CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases

Feng Zhang[◇], Weitao Wan[◇], Chenyang Zhang[◇], Jidong Zhai[★], Yunpeng Chai[◇], Haixiang Li[+], Xiaoyong Du[◇]
[◇]Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China
[★]Department of Computer Science and Technology, Tsinghua University
[+]Tencent Inc., China
fengzhang@ruc.edu.cn,wanweitao@ruc.edu.cn,chenyangzhang@ruc.edu.cn,zhaijidong@tsinghua.edu.cn
ypchai@ruc.edu.cn,blueseali@tencent.com,duyong@ruc.edu.cn

ABSTRACT

In modern data management systems, directly performing operations on compressed data has been proven to be a big success facing big data problems. 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.

We develop a new storage engine, called CompressDB, which 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, LevelDB, MongoDB, and ClickHouse. We evaluate our method using six 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 1.81 compression ratio on average.

CCS CONCEPTS

• **Information systems** → **Data compression**; **Database management system engines**; • **Theory of computation** → Data compression.

KEYWORDS

compression, compressed data direct processing, database systems

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526130

ACM Reference Format:
Feng Zhang[◇], Weitao Wan[◇], Chenyang Zhang[◇], Jidong Zhai[★], Yunpeng Chai[◇], Haixiang Li[+], Xiaoyong Du[◇] . 2022. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526130

1 INTRODUCTION

Modern big data systems are facing an exponentially growing data volume and use data compression to reduce the storage footprint. 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 [13, 72, 98–103, 105]. These systems have demonstrated significant compression ratio and performance improvement at the same time for data analytics applications.

While existing solutions have shown great potential on *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 [13] is a database supporting queries over compressed data; the compression technique is based on index and suffix array [13, 21, 26, 30, 43, 68] 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) [101, 102, 105], which achieves a similar goal as Succinct but uses a rule-based compression strategy. TADOC uses a rule to represent content that appears multiple times.

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. 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 [105], 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-based 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.

We validate the efficacy of CompressDB by supporting various kinds of database systems, including SQLite [10], LevelDB [37], MongoDB [8], and ClickHouse [5]. We evaluate our method using six 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 [9], 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 1.81 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.

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

2 PRELIMINARY

After a quick scan at these algorithms, we find that grammar-based compression [64, 65, 85] is naturally suitable for random update directly on compressed data. We use TADOC [101], a representative rule based compression, for illustration.

2.1 Rule-Based Compression

TADOC is a novel rule-based solution for compression-based direct processing [101–105], which can be explained from three levels: elements, rules, and DAG.

- *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.
- *Rule: String of elements.* TADOC uses a rule to represent repeated content, and a rule consists of subrules and data units.
- *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 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 Figure 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 Figure 1 (b). In Figure 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 Figure 1 (c).

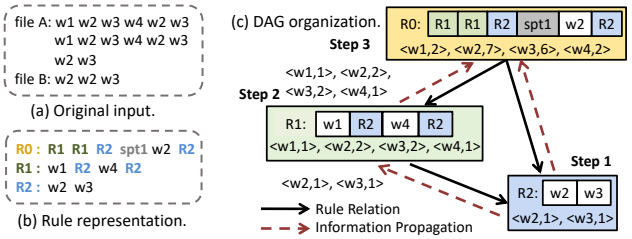


Figure 1: An example of TADOC compression.

Data analytics. Based on the rule representation, we can perform common data analytics directly on the rule compressed data [103]. 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 Figure 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 [103, 105] for data query, which is essential for data analytics systems. Zhang et al. [103] 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 1KB or 4KB, 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 2GB dataset detailed in Section 6.1) 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.

2.2 Motivation

Idea. 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. However, in developing our idea, we meet the following three challenges.

Challenge in element level: Complexities in using large element granularity. Previous TADOC processes data at word granularity, while the storage systems usually organize data at a much larger block granularity, such as 1KB or 4KB. Simply increasing the processing granularity can reduce the compression effect, because two large data blocks can have part of the same data. Worse still, when misalignment occurs, even two pieces of the same data cannot be expressed by the same rule.

Challenge in rule level: Operating rules for random update. 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 [103]. Assume the depth of the rules is d and the average number of parent nodes of each point is n, which relates to the redundancy between rules. Then, the complexity of recursive rule split for random update in a rule is $O(n^d)$. In practice, even for a small file of 2GB (dataset D in Table 1), d can reach 12 and n can reach 1,211,546. Hence, the higher the depth of the rules, the greater the overhead of updating the compressed data.

Challenge in DAG level: Performance maintenance of real-time operation on the DAG in storage layer. 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.

3 DESIGN OVERVIEW

CompressDB: We develop a new storage engine, called CompressDB, which can perform data processing for databases without decompression.
Idea: Our basic idea is 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 2.1. 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 2.2. 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 used in [105] 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 4.1.

