electronics

MDPI

Review

Requirements and Trade-Offs of Compression Techniques in Key–Value Stores: A Survey

Charles Jaranilla and Jongmoo Choi *

Department of Software, Dankook University, Yongin 16890, Republic of Korea; cdjaranilla@dankook.ac.kr
* Correspondence: choijm@dankook.ac.kr; Tel.: +82-31-8005-3242

Abstract: The prevalence of big data has caused a notable surge in both the diversity and magnitude of data. Consequently, this has prompted the emergence and advancement of two distinct technologies: unstructured data management and data volume reduction. Key–value stores, such as Google's LevelDB and Meta's RocksDB, have emerged as a popular solution for managing unstructured data due to their ability to handle diverse data types with a simple key–value abstraction. Simultaneously, a multitude of data management tools have actively adopted compression techniques, such as Snappy and Zstd, to effectively reduce data volume. The objective of this study is to explore how these two technologies influence each other. For this purpose, we first examine a classification of compression techniques and discuss their strength and weakness, especially those adopted by modern key–value stores. We also investigate the internal structures and operations, such as batch writing and compaction, in order to grasp the characteristics of key–value stores. Then, we quantitatively evaluate the compression ratio and performance using RocksDB under diverse compression techniques, block sizes, value sizes, and workloads. Our evaluation shows that compression not only saves storage space but also decreases compaction overhead. It also reveals that compression techniques have their inherent trade-offs, meaning that some provide a better compression ratio, while others yield better compression performance. Based on our evaluation, a number of potential avenues for further research have been identified. These include the exploration of a compression-aware compaction mechanism, selective compression, and revisiting compression granularity.

Keywords: key–value store; compression; log-structured merge tree; evaluation

Citation: Jaranilla, C.; Choi, J. Requirements and Trade-Offs of Compression Techniques in Key–Value Stores: A Survey. *Electronics* **2023**, *12*, 4280. https://doi.org/10.3390/electronics12204280

Academic Editors: Juan M. Corchado, Byung-Gyu Kim, Carlos A. Iglesias, In Lee, Fuji Ren and Rashid Mehmood

Received: 28 August 2023
Revised: 3 October 2023
Accepted: 14 October 2023
Published: 16 October 2023

1. Introduction

Data compression is the technology of condensing information into a more concise format, hence minimizing the storage space, the amount of storage required to store, and the number of I/Os needed to access data [1–3]. One prominent advantage of data compression is the efficient utilization of storage space, resulting in a proportional expansion of the storage medium's capacity. Furthermore, it has the potential to enhance performance by reducing the volume of data that needs to be accessed and transmitted through the I/O system and interconnection network. Nevertheless, the process of compression and decompression necessitates supplementary CPU and memory resources, potentially leading to a decline in data access latency. Hence, it is imperative to thoroughly evaluate these trade-offs prior to implementing compression in data management systems.

A key–value store is a de facto standard database for unstructured big data [4,5]. Several data service providers have their own key–value stores, including Google's LevelDB [6], Meta's RocksDB [7], Amazon's Dynamo [8], and LinkedIn's Voldemort [9], among others. Due to their ability to accommodate many data formats and their support for a straightforward key–value abstraction, they are actively adopted for various services, including social graph analysis, AI/ML services, and distributed databases [10]. Key–value stores utilize a range of compression techniques [11–13], including Snappy [14], Zstandard (Zstd) [15], and LZ4 [16]. These approaches will be discussed further in Table 1.

Table 1. Compression techniques adopted by modern key–value stores.

	Brotli	Gzip	LZ4 [1]	LZMA	LZO	Snappy	Zlib	Zstd
LevelDB [6]						✓		✓
RocksDB [7]			✓			✓	✓	✓
Cassandra [17]			✓			✓	✓	✓
WiredTiger [18]						✓	✓	✓
HBase [19]	✓	✓	✓	✓	✓	✓		✓

[1] Implementations of LZ4 also include LZ4HC (LZ4 High-Compression version).

The fundamental data structure used by most key–value stores is the log-structured merge tree (hereafter, LSM-tree) [20], which has distinct characteristics suitable for compression. Initially, the system records key–value pairs in a log-structured manner. More precisely, it manipulates two fundamental components, namely *Memtable*, which is a memory component, and *SSTable*, which is a storage component. The concept of a Memtable refers to a specific type of write buffer that serves as a cache for key–value pairs. These pairs are then written to storage in a batch style, with a default size of 64 MB. The utilization of bigger block sizes in compression techniques can be leveraged to improve the compression ratio in the context of this expansive batch writing. Nevertheless, the increased block size has both advantages and disadvantages, with one of the drawbacks being a detrimental effect on read latency. The consideration of this trade-off necessitates careful consideration when employing compression in a key–value store.

Another notable characteristic of the LSM-tree is its utilization of the out-of-place update method, wherein updated data is written to a new location while simultaneously rendering the old data in the original location invalid. Consequently, in order to reclaim the invalidated location, it introduces an internal operation called *compaction*. SSTable files are structured hierarchically, encompassing numerous levels. A Memtable is flushed into storage, transforming into an SSTable file located at level 0. Compaction is triggered when the size of a given level exceeds a predetermined threshold. It consists of three steps: (1) read existing SSTable files from the level and one higher level; (2) conduct merge-sort while removing the outdated key–value pairs; and (3) write new SSTable files into the higher level and delete the compacted files. This implies that compression and decompression operations are frequent in a key–value store, occurring not just during data retrieval but also during the compaction process. Hence, compression time becomes more critical than compression ratio, especially for SSTable files at lower levels, which will be compacted and deleted in the near future.

In this paper, we explore how these characteristics, batch writing and compaction, affect the compression ratio and how the compression ratio impacts the performance of key–value stores. We first survey various compression techniques used by key–value stores and investigate how each technique works, along with its strengths and weaknesses. Then, we analyze the internal structure and operations of key–value stores in order to identify primary factors, such as block size, value size, and workload, that influence compression behaviors significantly. Finally, we evaluate the impact of compression on the performance of key–value stores, including storage usage, compaction count, throughput, and access latency. In the evaluation process, RocksDB, a widely recognized key–value store, is tested with three distinct workloads. The first workload is a db_bench [7] workload that utilizes synthetic data, while the remaining two workloads utilize real-world data sourced from Amazon [21] and Twitter [22].

To the best of our knowledge, this is the first survey paper that quantitatively investigates various compression techniques utilized by modern key–value stores. We provide a taxonomy of compression techniques and discuss their opportunities and requirements with the consideration of the LSM-tree characteristics. Then, we evaluate the compression ratio and performance under diverse compression techniques, block sizes, value sizes, and both synthetic and real-world workloads. Our purpose encompasses not only evaluating performance but also analyzing the inherent trade-offs linked to compression techniques

and identifying potential areas for enhancement. Our findings lay the groundwork for future research endeavors focused on fully leveraging the capabilities of compression in key–value stores. We presents several lessons, insights, and recommendations for improving compression efficiency and raising the overall performance of key–value stores, thereby paving the way for novel breakthroughs in this field.

The remainder of this paper is organized as follows: Section 2 explains data compression in general and examines compression techniques used by key–value stores. In Section 3, we describe the characteristics and factors of key–value stores that have an influential impact on the compression ratio and performance, as well as the literature related to our work. Then, we present the evaluation results and observations in Section 4. Section 5 discusses lessons we have learned and future research directions. Finally, we conclude this paper in Section 6.

2. Background

In this section, we first describe two representative approaches to compression, lossy and lossless, and further classify the lossless approach into four categories. Then, we discuss each technique in detail, including its strengths and weaknesses.

2.1. Compression in General

The earliest evolution of data compression can be associated with Morse code [1]. It was introduced by Samuel Morse in 1838 to compress letters in telegraphs. In earlier days, when messages were sent by telegraph, letters were encoded with dots and dashes. Morse noticed that certain letters occurred more often than others. In order to reduce the average time spent sending messages, shorter and longer sequences were assigned to more frequent and less frequent letters, respectively. This idea of using shorter codes for more frequently occurring characters is also used in Huffman coding [23].

Through the years, compression has been continually developed with consideration for advanced technology and applications. Nevertheless, its main purpose stays the same: reducing storage space and the number of I/Os by removing redundancies in the given data. A lot of compression techniques exist, and they can be broadly divided into lossy and lossless techniques, as shown in Figure 1. In the figure, we intentionally highlight the compression techniques that are popularly used by key–value stores.

