2 such that N_1 and N_3 remain unaware of its presence. Due to the ease of correcting minor errors, N_2 necessitates a high-performance data integrity system operating at the finest granularity. Additionally, there should be no changes in the API or the code of the other nodes. Given that the data stored in this cloud comprises high-profile banking information, the integrity system must not alter the values' nature. MetaHive offers a management solution that allows the cloud owner to satisfy all these requirements concurrently.

[1] In general, MetaHive works on all databases that store keys in a sorted manner: LevelDB [1], RocksDB [8], WiredTiger (used by MongoDB) [16], BadgerDB, TiKV, and DuckDB [17].

Objectives. Three main objectives should be considered for metadata management in cloud KV stores:

(i) **Performance**: The metadata for each KV should be located close to its corresponding data because, in many instances, the application requires access to the metadata immediately after reading the KV (e.g., verifying data correctness). To enhance cache efficiency and minimize cache misses and memory page (block) lookups, the metadata should be placed on the same memory block as the corresponding KV.

(ii) **Heterogeneity**: Cloud databases often employ a distributed KV store architecture, where each KV store shard is hosted on a separate node. These nodes can have varying hardware specifications, and the software version of the KV store on each node might differ. Within this diverse KV store cluster, it is crucial to ensure that introducing the KV metadata does not affect any version of KV store applications. This necessitates *backward compatibility*, allowing older KV data to function with current data, as well as *forward compatibility*, ensuring that the new version of KV operates with previous software code.

(iii) **Privacy**: In a clustered environment, each KV shard on a node contains a subset of all KVs. These KVs could represent private data specific to each node. Consequently, the metadata containing information about the key values of that node should be stored on the same edge node and should not be migrated to other shards.

Contributions. In this paper, we present *MetaHive, a metadata management solution designed with privacy and efficiency in mind for diverse KV stores*, which uniquely addresses all these objectives at once. While the proposed design is applicable to all KV stores, it is practically implemented and evaluated on a heterogeneous cluster of RocksDB nodes, a high-performance embedded database for key-value data. The focus is on ensuring data integrity in a cluster of RocksDB shards that constitute a Cloud KV store, where the DB provider must assure users of data correctness [10].

MetaHive introduces metadata as a KV structure that is specifically designed to work in a heterogeneous cluster while maintaining backward and forward compatibility. The design also ensures the privacy of each node's metadata. To ensure data integrity, a checksum part is included in the payload of the metadata entries. This checksum represents the checksum of the corresponding KV entry, providing a means to verify the integrity of the data. Additionally, MetaHive guarantees that both the KV store pairs and their corresponding metadata are written on the same memory page throughout various RocksDB processes, such as compaction. This ensures efficient and consistent handling of data and metadata within the system.

Organization. The remainder of the paper proceeds as follows: Section 2 covers the background concepts. Section 3 discusses related work, and Section 4 presents an overview of our proposed architecture. In Section 5, we evaluate the implementation of our solution in RocksDB, and Section 6 concludes with a summary of the key points.

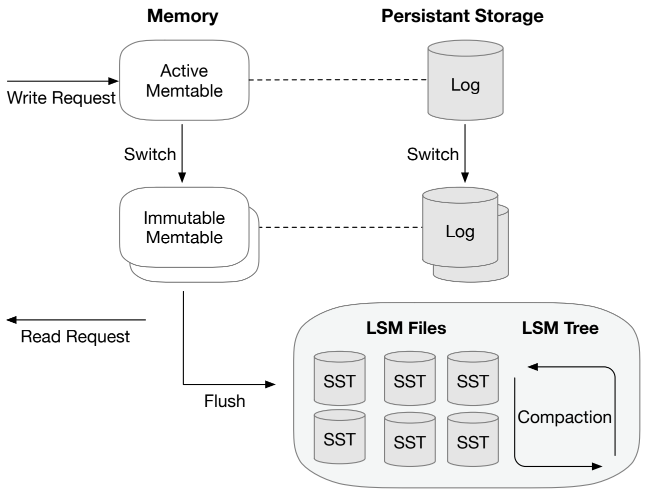


Figure 1: RocksDB Architecture

2 BACKGROUND

This section provides background information for the study and outlines the essential terminology and concepts used in this work.