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 [69], 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.

4 SYSTEM DESIGN

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

4.1 Overview of System Modules

We show the overview of our storage engine, CompressDB, in Figure 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.

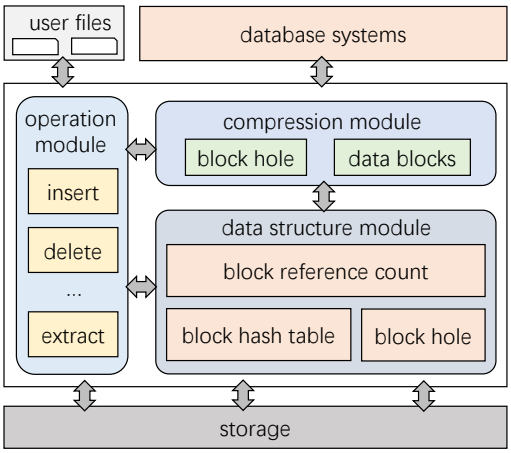


Figure 2: Overview of CompressDB.

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 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 processing 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 [32, 33, 91], cloud databases [12, 66, 95, 96], SSD [24, 36, 45, 52, 92, 94], and memory [38, 50]. 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 [15, 93]. 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 [54, 88]. However, these specialties have not been considered in previous TADOC research [101–103, 105].

Updating parent nodes. One prominent feature of CompressDB is its simplicity in updating parents. By limiting the number of parents of non-leaf nodes to one, we reduce the complexity of updating parents from $O(n^d)$, as discussed in Section 2.2, 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 Figure 3 for illustration. Figure 3 (a) shows the original data, while Figure 3 (b) shows the compressed state, assuming *db*2 and *db*5 contain the same content. Figure 3 (c) shows an insert example. Assume we insert content in db_3. We need to allocate a new db_3 first and then copy the content from db_2 before insertion. Note that the parents of the new block also need to be split. Accordingly, we recursively update its parents (the ptr page) since the whole segment (ptr'1, ptr'2 and ptr'3) has been changed. Fortunately, the depth in the file tree is not deep so the block split cost is marginal. Figure 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. Figure 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. Figure 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 4.2), compressor (Section 4.3), and operation pushdown (Section 4.4).

4.2 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 compressor module and the related operations that need to quickly locate content such as *search* in the operation pushdown module. Note that the hash value is determined by the content of the block. Accordingly, we can check whether there is a previous block with the same content by hashing the input block. 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 them 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.

4.3 Compression Module

In this part, we show the design of our 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 4.4).

Case study. We show an example of data compression in Figure 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, Figure 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.

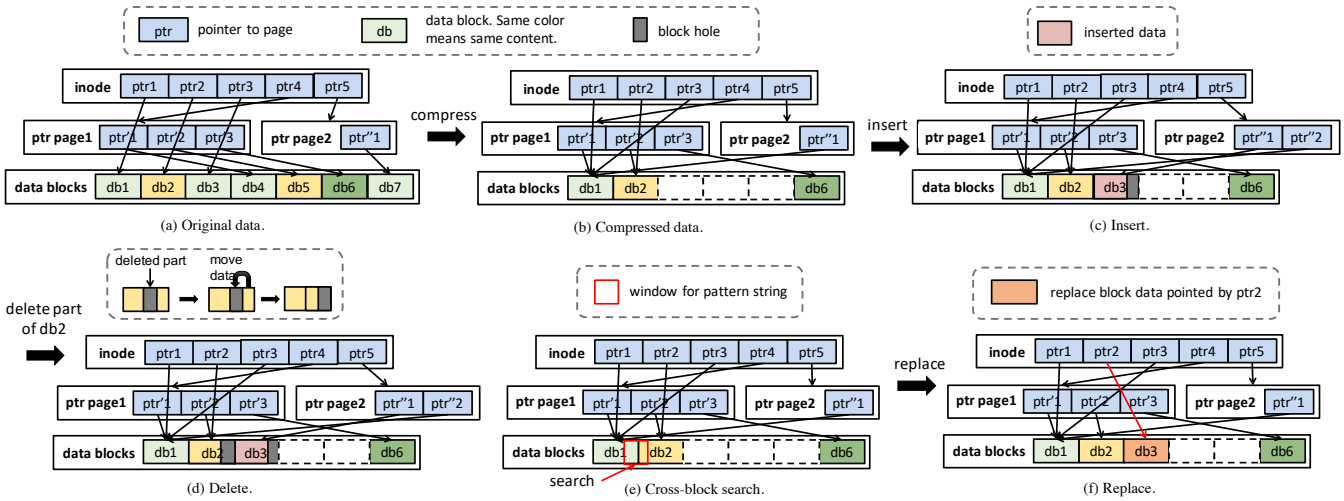


Figure 3: Illustration for compression and random updates.