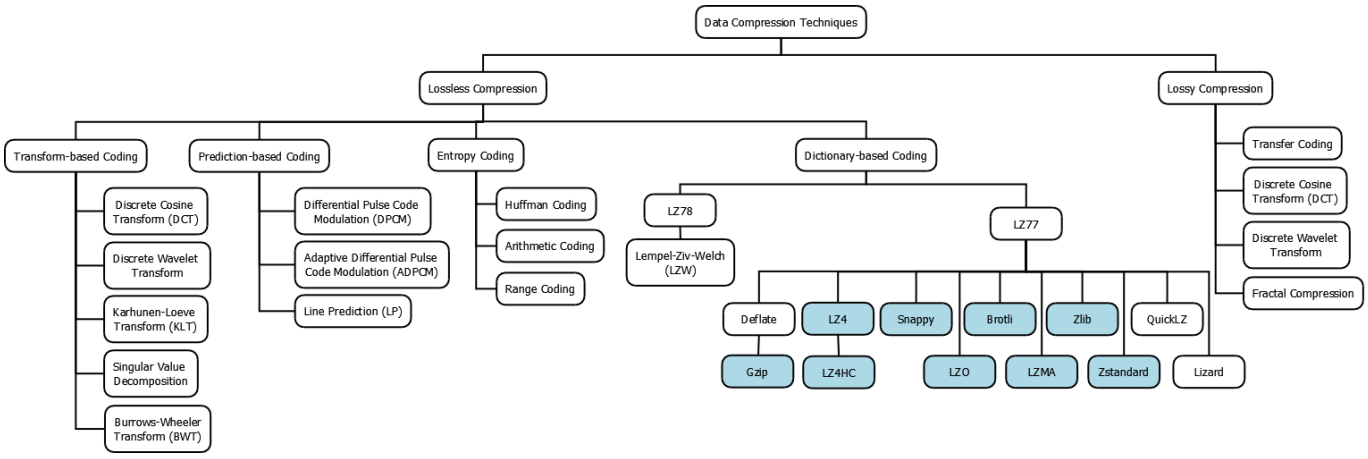


Figure 1. Taxonomy of compression techniques. They can be classified into lossy and lossless approaches, and then classified further based on their mechanisms. Note that compression techniques actively used by modern key–value stores are highlighted in blue.

Lossy compression is used for data that can afford to lose some parts. It removes some parts of data irreversibly, resulting in the original data not being reconstructed upon decompression. This approximate reconstruction can be desirable as it leads to better performance and resource utilization. Lossy compression is commonly applied to videos,

images, and audio data sources where humans perceive data as complete even though some portions have been discarded by compression. Commonly used lossy compression techniques include transfer coding, discrete cosine transform, discrete wavelet transform, and fractal compression [1,2].

Sensitive information has to be processed with lossless compression, where no data is lost in the compression process and the original data can be fully reconstructed after decompression [24]. In text compression, omitting or changing a single character results in a non-negligible impact on the intended meaning, so lossless compression should be used. Similarly, data sources containing sensitive information, such as medical images and data intended for legal purposes, should be compressed in a lossless manner. Key–value stores use lossless compression since they manipulate diverse types of sensitive data.

Lossless compression can be categorized into four distinct classes, namely transform-based coding, prediction-based coding, entropy coding, and dictionary-based coding [2,24]. Transform-based coding is a type of data compression technique wherein input data are transformed into the frequency or wavelet domain, preserving the most important and visually significant coefficients while discarding less significant ones to achieve compression. Prediction-based coding takes advantage of statistical regularities through the analysis of correlations between adjacent symbols or samples, which are used to predict the value of the next symbol or sample. Entropy coding is a technique that exploits the statistical properties of the input data and attempts to minimize the number of bits for data representation by assigning shorter codes to more frequently occurring symbols. Finally, dictionary-based coding compresses data by encoding pre-existing data patterns using a dictionary and replacing these patterns with corresponding entries in the dictionary.

It should be noted that this kind of classification is inherently complex due to the fact that some techniques take both lossy and lossless approaches. For example, the discrete cosine transform can be used for either lossy or lossless data, as depicted in Figure 1, by governing a threshold when it transforms original data into the frequency domain. Furthermore, some lossless techniques make use of a hybrid mechanism derived from two or more categories. A typical example is Zstd, which, even though it is commonly categorized as dictionary-based coding, provides compression capabilities based on both a dictionary and an entropy by combining both at different stages. Certain techniques utilize a hierarchical mechanism, applying transform-based coding at the first level and then applying Huffman coding to the transformed data to compress it further. Nevertheless, we make this classification, since it gives us a global viewpoint, allowing us to discuss each compression technique in a systematic way.

Now, let us compare four lossless categories further. Entropy coding is known to be effective at compressing data that contains a wide range of symbols with varying probabilities of occurrence. On the other hand, prediction-based coding is more effective at compressing data that contain patterns or redundancies. Transform-based coding is a highly effective technique for compressing data that exhibit spatial or frequency correlations, such as images, audio, and video. However, transform-based coding requires more computation and memory than entropy coding or prediction-based coding. Dictionary-based coding, compared to other categories, has the lowest compression overhead, hence demanding less memory and computing resources. This is why most key–value stores choose techniques based on dictionary-based coding.

Two widely used implementations of dictionary-based coding are LZ77 and LZ78 [25]. The distinction between these techniques lies in how they build and use dictionaries. LZ77 employs a sliding window mechanism to find recurring patterns in the input data and replace them with references to previously seen patterns stored in a dictionary. On the contrary, LZ78 builds a complete dictionary of all potential patterns in the input data. Therefore, LZ77 is faster and more memory-efficient than LZ78, but yields a relatively lower compression ratio. Since fast compression is indispensable in key–value stores, they prefer to utilize LZ77-based techniques, as highlighted in Figure 1. In the next subsection, we will discuss these techniques in detail.

2.2. Compression Techniques Used by Key–Value Stores

Table 1 summarizes the compression techniques employed by various key–value stores. LevelDB originally supported Snappy [14] only due to its simplicity, but it has added Zstd [15] since version 1.23. RocksDB, the successor of LevelDB, supports not only Snappy and Zstd but also LZ4 [16] and Zlib [26]. Both Cassandra and WiredTiger support similar techniques as RocksDB. The recent version of HBase equips users with more techniques, including Brotli [27], Gzip [28], LZMA [29], and Lempel–Ziv–Oberhumer (LZO) [30]. From now on, we will examine how each technique works, taking into consideration their strengths and weaknesses.

2.2.1. Snappy

Mechanism. Snappy [14], previously known as Zippy, is a compression/decompression library developed and open-sourced by Google. It aims for high speed and a reasonable compression ratio. It is based on LZ77, categorized as dictionary-based coding, but also makes use of Huffman coding. In addition, Snappy can include a checksum at the end of each compressed block to ensure data integrity.

Strength. In terms of speed, Snappy is fast compared to other compression techniques, which makes it ideal for use in applications that require quick data access even with compression and decompression. It provides a reasonable compression ratio, particularly for smaller files. Moreover, Snappy is designed to use less CPU and memory resources during both compression and decompression.

Weakness. Snappy's limited compression ratio, particularly for larger files, can be a disadvantage. However, since key–value stores compress data per block, this disadvantage is not that serious. Another weakness is that it does not support all file formats, which limits its usefulness in certain applications.

2.2.2. Zstd

Mechanism. Zstd is a popular dictionary-based compression technique developed by Meta [15]. It uses a sliding window of up to 128KB as a dictionary. Furthermore, it supports dictionary compression using a variant of finite state entropy and Huffman coding. It offers 22 levels of compression (e.g., 1 being the fastest but lowest compression ratio), providing a knob to balance the compression ratio and resource utilization.

Strength. Zstd provides an excellent compression ratio, particularly for text data. This is advantageous in scenarios where storage space is a prioritized concern. For interoperability, it supports multiple programming languages, including Java, Python, Rust, Ruby, and so on.

Weakness. One potential drawback of Zstd is its significant computational overhead, which necessitates substantial processing capabilities in order to achieve a desirable compression ratio. This could be an concern, especially in resource-constrained environments.

2.2.3. LZ4 and LZ4HC

Mechanism. LZ4 is a dictionary-based coding technique that works by identifying repeated patterns in data and replacing them with a shorter reference to the previous occurrence of the same pattern [16]. With LZ4, speed can be tuned by selecting an acceleration factor that trades compression ratio for faster speed. It supports a derivative called LZ4HC (High Compression), which trades CPU time for an improved compression ratio.