2.1 Block, Memory Page, and Cache Optimality

A memory block is a contiguous chunk of memory allocated for specific purposes in a computer system, managed by the operating system, and used to store data or instructions. Block size, a critical parameter in file systems and databases, affects data access and storage efficiency. A memory page is a contiguous block of virtual memory described by a single entry in the page table, typically sized in powers of two (e.g. 4 or 8 KB), and is crucial for memory management. Cache optimality refers to the degree to which an algorithm or data structure uses cache memory to frequently access data within the same memory page to minimize page faults and improve performance[4, 22, 23]. Designing with memory blocks and pages in mind can improve cache usage and system efficiency.

2.2 RocksDB Architecture

RocksDB, created by Meta in 2012, is a highly efficient key-value storage engine [7]. It is specifically optimized to utilize the capabilities of Solid State Drives (SSDs). This storage engine is mainly designed for large-scale distributed applications and is implemented as a library component that can be integrated into higher-level applications. RocksDB uses log-structured merged trees (LSM) as its core data structure [18].

When data is written to RocksDB, it undergoes two main processes. Initially, the data are placed into an in-memory write buffer known as the MemTable. At the same time, a Write-Ahead Log (WAL) is generated on the disk (see Figure 1). The MemTable is structured as a skiplist, maintaining the data in an ordered fashion with an insertion and search complexity of $O(log n)$. The WAL acts as a recovery tool in the event of failures, although its use is optional. Once the MemTable hits a predetermined size threshold, both it and the WAL are set to an immutable state. Then, new MemTable and WAL instances are created for future writes. The data in the

MemTable are then transferred to a 'Sorted String Table' (SST) file on the disk, and the old MemTable and WAL are discarded. Each SST organizes data in a sorted sequence and is divided into equally sized blocks. Furthermore, each SST contains an index block that contains one index entry per data block, enabling efficient binary search operations.

The LSM tree in RocksDB is made up of several levels, each of which is built from multiple SSTs. The newest SSTs are generated by flushing MemTables and placed in Level-0. Levels higher than Level-0 are generated through a process called compaction [24]. The size of the SSTs on each level is restricted by configurable parameters. If the target size for a specific level, such as Level-L, is exceeded, a subset of Level-L SSTs is chosen and combined with overlapping Level-(L + 1) SSTs. This compaction process eliminates deleted and overwritten data, optimizing the table for improved read performance and space efficiency. Consequently, the written data are gradually migrated from Level-0 to the highest level. The compaction I/O operations are efficient as they can be parallelized and involve bulk reads and writes of entire files. However, this process requires fetching multiple data from the disk and creating new SSTs, which are then written back to the disk. These operations are prone to errors caused by software and hardware failures.

2.3 Heterogeneous KV store Cluster

As the operational load of the system grows, vertically scaling a single node might not be enough to solve scalability problems. This necessitates the formation of a cluster of KV stores. Within this cluster, various KV store applications are installed on distinct nodes, allowing them to manage the increased loads together. These nodes interact with one another or with a dispatcher edge node to efficiently address queries. Privacy concerns also motivate the use of KV store clusters. When various applications or users need access to certain parts of the data or specific KV nodes, clustering offers essential isolation and management. Dividing the data among the clusters allows each application or user to reach their specific portion of the data, guaranteeing privacy and data separation.

KV store clusters frequently consist of heterogeneous clusters, characterized by varying hardware specifications of the deployment nodes, with each node potentially running a distinct version of the KV store software. The diversity in hardware and software versions complicates the process of updating the KV store. Updates to the KV store software must be compatible with the data of the current versions running on other nodes in the cluster. Incompatibility could lead to cluster failures, resulting in interruptions in data access and availability. Thus, when adding metadata per key-value in a KV store cluster, it is crucial to design the system so the data remains unobservable from other shards or nodes. This ensures both privacy and compatibility, facilitating efficient data management and access within the cluster while preserving the system's overall integrity and consistency.

3 EXISTING DATA INTEGRITY METHODS FOR KV STORES

In this section, we focus on maintaining the integrity of key-value data, using it as an example to demonstrate the integration of metadata for each KV store pair. To elucidate these solutions and make them more concrete, we illustrate them using RocksDB. Currently, RocksDB employs two main methods to verify the integrity of the data: (1) incorporating a checksum into the KV payload and (2) embedding a checksum within the metadata block. However, these methods face challenges related to heterogeneous clusters, privacy issues, and performance enhancement.