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 Figure 3 (b).

We identify and merge the redundant blocks by letting their pointers point to only one block. In Figure 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 Figure 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 before 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. We in this part 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 bit 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, we can see that there is no pointer pointing to the current block so the current block can be released (Lines 3 to 4). Otherwise, we need to subtract one from the reference count and set the current pointer to the repeated block (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 are 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).

Algorithm 1: Real-time 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);
```

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 $O(n)$ time, where n is the size of the block. Similarly, the complexity for the update in Line 18 is also $O(n)$, while the complexity for the other parts is $O(1)$. Hence, the complexity for Algorithm 1 is $O(n)$.

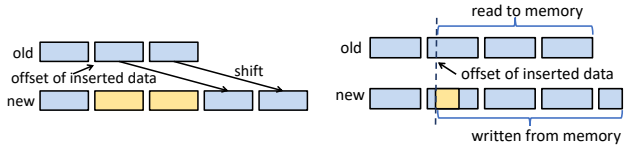
4.4 Operation Module

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.

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 Figure 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 this design.



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

Figure 4: Traditional solution for *insert*.

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 Figure 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 Figure 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 Figure 5 (b), every corresponding block of these two block sequences becomes different. Therefore, the benefits of compression technology are greatly reduced.

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

Figure 5: Loss of compression.

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 4.2, 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 Figure 3 (c). An *insert* is performed in the third data block 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 $ptr3$ from db_1 and db_3, 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.

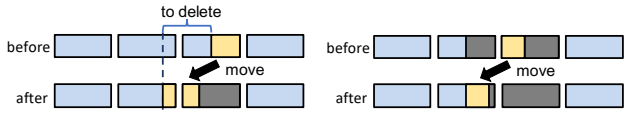
5) Influence of insert *to the other operations.* Next, we illustrate how our solution cooperates with the other operations. Since the holes caused by *insert* exist in the block sequence but with meaningless content, we need to skip these holes when performing the other operations such as *extract* 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 Figure 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 movement operations.

Our design. We develop *delete* with the *blockHole* data structure. We show an example in Figure 6 (a). The deletion can start in one block and end in another block, which are block#1 and block#2 in Figure 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* 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.



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

Figure 6: *Delete* in CompressDB.

Complexity analysis. The complexity of *delete* is influenced by three parts: (1) moving the remained data forward, which is the yellow part in Figure 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 Figure 3 (a). For example, assume the block size is 1024 bytes. Then, the offset 1536 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 Figure 3 (f). Assume that the file has three data blocks and the *replace* target lies in the third data block. The third block has been referenced two times, so we cannot change its content directly. In contrast, we allocate a new data block, set the third pointer to the new one, and then write the modified content to the new block. In Figure 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 rule nodes.

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 final result.

Complexity analysis. The complexity is determined by the string matching algorithm, which is KMP (Knuth Morris Pratt) [51] 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.

5 IMPLEMENTATION

We develop CompressDB in real storage systems, including FUSE [3] and MooseFS [9], for validation. The FUSE system (Filesystem in Userspace) [3] allows us to conveniently construct a file system in user space without caring about the code of the Linux system kernel. MooseFS [9] 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 scenario [58–60]. 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 4.2.

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 processing 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.

6 EVALUATION

We evaluate the efficacy of CompressDB, in terms of both time and space savings. We report performance in both single-node and distributed environments.

6.1 Methodology

In this part, we show the methodology and experimental setup, including the evaluated methods, database systems, datasets, benchmark, and platforms.