Strength. LZ4 is designed to use minimal CPU and memory resources, making it an efficient choice for resource-constrained environments. In addition, it provides fast compression and decompression speeds, which are well-suited for applications that have a fast or real-time data transfer requirement.

Weakness. LZ4 may not be as effective compared to other techniques when compressing certain types of data, such as images or videos, where the data do not have enough repeating patterns. Another weakness is its lack of error detection and correction mechanisms, such as cyclic redundancy checks, which are found in most compression techniques.

2.2.4. Gzip, Zlib, and Brotli

Mechanism. Both Gzip [28] and Zlib [26] are based on the Deflate compression algorithm [31], which uses a combination of LZ77 and Huffman coding. They differ in file formats, dictionary sizes, and usage contexts. Brotli [27] is a technique designed for superior compression ratio and faster decompression time compared to Gzip. It utilizes a dictionary and context model to exploit data patterns for better compression.

Strength. Gzip and Zlib support a good compression ratio. Gzip works well with most data types. It is particularly useful for applications where memory usage is an issue, as it has a relatively low memory overhead. Zlib provides additional features such as data integrity checks and sliding window processing. Brotli offers a higher compression ratio and faster decompression speed for text data.

Weakness. Both Brotli and Zlib incur high computational costs for compression and decompression. The distribution of Brotli is also limited, which can be a portability concern. For larger files, Gzip's compression and decompression speeds can be an issue.

2.2.5. LZMA

Mechanism. LZMA stands for Lempel–Ziv Markov chain Algorithm [29]. It is a compression technique that combines both statistical modeling and dictionary mechanisms. The statistical modeling allows for the examination of compressible patterns across entire datasets, whereas the dictionary enables the compression of small pieces simultaneously. Its recent version, LZMA2, supports multithreaded compression and decompression for scalability.

Strength. LZMA achieves a high compression ratio, which is great for applications that need to utilize available storage space efficiently for storing large amounts of data. Another important aspect is LZMA's support for scalability.

Weakness. To obtain a high compression ratio, LZMA is computationally expensive, which leads to a large computational overhead. It also needs a significant amount of memory to conduct the statistical modeling during compression.

2.2.6. LZO

Mechanism. LZO is a dictionary-based compression technique that is geared towards fast compression and decompression speed through simple and efficient algorithms that can be implemented in both hardware and software [30]. It also uses a mechanism called run-length encoding to compress repeated byte sequences, which replaces repeated bytes with a count and a single copy of the byte. This can be particularly effective for compressing data with long sequences of repeated bytes, such as zeros.

Strength. LZO provides excellent compression and decompression speeds. In addition, it can achieve a good compression ratio for certain types of data, such as executable files and data with long sequences of repeated bytes. Moreover, LZO requires a relatively small amount of memory.

Weakness. LZO is not suitable for compressing certain types of data, particularly data with a low degree of repetition. It also has limited support for compression concurrency. Since key–value stores need to handle multiple requests simultaneously, LZO may not be as effective as handling concurrent requests.

2.3. Literature Review

This subsection explains prior research related to compression. There exist studies that are specifically centered around data, such as examining the compression efficiency of text data [32], image data [33,34], and sensor data [35]. On the other hand, several studies concentrate on particular research domains, such as hardware acceleration [36,37], architectural approaches [38], and database design [39]. For instance, Kimura et al. examine how compression affects choices of appropriate database design, such as indexes [39]. Nevertheless, they focus on traditional relational databases, rather than key–value stores focused on in this paper.

Note that databases can be classified into two categories: traditional relational databases and NoSQL databases. NoSQL databases can be further classified into key–value stores, column databases, document databases, and graph databases. Among NoSQL databases, compression is actively studied in column databases [40–43]. However, the internal structure of column databases is different from that of key–value stores. They manage data based on columns instead of rows, and they do not have the compaction operation. Hence, the effect of compression is also different between column databases and key–value stores.

In the area of key–value stores, a number of prior studies have endeavored to either incorporate or enhance compression techniques. Dong et al. employ compression to reduce the space amplification in RocksDB [11]. In addition, they devised a mechanism known as dynamic level-size adaptation and compression, which serves to achieve a balance between space and read/write amplification. Kim and Vitter suggest a technique that integrates compression and encryption in a persistent key–value store for high-performance computing systems [12]. Their technique exploits deep memory hierarchy to achieve both data reduction and performance enhancement.

Ma et al. present several techniques to integrate data compression into in-memory key–value stores [44]. They reduce compression-induced read and write amplification, thereby mitigating compression-induced performance degradation. In order to further reduce write amplification, they propose to incorporate delta encoding within a background re-compression process. Zhang et al. develop a storage engine called CompressDB [45]. Since it is integrated directly into file systems, it is able to support various database systems, one of which is LevelDB. In the context of LSM-tree, since LevelDB uses LSM-tree as its core architecture, CompressDB, paired with LevelDB's default compression technique Snappy, is able to further improve performance as well as space savings.

Guler and Ozkasap propose a periodic incremental checkpoint algorithm with compression to address the communication costs of a backup replication protocol [46]. They set up replicated key–value stores on distributed nodes and develop compressed incremental checkpointing to support primary backup replication. In their setup, they use various performance metrics, which include compression ratio and compression/decompression times, among others. They conclude that Zstd is the most competent compression method under all the scenarios in their experiments.

Jia et al. propose a scheme called SlimCache that enhances the hit ratio in key–value caching [47]. By using dynamic online compression, it expands the usable cache space, which in turn increases the cache hit ratio and improves cache performance. Jin et al. identify that the write and space amplification caused by key–value stores can deteriorate the lifespan of flash-based storage devices [48]. To overcome this problem, they design a knob that directly controls how data is compressed in RocksDB.

Tkachenko analyzes various database compression techniques [49]. The analysis makes use of an open-source compression benchmark called lzbench [50] in evaluating techniques such as Zstd, Snappy, LZ4, Zlib, LZMA, Brotli, and QuickLZ. It shows several results, including compression ratio, speed, and the influence of different block sizes. However, a notable distinction between their work and ours lies in the absence of any findings pertaining to key–value stores, whereas this study delves into the intricate interplay between compression techniques and the internal processes of key–value stores.

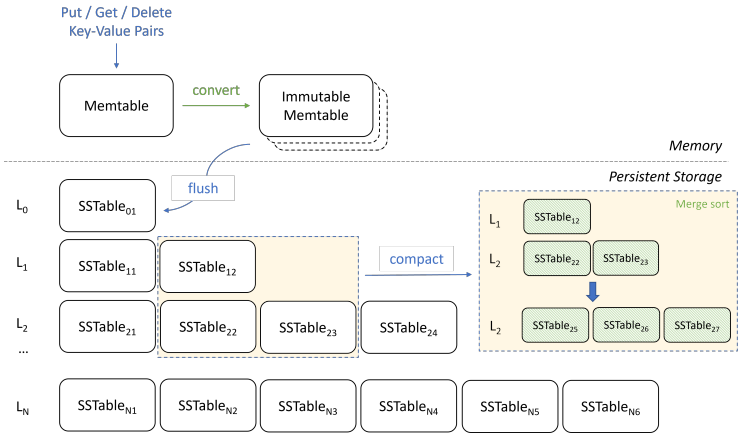
To the best of our knowledge, this is the first paper that explores various compression techniques using real evaluation, and analyzes how the characteristics of LSM-tree and compression influence each other.

3. Issues of Compression in Key–Value Stores

In this section, we explain the internal structures and operations of an LSM-tree-based key–value store. Then, we discuss how compression and key–value stores influence each other. Finally, we identify the primary factors that affect compression ratio and performance.

3.1. Internal Structure and Operations of Key–Value Stores

Figure 2 presents the internals of an LSM-tree-based key–value store. It stores and retrieves data using a set of key–value interfaces such as put(), get(), and delete(). It consists of two components: the Memtable in memory and the SSTable in storage. The Memtable is an in-memory data structure holding data before it is flushed to storage. Once a Memtable is full, it becomes immutable and replaced by a new Memtable. A background thread will flush the content of the immutable Memtable into an SSTable file, after which the Memtable is set to be removed. The SSTable is a collection of files that are organized in multiple levels, from L_0 to L_N in this figure, where the default value of N is six in RocksDB.



(**a**) LSM-tree data structure

(**b**) SSTable layout

Figure 2. Internals of an LSM-tree based key–value store and SSTable layout. This image is a modified version of the one presented by Zhang et al. [51].