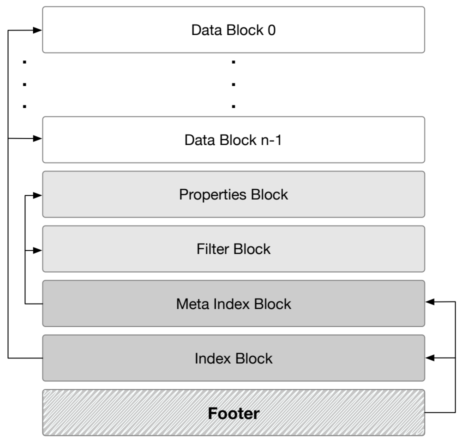


Figure 2: RocksDB block-based SST format

3.1 Adding checksum to the KV payload

A typical technique involves adding a checksum as metadata to the end of each KV value payload. Upon querying for data, the KV store strips the checksum and delivers the value to the user. Although this method appears straightforward and efficient for handling any metadata, it has some drawbacks.

First, it is incompatible with heterogeneous RocksDB clusters. This issue stems from changes in the data structure that other versions of RocksDB cannot interpret. If SST files are migrated from a RocksDB version with the KV checksum to an older version that does not recognize it, the older version will misinterpret the checksum as part of the value payload, leading to incorrect data interpretation. Furthermore, consider other types of key-value metadata, such as the statistics needed for ETL [12], such as the KV distance from the median. In these cases, retrieving the metadata value requires loading the entire value payload from storage to memory and processing it. RocksDB payloads can be as large as 3GB, but processing the actual value for these metadata types is unnecessary, causing avoidable overhead. Furthermore, this process alters the user value and requires enough information to determine the end of the value, which could pose security risks. Additionally, this method requires shared information on the extraction of value and metadata among all cluster nodes, which contradicts the principle of heterogeneity.

3.2 Adding checksum to the Footer Block

An alternative method to incorporate the checksum metadata while preserving the value payload is to append the checksum to the SST footer. Figure 2 illustrates a block-based SST structure. It comprises multiple data blocks containing key-value pairs, which are separated to enhance cache optimization for KV retrieval. The final

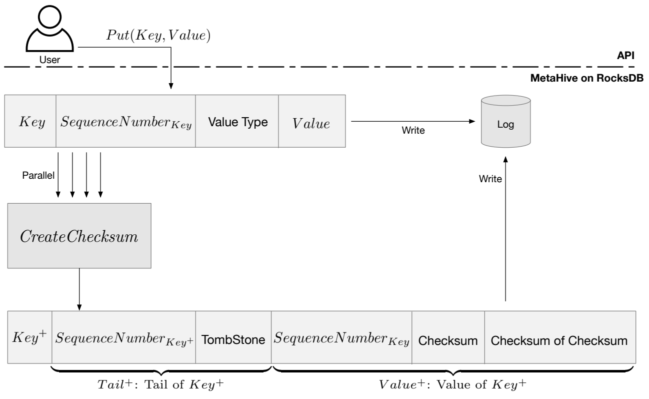


Figure 3: Inserting checksum metadata on PUT operation

block of each SST is a footer that can be extended to include various types of data. To ensure data integrity, the computed metadata (e.g., checksum) can be added to the footer block. Consequently, the key-value pairs remain unchanged, making this approach suitable for heterogeneous RocksDB clusters.

The issue with placing a checksum in the footer block is that it is not optimized for caching, as it disaggregates the metadata from the target KVs on different memory pages (see Section 2.1). Consequently, whenever we need to obtain KV metadata, such as checking its checksum, we have to access both the footer block and the data block, resulting in two memory page accesses. As the SST size increases, this problem is exacerbated, increasing processing costs. This method incurs a significant memory read overhead, as it requires loading two pages into memory for each KV.

4 METAHIVE DESIGN

In this section, we present MetaHive [2], *a cache-optimized method designed to store metadata in KV stores*, considering a heterogeneous KV store cluster. Although MetaHive is a versatile approach for managing metadata, we specifically apply it for data integrity and error detection of RocksDB data. We define error detection as identifying the discrepancies between the stored value and the original value [13], which involves storing checksum metadata per key-value.

