Improving LZ4 for Effective Compression and Efficient Query

ZHIHENG LIU, Tsinghua University, China
SHAOXU SONG*, Tsinghua University, China

LZ4 is a compression algorithm widely adopted in many database systems, which surprisingly has no support for directly querying the compressed data. Existing systems rely on full decompression for query processing, leading to increased query latency. Moreover, the LZ4 compression algorithm has issues like long match dependency, which reduces both compression effectiveness and the efficiency of compressed-data query processing. In this paper, (1) we propose LZV, a compression algorithm that employs a variable-length hash mechanism to identify maximal matches, significantly improving the compression ratio and reducing the compressed-data query overhead. (2) We propose compressed index search method that leverages auxiliary structure to efficiently query compressed data in LZ4 format directly. (3) Leveraging the ordered, quasi-uniformly spaced nature of the key column, we introduce compressed key search method that integrates prediction with binary search to retrieve the corresponding index for a given key efficiently. Finally, we implement the compressed-data query pipeline in Apache TsFile, an open-source KV storage system. Experimental results show that our approach significantly improves compression effectiveness and query efficiency.

CCS Concepts: • **Theory of computation** → **Data compression**; • **Information systems** → *Query optimization.*

Additional Key Words and Phrases: LZ4, Compression, Query Processing

ACM Reference Format:
Zhiheng Liu and Shaoxu Song. 2026. Improving LZ4 for Effective Compression and Efficient Query. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 46 (February 2026), 26 pages. https://doi.org/10.1145/3786660

1 Introduction

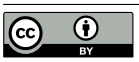
LZ4 [10, 22] is a widely adopted lossless compression algorithm known for its extremely fast compression and decompression speeds with low CPU overhead. It has been integrated as the default or recommended compression method in several modern database systems, such as Apache Cassandra [3], ClickHouse [9], RocksDB [27], Apache IoTDB [4, 34], and Apache TsFile [5, 37]. Although LZ4 is widely recognized for its fast compression and decompression performance, it still requires full decompression to process any query, leading to substantial time overhead, particularly on large-scale datasets [30, 36].

1.1 Motivation

Unlike LZ77, which searches for the longest match through traversal, LZ4 relies on a hash table to locate matching positions. Specifically, during the hash update phase, LZ4 computes a hash for each unmatched 4-byte sequence and stores the positions of these sequences in a hash table. In the hash lookup phase, LZ4 calculates the hash of the current sequence and quickly checks for matches using the hash table. While the use of a simple hash method significantly reduces compression time,

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

Authors' Contact Information: Zhiheng Liu, Tsinghua University, Beijing, China, zh-liu24@mails.tsinghua.edu.cn; Shaoxu Song, Tsinghua University, Beijing, China, sxsong@tsinghua.edu.cn.



ACM 2836-6573/2026/2-ART46
https://doi.org/10.1145/3786660

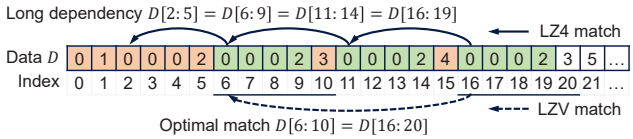


Fig. 1. Motivating example, where LZ4 compressor greedily matches the most recent occurrence of each 4-bytes, e.g., $D[11:14] = D[16:19]$, leading to weaker compression ratio and longer dependency.

it also introduces adverse effects on the LZ4 algorithm. Specifically, LZ4 always selects the most recent match during hash table lookups, which often (1) leads to suboptimal matches, reducing compression effectiveness, and (2) creates long dependency chains, increasing compressed-data query overhead.

Example 1. *Figure 1 presents a motivating example. For instance, the orange-highlighted parts, $D[0:5], D[10:10], D[15:15]$, represent unmatched subsequences that must be stored explicitly, while the green-highlighted parts, $D[6:9], D[11:14], D[16:19]$, indicate matched subsequences that can be referenced from previously seen data to achieve compression.*

Note that the longest match for the subsequence starting at index 16 is the one starting at index 6, i.e., $D[16:20] = D[6:10]$, with a match length of 5. However, the match found by LZ4 only has a length of 4, i.e., $D[16:19] = D[11:14]$. These suboptimal matches reduce the compression ratio.

On the other hand, the subsequence $D[16:19]$ matches with $D[11:14]$, which in turn matches with $D[6:9]$, and finally $D[2:5]$. Consequently, LZ4 forms a long dependency chain for $D[16:19]$, which hinders the efficiency of query processing on compressed data.

1.2 Intuition

This motivates our core research question: Can we optimize LZ4 to improve the compression ratio while shortening the data dependency chain for efficient querying?

