Compressed Data Direct Computing for Databases

Weitao Wan, Feng Zhang, *Member, IEEE*, Chenyang Zhang, Mingde Zhang, Jidong Zhai, Yunpeng Chai, *Member, IEEE*, Huanchen Zhang, Wei Lu, Yuxing Chen, Haixiang Li, Anqun Pan, and Xiaoyong Du, *Member, IEEE*

***Abstract*—Directly performing operations on compressed data has been proven to be a big success facing Big Data problems in modern data management systems. These systems have demonstrated significant compression benefits and performance improvement for data analytics applications. However, current systems only focus on data queries, while a complete Big Data system must support both data query and data manipulation. To solve this problem, we develop CompressDB, which is a new storage engine that can support data processing for databases without decompression. CompressDB has the following advantages. First, CompressDB utilizes context-free grammar to compress data, and supports both data query and data manipulation. Second, for adaptability, we integrate CompressDB to file systems so that a wide range of databases can directly use CompressDB without any change. Third, we enable operation pushdown to storage so that we can perform data query and manipulation in storage systems without bringing large data to memory for high efficiency. We validate the efficacy of CompressDB supporting various kinds of database systems, including SQLite, MySQL, LevelDB, MongoDB, ClickHouse, and Neo4j. We evaluate our method using seven real-world datasets with various lengths, structures, and content in both single node and cluster environments. Experiments show that CompressDB achieves 40% throughput improvement and 44% latency reduction, along with 1.75 compression ratio on average.**

***Index Terms*—Compression, compressed data direct processing, database systems.**

Manuscript received 7 January 2023; revised 29 August 2023; accepted 11 September 2023. Date of publication 18 September 2023; date of current version 5 April 2024. This work was supported in part by the National Natural Science Foundation of China under Grants 62322213 and 62172419, in part by Beijing Nova Program, and Public Computing Cloud, Renmin University of China. This work was also supported in part by funds for building world-class universities (disciplines) at Renmin University of China. Recommended for acceptance by F. Ozcan. *(Corresponding author: Feng Zhang.)*

Weitao Wan is with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and the School of Information, Renmin University of China, Beijing 100872, China, and also with the Tsinghua University, Beijing 100084, China (e-mail: wanweitao@ruc.edu.cn).

Feng Zhang, Chenyang Zhang, Mingde Zhang, Yunpeng Chai, Wei Lu, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and the School of Information, Renmin University of China, Beijing 100872, China (e-mail: fengzhang@ruc.edu.cn; chenyangzhang@ruc.edu.cn; 2019202191@ruc.edu.cn; ypchai@ruc.edu.cn; lu-wei@ruc.edu.cn; duyong@ruc.edu.cn).

Jidong Zhai and Huanchen Zhang are with the Tsinghua University, Beijing 100084, China (e-mail: zhaijidong@tsinghua.edu.cn; huanchen@tsinghua.edu.cn).

Yuxing Chen, Haixiang Li, and Anqun Pan are with the Tencent Inc., Shenzhen 518052, China (e-mail: axingguchen@tencent.com; blueseali@tencent.com; aaronpan@tencent.com).

Digital Object Identifier 10.1109/TKDE.2023.3316274

I. Introduction

MODERN Big Data systems use data compression to shrink their storage footprint in the face of ever-increasing data volumes. To avoid the overhead of constant compression and decompression operations, existing research systems have explored the idea of directly performing Big Data operations on compressed data [1], [2], [3], [4]. These systems have demonstrated significant compression ratio and performance improvement at the same time for data analytics applications. Although existing solutions have demonstrated significant potential for *read-only* query processing, a feature-complete Big Data system must support both *data query* and *data manipulation*. In particular, a system must support updates of random records as well as insertion and deletion of records. Previous solutions, however, do not natively support these functionalities, and thus must decompress and re-compress a relatively large chunk of data each time modification is incurred, leading to significant performance overhead.

In this paper, we strive to fill in the missing piece by developing a highly efficient technique to support updates, inserts, and deletes directly on compressed data, thus enabling a space-efficient Big Data system that supports both data query and manipulation. This is a challenging task because existing compression technologies are mostly optimized for compression ratio or read operations; the data structures used for compressed data are not amenable to modifications. For example, Succinct [1] is a database supporting queries over compressed data; the compression technique is based on index and suffix array [1], [5], [6], [7], [8], [9] where compressed elements are dependent on each other, making it extremely inefficient if a small unit of data needs updates. Another example is TADOC (text analytics directly on compression) [2], [3], which achieves a similar goal as Succinct but uses a rule-based compression strategy. Rule-based compression is a data compression technology that uses a set of rules to represent the elements of the original data. The dependencies among rules can be organized in a directed graph structure without any cycles, denoted as a DAG of rules. We plan to adopt the rule-based compression, which can be explained from three levels: elements, rules, and DAG.

Since large data are usually stored in disk, our idea is to develop random update over compressed data in storage layer based on rule compression. If we can integrate the compressed data direct computing technology with the underlying storage layer, then the databases built on it can automatically apply our

technology. Besides, data operations can also be pushed down to the storage layer for near-data computing.

However, in developing our idea, we meet the following three challenges. The first challenge comes from the element level. Previous TADOC processes data at word granularity, while the storage systems usually organize data at a much larger block granularity, such as 1 KB or 4 KB. Simply increasing the processing granularity can reduce the compression effect, because two large data blocks can have part of the same data. The second challenge comes from the rule level. Random updates involve great difficulties to handle, especially dealing with a large amount of rules. In detail, random update on hierarchically compressed data needs recursive rule split, which is extremely inefficient when the DAG is deep [4]. The third challenge comes from the DAG level. When we develop random updates over compressed data in storage layer on the fly, the operations also need to be implemented in the storage layer for efficiency. Because the access speed of the disk is much lower than the memory speed, we need to guarantee that the amount of data taken from disk by random access is as small as possible. Unfortunately, not only the rule that needs to be modified, but also its recursive parents need to be read into memory for updates. Even worse, data holes or unfull data blocks can be generated. It is hard to integrate these new data structures to the compressed data in real time without incurring extra overhead.

We develop a new storage engine, called *CompressDB*, which can support both data query and data manipulation directly on compressed data and can support various database systems. We make a key observation that rule-based compression method is suitable for data manipulation if the depth[1] of the DAG of rules is limited to a shallow degree. First, it can provide direct data manipulation without decompression. Second, it involves data reuse that saves both time and space. Third, data manipulation can be supported efficiently with simple rule operations. Specifically, CompressDB adopts rule-based compression and limits its rule generation depth. Meanwhile, CompressDB can compress and manipulate data in real time by operating grammatical rules. Compared with the previous rule-based compression targeting data analytics [3], we develop a series of novel designs: In element level, we propose a new data structure of data holes within rules. In rule level, we enable efficient rule positioning and rule split for random update. In DAG level, we reduce the depth of rule organization for efficiency. By leveraging new data structures and algorithm designs, CompressDB is highly efficient in data manipulation without decompression, which has not been supported by previous compression systems. To enable CompressDB to seamlessly support various databases, we develop CompressDB in file systems. At the file system layer, CompressDB can handle system calls like read and write, as they can be reimplemented with operations like *extract*, *replace*, *append*, etc. Accordingly, CompressDB can support different types of database systems that run on the file systems (e.g., SQLite, MySQL, MongoDB, etc). These database systems rely on the system calls handled by CompressDB. Thus, various data types (e.g., integer, float, string, etc.) and operations (e.g., join, select, insert, etc.) of database systems can be supported by CompressDB. In addition, we develop more general operations for CompressDB that are not supported by the file system, such as *insert* and *delete*. Because these operations do not have corresponding POSIX interface, we also provide a separate set of APIs, which can be used efficiently. Our preliminary work has been presented in [10]. Compared to the previous work, this paper adds many technical details, including the tradeoffs such as block size selection and the space-performance cost in the experiments, and the validation of the performance improvement of CompressDB on MySQL and Neo4j.

We validate the efficacy of CompressDB by supporting various kinds of database systems, including SQLite [11], MySQL [12], LevelDB [13], MongoDB [14], ClickHouse [15] and Neo4j [16]. We evaluate our method using seven real-world datasets with various lengths, structures, and content in both single node and cluster environments. We use a five-node cluster in the cloud with MooseFS [17], a high-performance network distributed file system. MooseFS spreads data on cloud and provides high-throughput accesses to data. Compared to the original baseline of MooseFS, our method achieves 40% throughput improvement, 44% latency reduction, and 1.75 compression ratio on average, which proves the effectiveness and efficiency of our method. The paper makes the following key contributions:

- We develop efficient data manipulation operations, such as insert, delete, and update, directly on compressed data. Along with previous random access support, we enable both data query and data manipulation.
- We develop CompressDB, a storage engine that is integrated into file systems. CompressDB can support various database systems seamlessly without modifying the databases.
- We enable operation pushdown to storage systems, which avoids unnecessary data movement between memory and disks, thus improving processing efficiency on compressed data.

II. Preliminary

After a quick scan at these algorithms, we find that grammar-based compression [18], [19], [20] is naturally suitable for random update directly on compressed data. We use TADOC [2], a representative rule based compression, for illustration.

A. Rule-Based Compression

TADOC is a novel rule-based solution for compression-based direct computing [2], [3], [4], which can be explained from three levels: elements, rules, and DAG. 1) *Element: The smallest indivisible minimum processing unit.* An element can be either a rule or a data unit such as a word from the original file. 2) *Rule: String of elements.* TADOC uses a rule to represent repeated content, and a rule consists of subrules and data units. 3) *DAG: The rule-compressed representation.* The relations between different rules can be organized as a directed acyclic graph (DAG).

Such a rule-based representation is much smaller than the original data. With this method, TADOC recursively represents

[1]We define the depth of a DAG as the maximal length of a directed path from the root to leaves.

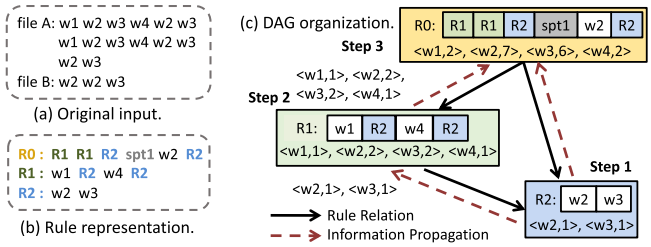


Fig. 1. Example of TADOC compression.

pieces of input data into a hierarchically compressed form, which is of great benefits for analytics over compressed data, since a rule can always be restored freely by explaining its elements, regardless of the context in which the rule appears.

Rule Compression: Rule expression in TADOC is a kind of compression, because each rule represents a repeated data segment in the input. In Fig. 1(a), we show the original input, which has two files. The symbol w_i can represent a minimal data unit, which can be a word, a character, or a data segment. TADOC can transform the input to a set of rules, as shown in Fig. 1(b). In Fig. 1(b), Ri represents a rule. $R0$ does not have in-edges, so we also call $R0$ the root. To utilize the redundancy between files, TADOC can compress different files together with a file boundary spt_i inserted in $R0$. For example, $spt1$ in $R0$ is used to indicate that the first file is represented as "$R1, R1, R2$" while the second file "$w2, R2$". Such a representation is lossless compression. To restore data, we can recursively restore the rules from $R0$. For example, to restore $fileA$, we need to restore $R1$ with $R2$ before restoring $R0$, which is "$w1, w2, w3, w4, w2, w3$". The relations between the rules can be represented as a DAG, as shown in Fig. 1(c).