4.1 Creating Checksum Metadata

We need to describe the key-value entry of RocksDB before explaining how we design MetaHive *to disaggregate metadata from their corresponding data*[5, 9, 15]. RocksDB appends a 56-bit integer *sequence number* to know the order of the entries with similar keys, and an 8-bit integer indicating the specific *operation type* of the entry (e.g., PUT, Delete, Merge). MetaHive uses the RocksDB entry design and creates metadata (Figure 3) as separate KVs to be stored close to the original data. We calculate the checksum of the corresponding KV and add it to the metadata payload to verify data integrity.

[2]A data integrity mechanism created following the MetaHive framework, with a patent application filed (U.S. Patent Serial No. 18/769665)

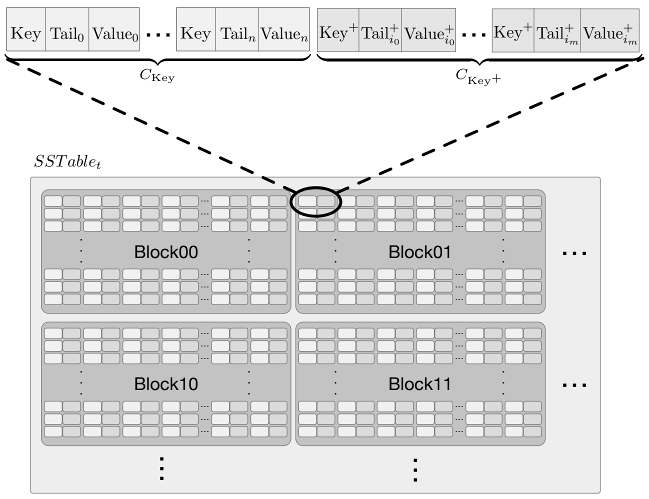


Figure 4: Clusters of KV and corresponding metadata

4.1.1 Metadata Key Generation. To ensure that the associated metadata is placed immediately after its KV in both the MemTable and the underlying SST files, we append a special character called the "Start of Heading" (SOH) symbol, represented by '\001', to the end of the original key. This special character acts as the metadata identifier and we impose the rule that standard keys do not terminate with this character. Additionally, it is generally known that many APIs and systems already reject such keys. Since the SOH symbol is the smallest character following the NUL symbol ('\000'), which is also prohibited at the end of strings on all systems, it ensures that the keys and their associated checksum metadata are placed consecutively. As illustrated in Figure 3, the metadata KV record's key is represented as Key^+, and is followed by a $Tail^+$ that includes the typical sequence number and value type for the metadata KV record.

4.1.2 Metadata Payload Generation. We calculate the hash values of the key, value, sequence number, and type individually. These values are then XORed together and added to the checksum payload. In addition, we compute the checksum of the checksum to verify the integrity of the checksum itself at a later stage. The final payload contains the sequence number of the original key, the checksum value, and the checksum of the checksum. We use XXhash3 as the hash function, which is a fast hash algorithm with processing at RAM speed limits. All these variables constitute the value component of the key-value pair, which includes the checksum information depicted in Figure 3 as $Value^+$.

4.2 KV and Metadata Clusters

The keys and their corresponding checksums can generate *two sequential clusters* as shown in Figure 4. When multiple updates are applied to the same key and these updates are stored in the same MemTable or SST file, a cluster of identical keys C_{Key} will be formed. When the checksum metadata for a KV pair, K^+V, is added to the MemTable for all KV records, the checksum pairs that share the duplicate key Key^+ form a cluster C_{Key^+} immediately following the cluster C_{Key}. In such cases, it is necessary to determine which checksum key corresponds to each original key. This is achieved by including the sequence number of the original key, which is a unique number, in the initial part of the checksum payload $Value^{+}$ (see Figure 3).

4.2.1 Cluster Analysis. The sequence of keys in C_{Key} does not correspond directly to the sequence of keys in $C_{Key^{+}}$. This is because some keys are deleted or filtered, but their metadata is in the SST, which we call *orphan metadata*. There might also be some keys that do not have any corresponding metadata (originally created from nodes without the MetaHive design). MetaHive uses a single-pass algorithm to correspond the KVs with their metadata (Section 4.4). MetaHive also removes orphan metadata to enhance storage efficiency, since the associated keys for these metadata have already been deleted.