(1) **A novel hashing mechanism in compression:** As illustrated in the motivating example, LZ4 suffers from suboptimal compression and inefficient query due to its simplistic hash update and lookup mechanisms. Our key insight is that by adopting a variable-length hash strategy, we can record subsequences that start at every position, thereby constructing a more comprehensive view of the input data and enabling longer matches over greater distances. At the same time, we can remain the LZ4 compressed format, allowing reuse of standard decompression procedures.

(2) **Query on compressed data in LZ4 format:** LZ4 encodes data into a compact byte stream according to fixed-format rules. This format is designed primarily for space efficiency, making random access or point queries over compressed data inherently difficult, as the compressed data lacks explicit block boundaries or indexing structures. Our intuition is that by building a lightweight online index over the compressed data, we can map positions in the original data to the corresponding index of matched and unmatched blocks in the compressed data. This allows us to perform queries, such as point access, without full decompression.

(3) **Optimization for queries in key-value storage systems:** In a typical key-value store query, the system must first locate the index corresponding to a given key, and then use this index to retrieve the associated value [26, 36]. Therefore, efficiently returning the index for a specific key from the compressed key column is also a challenge. Our intuition is that, by leveraging the fact that the key column is sorted [2] and often exhibits a nearly uniform distribution[25], we can reduce the number of query operations through an efficient search strategy. Furthermore, by reusing our compressed-data query technique, we can effectively locate the target key within the compressed key column.

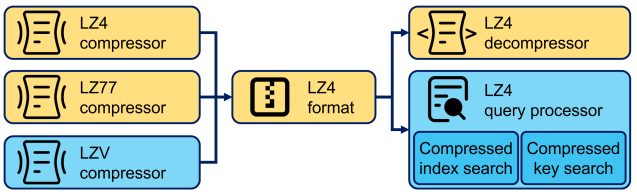


Fig. 2. An overview of contributions (in blue).

1.3 Contribution

To the best of our knowledge, this is the first study on querying compressed data in LZ4 format. Our major contributions are illustrated in Figure 2 as follows.

(1) **Maximal-match compression algorithm LZV** (Section 4): The main contribution of our work lies is the LZV compression algorithm, which not only improves the compression ratio compared to LZ4 but also facilitates efficient compressed querying. Specifically, by ensuring longer matches and shorter dependency chains, LZV significantly improves the efficiency of compressed query methods by executing with substantially fewer jumps. Notably, LZV still outputs compressed data in LZ4 format, and thus can be decompressed by existing LZ4 decompressor.
(2) **Compressed index search over LZ4 format** (Section 5.1): CIS method is the first method to query compressed data in LZ4 format. It begins by constructing an index via a fast preprocessing procedure. Using the index, it can retrieve the value without full decompression of the compressed data. Theoretical analysis (Propositions 5.2 and 5.3) shows that the time complexity of compressed index search is linear to the size of the compressed data.
(3) **Compressed key search over LZ4 format** (Section 5.2): CKS is a method for querying the index of a given key in the compressed data. It combines interpolation search with binary search in an alternating manner. For compressed keys, interval lengths may vary across segments (e.g., regular or irregular), and our alternating approach is robust to such variations.
(4) **System implementation in Apache TsFile** (Section 6): We implement LZV in Apache TsFile [5, 37], a state-of-the-art key-value storage system for time series. To support direct query on compressed data, we design and implement a compressed-data query pipeline, as shown in Figure 8.
(5) **Extensive evaluation on real-world datasets** (Section 7): We conducted extensive algorithmic and system-level experiments across a wide range of datasets. The experimental results demonstrate that LZV has higher compression time for better compression ratio and more efficient query processing, suitable to write-once-read-many (WORM) scenarios of databases.

2 Related work

This section reviews related works about compression algorithms and existing methods for querying compressed data.

2.1 Compression Algorithm

The LZ77 algorithm [38] is a classical lossless compression scheme that forms the foundation of many modern compression formats. It works by replacing repeated sequences of data with references to previous occurrences within a fixed-size sliding window. The encoded output consists of triples in the form of *(offset, length, next symbol)*, representing a backward pointer to the match location, the length of the match, and the next unmatched symbol.

Building on the LZ77 framework, several derivative algorithms have been developed and widely applied in data compression. LZ78 [39] introduced a dictionary-based approach for simpler encoding,

while LZW [23] enhanced efficiency by outputting only dictionary indices. These foundational methods influenced modern compression algorithms such as GZIP, LZMA [28], Zstandard, and Snappy [15], which combine LZ77-style pattern matching with advanced techniques like Huffman coding, entropy modeling, and range encoding.

Most existing optimization efforts for LZ4 focus primarily on further improving its compression speed. For instance, at the hardware level, researchers have proposed a high-throughput and low-latency LZ4 compression architecture on FPGA [8]. At the algorithmic level, techniques such as rolling hash and hash computation fusion have been introduced to accelerate the hash-based match-finding process [16]. However, these approaches often compromise the quality of match selection to some extent. In particular, there is a lack of research on how to leverage hash-based methods to find maximal matches while also improve compressed-data query performance.