Data Analytics: Based on the rule representation, we can perform common data analytics directly on the rule compressed data [4]. In detail, TADOC can convert the analytics task to a DAG traversal problem with rule interpretation. We use *word count* as an example to illustrate how to perform analytics directly on TADOC compressed data by traversing the DAG, as shown in Fig. 1(c). First, $R2$ transmits its local word counts to its parents, $R0$ and $R1$. Second, $R1$ transmits its accumulated word counts to its parent, $R0$. Third, $R0$ outputs the final result. For the other data analytics tasks, they can also be transformed into similar traversals based on rules.

Random Access: TADOC supports random access [3], [4] for data query, which is essential for data analytics systems. Zhang et al. [4] built special indexes for hierarchical compressed data. They built indexes on word granularity, including *word2rule* for words and *rule2location* for rules. In detail, the data structure *word2rule* can be used to locate the rules containing a word, and *rule2location* can be used to locate the offsets a rule appears in the original input.

Limitation: Although TADOC already supports common data analytics and random accesses, TADOC still has the following three disadvantages, which limit its application to Big Data systems.

- *Element level: The smallest processing element of TADOC is a word*. The word granularity is too small to integrate TADOC into current Big Data systems. Current systems process data at block granularity, and the size of a block can be 1 KB or 4 KB, regardless of the semantic and grammatical rules.
- *Rule level: The complicated organization of rules limits TADOC's efficient real-time random updates, which are necessary to Big Data applications*. The DAG of the rules in TADOC can be very deep. For example, the depth for dataset A (a 2 GB dataset detailed in Section VI-A) reaches 939 layers. Random updates require a bottom-up recursive rule split, which causes serious impact on performance, especially for deep rules. Currently, TADOC does not support *delete*, and can only insert new data into a separate file that will be merged to the compressed format by recompression.
- *DAG level: TADOC needs to load the whole DAG of compressed data into memory before processing*. This is inefficient for operations that utilize only partial data stored on disk. Specifically, due to the scale of Big Data, large data are usually stored on disk. Hence, current TADOC is extremely inefficient for Big Data applications.

B. Various Database Systems

One of the distinctive features of Big Data applications is the diversity of data [21], [22], including structured relational data, key-value data, and even unstructured text data. With the development of Big Data technology, various database systems appear [23], [24], and they have different advantages over diverse situations. For example, relational databases support applications that require fine-grained data management, such as people, finances, and objects, while key-value pair databases are suitable for data management such as unstructured or wide tables that do not require the definition of data schemas or highly variable schemas. Therefore, it can be expected that more and more databases will appear in the future.

Opportunity: There are now a variety of databases. Although it is not practical to combine compressed data direct computing technology with each database, we can still provide a way to support different databases. In the context of new Big Data applications, organizing, storing, and managing various types of data with different management systems still depend on file systems. Accordingly, we can provide a unified support in the storage layer, which is also the motivation of this work.

III. Design Overview

CompressDB: We develop a new storage engine, called CompressDB, which can perform data processing for databases without decompression. *Insight.* By redesigning the system from the three levels for efficient random updates, we can compress and manipulate data in real time by operating grammatical rules.

Despite these difficulties, we still decide to develop CompressDB in the storage layer, because storage space is much larger than memory size. Accordingly, an element in CompressDB represents a data unit like a data block, a rule represents repeated content consisting of elements and subrules, and DAG is the organization of rules, as discussed in Section II-A. In detail,

in element level, we introduce the concept of data holes to allow updates in large data blocks. In rule level, we develop hashing and counting data structures for efficiently locating rules. In DAG level, we limit the depth of the DAG to retain the cost of rule split and merge within a small range. Such a design is of great benefits to database applications, and can solve the challenges mentioned in Section I. Database systems built on our storage engine can enjoy the time and space benefits of compressed data direct processing.

Novelty in Element Level Design: Allowing Data Holes Within Rules. The block granularity used by the storage system is fixed, unlike the word granularity in TADOC that has variable length. Accordingly, we need to solve the alignment problem, because the data that was originally aligned with the block size can no longer be aligned with the block size after a random update. However, the current storage system does not solve the alignment problem: it only supports aligned insertion of data blocks. Therefore, we propose a hole structure that allows data holes in the block to be filled and aligned when misalignment occurs, so that the storage system supports flexible random updates. The operations of *extract, search, count, insert*, and *append* used in [3], [4] have also been pushed down to the storage layer utilizing the new design. Our method is compatible with the file system to the greatest extent, detailed in Section IV-A.

Novelty in Rule Level Design: Efficiently Locating and Merging Rules. The rule organization of the DAG structure of TADOC is too complicated for CompressDB. In TADOC, because a node can correspond to multiple parents, the recursive rule split makes the update extremely inefficient. Hence, we propose a new design in rule level organization: Except the leaves, the rest nodes are organized into a tree structure, and only the leaves can contain data blocks. Such a design is of great help for CompressDB. First, the split and merge operations for update on rules can be greatly simplified, because each node has a unique parent. Second, for locating data blocks, we can adopt a hash table to track the data in leaves, and it can also quickly justify whether a data block exists in a certain leaf node. Third, the data holes in element level design can exist in only leaves, making the data manipulation conducive.

Novelty in DAG Level Design: Limiting the Depth of Rules for Efficient Random Updates: The depth of rules in previous TADOC is very deep, which is mainly for reducing storage space. The compression of TADOC comes from Sequitur [25], and a deep depth can help reduce redundancy including rules. The current TADOC is only used for data queries. The compressed data are static and do not change. However, when we update the compressed data, such as inserting a piece of content into a rule, this can incur disastrous performance degradation. In detail, if we directly insert the content to a rule, the inserted rule needs to be split to two rules since it represents repeated content. Even worse, the parents of the rule all need to be split recursively, so the delicately compressed structure can also be disrupted. Hence, we limit the depth of DAG.

Difference From String Compression: CompressDB is different from string compression algorithms. First, the types of data to be processed are different. CompressDB works in the storage layer, which divides data into different data blocks, regardless of data types. In contrast, string compression usually handles text data. Second, the target platforms are different. CompressDB is a storage engine aiming to support various Big Data systems, which cannot be achieved by string compression algorithms. Third, CompressDB provides a series of operations and can interact with users to perform different data analytics and manipulation tasks, while string compressions are more focused on compression and specific tasks such as indexing.

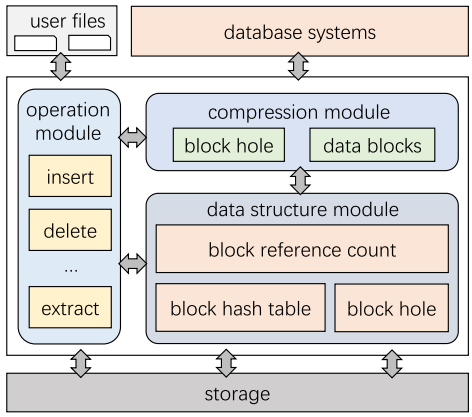


Fig. 2. Overview of CompressDB.

IV. System Design

CompressDB works in the storage layer, so the database systems built on CompressDB can employ its ability of compressed data direct computing automatically. In this section, we show the detailed system design of CompressDB.

A. Overview of System Modules

We show the overview of our storage engine, CompressDB, in Fig. 2, which can support various database systems. CompressDB consists of three major modules: 1) data structure module, 2) compression module, and 3) operation module. These three modules support the database systems built on our storage engine. The first data structure module provides necessary data structures to both the compression module and the operation module. The provided data structures include *blockHashTable* for indicating the mapping relation from data content to block location, *blockRefCount* recording the referenced number of times of a block, and *blockHole* for handling holes caused by update operations. The second compression module supports hierarchically compression in file systems and can be applied to various block-based file systems. The third operation module can push down user operations to file systems. Importantly, our operation pushdown techniques are transparent to users, so users can still use the system in the same way as TADOC with random update support.

Note that these modules are not independent of each other. They work synergistically to address the various complexities in all types of random updates. First, the data structure module is the basis of the system, which provides necessary data structures to both modules. Second, the compression module compresses the input with the support of the data structure module, and stores the compressed data in file systems for the operations after pushdown. Third, the operation module operates on the

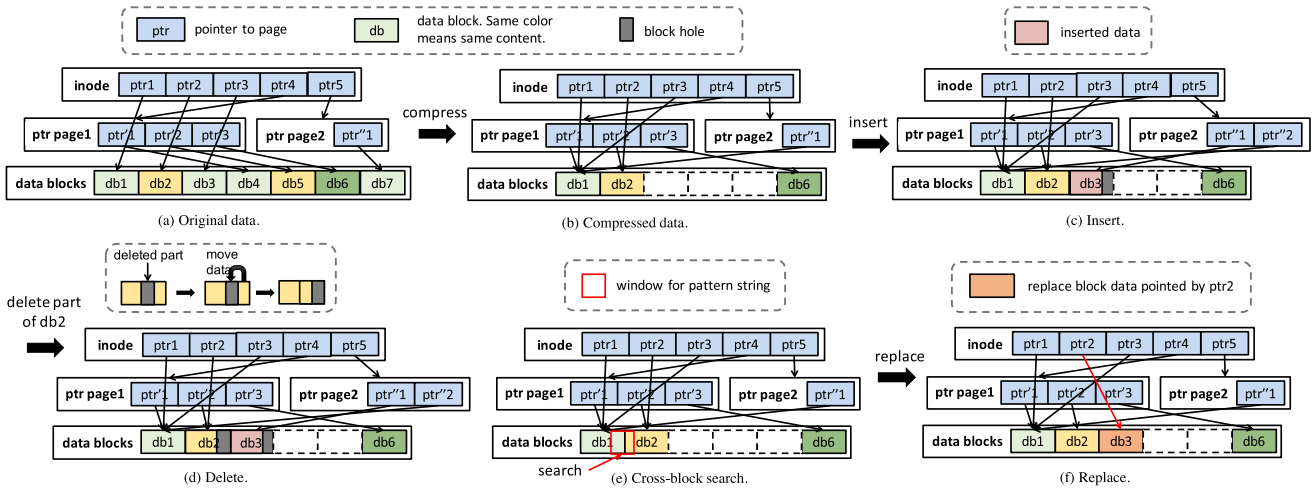


Fig. 3. Illustration for compression and random updates.

TABLE I
DATASETS

Dataset	A	B	C	D	E	F	G
Size	50GB	150GB	300GB	2.1GB	580MB	26GB	1.9GB
File#	109	309	618	4	134631	36	405

TADOC-compressed data from compressor with the data structure support.