4.2.2 Cache-Optimized Metadata Retrieval. MetaHive put the metadata entry in the vicinity of the actual entries by appending the specific character (Section 4.1.1). However, this does not guarantee that entries and their checksum are written in the same SST block to be efficiently retrieved in memory by reading one SST block file. RocksDB determines the maximum size of the block as a constant and dumps entries into the block files in the compaction process (see Section 2.2) whenever it reaches this capacity. This process may separate some entries with their corresponding metadata. We modify the RocksDB code to ensure that the actual KV entries and their corresponding checksum flush in the same block file during the compaction process. Therefore, reaching the metadata of an entry does not require reading two block files.

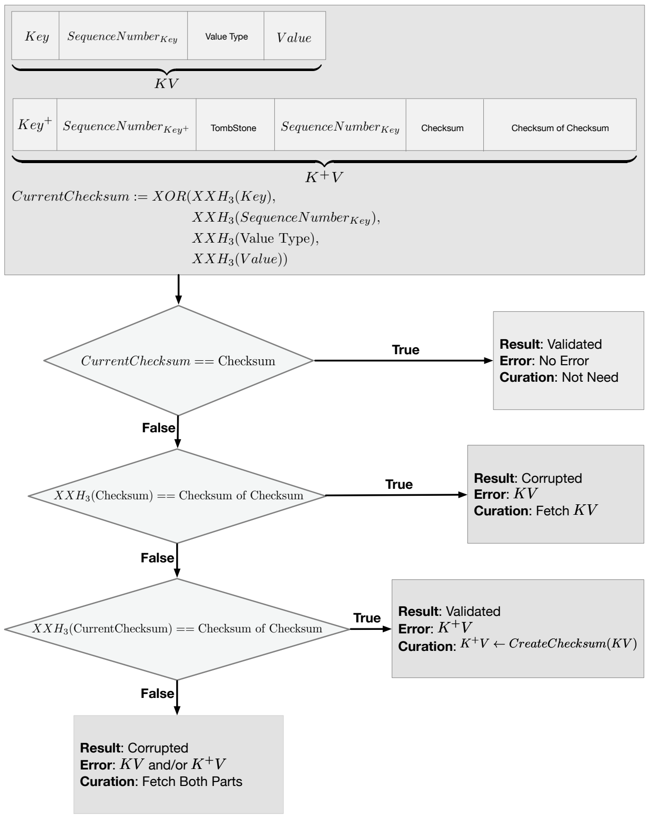
4.3 Metadata within a Heterogeneous Cluster

The significant goal of MetaHive is to achieve data integrity in a heterogeneous cluster (Section 2.3). In our scenario, each node might run a different version of the RocksDB code. Hence, it is vital to ensure that nodes without checksum metadata support are unable to detect it. It is important that the checksum metadata stays hidden, guaranteeing that (i) key iteration skips the checksum metadata, and (ii) automatic cleanup happens when SST files from the MetaHive version of RocksDB are transferred to an older version of RocksDB.
Tombstone to Skip and Auto-cleaning Metadata LSM-based KV stores, such as RocksDB, frequently employ a particular Tombstone type for key deletion. This type ensures that the keys marked with it are automatically purged from the lower SST levels, and their iterator bypasses these keys. Hence, using Tombstone, we fulfill the two objectives in the diverse cluster on the RocksDB nodes that do not include the MetaHive design. It is important to note that we cannot introduce a new type to RocksDB because it would not be recognized by older versions of RocksDB. MetaHive uses the following criteria to identify a checksum metadata entry:

$$is\_metadata = key.endsWith(\text{“}\backslash 001\text{”}) \land key.type == Tombstone$$

4.4 Data Integrity Check during Compaction

RocksDB is a write-intensive application designed for rapid data ingestion. To ensure consistency and fuse [14] duplicated data, it periodically performs a compaction process, selecting data from both memory and disk to create a new sorted structure. This process is prone to errors, as numerous KV reads and writes from memory and storage occur during this process. In this process, we first check the integrity of the data using the checksum metadata and then we use a repair mechanism for erroneous data.