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 [3]. For the distributed environment, the baseline refers to the original MooseFS [9]. 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 processing.

Database systems. We use four common databases in our evaluation, including SQLite [10], LevelDB [37], MongoDB [8], and ClickHouse [5]. SQLite [10] is an embedded relational database management system (RDBMS), and it is widely used by web browsers, operating systems, and embedded systems. SQLite is a representative RDBMS and is almost the most widespread database engine. LevelDB [37] is a NoSQL [46] key-value database management system (DBMS) developed by Google and is used as the backend database for Google Chrome's IndexedDB. As a NoSQL database, LevelDB does not apply relational data model or support SQL queries. MongoDB [8] is a NoSQL distributed document DBMS, using flexible JSON-like documents with optional schemas to store data. As a distributed database, it has high availability, horizontal scaling, and geographic distribution. ClickHouse [5] is a fast column-oriented database management system supporting SQL queries in real-time.

Datasets. We use six datasets in our evaluation, as shown in Table 1. The sizes shown in Table 1 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 [101–103, 105]. Datasets A, B, and C are collections of web documents downloaded from the Wikipedia database[1]. 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 [61]. 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.

Table 1: Datasets.

Dataset	Size	File#	Inode#	Block#
A	50GB	109	109	51,389,150
B	150GB	309	309	146,303,136
C	300GB	618	618	292,606,272
D	2.1GB	4	4	2,097,152
E	580MB	134,631	134,631	593,283
F	26GB	36	36	27,329,745

Benchmark. Datasets A, B, C, D, and E are all unstructured document datasets, and are used to evaluate the performance of databases, including SQLite, LevelDB, and MongoDB. Dataset F is a structured dataset and is employed to evaluate the performance of ClickHouse. For each database, we randomly generate 500,000 query statements, of which 50% are write and 50% are read. 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 and ClickHouse. In LevelDB, we use *Get* and *Put* for read and write respectively. In MongoDB, we use the interfaces *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 Platinum 8369HB CPU at 3.30GHz with 16 GB memory. The operating system is Ubuntu 18.04.5. The hard disk on each node is 400GB ESSD with 50 thousand IOPS. For small datasets D and E, along with the structured dataset F, we use a single node machine equipped with an Intel i9-9900K CPU and 64 GB memory. The operating system is Ubuntu 18.04.5. The hard disk in this machine is WDC WD60EZAZ, a 6TB WD BLUE 3.5" PC HARD DRIVE with 5400 RPM and 256 MB Cache.

6.2 End-to-End Performance Evaluation

We evaluate the performance of CompressDB upon four popular and diversified databases introduced in Section 6.1. We evaluate the performance of these systems with and without our storage engine from the throughput and latency perspectives.

Throughput. We show the throughput results in Figure 7. 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, and F, 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

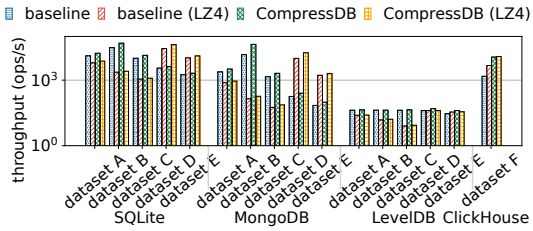


Figure 7: Throughput of CompressDB in supporting different databases.

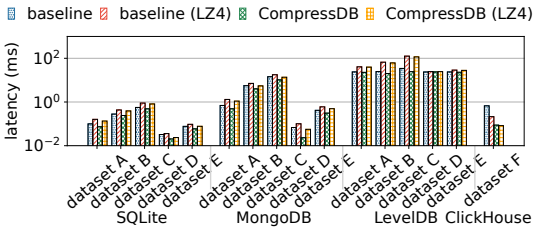


Figure 8: Latency of CompressDB in supporting different databases.

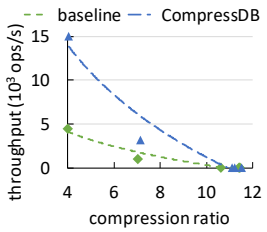


Figure 9: Improving database capacity.

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 Figure 8. 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.

Improving compression in databases. CompressDB can improve the capability of traditional databases from both time and space perspectives. Figure 9 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.

Range scan. Range scan is an important operation in the context of databases. With the help of CompressDB, range scan can also enjoy the benefits of data reuse. In the evaluation, we use the query `"select id, sum(cnt)/count(dt) avg_cnt from tbl where idx >= 0 and idx <= 8 group by id order by avg_cnt desc;"`, which covers range scan. For this query on ClickHouse, CompressDB provides 15.48% performance improvement, and on SQLite, it generates 9.62% performance improvement over the baseline.

6.3 Evaluation of Individual Operations

We show the throughput for different datasets in Figure 10. Generally, the operations of CompressDB achieve much higher throughput over the original file system. 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 that they have relatively high complexities: they need to handle data holes with the *blockHole* data structure, as discussed in Section 4.4.

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.

Figure 11 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.

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 processing of compressed data does not change, and 18.82% performance improvement can still be maintained under mixed workloads.