Operation Pushdown: To reduce data transmission cost, we push down the operations to the storage layer. The operation pushdown refers to that the data processing happens directly in the file system layer (a lower software layer). It allows the direct computing techniques to occur closer to data, with which CompressDB can significantly reduce the amount of data accesses to disk and accelerate all upper-level database applications. Computation pushdown has been widely applied in database machines [26], [27], [28], cloud databases [29], [30], [31], [32], and memory [33]. We integrate CompressDB to systems like FUSE and MooseFS so CompressDB can work in both single and distributed environments. Similar idea has also been used in different scenarios. For example, in relational model, storage-related operators in relational algebra, such as scanning, filtering, and projection, can be embedded into the storage layer [34], [35]. In graph models, the processes for obtaining vertices, edges, and attributes of the graph can be integrated to the storage layer reducing the amount of data transmitted to the upper layers [36], [37]. However, these specialties have not been considered in previous TADOC research [2], [3], [4].

Updating Parent Nodes: One prominent feature of CompressDB is its simplicity in updating parents. In practice, even for a small file of 2 GB (dataset D in Table I), d can reach 12 and n can reach 1,211,546. By limiting the number of parents of non-leaf nodes to one, we reduce the complexity of updating parents from $O(n^d)$ to $O(d)$.

Depth of DAG: With the rule- and DAG-level design of CompressDB, the changes caused by splitting and merging of nodes are integrated to the same layer or upper layers, and thus the depth of DAG does not increase.

Example: We show an example in Fig. 3 for illustration. Fig. 3(a) shows the original data, while Fig. 3(b) shows the compressed state, assuming $db2$ and $db5$ contain the same content. Fig. 3(c) shows an insert example. Assume we insert content (marked red in $db3$) before the $db1$ pointed by ptr"1. We need to allocate a new $db3$ first and then write the content to it. Note that the parent pointer (ptr"1 in ptr page2) of $db1$ needs to be split into two pointers, which respectively point to $db1$ and the new block. Accordingly, we update the parent and add a pointer ptr"2. This process should be performed recursively (split the parent pointer of ptr page2) if ptr page2 does not have enough space for ptr"2. Fortunately, the depth in the file tree is not deep so the block split cost is marginal. Fig. 3(d) shows a *delete* example. We delete the piece of content after we copy the data block to a new one. Then, we update its parents recursively. Fig. 3(e) shows a *search* example. We develop a novel parallel block-level search strategy, which includes two phases. In the first phase, we search the given content within each block in parallel. In the second phase, we perform cross-block search. We use a window with the same length as the pattern string, and then let it slide at the junction of two adjacent blocks to search for the cross-block occurrences of the given pattern string. Fig. 3(f) shows a *replace* example, which can be achieved by *insert* and *delete*.

Next, we explain in detail the three modules of data structures (Section IV-B), compressor (Section IV-C), and operation pushdown (Section IV-D).

B. Data Structure Module

We show the data structure module in this part, which provides necessary data structures to both the compression module and the operation module. When designing the data structures required to support our method, we keep space overheads in mind and try to make each newly introduced data structure useful for more than one type of operation. Specifically, we introduce three data structures, which we briefly explain as follows. We provide more details on each data structure when we explain our techniques for the five random update operations.

1) *blockHashTable:* This data structure provides the mapping from the hash value of a block to its block number. It can be used to quickly locate a block location in both the compression process of the compression module and the related operations that need to quickly locate content such as *search* in the operation pushdown module. Specifically, we use the hash value of the block as the key and the block number as the value. In such design, one key may have multiple values, as different blocks (with different content) may share the same hash value. For blocks with the same hash value, we need to further compare the block contents to determine whether they are identical. Hence, the system is resilient to hash collisions.

2) *blockRefCount:* This data structure records the number of times a block is referenced in the file system. For example, if a block is referenced twice, it means that the block occurs twice in a file or occurs in two files. This indicator is extremely useful in compressed storage systems, because a block can be shared by multiple rules in TADOC. For example, it provides the information on whether a block can be released before *delete* operation, or whether a block can be modified before *insert* and *replace*.

3) *blockHole:* This data structure provides the necessary information in update operations when "block hole" is generated. This structure records the hole structure caused by *insert* and *delete* operation. Note the file systems usually support only *write* and *read* operations, with no *insert* or *delete* operations that can generate a "hole" in blocks. In our random update such as *insert*, the *offset* and *size* of insertion are not required to be aligned with block size. Therefore, we have to add the data structures for holes to make the *offset* and *size* aligned with block size so that the insertion does not destroy the compression.

Computing Offset With Data Holes: After introducing data holes, a common question is how to compute offset with data holes. In our design, we add extra meta data to trace the offset and size of each hole. Experiments show that the overhead of data holes is less than 3%.

Space Consideration: The data structures, such as block hash table, occupy large space and can be frequently updated. To save storage space, these data structures are selectively stored on disk or in memory. In detail, the data structure *blockRefCount* is usually the largest one because it stores the reference counts of all blocks. Therefore, we allocate a partition on disk to store all the reference counts so that the compressed data will not be destroyed in practice even after a *remount* (unmount and mount) or failure (crash or poweroff) of file system. As for *blockHashTable*, we put it in memory because they are not required to be kept after a remount. As for *blockHole*, it takes up very little storage space, so we keep it both in memory and on disk.

C. Compression Module

General Design: Our general design is that when data are input into CompressDB for the first time, we use our rule based method to compress the file to the system. In detail, we regard each block as a node. The indirect nodes represent rules while the data nodes represent leaves in TADOC. To limit the depth of the DAG, we first view the original file organization, which is usually organized as a tree structure. Second, we use the *blockHashTable* to identify the identical blocks, and merge their upper-level pointers pointing to the same block. Third, we update the *blockRefCount* for references. With such design, the compression process becomes lightweight and the depth of the DAG is shallow, which is practical to file systems. Then, when users update the data, we can perform the operations on the compressed data directly (detailed in Section IV-D).

Case Study: We show an example of data compression in Fig. 3(a), and use a simplified inode map for illustration. There are five pointers in the root. Three of them are direct pointers, and each direct pointer points to one data block leaf. The other two are indirect pointers pointing to subrules, and every subrule node further points to three leaves of data blocks. The colors of the data blocks indicate the content, and blocks with the same color have the same content. Therefore, Fig. 3(a) indicates that a file includes seven data blocks and more than half of them are redundant. Note that both indirect rules and leaf nodes need to be stored in file systems. However, we find that leaf nodes account for more than 99% of the storage space, so we mainly consider how to compress the leaves by managing the map from indirect rules to leaves, as shown in Fig. 3(b).

We identify and merge the redundant blocks by letting their pointers point to only one block. In Fig. 3(b), although different colors of data blocks appear more than once, we only store them once, with multiple pointers pointing to these blocks. Now, except for indirect rules, only three data blocks need to be stored in the file system for this file, thus achieving data compression. Further, the indirect rules can also be compressed. For example, the two indirect nodes in blue point to the same blocks, which can be merged.

Integration to File Systems: When we integrate our solution to file systems, we have two options about when to launch the compressor for modification: file-level compression, and block-level compression.

The first option is file-level compression, which is to check all data blocks of a file after all modifications of the file. In file systems, read or modification to a file should be performed after an *open* call, and ends with a *close* call. This option means that we need to check all the blocks of a file after a *close* call, even for a small modification. Moreover, as shown in Fig. 3, the index structure in inode could be complicated, so interaction between different modifications can be time-consuming, even is performed only after a *close* of a file. Hence, we abandon this design.

The second option is block-level compression, which is to check the related data block after each modification. At block level, any read or modification to a block should be performed after a block *get* call, and ends with a block *release* call. Here, the *get* call loads the block from disk to memory, while the *release* call releases the block after any read or modification by callers. With this design, we only need a single check for each *release*, without the requirement to interact or trace all modified blocks. Therefore, we use this design to launch our compressor for each modification.

Detailed Design: In this part, we show our detailed design about how to use the *blockHashTable* data structure to identify repeated content, and then perform compression. Assume that

every block has its own unique block number and can be accessed by the block number. For the hash table, we create a map from the content of blocks to the block numbers, and use string s to represent the content of a block. The key is the content of the block, denoted as s, while the value is the block number. In the hashing process, we hash s to a 64 b long unsigned integer $hashed\_s$ and calculate $bucket\_number$ by "$hashed\_s$ mod $length$" to assign a bucket for each key. The parameter $length$ is the size of hash table.

After the hashing and modular operation, db1 and db2 are hashed to bucket 1, indicating that a hash collision occurs. We use linked lists to solve the collision problem. When a data block is released, we can perform a hashing and a modular operation to obtain the value. If a previous data block has repeated content, its block number should be in the corresponding bucket. Accordingly, we need to traverse the bucket to identify the repeated blocks. In detail, we check whether the content of the block corresponding to each block number in the bucket is duplicated with the current block.

Algorithm: The pseudocode of the compression is shown in Algorithm 1. Any modification to the file triggers the compression. The input of Algorithm 1 includes the current block, a temporary block, and a pointer to the current block in the inode. The current block is the block to be changed. We have updated the *release* call in the block-level process, so the modification does not take effect immediately in file systems. The modification is stored in a temporary block, so we need to cooperate with the temporary block. Moreover, if the current block is the same as another block, the pointer of the current block in the inode also needs to point to the identified block. The detailed process is shown as follows.

First, we perform a hashing process for the content of the temporary block to find if there is a repeated block via `hash_find_duplicate`, as shown in Line 1.

Second, if a repeated block is found, we then check the reference count of the current block, as shown from Lines 2 to 8. If the reference count is one, there is no other pointer pointing to current block and the block can be released (Lines 3 to 4). Otherwise, we subtract the reference count by one (Line 6). Next, we need to set the current pointer pointing to the repeated block by setting the value of the current pointer to the repeated block's number, and add one to its reference count (Lines 7 and 8).

Third, if a repeated block is not found, we still need to check the reference count of the current block. If the reference count is one, we can see that no pointer points to the current block so we can modify the block (Line 12). In detail, we need to change the record of the current block in the hash table, as the content is changed (Lines 11 to 13). If the reference count is larger than one, we need to subtract one from the reference count and allocate a new block to store the modified content. Then, the pointer of the current block should be set to the new block and we should add the record of the new block to the hash table (Lines 15 to 18).

After these steps, the modification to the current block has been handled for data compression.

Complexity Analysis: The complexity in the first and second steps of Algorithm 1 is $O(1)$, since these steps involve only limited operations such as hashing, fetching or modifying the reference count of blocks, deleting a record in the hash table, and freeing blocks. For the third step, the update operation in Line 12 takes $O(1)$ time, as the size of the block is fixed. Similarly, the complexity for the update in Line 18 is also $O(1)$, while the complexity for the other parts is $O(1)$. Hence, the complexity for Algorithm 1 is $O(1)$.

Algorithm 1: Compression in CompressDB.

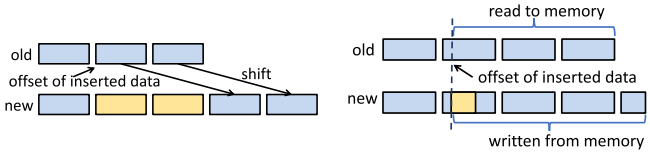
```
   input: curr ← current block.
          tmp ← temporary block with data to write to
   curr.
          ptr ← pointer to current block in inode.
1  if dup = hash_find_duplicate(tmp) then
       // Duplicate block found.
2      if curr.reference_count == 1 then
           // Delete record in hash table.
3          delete_record(curr);
4          free_block(curr);
5      else
6          curr.reference_count -= 1;
       // set the pointer to duplicate block.
7      ptr = dup;
8      dup.reference_count += 1;
9  else
10     if curr.reference_count == 1 then
           // Delete the record of curr.
11         delete_record(curr);
           // Update the content of curr with
              tmp.
12         curr.update(tmp);
           // Renew the record with new data.
13         add_record(curr);
14     else
15         curr.reference_count -= 1;
           // Copy on write.
16         ptr = allocate_block();
17         ptr.update(tmp);
18         add_record(ptr);
```

Additional I/Os: When a block shares the same hash value as a record in the hash table, additional I/Os may occur. In our complexity analysis, we treat these I/Os as regular operations, like arithmetic add and division. Therefore, the performance of these hash table operations may be primarily influenced by the number of I/Os required when the relevant blocks are not in memory.