LSM-tree is a write-optimized data structure. The basic idea of LSM-tree is writing data in a log-structured manner and conducting out-of-place updates instead of in-place updates. Specifically, it keeps incoming key–value pairs in Memtable. When the size of Memtable is beyond a threshold (e.g., 64 MB), the Memtable becomes immutable and is eventually written to storage. This operation is called *flush*. One characteristic of LSM-tree is to buffer incoming writes in memory using Memtable, and flush it to storage as a new SSTable file in a batch style. Note that when a key is updated with a new value, a new key–value pair is written into a new SSTable file, while the outdated pair is also maintained in an existing SSTable file.

SSTable files are then merge-sorted on a regular basis using an internal operation called *compaction*. The compaction operation is illustrated in a box shown on the right side of Figure 2a. When the number of SSTable files at level L_i is beyond a threshold, compaction is triggered. It first selects a file at level L_i according to a selection policy (e.g., the oldest file or the largest file). Then, it reads all files, from both level L_i and level L_{i+1}, that have overlapped key ranges with the selected file. Then, it performs merge-sort while deleting outdated key–value pairs. Finally, it writes new SSTable files at level L_{i+1}. After compaction, old SSTable files are removed and their space is reclaimed.

When a lookup request arrives, it first checks the Memtables (both mutable and immutable) and serves it if the requested key–value pair exists there. Otherwise, it checks SSTable files, from the lowest level (L_0) to the higher levels, to identify a file whose key range contains the requested key. Note that the lower level always has the most up-to-date pair in the LSM-tree. Finally, it reads the data block from the identified file with the help of

an index block stored at the end of the file that maps keys into data blocks. Note that, since the SSTable file is identified using key ranges, it does not actually contain the requested key, which causes an unnecessary read. To avoid this, key–value stores make use of the bloomfilter to test the non-existence of a key in a file.

The put interface is used for either adding a new key–value pair or updating an existing pair. Since a key–value pair is first managed in Memtable and later flushed into a SSTable file, a sudden power failure may cause data loss. To overcome this problem, most key–value stores employ a write-ahead log (WAL) on storage in order to support durability. The get interface is used for a lookup, checking the Memtables first and then SSTable files from Level 0 to the higher levels until the key is either found or determined to be absent. Finally, the delete interface is used for removing an existing pair. A specific tombstone marker is inserted into Memtable and written to the WAL during a deletion, indicating that the key should be deemed deleted.

Updating data in a log-structured manner in key–value stores can enhance write performance by converting random writes into sequential ones. However, it increases the write amplification due to compaction. Furthermore, it degrades storage utilization by storing not only up-to-date data but also old data until compaction. Compression can be a practical solution to overcome these issues. In the next subsection, we will discuss how compression is integrated into key–value stores.

3.2. Compression in Key–Value Stores

Data compression is a powerful tool with numerous advantages that can be used for a wide range of applications. Saving storage space is the most important benefit that can be expected from compression. Another advantage is the reduced number of I/Os, which is especially beneficial for key–value stores when we consider compaction. However, compression has its own drawbacks. We need to consider the compression overhead and increased CPU usage. Compressed data also result in increased latency when data need to be decompressed before it can be accessed, which may also increase the memory footprint.

In the context of key–value stores, we need to contemplate two characteristics driven by the LSM-tree. First is the bulky batch writing. Incoming key–value pairs are buffered in Memtable, whose default size is 64 MB, and written into storage in batch style. It gives the encoder a chance to find a lot of repeated patterns, providing a positive impact on compression. The second characteristic is that, in key–value stores, compression and decompression occur not only during data access time but also during flush and compaction time. It implies that key–value stores carry out compression and decompression more frequently than other databases.

As a result, this high frequency leads to compression speed being considered as a top priority. This is why most key–value stores choose dictionary-based coding among the four categories shown in Figure 1, since it has the lowest compression overhead. In addition, key–value stores apply compression based on a data block instead of a SSTable file, as illustrated in Figure 2b. This block-based compression can provide better read latency since, when a lookup is requested, it requires access to only the related block. On the contrary, file-based compression may degrade read latency to access whole blocks for decompression, whereas it can yield a better compression ratio.

In key–value stores, compression and decompression are conducted at three main points: flush, compaction, and block read. For instance, when we enable compression, compaction procedures can be divided into five steps: (1) read, (2) decompress, (3) merge–sort, (4) compress, and (5) write [51]. During compression, a block is fed into an interface where a repeated pattern is replaced with a reference to the previous offset and length of the pattern. During decompression, the reference is converted into a pattern, recreating the original data. Note that, in the current implementation, decompression and merge–sort are performed separately, meaning that the merge–sort step starts after completing decompression.

3.3. Key Factors That Affect Compression in Key–Value Stores

There are a variety of factors in key–value stores that influence compression ratio and performance significantly. In this subsection, we describe the five most influential factors.

First, the choice of compression technique can significantly affect the overall behavior of key–value stores. Techniques such as Snappy and LZ4 are designed to provide fast compression and decompression with low CPU usage, making them ideal for systems that require high read and write throughput. However, these techniques do not necessarily provide a superior compression ratio. On the other hand, techniques such as Zstd or LZ4HC can provide a higher compression ratio, but compression and decompression can take longer and require more CPU.

Second, the size of blocks is another influential factor that affects the compression ratio and throughput of key–value stores. For dictionary-based compression techniques, having a bigger block size positively impacts the compression ratio since more data can be fed through compression. Within a larger pool of data, more data patterns can potentially be identified as repetitive, which improves the compression ratio. However, larger blocks require more memory and increase read latency for decompression.

The third factor is workload. We recognize two features of a workload that determine the effectiveness of compression in key–value stores, one of which is compressibility [52]. If the data has high randomness and does not have enough repeated patterns, compression may not be beneficial. The other feature is the lifetime. In key–value stores, SSTable files in lower levels, especially level 0, will be compacted and deleted in the near future. In contrast, SSTable files at higher levels have a long lifetime. The compression ratio is more critical for long-term data, while compression speed is more important for short-term data.

Fourth, system resources such as CPU capability and storage type are important factors. Note that compaction itself is a CPU-intensive job, consuming a large portion of CPU utilization. Therefore, it is better to use a lightweight compression technique unless there are abundant CPU resources. For slow storage such as HDDs (hard disk drives), the latency for compression is relatively small compared to the I/O latency, meaning that it is possible to employ more powerful techniques to enhance the compression ratio. On the contrary, for fast storage such as NVMe SSDs (solid state drives), employing more powerful techniques could increase the overall data access latency considerably.

Finally, key–value store configurations are also crucial factors. For instance, key–value stores use two different compaction mechanisms: leveled and tiered compaction [10]. The leveled mechanism applies compaction aggressively, obtaining better storage utilization at the cost of increased compaction counts, while the tiered mechanism applies compaction in a lazy manner, reducing compaction counts at the cost of worse space utilization. Hence, when we select a compression technique, we need to consider this trait. As another example, the block cache of key–value stores maintains uncompressed data in memory, which can be used effectively to hide the decompression overhead.

4. Analysis

This section presents our experimental results with one of the most popular key–value stores, RocksDB. Table 2 summarizes our experimental environment. The hardware consists of an Intel i7 processor with 16 cores, 16 GB of DRAM, and 1 TB of NVMe SSD. In this hardware, we install Ubuntu 20.04.4 LTS with Linux kernel version 5.4 and use RocksDB version 7.7.3. We chose RocksDB for our experiments since it is a well-known key–value store utilized not only in industry [7] but also for research purposes [48,53]. It is derived from LevelDB, having the same LSM-tree based internal structure, but supporting more diverse compression techniques.

Table 2. Experimental system.

CPU	Intel(R) Core(TM) i7-10700K CPU @ 3.80 GHz
Memory	16,384 MB
Storage	1 TB Samsung V-NAND NVMe M.2 SSD 970 PRO
OS	Ubuntu 20.04.4 LTS (Focal Fossa)
Key-value store	RocksDB 7.7.3
Workloads	Synthetic (from db_bench), Amazon data (from [21]), Twitter data (from [22])

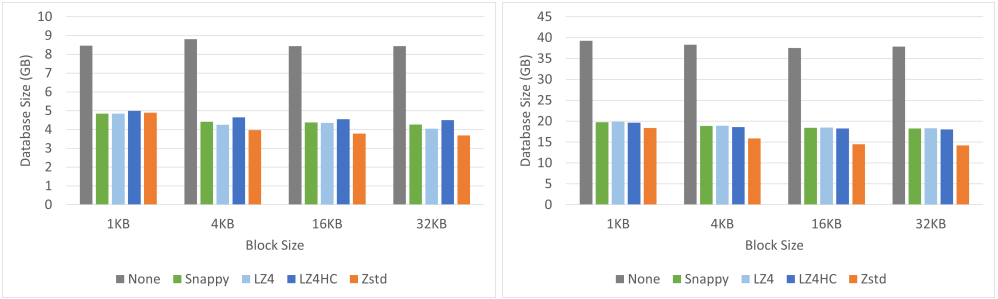
RocksDB has a micro-benchmark tool called db_bench. It provides a facility to evaluate the overall performance of RocksDB through various benchmarking options with synthetic workloads. For our experiments, we insert 100 million key–value pairs. We specify 16 Bytes for the key size and 100 and 500 Bytes for the value sizes, respectively. While db_bench is able to evaluate RocksDB's overall performance, it uses synthetic data by randomly generating both keys and values. For evaluating real-world data, we use two additional workloads: Amazon Review data [21] and Twitter Trace data [22].