Key
$SequenceNumber_{Key}$
Value Type
$Value$
KV
Key^{+}
$SequenceNumber_{Key^{+}}$
TombStone
$SequenceNumber_{Key}$
Checksum
Checksum of Checksum
$K^{+}V$
$CurrentChecksum := XOR(XXH_3(Key), XXH_3(SequenceNumber_{Key}), XXH_3(\text{Value Type}), XXH_3(Value))$
$CurrentChecksum ==$ Checksum
True
Result: Validated **Error**: No Error **Curation**: Not Need
False
$XXH_3(\text{Checksum}) ==$ Checksum of Checksum
True
Result: Corrupted **Error**: KV **Curation**: Fetch KV
False
$XXH_3(\text{CurrentChecksum}) ==$ Checksum of Checksum
True
Result: Validated **Error**: $K^{+}V$ **Curation**: $K^{+}V \leftarrow CreateChecksum(KV)$
False
Result: Corrupted **Error**: KV and/or $K^{+}V$ **Curation**: Fetch Both Parts

Figure 5: $DataIntegrity(KV, K^{+}V)$ **the MetaHive error detection module for data integrity procedure.**

4.4.1 Error Detection Module. We utilize a single-pass approach for error detection. For each key within the cluster C_{Key} and its associated metadata cluster $C_{Key^{+}}$, we first traverse C_{Key}, compute their checksums, and store them in a temporary map data structure using the key sequence number of the KV. Subsequently, we use the sequence number stored in $Value^{+}$ of each $K^{+}V$ in $C_{Key^{+}}$ to retrieve the corresponding key from the map.

Next, we initiate the verification process for the key and its associated checksum, as depicted in Figure 5. Initially, if the checksum of the current KV, $CurrentChecksum$, matches the checksum stored in $K^{+}V$, the process returns $Validated$ to indicate the correctness of KV. If not, it checks the data in $K^{+}V$ by computing the checksum of its checksum and comparing it with the corresponding part. A match implies that KV is erroneous, which prompts the module to return $Corrupted$. If there is no match, it recalculates the checksum of $CurrentChecksum$ and compares it with the checksum of the checksum in $K^{+}V$. If these are equal, it indicates that $K^{+}V$ is incorrect; Hence, it prevents the need to retrieve KV, which is generally quite large, and $K^{+}V$ can be recalculated using $CreateChecksum(.)$, returning a verdict $Validated$ for KV. Finally, if all comparisons fail,

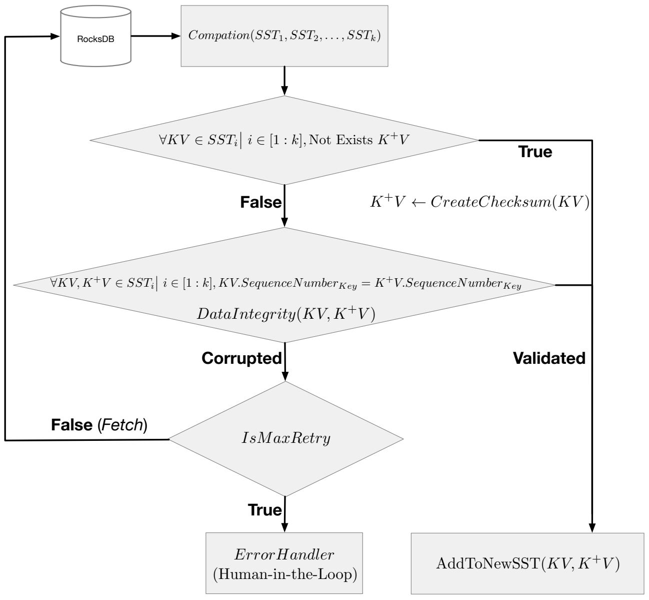

RocksDB
$Compation(SST_1, SST_2, \ldots, SST_k)$
$\forall KV \in SST_i \big| \ i \in [1:k]$, Not Exists K^+V
True
False
$K^+V \leftarrow CreateChecksum(KV)$
$\forall KV, K^+V \in SST_i \big| \ i \in [1:k], KV.SequenceNumber_{Key} = K^+V.SequenceNumber_{Key}$
$DataIntegrity(KV, K^+V)$
Corrupted
Validated
False (Fetch)
$IsMaxRetry$
True
$ErrorHandler$ (Human-in-the-Loop)
AddToNewSST(KV, K^+V)

Figure 6: The MetaHive design cycle within $Compation(.)$ and the repair process.

it indicates that both K^+V and KV are erroneous, and the process returns *Corrupted*.