Specifically, the operation `hash_find_duplicate()` compares blocks only when multiple blocks share the same hash value. This comparison checks for identical content or hash collisions. If the block is not in memory, additional I/Os are required. Similarly, `delete_record()` searches the hash table, requiring additional I/Os if the block is not in memory. In contrast, `add_record()` does not require comparisons or additional I/Os since it is used when the block content is not in the hash table.

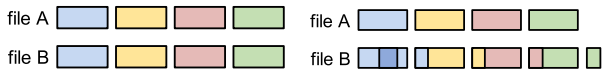
D. Operation Module

We next introduce the operations in the operation module. These operations, including *extract*, *replace*, *insert*, *delete*, *search*, and *count*, can be pushed down to the storage layer.



(a) *Insert* by *fallocate*. (b) *Insert* by overwrite.

Fig. 4. Traditional solution for *insert*.



(a) Before *insert*. (b) After *insert*.

Fig. 5. Loss of compression.

Insert: This operation inserts content directly to the compressed data, without interfering with other operations. In this part, we first analyze the current file update operations in file systems. Second, we introduce our *insert* solution. Third, we analyze the influence of *insert* on the other operations.

1) Analysis on Previous Update Operations: We analyze previous update operations before showing our design.

The first update operation we analyze in file systems is the *fallocate* system call, which can quickly preallocate or deallocate data blocks in file systems. We show an example in Fig. 4(a), which inserts two data blocks into the data block sequence by modifying metadata in inode. However, *fallocate* has a critical drawback that the size of the inserted data needs to be an integer multiple of the block size. This restriction makes *fallocate* less practical so we abandon it.

The second update operation we analyze is the *read* and *write* system calls. Assume that we use *read* and *write* to insert data with the size of len at *offset*, as shown in Fig. 4(b). We have to read all content after *offset*, append the read content at the end of the data to be inserted, and then write all of these data at the *offset* of the file. Unfortunately, although this approach does not require the *offset* or len to be an integer multiple of the block size and can be applied for arbitrary insertion, it has two major disadvantages: (1) large read/write size cost, and (2) potential compression benefit loss. As shown in Fig. 5(a), block sequences A and B are the same, so we need to store only one of them, according to the compression technique. However, if we insert a piece of data to the first block of sequence B, as shown in Fig. 5 (b), every corresponding block of these two block sequences becomes different. Therefore, the benefits of compression technology are greatly reduced.

2) Our Design: Our design should overcome the shortcomings of the initial *update* operations of the file systems, and maintain the advantages of their methods. On the one hand, to ensure high performance, we should perform *insert* by inserting data blocks directly into the block sequence. Inserting blocks into a block sequence is time-efficient because only the metadata in inode has been modified. On the other hand, to support flexible *insert*, we should support *insert* for arbitrary *offset* and len. However, *inserts* that are not an integer multiple of the block size generate holes. As discussed in Section IV-B, the *blockHole* data structure is in-memory metadata stored in inode and records the location and size of each hole, which can be used to solve this problem. In our solution, if an *insert* is not aligned with block size, we add a *blockHole* instance to make sure the alignment for the insertion to data blocks.

3) Case Study: We show an example in Fig. 3(c). An *insert* is performed before the seventh data block ($db1$ pointed by ptr"1) of this file, and its length len is unaligned with the block size. We add a new block (db3) where the gray part represents a block hole (implemented by the *blockHole* data structure). By redirecting ptr"1 from $db1$ to $db3$, we have successfully inserted the unaligned content without disturbing the data layout of all the other blocks.

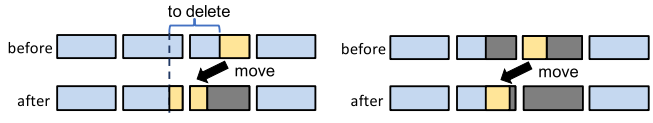
4) Detailed Design: The detailed design for inserting len data at *offset* is as follows. First, we allocate $\lceil (offset + len)/n \rceil$ new data blocks, where len represents the length of the inserted data and n is the block size. Second, if the len is not an integer multiple of block size, which means that a hole is going to be generated, we use *blockHole* to fill in the blank at the end. Third, we input the inserted data with the related block into the new blocks and redirect the pointers for the changed blocks to the new ones.

5) Influence of insert*to the Other Operations:* Next, we illustrate how our solution cooperates with the other operations. Since the holes caused by *insert* exist in the block sequence but with meaningless content, we need to skip these holes when performing the other operations such as *extract* and *replace*. The operations need to carefully check and skip the holes with the help of *blockHole* structure. To make *insert* compatible with the other operations, we need to regard the blocks with holes as regular blocks and perform hashing to adapt to the compression techniques in *replace*. After such a patch, our *insert* is both efficient and flexible because only metadata need to be modified about holes for alignment. Moreover, it is obvious that we have eliminated the compression loss in Fig. 5.

6) Complexity Analysis of *Insert:* We mainly focus on the IO operations, which include filling the blank part with *blockHole*, and writing the inserted data into new data blocks. These operations dominate the major time. Since we have to write the inserted data into storage and add a hole if necessary, the time complexity should be $O(m + n)$, where n is the block size and m is the size of data to be inserted. Because the size of *blockHole* is less than the data block size, the space cost is $O(n)$.

Delete: The *delete* operation removes len length data from the *offset* position. The *offset* and len can also be misaligned to the block size, so we need to utilize the *blockHole* data structure by adding data holes to make the remaining data aligned. As to data holes, repairing holes is equivalent to data movement since we need to move the data close to holes for merging. Data movement is expensive in file systems, and we should avoid frequent data movements.

Our design: We develop *delete* with the *blockHole* data structure. We show an example in Fig. 6(a). The deletion can start in one block and end in another block, which are block#1 and block#2 in Fig. 6(a). The remaining data in the blocks involved in the *delete* operation can also be misaligned, so we allow holes in the operated data blocks. Moreover, since *delete*



(a) *Delete* operation. (b) Remove a hole.

Fig. 6. *Delete* in CompressDB.

operations can incur a large number of holes, we develop a hole merging process. In detail, in a *delete* operation, when the blocks involved in a *delete* operation have more than one hole, we can rearrange the remaining data in the front part of the involved block sequence, and release the hole space if the size of the merged holes is larger than a block size.

Detailed Design: We have the following design to perform the *delete* operation in detail. First, we need to check if there are holes in the involved block sequence, which need to be merged into one hole. Second, we move the remaining data in the front part of the block sequence. Third, we check if there are redundant blocks because the size of the merged holes can be larger than the block size. If so, we remove the redundant blocks and update the *blockHole* data structure.

Complexity Analysis: The complexity of *delete* is influenced by three parts: (1) moving the remained data forward, which is the yellow part in Fig. 6(a), (2) adding a block hole to fill the blank part with *blockHole*, and (3) checking *blockHole*, rearranging *blockHole*, and releasing the hole space. For the first part, because the remained data size is less than block size, the time complexity for data movement is $O(n)$, where n is the block size. For the second part, the complexity of adding a block hole is $O(1)$. For the third part, although checking and releasing take $O(1)$ time, the rearrangement takes $O(n)$ time, since the rearranged data size is less than block size. Therefore, the time complexity of *delete* is $O(n)$.

Extract: In this part, we show how *extract* cooperates with the compression technique. To perform *extract*, our method traverses the data blocks in sequence from inode. The inode contains the mapping relations from file to data blocks, which means that any offset in a file can be transformed to an offset in a data block, as shown in Fig. 3(a). For example, assume the block size is 1,024 bytes. Then, the offset 1,536 in file corresponds to the offset 512 in the second block of the block sequence indicated by the mapping relation in inode. Accordingly, to perform the *extract* operation for a file, we just need to obtain the offset from inode and traverse from the offset in data block. We can see that any offset in a file still corresponds to an offset in the data block though their pointers have been changed.

Complexity Analysis: The complexity of *extract* is dominated by two parts. The first part is to find the data blocks from the hash table, whose complexity is $O(1)$. The second part is responsible for reading the data, whose complexity is $O(m)$, where m is the size of the extracted data. Hence, the time complexity of *extract* is $O(m)$.

Replace: Replace is an operation that is used to replace one or more bytes of the content of a file. Cooperating with inode, we can map the offset of the content to be replaced in the file to an offset in another data block, and then replace the content in the new data block. However, in TADOC compression, a rule of a data block can be shared and referenced by multiple pointers, so we cannot modify the content in a data block directly. To solve this problem, we allocate another block for modification and change the pointers for rule split. We show an example in Fig. 3(f). Assume that the file has seven data blocks and the *replace* target lies in the second data block ($db2$ pointed by ptr2). The second block has been referenced twice, so we cannot change its content directly. Accordingly, we allocate a new data block, set ptr2 to the new one, and then write the modified content to the new block. In Fig. 3(f), the orange block is a newly allocated one and the red arrow represents a pointer modification. Note that after the modification to a data block, we should also check if the new block is a repeated block by using *blockHashTable*.

Note that *replace* is different from "*delete+insert*". In CompressDB, *replace* directly overwrites the content on the old data instead of deleting and then inserting it. We do not use the "*delete+insert*" design because it needs to release space first, and then reallocate and write new content. This process can introduce additional overhead including creation and deletion of data blocks, and merging and splitting of rules.

Complexity Analysis: The *replace* complexity is influenced by three parts. The first part is to allocate new data blocks and store the modified content. Assume that the duplication ratio is r and the size of content to be modified is m. Then, its complexity is $O(r \times m)$, since the data volume to be stored is $r \times m$. The second part for redirecting pointers and the third part for checking with *blockHashTable* have $O(1)$ complexity. Therefore, the complexity for *replace* is $O(r \times m)$.

Search: The *search* operation returns the offsets for the content of a user's query. This operation consists of three major stages. The first stage is in-block search, which calculates the offsets when the content of the query completely falls within a block. In detail, each block can appear more than once in the original file and the offsets for each block can be obtained from *inode*. After obtaining the local offsets of the required content for each block in parallel, we can calculate the final offsets by adding the local offsets with the offsets of the blocks containing the required content. Note that repeated blocks appear only once in our compressed format. The reuse brings significant time saving for computation and data transmission. The second stage is a cross-block search. Assume the length of the content for the query is m and the block size is s. We need to check all the m-length cross-block content of the "$\lceil m/s \rceil$" consecutive blocks in parallel. The third stage is a merging stage, which merges the in- and cross-block results and returns the result.