6.4 Space Savings

We 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 2. CompressDB achieves 1.81 compression ratio, and CompressDB with LZ4 achieves 10.57 on average, which improves the space saving of LZ4 with 2.26%. We have the following findings. First, CompressDB achieves better

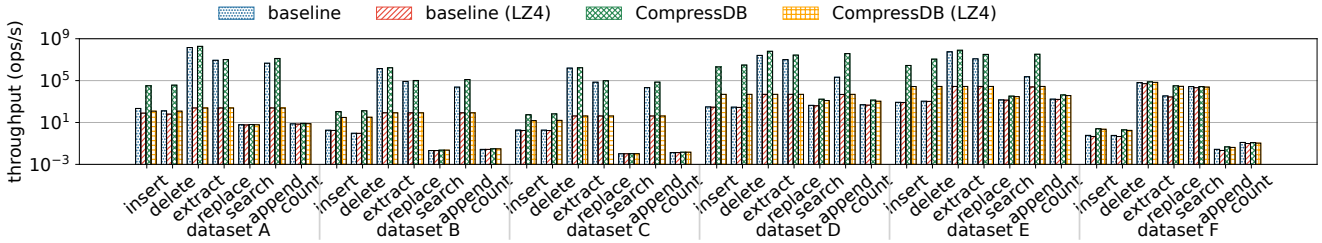


Figure 10: Throughput of different systems.

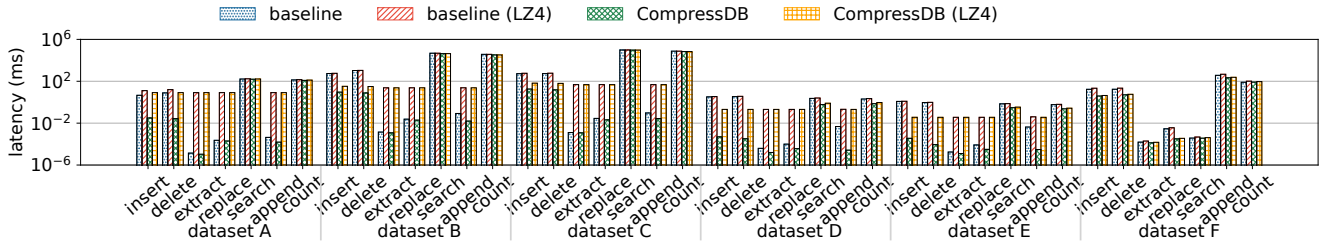


Figure 11: Latency of different systems.

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 system.

Table 2: Compression ratios for space savings.

Dataset	A	B	C	D	E	F	AVG
LZ4	10.64	11.45	11.41	11.05	4.03	14.88	10.57
CompressDB	1.30	1.77	2.58	1.34	1.12	2.80	1.81
CompressDB (LZ4)	11.11	11.54	11.54	11.48	4.06	14.95	10.78

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.

6.5 Detailed Analysis

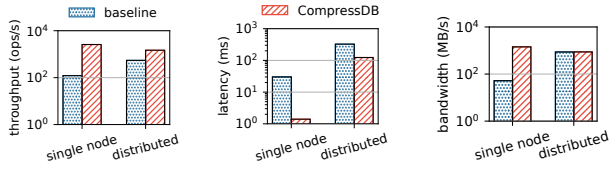
Memory consumption. We analyze the memory consumption of CompressDB in this part. As discussed in Section 4.2, the data structures of *blockHashTable* and *blockHole* are stored in memory for efficiency, which incurs memory cost. We show the memory overhead of them for different datasets in Table 3. The values for *blockHole* refer to the memory consumption for 1 GB changed data, since only data insertion and deletion generate holes. We have the following findings. First, the overall memory consumption is negligible, which occupies less than 2% of the original size of the datasets. Second, *blockHashTable*, which is used for fast data lookup, incurs the largest overhead. Third, the memory overhead for *blockHole* is marginal.

CPU utilization. The CPU utilization for CompressDB is between 0.06 and 0.26 out of 16 processors due to different workloads, mostly occupying no more than one processor. For comparison, the CPU utilization could be up to 0.21 out of 16 processors if two processes write data into a normal file system (without CompressDB) simultaneously. CompressDB does not incur high CPU utilization because the major workload lies in the hash function, which can be amortized by parallelism.

Table 3: Memory consumption. The *blockHole* size refers to the data structure size for 1 GB changed data.

Dataset	A	B	C	D	E	F
blockHashTable (MB)	1752.23	3813.80	5311.75	69.09	22.90	388.22
blockHole (KB)	82.21	77.66	80.17	81.50	82.77	79.82
total (MB)	1752.31	3813.87	5311.83	69.17	22.98	388.29

Read/write performance. We use Filebench [6] to evaluate the CompressDB in single node and distributed systems. Filebench allocates a file set with various directories and files to simulate real workload, and then performs read and write operations to measure the IO behaviors of given file systems. Filebench can output the results of throughput, latency, and bandwidth utilization, as shown in Figure 12. Compared to the baseline, our solution achieves better results in all aspects, including throughput, latency, and bandwidth utilization. Specifically, on average, the performance of pure reads is 1692.4 MB/s, which is 1.26× over that of the baseline. The performance of pure writes is 1688.1 MB/s, which is 1.28× over that of the baseline.



(a) Throughput. (b) Latency. (c) Bandwidth.

Figure 12: Filebench evaluation.

Index construction time. Although index construction takes time, when handling read and write system calls, CompressDB can keep the data compressed in real-time while performing updates, which can amortize the overhead. Moreover, please note that the index needs to be built only once. With this guarantee, even frequent reads and writes during index construction do not cause much overhead. We measure the index construction time for using CompressDB. The incurred overhead ranges from 3% to 15%, which is acceptable for users given its benefits.

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 [13] 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 5. Accordingly, Succinct can compress data and write the Succinct-compressed data into CompressDB. Then, CompressDB can further compress the data, and provide direct processing 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.

7 RELATED WORK

To our knowledge, this work is the first to support compressed data direct processing for various database systems.

Compression in traditional databases. Compression techniques in traditional databases are mainly for storing more data, not for faster data access [40, 79]. Under the premise of given system resources, such as storage space, more data can be stored in database systems [14, 70, 79]. Most compression techniques in databases depend on context [4, 11, 23, 48, 70]. 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 [4], lz4 [70], gzip [23], bzip2 [48], Huffman coding [78, 90], and zstd [11], have been applied to various big data management systems [14, 16, 86], such as Hadoop [87], Spark [97], Oracle Database [41], SQL Server [7], and IBM DB2 [77]. 