The Amazon Review data comprises a comprehensive compilation of 233.1 million reviews, product details, and associated links. The data was gathered during the months of May 1996 and October 2018, resulting in a total size of 34 GB. On the other hand, the Twitter data is a collection of key–value traces reflecting the activity directed to key–value stores from Twitter's in-memory caching clusters. These data was obtained from a total of 54 distinct clusters during the month of March in the year 2020, over a period of one week. In our experiments, a subset of each data, approximately 10 GB, is utilized for evaluating the compression capability of real-world data.

4.1. Compression Ratio

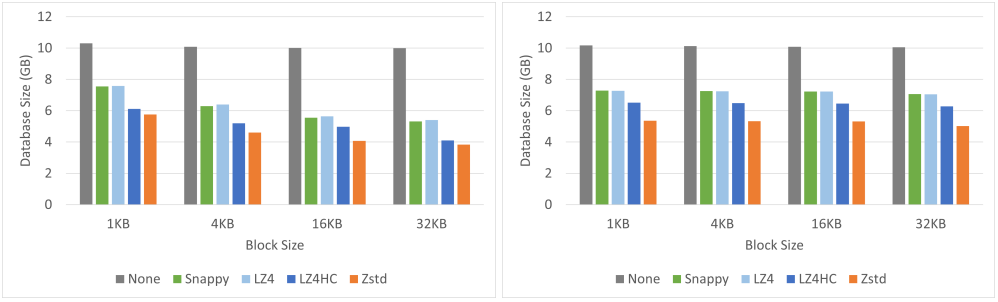
Figure 3 presents our first experimental result, the compression ratio. As we have discussed in Section 3.3, the compression ratio depends on various factors, including compression technique, block/value size, workload, and so on. In this experiment, we used four compression techniques, namely Snappy, LZ4, LZ4HC, and Zstd. Also, we set up four block sizes, 1 KB, 4 KB, 16 KB, and 32 KB, and two value sizes, 100 and 500 bytes. We chose these four compression techniques since they are the most popularly used ones, as shown in Table 1. Note that the default block size is 4 KB and we select 1 KB, 16 KB, and 32 KB to assess the effect of smaller and larger block size. Also note that the average value size is reported as 126 bytes in Meta's workload analysis [10]. Finally, we use three workloads: db_bench, Amazon, and Twitter.

From Figure 3, we can make the following observations:



(**a**) db_bench (value size=100 Bytes) (**b**) db_bench (value size=500 Bytes)

Figure 3. *Cont.*



(**c**) Amazon (**d**) Twitter

Figure 3. Impact of compression techniques, block sizes, and value sizes on compression ratio under synthetic (db_bench (**a**,**b**)) and real workloads (Amazon (**c**) and Twitter (**d**)).

Effect of compression techniques: Compression can reduce storage usage, and it is governed by compression techniques. In this paper, we use the space-saving rate as a metric for comparison, which is defined as Equation (1).

$$(1 - \frac{compressed\ size}{uncompressed\ size}) \times 100 \quad (1)$$

For example, in the case of the Amazon workload with a 4 KB block size, both Snappy and LZ4 obtain a similar storage saving rate of 39%, while LZ4HC and Zstd obtain better rates of 50% and 56%, respectively. These trends are also observed in other techniques under real-world workloads. Hence, we refer to the latter two techniques, LZ4HC and Zstd, as ratio-oriented techniques. In contrast, the former two techniques, Snappy and LZ4, provide better throughput, which will be discussed more evidently in Section 4.3, so we refer to them as performance-oriented techniques.

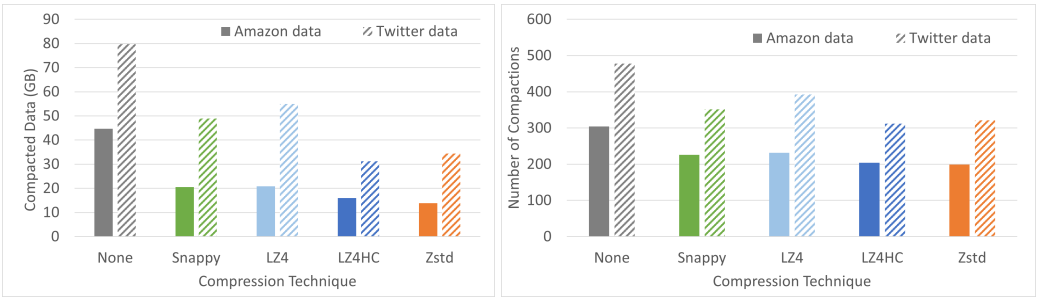
Effect of block sizes: Since RocksDB compresses data on block granularity, how the block sizes are set up makes a difference. For instance, in the case of the Amazon workload under Snappy, when the block sizes are set to 1 KB, 4 KB, 16 KB, and 32 KB, the space-saving rates are improved to 22%, 39%, 42%, and 47%, respectively. These improvements are also observed in other techniques (e.g., Zstd in Figure 3c) and other workloads (e.g., Twitter in Figure 3d). It means that larger blocks provide more chances to find repeated patterns, which leads to an enhanced compression ratio. When we increase the value size from 100 bytes to 500 bytes (difference between Figure 3a and Figure 3b), we observe some improvements, but the difference is small, with an average difference of 2%.

Effect of workloads: Our experiments demonstrate that workloads impact compression ratios considerably. The Amazon workload is the most compressible, showing space-saving rates ranging from 22% to 61% (the best case is obtained under Zstd with a 32 KB block size). It also shows the strongest sensitivity to different techniques and block sizes. The Twitter workload is modest, obtaining savings rates ranging from 26% to 50%. It is sensitive to different techniques but less sensitive to block sizes. The db_bench workload provides a consistent rate but is least sensitive to techniques and block sizes. This is because it is a synthetic workload, generating random data based on our configuration. In this experiment, we set the compression ratio at 50%, resulting in a similar amount of space savings. The small difference obtained by changing the value size from 100 to 500 Bytes is also due to the synthetic workload trait.

4.2. Impact on Compaction

Compaction is a core internal operation that significantly determines the performance of key–value stores [53]. During compaction, it generates a large number of I/Os to read and write SSTables, which can hinder I/O requests triggered by users. In addition, compaction is a CPU-consuming job, which may cause long-tail latency. This long-tail latency deteriorates the quality of service, becoming a serious problem in key–value stores.

To assess how compression impacts compaction, we make two measurements: one is the amount of compacted data, and the other is the number of compactions, while running the Amazon and Twitter workloads with a 4 KB block size, as shown in Figure 4. Compression can reduce the amount of compacted data by up to 69% (Zstd on Amazon workload), with an average of 53%. This reduction lessens the number of I/Os, eventually reducing the possibility of hindering user I/O requests. In addition, compression decreases the number of compactions by up to 33% (Zstd on Amazon workload) and 28% on average. This decrease can have a positive effect on preventing long-tail latency.



(**a**) Amount of compacted data (**b**) Number of compactions

Figure 4. Impact of compression to the size of data cumulatively processed during compaction (**a**) and the number of compactions (**b**) while running the Amazon and Twitter workloads in RocksDB.