Complexity Analysis: The complexity is determined by the string matching algorithm, which is KMP (Knuth Morris Pratt) [38] in our solution. The KMP complexity is $O(M + N)$, where M is the length of the pattern string, and N is the length of the long text. As in our *search* scenario, the time of the first stage is $O(k \times (m + n))$, where m is the size of the pattern string, k is the number of blocks, and n is the block size. For the second stage, a cross-block search takes $O(m)$ time. The data has $\lceil s/n \rceil$ data blocks, so the second stage performs cross-block searching $\lceil s/n \rceil - 1$ times, taking $O(\lceil s/n \rceil \times m)$ time. The third stage

takes $O(1)$ time. Therefore, the complexity of *search* is $O(k \times (m+n) + \lceil s/n \rceil \times m) = O(\lceil s/n \rceil \times m + k \times n)$.

Append: This operation appends data at the end of a file. Similar to *replace*, it can be implemented by allocating new blocks and performing a *replace* on the newly allocated content. However, to achieve better performance, we do not want to allocate first. Instead, we check whether it is necessary to allocate. For example, if the content to append is repeated with a previous block, we just need to add a pointer pointing to the previous block without block allocation. Otherwise, we allocate new blocks for *append* and fill the new blocks with the appended content. Note that *append* is different from *insert*. CompressDB stores the end positions of files in meta data, and *append* uses this data structure to quickly locate the end of the file and add data. In contrast, *insert* needs to support inserting data in different positions, which takes more time to find the insert position. Besides, *insert* is more likely to bring new data holes due to the alignment issues.

Count: This operation returns the frequency for the content of a user's query. Similar to *search*, we can obtain the frequencies from both in- and cross-block traversals. Note that the frequencies for each block can be provided by *blockHashTable*, which can be used directly among in-block scan for time-saving.

V. Implementation

We develop CompressDB in real storage systems, including FUSE [39] and MooseFS [17], for validation. The FUSE system (Filesystem in Userspace) [39] allows us to conveniently construct a file system in user space without caring about the code of the Linux system kernel. MooseFS [17] is a high-performance network distributed file system implemented in C language. MooseFS spreads data on commodity hardwares, and provides high-throughput accesses to data, which is suitable for Big Data scenarios. After integration, each operation acts as a separate call in userspace. The *extract* operation extracts a piece of content, which is similar to the *read* system call. The *replace* operation modifies a file at arbitrary *offset*, which is similar to the *write* system call. The *insert* operation performs insertions, and the *delete* operation performs deletions. The *search* operation returns offsets for a certain word. The *count* operation counts the appearances of a given word. For each of these operations, we develop sequential and distributed versions, written in C/C++. In addition to these six modules, our solution generates necessary data structures in memory on the fly, such as *blockHashTable*, *blockRefCount*, and *blockHole*, as discussed in Section IV-B.

Interaction With Database Systems: Databases can seamlessly use our processing on compression technology. In detail, we mount a file system in a directory, and then system calls on this directory are handled by CompressDB. If the database system is set to store data in this directory, it can automatically enjoy the benefits of direct computing on compressed data, because CompressDB can handle the system calls like *read* and *write*. We also develop a separate set of APIs for the other operations with no corresponding posix interface (e.g., *insert*, *search*, and *delete*). In our experiment, we call these interfaces and pass parameters and results through unix sockets. In practice, the database can also interact with CompressDB through the unix socket, but the database needs to be modified to adapt to this design.

Applicability: CompressDB is a storage engine whose major application is to support diverse database systems. It seamlessly supports different databases without modifying their code. The only thing users need to do is to set the system directory to that of CompressDB. In general, CompressDB is suitable for analytics and manipulation on data with a large amount of redundancy. For other domains, it may still work, but has not been verified.

VI. Evaluation

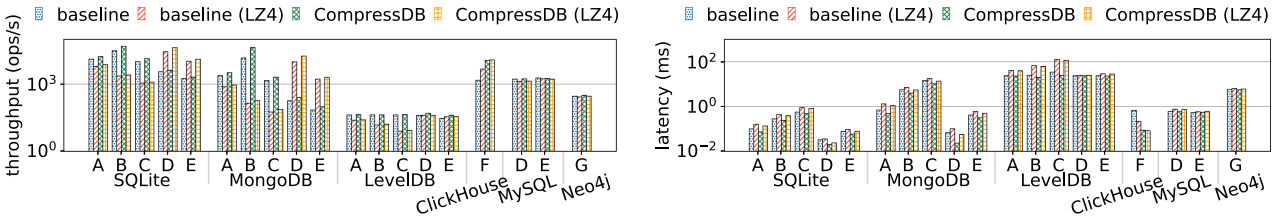
A. Methodology

Evaluated Methods: The baseline used in our evaluation is the original system without CompressDB. For single node evaluation, the baseline refers to the original FUSE [39]. For the distributed environment, the baseline refers to the original MooseFS [17]. We denote the search using linear scan over files in parallel as "baseline", and the baseline with LZ4 compression as "baseline (LZ4)". We develop CompressDB based on existing systems, including FUSE on single node and MooseFS in distributed environment, denoted as "CompressDB". Moreover, we can perform normal compression in CompressDB, which is denoted as "CompressDB (LZ4)". Note that applications need to decompress the data with LZ4 before using the data. Our comparison to baseline examines whether our proposed solution can deliver higher or comparable performance to the normal systems on random updates and accesses. If so, it validates the promise of our solution in making CompressDB the first storage engine that efficiently supports both data query and data manipulation while preserving hierarchically-compressed data direct computing.

Database Systems: We use six common databases in our evaluation, including SQLite [11], MongoDB [14], LevelDB [13], ClickHouse [15], MySQL [12], and Neo4j [16].

Datasets: We use seven datasets in our evaluation, as shown in Table I. The sizes shown in Table I represent the original uncompressed sizes of the files. These datasets are composed of real-world documents of various lengths, structures, and content, and have been widely used in previous studies [2], [3], [4]. Datasets A, B, and C are collections of web documents downloaded from the Wikipedia database [40]. Dataset D is a Wikipedia dataset composed of four large files. Dataset E represents NSF Research Award Abstracts (NSFRAA) dataset downloaded from UCI Machine Learning Repository [41]. Dataset E consists of a large number of small files, and is used to evaluate performance on small files. Dataset F is a real-world structured dataset from an Internet company, which is used for traffic forecasting and intervention. Dataset G is a graph dataset from International Consortium of Investigative Journalists [42].

Benchmark: Datasets A, B, C, D, and E are all unstructured document datasets, and are used to evaluate the performance of databases, including SQLite, MySQL, LevelDB, and MongoDB. Dataset F is a structured dataset and is employed to evaluate the performance of ClickHouse. Dataset G is used to evaluate the performance of Neo4j. For datasets A to F, we



(a) Throughput. (b) Latency.

Fig. 7. CompressDB in supporting different databases.

randomly generate 500,000 query statements, of which 50% are write and 50% are read. For dataset G, we use Galaxybase benchmark [43] for evaluation. We use statements from different databases to simulate reading and writing. For example, we use the SQL statement *select* for read and the statement *update* for write in SQLite, MySQL and ClickHouse. In LevelDB, we use *Get* and *Put* for read and write respectively. In MongoDB, we use *find_one* and *insert_one* in the pymongo library for read and write.

Platforms: We use two platforms in our experiments. For large datasets A, B, and C, we build a five node cluster in cloud. Each node includes an Intel Xeon 8369HB CPU at 3.30 GHz with 16 GB memory. The hard disk on each node is 400 GB ESSD with 50 thousand IOPS. For small datasets D to G, we use a single node machine equipped with an Intel i9-9900K CPU and 64 GB memory. The hard disk in this machine is WDC WD60EZAZ, a 6 TB WD BLUE 3.5" PC HARD DRIVE with 5400 RPM and 256 MB Cache. The operating system of both platforms is Ubuntu 18.04.5.

B. End-to-End Performance Evaluation

We evaluate the performance of CompressDB upon four popular and diversified databases introduced in Section VI-A.

Throughput: We show the throughput results in Fig. 7(a). On average, the databases using CompressDB achieve 40% throughput improvement over the baseline. We have the following findings. First, CompressDB achieves the highest throughput for the large datasets A, B, and C. For small datasets D, E, F and G, CompressDB (LZ4) or CompressDB can achieve the highest throughput. Second, different databases exhibit various performance behaviors. For example, the throughputs for different datasets are close in MongoDB, but are diverse in SQLite and LevelDB. The reason is that MongoDB is a document-based database, which is more complicated to search for an item by a key. Third, the throughputs of small datasets can be higher than those of large datasets. The reason is that the small datasets are processed on single node, which does not involve extra data transmission overhead between different nodes.

Latency: We show the latency results in Fig. 7(b). We measure the latency by recording the latency of each operation, and then calculating the average latency of each type of operation. On average, the databases using CompressDB achieve 44% latency reduction over the baseline. CompressDB achieves latency benefits in all cases. First, because our storage engine reduces the amount of data read from disk, the data preparation time has been reduced. On average, the latency is 9.41 ms, and the standard deviation is 11.43. The latencies of 90% operations are within 43.56 ms. For tail latencies, 5% operations are more than 55.58 ms. Second, large datasets exhibit higher latency. The reason is that datasets A, B, and C are processed in the distributed environment, which have data transmission overhead. Third, SQLite exhibits the lowest latency. The reason is that we perform data search by its primary key and the data are arranged sequentially in the order of primary key.

C. Evaluation of Individual Operations

We show the throughput for different datasets in Fig. 8. Generally, the operations of CompressDB achieve much higher throughput over the original file system and the performance of different operations varies. In detail, *extract* achieves the highest performance. The reason is that the read operation is usually faster than the write operation. *Append* and *replace* achieve moderate throughputs. The reason is that they can reuse the repeated content with minimal modification. For *replace*, it operates on the compressed DAG structure with the overhead of rule split. *Insert* and *delete* have the relatively low performance. The reason is the relatively high complexities when handling data holes with the *blockHole* data structure, as discussed in Section IV-D.

We have the following findings. First, experiments show that CompressDB significantly outperforms the baseline system. On average, for large datasets A, B, and C, CompressDB achieves 34.41× speedup over the baseline. For small datasets D and E, CompressDB achieves 43.79× higher throughput over the original file system. Second, although the throughput of *insert* and *delete* is relatively moderate, they achieve extremely high speedups, which relates to the benefits of data reuse between data blocks. In the original system, *insert* and *delete* involve massive reads and writes of data blocks. Third, *extract*, *append*, and *replace* achieve moderate speedups, though their throughput is high. The reason is that data in MooseFS are separated into different nodes, which decreases the write benefits.

Fig. 9 shows the latency results of the operations on the cloud. The latency is defined as the duration from the start time to the end time of the operation. On average, CompressDB achieves 93.16 ms latency. The standard deviation is 55.14, and the latencies of 90% operations are within 180.94 ms. For tail latencies, 5% operations are more than 337.83 ms. In detail, *extract* achieves the lowest latency among the datasets. The reason is that *extract* does not involve write operations. *Search* and *count* have the highest latency, which is due to the full range traversal.