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. Therefore, a cache of lightweight encoded data blocks is proposed in Pixels [17, 49] as a tradeoff between space efficiency and processing speed. Our solution can address the above problem and can be applied in Pixels to further improve the cache efficiency. 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 [13, 21, 25, 26, 30, 31, 39, 43, 55–57, 68]. Suffix trees [68, 84] consume less storage space, but with larger memory consumption [47, 53]. Burrows- Wheeler Transform [21, 30] and suffix arrays [63] provide more compact compression format, but still have memory consumption issue [47]. FM-indexes [2, 27–30] and compressed suffix array [42, 44, 81–83] are more efficient alternatives, and Succinct [13] 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 [18–20, 22, 34, 35, 64, 80, 85, 89, 101, 103]. For example, TADOC [101, 103] enables direct data analytics on hierarchically compressed data by providing efficient data traversal and random access operations on compression. However, TADOC cannot perform data updating operations efficiently since they are unable to change the compressed data. Note that inverted index can also be compressed [62, 67, 71, 73–76]. Different from these works, CompressDB focuses on how to support general update and access operations on compressed data in big data systems.

8 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.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (No. 2019YFE0198600), National Natural Science Foundation of China (No. 62172419, U20A20226, 61732014, 62072458, 61972402, and 61972275), and Beijing Natural Science Foundation (4202031). This work is also sponsored by CCF-Tencent Open Research Fund. Jidong Zhai and Xiaoyong Du are the corresponding authors of this paper.

REFERENCES

[1] 2017. Wikipedia HTML data dumps. https://dumps.wikimedia.org/enwiki/.
[2] 2018. FM-index. https://en.wikipedia.org/wiki/FM-index.
[3] 2020. FUSE — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.
[4] 2020. Snappy, a fast compressor/decompressor. https://github.com/google/snappy.
[5] 2021. ClickHouse Home Page. https://clickhouse.tech/.
[6] 2021. Filebench - Linux man page. https://linux.die.net/man/1/filebench.
[7] 2021. Microsoft SQL Server. https://www.microsoft.com/zh-cn/sql-server.
[8] 2021. MongoDB: The application data platform. https://www.mongodb.com/2.
[9] 2021. MooseFS. https://moosefs.com/
[10] 2021. SQLite Home Page. https://www.sqlite.org/index.html.
[11] 2021. zstd. https://github.com/facebook/zstd.
[12] 2022. Cloud Data Warehouse - Amazon Redshift. https://aws.amazon.com/redshift/.
[13] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *NSDI*. 337–350.
[14] Sushila Aghav. 2010. Database compression techniques for performance optimization. In *2010 2nd International Conference on Computer Engineering and Technology*, Vol. 6. IEEE, V6–714.
[15] Oreoluwatomiwa O Babarinsa and Stratos Idreos. 2015. JAFAR: Near-data processing for databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2069–2070.
[16] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, George Mihaila, Kenneth Ross, Sherman Lau, Cathy McArthur, Zoltan Toth, and Reza Sherkat. 2009. Efficient index compression in DB2 LUW. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1462–1473.
[17] Haoqiong Bian and Anastasia Ailamaki. 2022. Pixels: An Efficient Column Store for Cloud Data Lakes. In *ICDE*.
[18] Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. 2015. Finger search in grammar-compressed strings. *arXiv preprint arXiv:1507.02853* (2015).
[19] Philip Bille, Gad M Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. 2015. Random access to grammar-compressed strings and trees. *SIAM J. Comput.* (2015).
[20] Nieves R Brisaboa, Adrián Gómez-Brandón, Gonzalo Navarro, and José R Paramá. 2019. Gract: a grammar-based compressed index for trajectory data. *Information Sciences* (2019).
[21] Michael Burrows and David J Wheeler. 1994. A block-sorting lossless data compression algorithm. (1994).
[22] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Transactions on Information Theory* (2005).
[23] Peter Deutsch et al. 1996. GZIP file format specification version 4.3. (1996).
[24] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1221–1230.
[25] Andrea Farruggia, Paolo Ferragina, and Rossano Venturini. 2014. Bicriteria data compression: efficient and usable. In *European Symposium on Algorithms*.
[26] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. 2009. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)* (2009).
[27] Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*.
[28] Paolo Ferragina and Giovanni Manzini. 2001. An experimental study of a compressed index. *Information Sciences* (2001).
[29] Paolo Ferragina and Giovanni Manzini. 2001. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*.
[30] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *Journal of the ACM (JACM)* (2005).
[31] Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2009. On the bit-complexity of Lempel-Ziv compression. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*.
[32] Phil Francisco et al. 2011. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics.