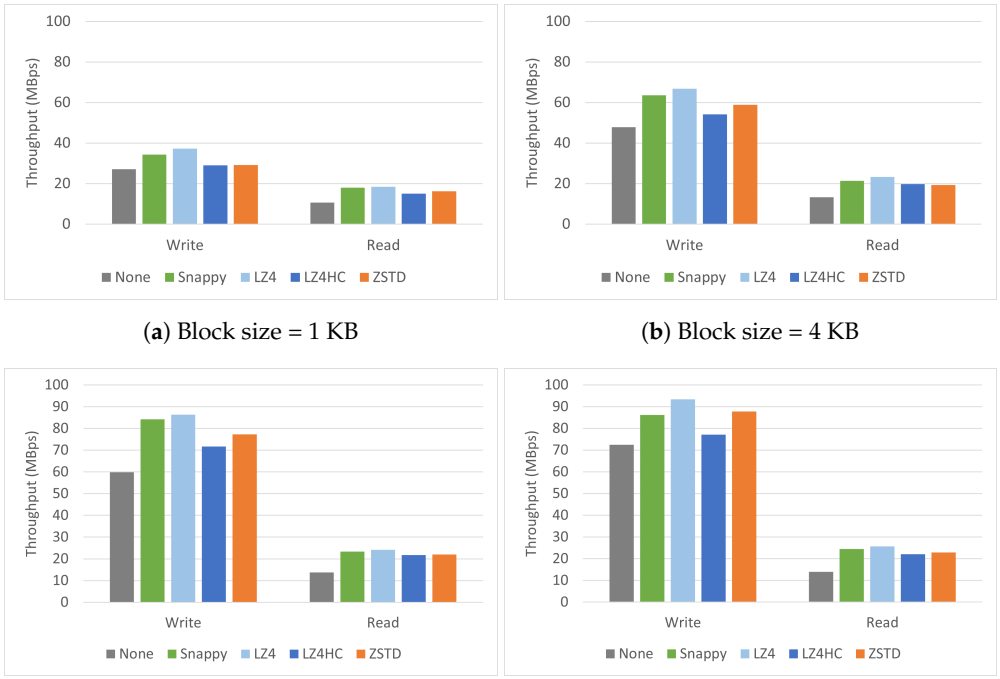
In Figure 4, we also observe that the decrease rate of the compaction numbers is relatively smaller than the reduction rate of the compacted data. This implies that, in the current implementation of compaction in RocksDB, the compaction numbers do not make full use of the benefit of compression compared with the compacted data. Our detailed examination reveals that compaction is triggered based on the number of SSTable files at the L_0 level. At the higher levels, it is triggered based on either the number of SSTable files or the total size of SSTable files at that level. Note that compression does not change the number of files, even though it lessens the size of each file. Therefore, with compression, the size-based triggering can reduce the number of compactions, while the number-based triggering cannot. Also note that, if we merge multiple files into one during compression, we can fully utilize compression in terms of not only the compacted data but also the compaction counts.

4.3. Performance of Key–Value Stores

Compression has both merits and demerits for the performance of key–value stores. The merit is that it can decrease the number of I/Os, which can lead to better throughput. In addition, it can reduce the number of compactions and compacted data, which can also enhance throughput. However, the downside is that the compression and decompression overhead makes the I/O latency longer, which in turn degrades throughput.

To evaluate this trade-off, we measure the write and read throughput as shown in Figure 5. In this experiment, we execute the Amazon workload under four compression techniques and four different block sizes. From this figure, we can make the following observations: First, compression improves the write and read throughput by an average of 22% and 50%, respectively. This means that the merit of compression outweighs the demerit.

Second, Snappy and LZ4 provide better performance than LZ4HC and Zstd. As we have already discussed in Section 2.2, the former two techniques have their strengths in compression speed, while the latter two have a strong point in compression ratio, which results in these outputs. Especially in key–value stores, due to compaction, we need to prioritize compression speed over ratio to achieve high throughput. On the other hand, in an environment with limited storage, ratio-oriented techniques will be an appropriate choice, as shown in Figure 3.



(**a**) Block size = 1 KB (**b**) Block size = 4 KB

(**c**) Block size = 16 KB (**d**) Block size = 32 KB

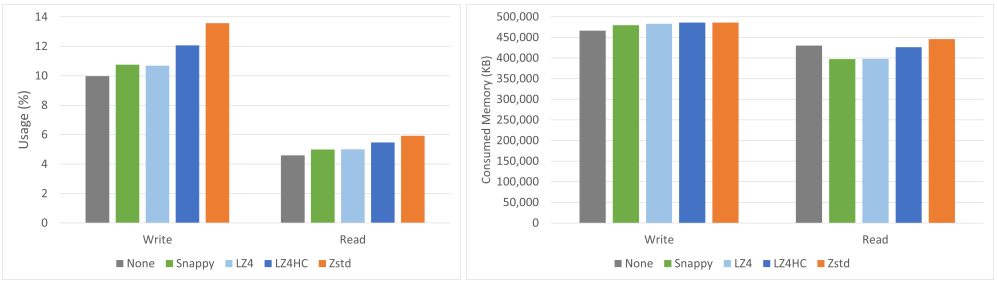
Figure 5. Impact of compression techniques and block size to the performance of RocksDB. We use the Amazon workload in this experiment.

Third, a larger block size improves the write throughput, from 28∼37 MB/s with 1 KB block size to 72∼92 MB/s with a 32 KB block size. We also notice that Zstd performs well at 32 KB block size due to its high compression ratio. However, such improvement is not noticeable for the read throughput. This is because a write request is handled asynchronously, whereas a read request is handled synchronously. Specifically, larger block sizes give several advantages, such as a higher compression ratio, a reduced number of I/Os, and the ability to exploit the internal parallelism of SSDs. However, it makes the I/O latency take longer to compress or decompress a larger block. The asynchronous feature of a write request enables it to hide the longer latency, making full use of the advantages of the larger block size. However, a synchronous read request is affected directly by the longer latency, which compensates for the advantages of the larger block size for the read throughput.

We coducted the same experiment using the Twitter workload and observe similar trends, as shown in Figure 5. We also identify that compaction and compression are conducted separately. In RocksDB, there is a special thread, called the compaction thread, that takes charge of the overall compaction procedures. It first reads data from SSTable files, decompresses it, performs merge–sort using the decompressed data, compresses it, and finally writes it to new SSTable files. Note that both merge-sort and compression are time-consuming jobs. We think that, if they are integrated in a pipelined manner, compression can enhance throughput further.

4.4. Impact on Resource Utilization

Figure 6 shows the impact of compression techniques on resource consumption. Compression increases CPU utilization, but the extent varies according to compression techniques. Since the ratio-oriented techniques, LZ4 and Zstd, try to enhance compression ratio while utilizing rather complex algorithms, they show a larger CPU utilization increment. In contrast, the performance-oriented techniques Snappy and LZ4 show a relatively smaller increment since they focus on reducing compression overhead while achieving a moderate compression ratio.



(**a**) CPU Utilization (**b**) Memory Consumption

Figure 6. Impact of compression techniques on CPU utilization and memory consumption.

The memory consumption presented in Figure 6b shows intriguing behaviors. For writes, compression increases memory consumption, and the ratio-oriented techniques use up more, as expected. On the other hand, for reads, the setting without compression (denoted None in the figure) takes up more memory than performance-oriented techniques like Snappy and LZ4. We think this is because performance-oriented techniques are geared towards minimal resource consumption, and using no compression means that there is more data being moved from storage to memory to be fed back to the user.

5. Lessons and Suggestions

In this section, we summarize the lessons we have learned during our evaluation. In addition, we give four suggestions for future research direction.

5.1. Lessons

One of the primary findings from our study is that the selection of a compression strategy has a significant impact on the rate of storage savings, the number of compactions, and the overall throughput of key–value stores. Zstd consistently offers the smallest storage saving rate across all block sizes, shown in Figure 3, and the fewest number of compactions, shown in Figure 4. However, it incurs the highest CPU utilization, as shown in Figure 6a. LZ4 provides the highest write and read throughput, but its compression ratio is relatively low. LZ4HC enhances compression ratio at the cost of degraded throughput. Snappy, prioritizing speed and simplicity, achieves a competitive throughput with a moderate compression ratio. We need to consider this trade-off between performance and compression ratio when we integrate compression into key–value stores.

Another lesson is that, in the current implementation of RocksDB, compaction and compression work independently. This is inevitable, since they have been designed for different objectives and at different times without awareness of each other. However, this unawareness makes it hard for compaction to fully utilize the benefits of compression.

As illustrated in Figure 4, the decrease in compaction counts is not in proportion to the reduction of compacted data. This implies that compaction is triggered more frequently than desired. Assume that the optimal trigger point for compaction at a level is N SSTable files if there is no compression. Then, with compression, the optimal trigger point needs to be changed to $2 \times N$ when the compression ratio is 50%. However, in the current version, it is still triggered at N, more frequently. Another example of unawareness is that merge–sort starts only after decompression is completely finished.