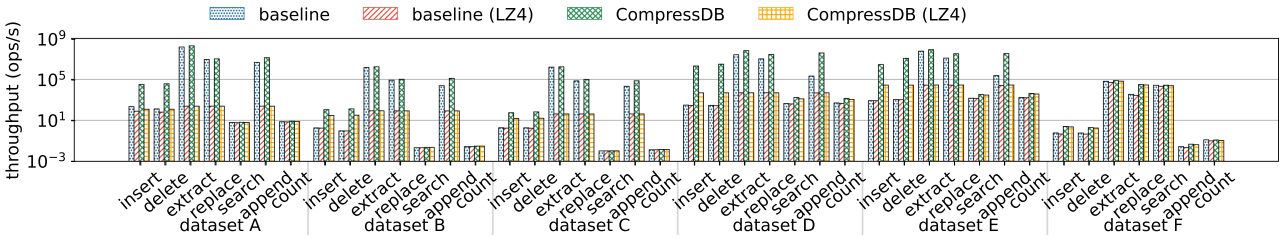


Fig. 8. Throughput of different systems.

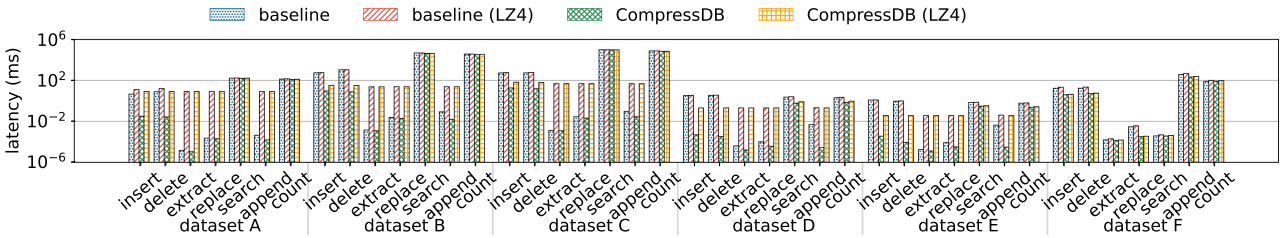


Fig. 9. Latency of different systems.

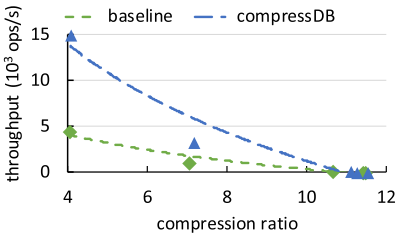


Fig. 10. Improving database capacity.

Interleaving Operations: We measure the influence of interleaving operations on the overall performance. In detail, we mix the seven types of operations, and each type of operation accounts for around 14%. Performing 100,000 mixed random operations, we find that compared with executing each operation independently, the performance of *extract*, *replace*, *search*, *append*, and *count* decreases by 13.57%, 7.36%, 14.17%, 4.41%, and 18.44%, respectively, while *insert* and *delete* remain the same. However, the trend of performance gains for direct computing of compressed data does not change, and 18.82% performance improvement can still be maintained under mixed workloads.

D. Space Savings

Improving Compression in Databases: CompressDB can improve the capability of traditional databases from both time and space perspectives. Fig. 10 shows the performance of baseline and CompressDB under different compression ratios. We can see that CompressDB significantly improves the baseline performance under the same compression ratio, especially when the compression ratio is low. In the case of providing the same performance, CompressDB can also deliver a higher compression ratio.

We also measure the space savings with the metric of compression ratio, which is defined as the size of the original data divided by the size of the compressed data. The default block size is 1024 bytes. The result is shown in Table II. CompressDB achieves 1.75 compression ratio, and CompressDB with LZ4 achieves 9.88 on average, which improves the space saving of LZ4 with 1.85%. We have the following findings. First, CompressDB achieves better results with larger dataset. The reason is that large datasets can involve more redundant data, which provides more compression opportunities. Second, even for small datasets, CompressDB still achieve clear space savings. Third, CompressDB with LZ4 achieves better compression ratio than LZ4, which provides a new chance to enhance compression.

We have the insight that "CompressDB (LZ4)" is a good choice for applications requiring high compression ratio because "CompressDB (LZ4)" can compress all datasets well and offer high throughput on smaller datasets.

TABLE II
COMPRESSION RATIOS FOR SPACE SAVINGS

Dataset	A	B	C	D	E	F	G	AVG
LZ4	10.64	11.45	11.41	11.05	4.03	14.88	4.50	9.70
CompressDB	1.30	1.77	2.58	1.34	1.12	2.80	1.36	1.75
CompressDB (LZ4)	11.11	11.54	11.54	11.48	4.06	14.95	4.51	9.88

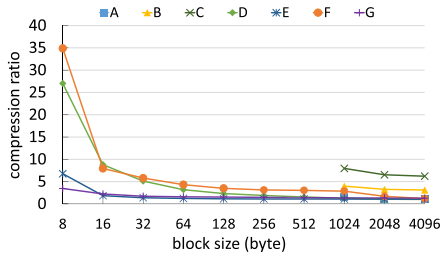


Fig. 11. Block size influence on compression ratio.

E. Design Tradeoffs

Block Size Influence: We analyze the block size influence on compression ratio in this part. Fig. 11 shows the compression ratio of different datasets with different block sizes. For large

datasets A, B, and C, we do not show the compression ratio with small block size due to long measurement time. We observe that smaller block size can increase the compression ratio, since data are more likely to be repeated in the case of small granularity. For small datasets, a small block size is important due to the undesirable compression ratio when the block is large. Please note that in Fig. 11, we have not factored in the reference count overhead for different block sizes. Therefore, when selecting the block size based on compression ratios, it is also essential to ensure that this overhead remains both acceptable and negligible. We utilize a 4-byte integer to store the reference count for each block. Typically, the storage space overhead for reference counting amounts to approximately $4 \times \texttt{capacity/block\_size}$. For instance, when using a block size of 1,024 bytes, this results in an insignificant space overhead of around 0.39%, which we consider negligible.

Theoretical Analysis: We attempt to analyze the relationship between block size and compression efficiency theoretically. Assume that duplicate blocks follow a uniform distribution. With any given block, we denote the probability that another block has identical content as p_n, when block size is n. We believe that p_n mainly relies on the specific dataset for a fixed block size, n. Now, if we adjust the block size to $2 \times n$, combining 2 blocks into a block group, with any given block group, the probability that another block group has identical content is p_n^2. Consequently, as the block size increases, the probability of finding duplicate blocks decreases rapidly, leading to a worse compression ratio with larger block sizes. As discussed in Section IV-C, we employ a hash table in compression. If the block size decreases from n to $\frac{n}{2}$, potentially, twice as many hash table operations are required to handle the same amount of data. Since hash table operations may involve I/Os, the operational overhead becomes unacceptable if the block size becomes too small.

Time versus Space: CompressDB focuses on direct computing on compressed data for various databases, and seeks a balance between compression ratio and processing time, which is beneficial in practice. In contrast, the LZ4 method targets a high compression ratio. Consequently, its compression and decompression processes cause large time overhead, especially on large datasets, making it less practical when dealing with large data. As shown in Fig. 7, "baseline (LZ4)" decreases the performance of the baseline without compression by 52.4% to 88.9% on the large datasets A, B, and C. As for CompressDB, it delivers the highest performance on large datasets, which is 30.1% to 59.8% higher over the performance of "baseline". Although its compression ratio is moderate, it can provide stable performance improvement. As discussed in Section VI-B, for the large datasets A, B, and C, CompressDB can provide the optimal performance. For the small datasets D, E, F and G, CompressDB (LZ4) or CompressDB can achieve the highest performance. The tradeoff between time and space is also affected by the block size. As discussed in the above paragraph, smaller block size can increase the compression ratio. However, small block size also brings large metadata load, massive pointer operations, large hash table, and a large number of rules, which incurs time cost. We show the relation between space overhead and performance exchange of CompressDB in Fig. 12. The space overhead refers

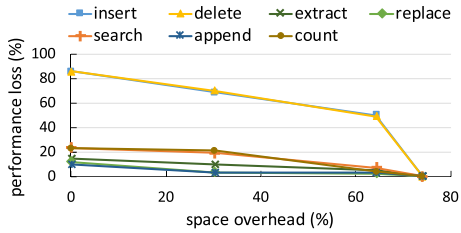


Fig. 12. Tradeoff between space and performance.

TABLE III
MEMORY CONSUMPTION

Dataset	A	B	C	D	E	F	G
HashTableSize (MB)	1752.23	3813.80	5311.75	69.09	22.90	388.22	65.10
DatasetSize	50GB	150GB	300GB	2.1GB	580MB	26GB	1.9GB

to the extra space overhead compared to the storage size with the 1 KB granularity. The performance exchange is denoted as the ratio difference from the optimal to the test value. Fig. 12 shows that we can trade performance for space, and vice versa.

Space Wasted in Data Holes: The design of *blockHole* brings high speed of operations but causes memory overhead. We add an analysis of space-wastage of different data update patterns, including 1) 100% *insert*, 2) 30% *insert* with 70% *delete*, 3) 50% *insert* with 50% *delete*, 4) 70% *insert* with 30% *delete*, and 5) 100% *delete*. These two types of operations mainly influence the generation of data holes. We generate ten thousand operations for each pattern analysis, and each operation manipulates 128 bytes on average. Experiments show that their average ratios of space-wastage are 0.043%, 0.026%, 0.025%, 0.024%, and 0.027%, respectively.

Memory Consumption: We evaluate and show the memory overhead of hash table for different datasets in Table III. Note that for the large datasets of B and C, we only consider the memory consumption on single machine. We find that the *blockHashTable* memory consumption is negligible, which occupies less than 2% of the original size of the datasets.

As for block hole, the memory space occupied by each block hole does not exceed the space it occupies on the disk. On average, assuming each *insert* and *delete* operation utilizes half a block (typically 512 bytes), this significantly surpasses the memory usage of each hole (32 bytes). As discussed in the space wasted in data holes, the disk space wasted on holes is minimal. Therefore, the memory overhead caused by block holes should also be minimal.

Depth of Rules: We explore the depths from 2 to 10 for different datasets and experiments show that 4 can be the optimal in most cases. The reason is that deep levels of DAG can incur high latency to locate the required content. The time complexity of locating data is $O(d)$, where d represents the depth. We find the time used on locating 2^{20} blocks can increase about 10 ms each time the depth increases by one. For comparison, performing *extract* operations on 2^{20} blocks consumes 55 ms with depth set to 4. This means that the locating overhead can be unacceptable if the levels are too deep. In contrast, shallow levels of depth cannot reduce data redundancy efficiently. Besides,

since we choose the block as the basic element in DAG, the block size limits the maximum number of a node's children.

Hash Table Rebuild: During the mounting process, we do not rebuild the hash table. Instead, we create a new empty hash table. Reference count and hole locations are stored both in memory and on disk, ensuring that CompressDB maintains data consistency with all existing compressed files. Although using an empty hash table may affect the compression effectiveness potentially, it greatly improves the mount time. Note that in all our experiments, we assume that the file system is mounted once and will not be remounted.