4.4.2 Repair Module Strategy. This module uses the error detection module as a subprocess to identify erroneous data during the compaction process. As shown in Figure 6, for each key-value KV in the SSTs that undergo compaction, if the associated K^+V is located, it is forwarded to the error detection module (Figure 5); otherwise, it is added to the new SST. If the data are validated, no further action is required and KV and K^+V are added to the new SST. Otherwise, a maximum retry counter is set, and the system retrieves the data specified by the error detection module. If the error persists after retries, the *ErrorHandler* module is activated, responsible for deeper investigation. This module often involves human-in-the-loop, bringing in domain experts for further analysis. Given the heterogeneity of the cluster of nodes, there is no requirement to locate K^+V. If K^+V is not found, the *CreateChecksum* function is applied to KV, and both are incorporated into the SST.

5 EVALUATIONS

In this section, we first explain our experimental settings to evaluate MetaHive for metadata management. Then, we demonstrate its effectiveness in meeting the three metadata management objectives (defined in Section 1) for a heterogeneous cloud cluster.

5.1 Experimental Settings

5.1.1 Environment. We implement MetaHive on the RocksDB version 8.1.1. We perform our evaluation on an Ubuntu 20.04 Linux machine with AMD Ryzen ThreadRipper Pro 5995WX 64-core 2.7GHz CPU and 256GB DDR4 RAM.

5.1.2 Dataset and Workloads. We utilize YCSB [6] to produce key-value pairs and simulate workloads. The key and value sizes are configured to 20 and 100 bytes, respectively. Our evaluations are conducted on the following YCSB workloads using a zipfian distribution: *(1) Read-Only* with no PUT operations, *(2) Read-Heavy* with 5% PUT operations, and *(3) Update-Heavy* with 50% PUT operations. Each workload is executed for 2 million operations.

5.2 Runtime Analysis

5.2.1 Performance Overhead. Table 1 shows the performance results of different workload executions on RocksDB with and without MetaHive. The MetaHive version includes the insertion of checksum metadata and the execution of the error detection algorithm. We excluded 1% of outlier data and utilized the median times of PUT and GET operations to measure performance.

Our results show that MetaHive has a negligible impact on GET operations with less than 0.5% overhead on the latency. This is because MetaHive does not generate K^+V (Section 3.1), therefore, fetching a key-value does not have the overhead of getting the metadata payload. We ran the same experiments by adding metadata to the value payload. In this case, we observe a slower throughput of more than 10% through all three workloads.

MetaHive also has less than 0.8% impact on PUT operations. We migrate the addition of checksum metadata to when the SST is created from the immutable MemTables (Figure 1). This ensures that MetaHive does not need to locate the correct position of the keys for any new entries in the current MemTable. Writing the *Immutable Memtable* to new SST files occurs as a background process, which minimally affects users' PUT operations. Also, as the keys are flushed in a sorted manner and the checksum metadata is always placed after the keys, there is no overhead for looking up the metadata place.

5.2.2 Memory and Storage Overhead. MetaHive storage overhead is as follows: *(1)* the key length plus one special character to put the metadata right after the key in the SST files, and *(2)* eight bytes for the sequence number to find the corresponding entry in the metadata in clusters. The rest are the payloads that we need to put in the metadata. In our data integrity scenario, we add eight bytes of checksum and eight bytes of payload checksum to the metadata payload to check the data correctness.

MetaHive incurs minimal memory overhead during the compaction process, which is influenced by the number of keys within a cluster. By precalculating the checksum of each key and storing it in the map using its sequence number as the map's key (Section4.4), the memory overhead is 16 Bytes per key in the cluster ($16 \times |C_{key}|$). Consequently, even with a cluster containing 50K keys, the overhead still remains below 0.001%.

5.3 Error Detection in Heterogeneous KV Cluster

To evaluate the heterogeneity feature of MetaHive, we design an experiment with three RocksDB key-value nodes and one load-balancing node that distributes the data between RocksDB shards based on key prefixes. We added the MetaHive feature to one of the nodes, while two other KV nodes have the lower software version without MetaHive. We also embed a fault injection code into the MetaHive that modifies one bit of the checksum payload with a

Table 1: Comparison of performance results of RocksDB with and without MetaHive

Workload	PUT Operation (Nanosecond)			GET Operation (Nanosecond)		
	RocksDB	MetaHive	Difference (%)	RocksDB	MetaHive	Difference (%)
Read-Only (0% write rate)	-	-	-	172124	172829	**0.41%**
Read-Heavy (5% write rate)	83530	84001	**0.56%**	163417	164021	**0.37%**
Update-Heavy (50% write rate)	100184	100931	**0.74%**	159856	160543	**0.43%**

chance of 1%. We also periodically assigned different prefixes to each shard to migrate the data between shards.