The third lesson pertains to the dependence of compression efficiency on diverse factors. A larger block size can improve the compression ratio, which results in enhanced write throughput. However, read throughput does not show such enhancement, since the increased decompression overhead due to the larger block compensates for the benefit of the improved compression ratio. It is also affected by workload, especially the lifetime of data. Note that, in key–value stores, the lifetime is closely associated with the levels due to the behavior of LSM-tree, which provides an opportunity to boost compression efficiency.

5.2. Suggestions

These lessons lead us to suggest the following four research directions: The first suggestion is a **compression-aware compaction mechanism**. It is devised to overcome the unawareness between compaction and compression. It consists of two ideas. First, it triggers compaction with the consideration of compression. One solution is that, if the compression ratio is R%, we set the trigger point to $100/R \times N$. However, even though this can avoid unnecessarily frequent triggering, it causes an increased number of SSTable files at a level, which has a negative impact on reads, especially at level 0. An alternative solution is coalescing multiple SSTable files into one during compression. For example, if we coalesce $100/R$ files into one, the number of SSTable files in a level does not increase, and frequent compaction triggering can be prevented. During coalescing, we need to consider the key ranges of a SSTable file to reduce compaction overhead further.

The second idea is integrating merge-sort and compression/decompression in a pipelined manner. Compaction is composed of five steps: (1) read, (2) decompress, (3) merge-sort, (4) compress, and (5) write, as we have already discussed in Section 3.2. Note that compression and merge-sort are time-consuming jobs. Hence, instead of passing all data after finishing decompression, our suggested mechanism divides the decompression step into multiple stages and passes data to the next step at the stage granularity. In addition, the merge-sort and compress steps also process data with the same granularity. This fine-granularity processing enables the merge-sort and compression/decompression times to overlap, eventually reducing the overall compaction latency.

Our second suggestion is **a selective compression approach**. We have learned that each compression technique has its own strengths. Some have a strong point in terms of performance, while others excel in terms of compression ratio. These observations give us an effective guideline when we choose a compression technique based on our requirements. For example, Zstd may be the ideal choice when space efficiency is the primary concern, while LZ4 will be a proper choice to obtain higher throughput.

In addition, the characteristics of key–value stores drive us to propose a new selective compression approach. Our analysis shows that the lifetime of SSTable files in the lowest level, L_0, is quite short since they are compacted shortly after being created. It implies that employing a ratio-oriented technique at the lowest level with heavy compression overhead is meaningless. Our analysis also uncovers that 90% of the data is stored at the highest level. This analysis leads in the opposite direction in that a ratio-oriented technique is indispensable at the highest level. To sum up, key–value stores require a selective compression approach based on levels and/or other data features such as hotness and compressibility.

The third suggestion is to **utilize emerging hardware devices**. Our observation shows that a larger block can enhance the compression ratio, which in turn improves write throughout. However, it does not improve read throughput due to the increased decompression overhead for larger blocks. This problem can be addressed using emerging hardware devices such as hardware accelerators [37] or new memory devices called NVM (Non-Volatile Memory) [54] or PM (Persistent Memory) [55]. This paper is interested in how to exploit NVM for compression.

NVM has both storage and DRAM features. Specifically, it supports non-volatility, which is a storage feature, and byte-addressability, which is a DRAM feature. Non-volatility allows for the storage of SSTable files without concern for durability. Byte-addressability allows data to be read with fine granularity and written with coarse granularity. Hence, using NVM, we can design a compression scheme that conducts compression with a larger block size while decompressing it with a smaller block size. This can enhance read throughput by decreasing the decompression overhead while achieving the benefit of a higher compression ratio for a larger block. One challenge of this flexible compression is how to make a shared dictionary used for both fine- and coarse-grained compression.

The final suggestion is **to rethink the compression granularity**. Many database and file system developers agree that block-granularity compression is better than file-

granularity compression, since applications access data using a block unit. However, in key–value stores, data are accessed not only as a block unit for reading a key–value pair but also as a file unit for compaction. Therefore, it is not obvious which one has better granularity, block or file, in key–value stores.

This depends on various parameters, including not only compression techniques but also storage types (e.g., slow HDDs or fast SSDs) and request portions generated by compaction. When storage becomes faster, reading a whole file is a viable solution. If the page cache in operating systems or the block cache in key–value stores can handle more requests for reads, requests generated from compaction become dominant. Key–value stores open a new design space and lead to revisiting the compression granularity, a spectrum from a fine unit (e.g., block) to a course unit (e.g., file).

6. Conclusions

This paper presents a comprehensive analysis of compression techniques utilized in key–value stores. We first provide a classification of compression techniques and discuss their strength and weakness. Then, we investigate the internal structures and operations, such as LSM-tree and compaction, in order to identify fundamental characteristics of key–value stores. Finally, we conduct a quantitative evaluation in terms of compression ratio and performance, while observing how characteristics and compression techniques influence each other using both synthetic and real-world workloads.

From our evaluation, we make the following four observations. First, by applying compression, we can achieve a space saving rate in a range from 22% to 61%. This reduction makes it feasible to boost not only the performance but also the lifespan of storage media. Second, compression has the potential to mitigate the compaction overhead, resulting in a reduction in the overall size of the compacted data, ranging from 31% to 66%. Third, both the block size and value size influence the compaction ratio greatly. Larger blocks and/or value sizes yield better compression ratios, generating more storage savings and a greater I/O reduction. Finally, some compression techniques such as Zstd and LZ4HC show better compression ratios, while others such as Snappy and LZ4 show better compression performance.

Our observations uncover potential avenues for future research, such as compression-aware compaction mechanism, selective compression, and reconsideration of the compression granularity. For instance, we recognize that existing compaction strategies are compression-oblivious and do not fully utilize its advantages, leading to compactions being triggered more frequently than needed. A new compression-aware compaction mechanism can integrate them in a pipelined manner to overlap their latencies and can perform compaction in a lazy manner to decrease the compaction overhead and write amplification. Our findings also give us a guideline when we choose an relevant compression technique in key–value stores based on our requirements, whether space efficiency or high throughput.

Author Contributions: Conceptualization, C.J. and J.C.; methodology, C.J.; software, C.J.; validation, C.J. and J.C.; formal analysis, C.J.; investigation, J.C.; resources, J.C.; data curation, C.J.; writing—original draft preparation, C.J.; writing—review and editing, J.C.; visualization, C.J.; supervision, J.C.; project administration, J.C. All authors have read and agreed to the published version of the manuscript.

Funding: The present research was supported by the research fund of Dankook university in 2022, and by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-01475, (SW StarLab) Development of Novel Key-Value DB for Unstructured Bigdata), and by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MIST) (No. 2022-0-01022, Development of Collection and Integrated Analysis Methods of Automotive Inter/Intra System Artifacts through Construction of Event-based experimental system).

Data Availability Statement: Not applicable.