F. Detailed Analysis

Comparison With LSM Method: Prior compression methods can use LSM-style (log-structured merge tree) method to periodically merge and compress data. In this part, we explore the additional advantage of the proposed method with the LSM method. We use LevelDB for illustration. LevelDB uses LSM-style for data format, whose compression is orthogonal to CompressDB, so they can work together. Experiments show that if we enable default compression on LevelDB (Snappy Compression), "CompressDB+LevelDB" can provide extra 23.8% performance improvement on random reads, 5.3% on random writes, and 10.8% space savings, compared with "Baseline+LevelDB". If we disable default compression on LevelDB, "CompressDB+LevelDB" can provide extra 18.3% performance improvement on random reads, 16.7% on random writes, and 24% space savings, compared with "Baseline+LevelDB".

Comparison With Succinct: Succinct [1] is a data store supporting queries directly on the compressed data. Succinct supports *extract*, *count*, and *search* operations, but it does not support data manipulation operations, such as *insert*, *delete*, and *update*. Note that CompressDB works at storage space, while Succinct works at userspace. Hence, they are orthogonal and can work together. In detail, we can set the storage directory of Succinct to CompressDB, as discussed in Section V. Accordingly, Succinct can compress data and write the Succinct-compressed data into CompressDB. Then, CompressDB can further compress the data, and provide direct computing on these data. When we compare Succinct and CompressDB separately, experiments show that CompressDB can provide 40.4× faster *extract* and 1.9× faster *search*, but 90% slower *count*. The reason is that Succinct involves array of suffixes, which can calculate the occurrence of a string efficiently without traversing the compressed file. When we use them together, "CompressDB+Succinct" delivers 33%, 43%, and 3% performance improvements on *extract*, *count*, and *search*, along with 23.9% space savings, compared with Succinct.

VII. Related Work

Compression in Traditional Databases: Compression is one of the hot research topics in data science domain [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64]. Compression techniques in traditional databases are mainly for storing more data, not for faster data access [65], [66]. Under the premise of given system resources, such as storage space, more data can be stored in database systems [66], [67], [68], [69], [70]. Most compression techniques in databases depend on context [68], [71], [72], [73], [74]. The block/page size of the long-term IO-optimized B-Tree after disk optimization is relatively large, which enables the traditional data compression algorithm to obtain a relatively large context. Therefore, the compression techniques based on block/page granularity, including Snappy [71], lz4 [68], gzip [72], bzip2 [73], and zstd [74], have been applied to various Big Data management systems [67], [75], such as Hadoop, Spark, Oracle Database, SQL Server, and IBM DB2. However, when compression is enabled, the data processing speed drops with it. Although database systems can deliver a dedicated cache to cache the decompressed data to improve the access performance of hot data, this incurs another problem of caching space overhead. Worse still, when the cache misses, the entire block still has to be decompressed. Fortunately, our solution can address the above problem. CompressDB enables database systems enjoy both space savings and time savings due to data reuse.

Data Processing Directly on Compression: Classic data analytics on compression are built on suffix trees and arrays [8], [9], [50], [76], [77], [78], [79], [80]. Suffix trees [5], [81] consume less storage space, but with larger memory consumption [82], [83]. Burrows-Wheeler Transform [6], [7] and suffix arrays [84] provide more compact compression format, but still have memory consumption issue [82]. FM-indexes [7], [85], [86], [87], [88] and compressed suffix array [89], [90], [91], [92], [93] are more efficient alternatives, and Succinct [1] supports database queries on compressed data. Different from these works, CompressDB can be integrated into file systems with both random update and access support, which provides a much wider application scope. Another type of compression strategy is based on grammatical compression, so operations are directly built on grammar encoded strings [2], [4], [18], [19], [94], [95], [96], [97], [98], [99], [100], [101], [102]. Different from these works, CompressDB focuses on how to support general update and access operations on compressed data in Big Data systems. Moreover, the concept behind CompressDB has versatile applications extending beyond its original domain. It can be effectively applied in diverse areas such as stream processing [103], graph analytics [104], and heterogeneous computing [105].

VIII. Conclusion

We develop a novel storage engine, called CompressDB, for enabling random updates directly on compressed data. Specifically, we integrate CompressDB to file systems, which can support various database systems. Moreover, CompressDB can push down the operations to the storage layer. We discuss in detail how the idea of random updates on compressed data be materialized, and prepare a comprehensive experimental analysis to show the advantages of CompressDB. Experiments show that CompressDB significantly improves the performance of common database systems, and saves space at the same time.

REFERENCES

[1] R. Agarwal et al., "Succinct: Enabling queries on compressed data," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 337–350.

[2] F. Zhang et al., "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," in *Proc. VLDB Endowment*, vol. 11, pp. 1522–1535, 2018.

[3] F. Zhang et al., "TADOC: Text analytics directly on compression," *The VLDB J.*, vol. 30, no. 2, pp. 163–188, 2021.

[4] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Enabling efficient random access to hierarchically-compressed data," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1069–1080.

[5] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge, U.K.: Cambridge Univ. Press, 2016.

[6] M. Burrows et al., "A block-sorting lossless data compression algorithm," *SRS Res. Rep.*, vol. 124, 1994.

[7] P. Ferragina et al., "Indexing compressed text," *J. ACM*, vol. 52, pp. 552–581, 2005.

[8] R. Grossi et al., "When indexing equals compression: Experiments with compressing suffix arrays and applications," in *Proc. 15th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2004, pp. 636–645.

[9] P. Ferragina et al., "Compressed text indexes: From theory to practice," *J. Exp. Algorithmics*, vol. 13, 2009, Art. no. 12.

[10] F. Zhang et al., "CompressDB: Enabling efficient compressed data direct processing for various databases," in *Proc. Int. Conf. Manage. Data*, 2022, pp. 1655–1669.

[11] SQLite Home Page, 2021. [Online]. Available: https://www.sqlite.org/index.html

[12] MySQL, 2021. [Online]. Available: https://www.mysql.com/

[13] S. Ghemawat et al., "LevelDB," 2021. [Online]. Available: https://github.com/google/leveldb

[14] MongoDB, 2021. [Online]. Available: https://www.mongodb.com/2

[15] ClickHouse Home Page, 2021. [Online]. Available: https://clickhouse.tech/

[16] Neo4j Home Page, 2021. [Online]. Available: https://neo4j.com/

[17] Moosefs, 2021. [Online]. Available: https://moosefs.com/

[18] S. Maneth, *Grammar-Based Compression*. Cham, Switzerland: Springer International Publishing, 2018, pp. 1–8.

[19] S. Sakr et al., *Encyclopedia of Big Data Technologies*. Berlin, Germany: Springer International Publishing, 2019.

[20] C. McAnlis et al., *Understanding Compression: Data Compression for Modern Developers*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2016.

[21] M. Drosou et al., "Diversity in big data: A review," *Big Data*, vol. 5, pp. 73–84, 2017.

[22] J. Abawajy, "Comprehensive analysis of big data variety landscape," *Int. J. Parallel Emergent Distrib. Syst.*, vol. 30, no. 1, pp. 5–14, 2015.

[23] S. Allam, "Components and development in big data system: A survey," *Int. J. Innov. Eng. Res. Technol.*, vol. 4, no. 9, pp. 28–35, 2017.

[24] A. Moniruzzaman et al., "NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison," 2013, *arXiv:1307.0191*.

[25] C. G. Nevill-Manning et al., "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Intell. Res.*, vol. 7, pp. 67–82, 1997.

[26] P. Francisco et al., "The Netezza data appliance architecture: A platform for high performance data warehousing and analytics," 2011.

[27] S. Fushimi et al., "An overview of the system software of a parallel relational database machine GRACE," in *Proc. 12th Int. Conf. Very Large Data Bases*, 1986, pp. 209–219.

[28] M. Ubell, "The intelligent database machine (IDM)," in *Query Processing in Database Systems*. Berlin, Germany: Springer, 1985.

[29] S. Melnik et al., "Dremel: Interactive analysis of web-scale datasets," in *Proc. VLDB Endowment*, vol. 3, pp. 330–339, 2010.

[30] Y. Yang et al., "FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS," in *Proc. VLDB Endowment*, vol. 14, pp. 2101–2113, 2021.

[31] X. Yu et al., "PushdownDB: Accelerating a DBMS using S3 computation," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1802–1805.

[32] Cloud Data Warehouse - Amazon Redshift, 2022. [Online]. Available: https://aws.amazon.com/redshift/

[33] T. R. Kepe et al., "Database processing-in-memory: An experimental study," in *Proc. VLDB Endowment*, vol. 13, pp. 334–347, 2019.

[34] O. O. Babarinsa et al., "JAFAR: Near-data processing for databases," in *Proc. Int. Conf. Manage. Data*, 2015, pp. 2069–2070.

[35] S. L. Xi et al., "Beyond the wall: Near-data processing for databases," in *Proc. 11th Int. Workshop Data Manage. New Hardware*, 2015, pp. 1–10.

[36] J. Lee et al., "ExtraV: Boosting graph processing near storage with a coherent accelerator," in *Proc. VLDB Endowment*, vol. 10, pp. 1706–1717, 2017.

[37] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 531–543.

[38] D. E. Knuth et al., "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, pp. 323–350, 1977.

[39] FUSE – The Linux Kernel documentation, 2020. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/fuse.html

[40] Wikipedia HTML data dumps, 2017. [Online]. Available: https://dumps.wikimedia.org/enwiki/

[41] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[42] Panama Papers by ICIJ, 2021. [Online]. Available: https://github.com/neo4j-graph-examples/icij-panama-papers

[43] galaxybase benchmark, 2021. [Online]. Available: https://github.com/galaxybase/graph-database-benchmark

[44] H. Yin et al., "Efficient trajectory compression and range query processing," *World Wide Web*, vol. 25, pp. 1259–1285, 2022.

[45] W. Fan et al., "Query preserving graph compression," in *Proc. Int. Conf. Manage. Data*, 2012, pp. 157–168.

[46] J. Li et al., "Compressing information of target tracking in wireless sensor networks," *Wireless Sensor Netw.*, vol. 3, pp. 73–81, 2011.

[47] J. Li et al., "Data sampling control, compression and query in sensor networks," *Int. J. Sensor Netw.*, vol. 2, pp. 53–61, 2007.

[48] Q. Ren, J. Li, and J. Li, "An efficient clustering-based method for data gathering and compressing in sensor networks," in *Proc. 8th ACIS Int. Conf. Softw. Eng. Artif. Intell. Netw. Parallel/Distrib. Comput.*, 2007, pp. 823–828.

[49] J. Li et al., "Data sampling control and compression in sensor networks," in *Proc. Int. Conf. Mobile Ad-Hoc Sensor Netw.*. Springer, 2005, pp. 42–51.

[50] J. Li et al., "Aggregation algorithms for very large compressed data warehouses," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 651–662.

[51] S. Gao, F. Huang, J. Pei, and H. Huang, "Discrete model compression with resource constraint for deep neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 1896–1905.

[52] C. Yang et al., "A spatiotemporal compression based approach for efficient big data processing on cloud," *J. Comput. Syst. Sci.*, vol. 80, no. 8, pp. 1563–1583, 2014.

[53] W. Liu et al., "On compressing weighted time-evolving graphs," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage.*, 2012, pp. 2319–2322.

[54] H. Maserrat and J. Pei, "Community preserving lossy compression of social networks," in *Proc. IEEE 12th Int. Conf. Data Mining*, 2012, pp. 509–518.

[55] H. Maserrat et al., "Neighbor query friendly compression of social networks," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2010, pp. 533–542.