[33] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. 1986. An Overview of The System Software of A Parallel Relational Database Machine GRACE.. In *VLDB*, Vol. 86. 209–219.
[34] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. 2012. A faster grammar-based self-index. In *International Conference on Language and Automata Theory and Applications*.
[35] Moses Ganardi, Artur Jeż, and Markus Lohrey. 2019. Balancing straight-line programs. In *IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*.
[36] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 126–137.
[37] Sanjay Ghemawat and Jeff Dean. 2021. LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb.
[38] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. 2018. Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions. *arXiv preprint arXiv:1802.00320* (2018).
[39] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*.
[40] Goetz Graefe and Leonard D Shapiro. 1990. *Data compression and database performance*. University of Colorado, Boulder, Department of Computer Science.
[41] Rick Greenwald, Robert Stackowiak, and Jonathan Stern. 2013. *Oracle essentials: Oracle database 12c*. " O'Reilly Media, Inc.".
[42] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*.
[43] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2004. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*.
[44] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.* (2005).
[45] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 153–165.
[46] Jing Han, Ee Haihong, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *2011 6th international conference on pervasive computing and applications*. IEEE, 363–366.
[47] Wing-Kai Hon, Tak Wah Lam, Wing-Kin Sung, Wai-Leuk Tse, Chi-Kwong Wong, and Siu-Ming Yiu. 2004. Practical aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences. In *ALENEX/ANALC*.
[48] Marc Jakoby, Bernd Weisshaar, Wolfgang Dröge-Laser, Jesus Vicente-Carbajosa, Jens Tiedemann, Thomas Kroj, and François Parcy. 2002. bZIP transcription factors in Arabidopsis. *Trends in plant science* 7, 3 (2002), 106–111.
[49] Guodong Jin, Haoqiong Bian, Yueguo Chen, and Xiaoyong Du. 2022. Columnar Storage Optimization and Caching for Data Lakes. In *EDBT*.
[50] Tiago R Kepe, Eduardo C de Almeida, and Marco AZ Alves. 2019. Database processing-in-memory: an experimental study. *Proceedings of the VLDB Endowment* 13, 3 (2019), 334–347.
[51] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. 1977. Fast pattern matching in strings. *SIAM journal on computing* 6, 2 (1977), 323–350.
[52] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 219–231.
[53] Stefan Kurtz. 1999. Reducing the space requirement of suffix trees. *Software: Practice and Experience* (1999).
[54] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. 2017. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1706–1717.
[55] Jianzhong Li, Doron Rotem, and Jaideep Srivastava. 1999. Aggregation algorithms for very large compressed data warehouses. In *VLDB*, Vol. 99. 651–662.
[56] Jianzhong Li, Doron Rotem, and Harry KT Wong. 1987. A New Compression Method with Fast Searching on Large Databases. In *Proceedings of the 13th International Conference on Very Large Data Bases*. 311–318.
[57] Jianzhong Li and Jaideep Srivastava. 2002. Efficient aggregation algorithms for compressed data warehouses. *IEEE Transactions on Knowledge and Data Engineering* 14, 3 (2002), 515–529.
[58] Li Li, Ke Xu, Tong Li, Kai Zheng, Chunyi Peng, Dan Wang, Xiangxiang Wang, Meng Shen, and Rashid Mijumbi. 2018. A Measurement Study on Multi-path TCP with Multiple Cellular Carriers on High Speed Rails. In *Proceedings of ACM SIGCOMM*.
[59] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. 2020. TACK: Improving Wireless Transport Performance by Taming Acknowledgments. In *ACM SIGCOMM*.
[60] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. 2021. Revisiting acknowledgment mechanism for transport control: Modeling, analysis, and implementation. *IEEE/ACM Transactions on Networking (TON)* (2021).
[61] M. Lichman. 2013. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml.

[62] Joel Mackenzie, Antonio Mallia, Matthias Petri, J Shane Culpepper, and Torsten Suel. 2019. Compressing inverted indexes with recursive graph bisection: A reproducibility study. In *European Conference on Information Retrieval*.

[63] Udi Manber and Gene Myers. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* (1993).

[64] Sebastian Maneth. 2018. *Grammar-Based Compression*. Springer International Publishing, Cham, 1–8. https://doi.org/10.1007/978-3-319-63962-8_56-1

[65] Colt McAnlis and Aleks Haecky. 2016. *Understanding compression: Data compression for modern developers*. " O'Reilly Media, Inc.".

[66] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[67] Alistair Moffat and Matthias Petri. 2018. Index compression using byte-aligned ANS coding and two-dimensional contexts. In *WSDM*.

[68] Gonzalo Navarro. 2016. *Compact Data Structures: A Practical Approach*. Cambridge University Press.