Acknowledgments: The authors appreciate all the reviewers and editors for their precious comments and work on this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Sayood, K. *Introduction to Data Compression*, 5th ed.; Morgan Kaufmann: Cambridge, MA, USA, 2018; ISBN 978-0-12-809474-4.
2. Salomon, D. *Data Compression: The Complete Reference*, 4th ed.; Springer: London, UK, 2007; ISBN 978-1-84628-602-5.
3. Jayasankar, U.; Thirumal, V.; Ponnurangam, D. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *J. King Saud Univ. Comput. Inf. Sci.* **2021**, *33*, 119–140. [CrossRef]
4. Kleppmann, M. *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2017; ISBN 978-1-449-37332-0.
5. Ramadhan, A.R.; Choi, M.; Chung, Y.; Choi, J. An Empirical Study of Segmented Linear Regression Search in LevelDB. *Electronics* **2023**, *12*, 1018. [CrossRef]
6. LevelDB: A Fast Key-Value Storage Library Written at Google. Available online: https://github.com/google/leveldb (accessed on 30 July 2023).
7. Dong, S.; Kryczka, A.; Jin, Y.; Stumm, M. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21), Online Conference, 23–25 February 2021; pp. 33–49.
8. Elhemali, M.; Gallagher, N.; Gordon, N.; Idziorek, J.; Krog, R.; Lazier, C.; Mo, E.; Mritunjai, A.; Perianayagam, S.; Rath, T.; et al. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In Proceedings of the 2022 USENIX Annual Technical Conference (ATC'22), Carlsbad, CA, USA, 11–13 July 2022; pp. 1037–1048.
9. Sumbaly, R.; Kreps, J.; Gao, L.; Feinberg, A.; Soman, C.; Shah, S. Serving Large-scale Batch Computed Data with Project Voldemort. In Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12), San Jose, CA, USA, 14–17 February 2012.
10. Cao, Z.; Dong, S.; Vemuri, S.; Du, D. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20), Santa Clara, CA, USA, 25–27 February 2020; pp. 209–223.
11. Dong, S.; Callaghan, M.; Galanis, L.; Borthakur, D.; Savor, T.; Strum, M. Optimizing Space Amplification in RocksDB. In Proceedings of the CIDR, Chaminade, CA, USA, 8–11 January 2017.
12. Kim, J.; Vetter, J.S. Implementing efficient data compression and encryption in a persistent key-value store for HPC. *Int. J. High Perform. Comput. Appl.* **2019**, *33*, 1098–1112. [CrossRef]
13. Aghav, S. Database compression techniques for performance optimization. In Proceedings of the 2010 2nd International Conference on Computer Engineering and Technology, Chengdu, China, 16–19 April 2010.
14. Snappy: A Fast Compressor/Decompressor. Available online: https://github.com/google/snappy (accessed on 30 July 2023).
15. Zstandard—Fast Real-Time Compression Algorithm. Available online: https://github.com/facebook/zstd (accessed on 30 July 2023).
16. LZ4: Extremely Fast Compression Algorithm. Available online: https://github.com/lz4/lz4 (accessed on 30 July 2023).
17. Apache Cassandra: Open Source NoSQL Database. Available online: https://cassandra.apache.org/ (accessed on 30 July 2023).
18. WiredTiger Storage Engine. Available online: https://www.mongodb.com/docs/manual/core/wiredtiger/ (accessed on 30 July 2023).
19. Welcome to Apache HBase^{TM}. Available online: https://hbase.apache.org/ (accessed on 30 July 2023).
20. O'Neil, P.; Cheng, E.; Gawlick, D.; O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Inform.* **1996**, *33*, 351–385. [CrossRef]
21. Ni, J.; Li, J.; McAuley, J. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Hong Kong, China, 3–7 November 2019; pp. 188–197.
22. Yang, J.; Yue, Y.; Rashmi, K.V. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* **2021**, *17*, 1–35. [CrossRef]
23. Moffat, A. Huffman Coding. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–35. [CrossRef]
24. Pu, I.M. *Fundamental Data Compression*; Butterworth-Heinemann: Oxford, UK, 2006; ISBN 978-0-7506-6310-6.
25. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343. [CrossRef]
26. zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. Available online: https://zlib.net/ (accessed on 30 July 2023).
27. Alakuijala, J.; Farruggia, A.; Ferragina, P.; Kliuchnikov, E.; Obryk, R.; Szabadka, Z.; Vandevenne, L. Brotli: A general-purpose data compressor. *ACM Trans. Inf. Syst. (TOIS)* **2017** , *37*, 1–30. [CrossRef]
28. GNU Gzip: General File (de)Compression. Available online: https://www.gnu.org/software/gzip/manual/gzip.html (accessed on 30 July 2023).

29. LZMA: What Is LZMA Compression? Available online: https://www.winzip.com/en/learn/tips/what-is-lzma/ (accessed on 30 July 2023).
30. LZO. Available online: http://www.oberhumer.com/opensource/lzo/ (accessed on 30 July 2023).
31. Oswal, S.; Singh, A.; Kumari, K. Deflate compression algorithm. *Int. J. Eng. Res. Gen. Sci.* **2016**, *4*, 430–436.
32. Kodituwakku, S.; Amarasinghe, U. Comparison of lossless data compression algorithms for text data. *Indian J. Comput. Sci. Eng.* **2010**, *1*, 416–425.
33. Rana, K.; Thakur, S. Data compression algorithm for computer vision applications: A survey. In Proceedings of the 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, 5–6 May 2017; pp. 1214–1219.
34. Faria, L.N.; Fonseca, L.M.; Costa, M.H. Performance evaluation of data compression systems applied to satellite imagery. *J. Electr. Comput. Eng.* **2012**, *2012*, 18. [CrossRef]
35. Srisooksai, T.; Keamarungsi, K.; Lamsrichan, P.; Araki, K. Practical data compression in wireless sensor networks: A survey. *J. Netw. Comput. Appl.* **2012**, *35*, 37–39. [CrossRef]
36. Vaidya, M.; Walia, E.S.; Gupta, A. Data compression using Shannon-fano algorithm implemented by VHDL. In Proceedings of the 2014 International Conference on Advances in Engineering & Technology Research (ICAETR-2014), Unnao, Kanpur, India, 1–2 August 2014.
37. Chiosa, M.; Maschi, F.; Müller, I.; Alonso, G.; May, N. Hardware acceleration of compression and encryption in SAP HANA. *Proc. Vldb Endow.* **2022**, *15*, 3277–3291. [CrossRef]
38. Mittal, S.; Vetter, J.S. A survey of architectural approaches for data compression in cache and main memory systems. *IEEE Trans. Parallel Distrib.* **2015**, *27*, 1524–1536. [CrossRef]
39. Kimura, H.; Narasayya, V.; Syamala, M. Compression aware physical database design. In Proceedings of the 37th International Conference on Very Large Data Bases 2011 (VLDB 2011), Seattle, WA, USA, 29 August–3 September 2011.
40. Abadi, D.; Madden, S.; Ferreira, M. Integrating compression and execution in column-oriented database systems. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06), Chicago, IL, USA, 27–29 June 2006; pp. 671–682.
41. Binnig, C.; Hildenbrand, S.; Färber, F. Dictionary-based order-preserving string compression for main memory column stores. In Proceedings of the International Conference on Management of Data (SIGMOD/PODS '09), Providence, RI, USA, 29 June 2009–2 July 2009; pp. 283–296.
42. Mladenova, T.; Kalmukov, Y.; Marinov, M.; Valova, I. Impact of Data Compression on the Performance of Column-oriented Data Stores. *Int. J. Adv. Comput. Sci. Appl.* **2021**, *12*, 416–421. [CrossRef]
43. Müller, I.; Ratsch, C.; Faerber, F. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In Proceedings of the 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, 24–28 March 2014.
44. Ma, L.; Xie, R.; Zhang, T. ZipKV: In-Memory Key-Value Store with Built-In Data Compression. In Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management, Orlando, FL, USA, 18 June 2023; pp. 150–162.
45. Zhang, F.; Wan, W.; Zhang, C.; Zhai, J.; Chai, Y.; Li, H.; Du, X. CompressDB: Enabling efficient compressed data direct processing for various databases. In Proceedings of the 2022 International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022; pp. 1655–1669.
46. Guler, B.; Ozkasap, O. Compressed incremental checkpointing for efficient replicated key-value stores. In Proceedings of the 2017 IEEE Symposium on Computers and Communications (ISCC), Heraklion, Greece, 3–6 July 2017; pp. 76–81.
47. Jia, Y.; Shao, Z.; Chen, F. SlimCache: An efficient data compression scheme for flash-based key-value caching. *ACM Trans.* **2020**, *16*, 1–34. [CrossRef]
48. Jin, H.; Choi, W.G.; Choi, J.; Sung, H.; Park, S. Improvement of RocksDB Performance via Large-Scale Parameter Analysis and Optimization. *J. Inf. Process. Syst.* **2022**, *18*, 374–388. [CrossRef]
49. Tkachenko, V. Evaluating Database Compression Methods: Update. Available online: https://www.percona.com/blog/evaluating-database-compression-methods-update/ (accessed on 30 July 2023).
50. Skibiński, P. Lzbench. Available online: https://github.com/inikep/lzbench (accessed on 30 July 2023).
51. Zhang, Z.; Yue, Y.; He, B.; Xiong, J.; Chen, M.; Zhang, L.; Sun, N. Pipelined compaction for the LSM-tree. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, 19–23 May 2014; pp. 777–786.
52. Lim, H.; Andersen, D.G.; Kaminsky, M. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16), Santa Clara, CA, USA, 22–25 February 2016; pp. 149–166.
53. Balmau, O.; Dinu, F.; Zwaenepoel, W.; Gupta, K.; Chandhiramoorthi, R.; Didona, D. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19), Renton, WA, USA, 10–12 July 2019; pp. 753–766.

54. Kim, J.; Lee, S.; Vetter, J.S. PapyrusKV: A high-performance parallel key-value store for distributed NVM architectures. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 12–17 November 2017; pp. 1–14.
55. Yang, J.; Kim, J.; Hoseinzadeh, M.; Izraelevitz, J.; Swanson, S. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20), Santa Clara, CA, USA, 25–27 February 2020; pp. 169–182.