[56] J. Pei, A. W.-C. Fu, X. Lin, and H. Wang, "Computing compressed multidimensional skyline cubes efficiently," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, 2007, pp. 96–105.

[57] Y. Chen, G. Dong, J. Han, J. Pei, B. W. Wah, and J. Wang, "Regression cubes with lossless compression and aggregation," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 12, pp. 1585–1599, Dec. 2006.

[58] M. Stonebraker et al., "C-store: A column-oriented DBMS," in *Proc. Mak. Databases Work: Pragmatic Wisdom Michael Stonebraker*, 2018, pp. 491–518.

[59] C. Liu et al., "Decomposed bounded floats for fast compression and queries," in *Proc. VLDB Endowment*, vol. 14, pp. 2586–2598, 2021.

[60] S. Deep et al., "Comprehensive and efficient workload compression," in *Proc. VLDB Endowment*, vol. 14, pp. 418–430, 2020.

[61] T. Li et al., "Trace: Real-time compression of streaming trajectories in road networks," in *Proc. VLDB Endowment*, vol. 14, pp. 1175–1187, 2021.

[62] A. Shanbhag et al., "Tile-based lightweight integer compression in GPU," in *Proc. Int. Conf. Manage. Data*, 2022, pp. 1390–1403.

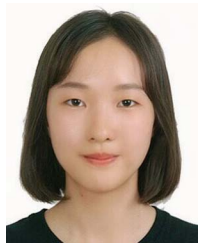
[63] H. Zhang et al., "Order-preserving key compression for in-memory search trees," in *Proc. Int. Conf. Manage. Data*, 2020, pp. 1601–1615.

[64] Q. Yin et al., "An adaptive elastic multi-model big data analysis and information extraction system," *Data Sci. Eng.*, vol. 7, pp. 328–338, 2022.

[65] G. Graefe et al., *Data Compression and Database Performance*. University of Colorado, Boulder, Department of Computer Science, 1990.
[66] M. A. Roth et al., "Database compression," *ACM SIGMOD Rec.*, vol. 22, pp. 31–39, 1993.
[67] S. Aghav, "Database compression techniques for performance optimization," in *Proc. Int. Conf. Comput. Eng. Technol.*, 2010, pp. V6–714–V6-717.
[68] W.-K. Ng and C. V. Ravishankar, "Block-oriented compression techniques for large statistical databases," *IEEE Trans. Knowl. Data Eng.*, vol. 9, no. 2, pp. 314–328, Mar./Apr. 1997.
[69] P. Hansert et al., "Ameliorating data compression and query performance through cracked Parquet," in *Proc. Int. Workshop Big Data Emergent Distrib. Environ.*, 2022, pp. 1–7.
[70] T. Siddiqui et al., "ISUM: Efficiently compressing large and complex workloads for scalable index tuning," in *Proc. Int. Conf. Manage. Data*, 2022, pp. 600–673.
[71] Snappy, "A fast compressor/decompressor," 2020. [Online]. Available: https://github.com/google/snappy
[72] P. Deutsch et al., "Rfc1952: Gzip file format specification version 4.3," 1996.
[73] M. Jakoby et al., "bZIP transcription factors in Arabidopsis," *Trends Plant Sci.*, vol. 7, no. 3, pp. 106–111, 2002.
[74] zstd, 2021. [Online]. Available: https://github.com/facebook/zstd
[75] B. Bhattacharjee et al., "Efficient index compression in DB2 LUW," in *Proc. VLDB Endowment*, vol. 2, pp. 1462–1473, 2009.
[76] A. Farruggia et al., "Bicriteria data compression: Efficient and usable," in *Proc. Eur. Symp. Algorithms*, 2014, pp. 406–417.
[77] P. Ferragina et al., "On the bit-complexity of Lempel-Ziv compression," in *Proc. 20th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2009, pp. 768–777.
[78] S. Gog et al., "From theory to practice: Plug and play with succinct data structures," in *Proc. Int. Symp. Exp. Algorithms*, 2014, pp. 326–337.
[79] J. Li et al., "A new compression method with fast searching on large databases," in *Proc. 13th Int. Conf. Very Large Data Bases*, 1987, pp. 311–318.
[80] J. Li and J. Srivastava, "Efficient aggregation algorithms for compressed data warehouses," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 3, pp. 515–529, May/Jun. 2002.
[81] K. Sadakane, "Compressed suffix trees with full functionality," *Theory Comput. Syst.*, vol. 41, pp. 589–607, 2007.
[82] W.-K. Hon et al., "Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences," in *Proc. ALENEX/ANALC*, 2004, pp. 31–38.
[83] S. Kurtz, "Reducing the space requirement of suffix trees," *Softw. Pract. Exp.*, vol. 29, pp. 1149–1171, 1999.
[84] U. Manber et al., "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, pp. 935–948, 1993.
[85] FM-index, 2018. [Online]. Available: https://en.wikipedia.org/wiki/FM-index
[86] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41st Annu. Symp. Found. Comput. Sci.*, 2000, pp. 390–398.
[87] P. Ferragina et al., "An experimental study of a compressed index," *Inf. Sci.*, vol. 135, pp. 13–28, 2001.
[88] P. Ferragina et al., "An experimental study of an opportunistic index," in *Proc. 12th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2001, pp. 269–278.
[89] R. Grossi et al., "High-order entropy-compressed text indexes," in *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2003, pp. 841–850.
[90] R. Grossi et al., "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM J. Comput.*, vol. 35, pp. 378–407, 2005.
[91] K. Sadakane, "Compressed text databases with efficient query algorithms based on the compressed suffix array," in *Proc. Int. Symp. Algorithms Comput.*, 2000, pp. 410–421.
[92] K. Sadakane, "Succinct representations of LCP information and improvements in the compressed suffix arrays," in *Proc. 13th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2002, pp. 225–232.
[93] K. Sadakane, "New text indexing functionalities of the compressed suffix arrays," *J. Algorithms*, vol. 48, pp. 294–313, 2003.
[94] W. Rytter, "Grammar compression, LZ-encodings, and string algorithms with implicit input," in *Proc. Int. Colloq. Automata Lang. Program.*, 2004, pp. 15–27.
[95] M. Charikar et al., "The smallest grammar problem," *IEEE Trans. Inf. Theory*, vol. 51, no. 7, pp. 2554–2576, Jul. 2005.
[96] T. Gagie et al., "A faster grammar-based self-index," in *Proc. Int. Conf. Lang. Automata Theory Appl.*, 2012, pp. 240–251.
[97] P. Bille et al., "Random access to grammar-compressed strings and trees," *SIAM J. Comput.*, vol. 44, pp. 513–539, 2015.
[98] P. Bille et al., "Finger search in grammar-compressed strings," *Theory Comput. Syst.*, vol. 62, pp. 1715–1735, 2018.
[99] N. R. Brisaboa et al., "GraCT: A grammar-based compressed index for trajectory data," *Inf. Sci.*, vol. 483, pp. 106–135, 2019.
[100] M. Ganardi, A. Jeż, and M. Lohrey, "Balancing straight-line programs," in *Proc. IEEE Annu. Symp. Found. Comput. Sci.*, 2019, pp. 1169–1183.
[101] Y. Takabatake et al., "A space-optimal grammar compression," in *Proc. 25th Annu. Eur. Symp. Algorithms*, vol. 87, Schloss Dagstuhl Leibniz-Zentrum fuer Informatik, 2017, pp. 67:1–67:15.
[102] A. Shanbhag et al., "Tile-based lightweight integer compression in GPU," in *Proc. Int. Conf. Manage. Data*, 2022, pp. 1390–1403.
[103] Y. Zhang, F. Zhang, H. Li, S. Zhang, and X. Du, "CompressStreamDB: Fine-grained adaptive stream processing without decompression," in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 408–422.
[104] Z. Chen et al., "CompressGraph: Efficient parallel graph analytics with rule-based compression," in *Proc. Int. Conf. Manage. Data*, vol. 1, no. 1, 2023, pp. 1–31.
[105] F. Zhang et al., "Optimizing random access to hierarchically-compressed data on GPU," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 1–15.

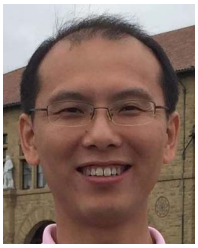
Weitao Wan received the bachelor's degree from the Renmin University of China, in 2022. He is now working toward the master's degree in computer science with Tsinghua University, and was a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE). His current research interests include cloud computing and database systems.

Feng Zhang (Member, IEEE) received the bachelor's degree from Xidian University, in 2012, and the PhD degree in computer science from Tsinghua University, in 2017. He is a professor with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. His major research interests include DBMS, data compression, and parallel and distributed systems.



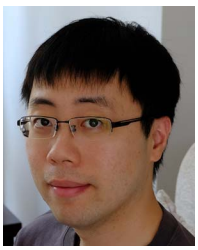
Chenyang Zhang received the bachelor degree from the Renmin University of China, in 2022. She is currently working toward the PhD degree in computer science with Tsinghua University, and was a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE). Her current research interests include parallel computing, heterogeneous computing, and database systems.

Mingde Zhang received the bachelor's degree from the Renmin University of China, in 2023, and he is currently working toward the PhD degree in computer science with the Renmin University of China. His current research interests include parallel computing, heterogeneous computing, and database systems.



Jidong Zhai received the BS degree in computer science from the University of Electronic Science and Technology of China in 2003, and the PhD degree in computer science from Tsinghua University, in 2010. He is a professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include performance evaluation for high performance computers, performance analysis and modeling of parallel applications.

Yunpeng Chai (Member, IEEE) received the BS and PhD degrees in computer science and technology from Tsinghua University, Beijing, China, in 2004 and 2009, respectively. He is currently a professor with the School of Information, Renmin University of China. His research interests include key-value storage systems, the emerging storage devices, and cloud resource allocation.



Huanchen Zhang received the BS degree in computer engineering from the University of Wisconsin - Madison, and the PhD degree from the Computer Science Department, Carnegie Mellon University. He is currently an assistant professor with the IIIS (Yao Class), Tsinghua University. Before joining Tsinghua, he worked with Snowflake as a postdoctoral research fellow. His research interest is in database management systems.

Wei Lu received the PhD degree in computer science from the Renmin University of China, in 2011. He is a professor with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. His research interests include query processing in the context of spatiotemporal, cloud database systems, and applications.

Yuxing Chen received the PhD degree in computer science from the University of Helsinki, Finland, in 2021. He currently works as a senior research engineer with the Database R&D Department, Tencent, China. His research interests focus on database performance and evaluation, HTAP database design, and distributed system design.

Haixiang Li is a senior expert with Tencent. His research interests include transaction processing, query optimization, distributed consistency, high availability, database system architecture, cloud database, and distributed database systems.

Anqun Pan is the technical director of the Database R&D Department, Tencent in China. With more than 15 years of experience, he has specialized in the research and development of distributed computing and storage systems. Currently, he is responsible for steering the research and development of the Tencent distributed database system (TDSQL).

Xiaoyong Du (Member, IEEE) received the BS degree from Hangzhou University, Zhejiang, China, in 1983, the ME degree from the Renmin University of China, Beijing, China, in 1988, and the PhD degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.