[69] Craig G Nevill-Manning and Ian H Witten. 1997. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research* 7 (1997), 67–82.

[70] Wee-Keong Ng and Chinya V. Ravishankar. 1997. Block-oriented compression techniques for large statistical databases. *IEEE Transactions on Knowledge and Data Engineering* 9, 2 (1997), 314–328.

[71] Harrie Oosterhuis, J Shane Culpepper, and Maarten de Rijke. 2018. The potential of learned index structures for index compression. In *Proceedings of the 23rd Australasian Document Computing Symposium*.

[72] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. Exploring data analytics without decompression on embedded GPU systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2021), 1553–1568.

[73] Matthias Petri and Alistair Moffat. 2018. Compact inverted index storage using general-purpose compression libraries. *Software: Practice and Experience* (2018).

[74] Giulio Ermanno Pibiri, Raffaele Perego, and Rossano Venturini. 2020. Compressed Indexes for Fast Search of Semantic Data. *TKDE* (2020).

[75] Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. 2019. Fast dictionary-based compression for inverted indexes. In *WSDM*.

[76] Giulio Ermanno Pibiri and Rossano Venturini. 2019. Techniques for Inverted Index Compression. *arXiv preprint arXiv:1908.10598* (2019).

[77] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[78] Cody Rivera, Sheng Di, Jiannan Tian, Xiaodong Yu, Dingwen Tao, and Franck Cappello. 2022. Optimizing Huffman Decoding for Error-Bounded Lossy Compression on GPUs. In *The 36th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2022)*.

[79] Mark A Roth and Scott J Van Horn. 1993. Database compression. *ACM Sigmod Record* 22, 3 (1993), 31–39.

[80] Wojciech Rytter. 2004. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *International Colloquium on Automata, Languages, and Programming*.

[81] Kunihiko Sadakane. 2000. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *International Symposium on Algorithms and Computation*.

[82] Kunihiko Sadakane. 2002. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*.

[83] Kunihiko Sadakane. 2003. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms* (2003).

[84] Kunihiko Sadakane. 2007. Compressed suffix trees with full functionality. *Theory of Computing Systems* (2007).

[85] Sherif Sakr and Albert Y Zomaya. 2019. *Encyclopedia of big data technologies*. Springer International Publishing.

[86] Ivan Schreter and Amarnadh Sai Eluri. 2018. Efficient block-level space allocation for multi-version concurrency control data. US Patent 9,875,024.

[87] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[88] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[89] Yoshimasa Takabatake, Hiroshi Sakamoto, et al. 2017. A space-optimal grammar compression. In *25th Annual European Symposium on Algorithms*.

[90] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. 2021. Revisiting Huffman coding: Toward extreme performance on modern GPU architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[91] Michael Ubell. 1985. The intelligent database machine (idm). In *Query processing in database systems*. Springer, 237–247.

[92] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.

[93] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. 2015. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*. 1–10.

[94] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. 2020. AQUOMAN: An Analytic-Query Offloading Machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 386–399.

[95] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2101–2113.

[96] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1805.

[97] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. *HotCloud* (2010).

[98] Feng Zhang, Zheng Chen, Chenyang Zhang, Amelie Chi Zhou, Jidong Zhai, and Xiaoyong Du. 2021. An efficient parallel secure machine learning framework on GPUs. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[99] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. G-TADOC: Enabling Efficient GPU-Based Text Analytics without Decompression. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE.

[100] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2016. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* (2016).

[101] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1522–1535.

[102] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Zwift: A programming framework for high performance text analytics on compressed data. In *Proceedings of the 2018 International Conference on Supercomputing*. 195–206.

[103] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2020. Enabling efficient random access to hierarchically-compressed data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1069–1080.

[104] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2022. POCLib: a high-performance framework for enabling near orthogonal processing on compression. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2022), 459–475.

[105] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.