This experiment shows us that data from the old nodes got metadata when they migrated to the MetaHive node, and no metadata was transferred to the old nodes. Even when migrating a full SST from MetaHive to the old nodes, metadata was removed automatically without users' notice since they have Tombstone type. In addition, we observe faulty data detected before sending it to the new node for the first time it is undergoing compaction.

6 CONCLUSION

We introduce MetaHive, a powerful method to manage and access metadata in key-value stores, considering heterogeneity and privacy. MetaHive is utilized to incorporate checksum metadata into key-value entries in RocksDB. Our findings indicate that MetaHive identifies corrupted data with negligible performance impact in a heterogeneous key-value store cluster.

REFERENCES

[1] 2011. LevelDB. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html.

[2] 2023. Cloud Computing Market Size, Share and Growth Report. https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry. Accessed: 2024-06-12.

[3] 2023. Global NoSQL Market Size, Share and Trends Analysis Report By Type, By Application (Web Apps, Data Analytics, Mobile Apps, Data Storage, and Others), By End User (IT, Retail, Gaming, and Others) By Regional Outlook and Forecast. https://www.kbvresearch.com/nosql-market/. Accessed: 2024-06-12.

[4] Yehuda Afek, Dave Dice, and Adam Morrison. 2011. Cache index-aware memory allocation. *ACM SIGPLAN Notices* 46, 11 (2011), 55–64.

[5] Fabrício B Carvalho, Ronaldo A Ferreira, Ítalo Cunha, Marcos AM Vieira, and Murali K Ramanathan. 2023. State disaggregation for dynamic scaling of network functions. *IEEE/ACM Transactions on Networking* 32, 1 (2023), 81–95.

[6] 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*. 143–154.

[7] 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)*. USENIX Association, 33–49. https://www.usenix.org/conference/fast21/presentation/dong

[8] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[9] Pavan Edara and Mosha Pasumansky. 2021. Big metadata: when metadata is big data. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3083–3095.

[10] Puya Ghazizadeh, Ravi Mukkamala, and Stephan Olariu. 2013. Data integrity evaluation in cloud database-as-a-service. In *2013 IEEE Ninth World Congress on Services*. IEEE, 280–285.

[11] Vishal Gour, SS Sarangdevot, Govind Singh Tanwar, and Anand Sharma. 2010. Improve performance of extract, transform and load (ETL) in data warehouse. *Int. Journal on Comp. Sci. and Eng* 2, 3 (2010), 786–789.

[12] Alireza Heidari, Shrinu Kushagra, and Ihab F Ilyas. 2020. On sampling from data with duplicate records. *arXiv preprint arXiv:2008.10549* (2020).

[13] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.

[14] Alireza Heidari, George Michalopoulos, Ihab F Ilyas, and Theodoros Rekatsinas. 2024. Record Fusion via Inference and Data Augmentation. *ACM/JMS Journal of Data Science* 1, 1 (2024), 1–23.

[15] Xin Jin, Zhihao Bai, Zhen Zhang, Yibo Zhu, Yinmin Zhong, and Xuanzhe Liu. 2024. : Efficient Resource Disaggregation for Deep Learning Workloads. *IEEE/ACM Transactions on Networking* (2024).

[16] Peter Membrey, Eelco Plugge, and Tim Hawkins. 2010. Definitive guide to mongodb. *Apress* (2010).

[17] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[18] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F Ilyas. 2023. Real-time LSM-trees for HTAP workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1208–1220.

[19] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[20] Alexander Suleykin and Peter Panfilov. 2020. Metadata-driven industrial-grade ETL system. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2433–2442.

[21] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.

[22] Idan Yaniv and Dan Tsafrir. 2016. Hash, don't cache (the page table). *ACM SIGMETRICS Performance Evaluation Review* 44, 1 (2016), 337–350.

[23] Çağkan Yapar, Kai Wan, Rafael F Schaefer, and Giuseppe Caire. 2019. On the optimality of D2D coded caching with uncoded cache placement and one-shot delivery. *IEEE Transactions on Communications* 67, 12 (2019), 8179–8192.

[24] 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. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[25] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2023. FoundationDB: A Distributed Key-Value Store. *Commun. ACM* 66, 6 (2023), 97–105.