2.2 Query on Compressed Data

To reduce query overhead on compressed data, significant effort has focused on enabling efficient direct access without full decompression. Traditional LZ77 compression offers strong compression on repetitive texts but lacks efficient random access. To address this, Kreft and Navarro [19] proposed LZ-End, which restricts phrase sources to phrase ends, allowing subsequence extraction in $O(\ell)$ time when the subsequence ends at a phrase boundary. However, this compression algorithm affects the compression ratio to some extent. They also explored self-indexing and direct query support over LZ77 and LZ-End [20] on its triplet representation. Notably, the LZ4 format is a highly compact data representation, where the smallest addressable unit is at the bit level, making query support particularly challenging.

On the query side, Han et al. [13] proposed the RELZ family of algorithms to support regular expression matching on LZ77-compressed texts using pruning techniques based on positive and negative factors. However, most existing approaches overlook the inherent ordering present in columnar and analytical databases. Leveraging this natural order remains an open and promising avenue for improving the efficiency of compressed-data query processing. It is worth noting that RELZ and related methods operate on the LZ77 compression format, which has worse compression ratio than LZ4 format. Unlike existing method, we do not change the compressed data format of LZ4, which offers better compression ratios and is more widely used in practice, and thus these related works are not compared in our experiments.

3 Preliminary

Table 1 summarizes the notations used throughout the paper. Some of these symbols are formally defined in Section 3.1. Section 3.2 provides a detailed introduction of the compressed data format used by LZ4, and Section 3.3 introduces the LZ4 compression algorithm.

3.1 Definition

This subsection introduces a set of fundamental definitions.

Definition 3.1 (Subsequence). Let Σ be a finite alphabet, and let $D \in \Sigma^*$ be a sequence of length n, i.e.,

$$D = d_0 d_1 \dots d_{n-1},$$

where each $d_i \in \Sigma$. The **subsequence** $D[i : j]$ is defined as

$$D[i : j] = d_i d_{i+1} \dots d_j, 0 \le i \le j < n.$$

In LZ77-based compression algorithms, matching is a key operation. The following definitions specify how to determine a match and calculate the match length between two positions.

Table 1. Notations.

Symbol	Description
D	original data sequence before compression
I	index of original data D
n	length of original data D
C	compressed data sequence in LZ4 format
P	index of compressed data C
m	length of compressed data C
b	number of basic blocks in compressed data C
l	length of a matched subsequence
o	the offset of a subsequence $D[i:j]$
$D[i:j]$	a subsequence of D starting from i ending at j
M, U	matched and unmatched subsequences in D
$Mat(D, i, j)$	match length of subsequences starting at i and j
$Max(D, i)$	maximal match position of position i in D

Definition 3.2 (Matched and Unmatched Subsequence). Let D be a sequence of length n. Two subsequences $D[i:j]$ and $D[i':j']$ are said to **match** each other if they have the same length, i.e., $i > i'$, $j - i = j' - i'$, and satisfy $d_{i+\alpha} = d_{i'+\alpha}$ for all $\alpha \in [0, j-i]$. In this case, $D[i:j]$ is called a **matched subsequence**. A subsequence $D[i:j]$ is said to be an **unmatched subsequence** in D if there exists no subsequence $D[i':j']$ with $i' < i$ such that $D[i:j]$ and $D[i':j']$ match.

Offset and match length are two key concepts used in LZ77-based compression algorithms to describe repeated patterns in the sequence.

Definition 3.3 (Offset). Given a matched subsequence $D[i:j]$, its **offset** $o = \alpha$ indicates that the subsequence $D[i:j]$ matches a previous subsequence $D[i-\alpha : j-\alpha]$.

Definition 3.4 (Match Length). $D \in \Sigma^*$ is a sequence over a finite alphabet Σ of length n, and α, β are two indices such that $0 \le \alpha < \beta < n$. The **match length** $Mat(D, \alpha, \beta)$ is defined as the length of the longest common prefix between the subsequences of D starting at indices α and β, that is,

$$Mat(D, \alpha, \beta) = \max\{k \in \mathbb{N} \mid \beta + k < n, \text{ and } D[\alpha : \alpha + k] = D[\beta : \beta + k]\}.$$

During compression, an ideal match should have both a long length to improve efficiency and a large distance to reduce data dependencies. We define such ideal match position as follows.

Definition 3.5 (Maximal Match Position). $D \in \Sigma^*$ is a sequence over a finite alphabet Σ of length n, and α is an index such that $0 \le \alpha < n$. The **maximal match position** of α in D, denoted as $Max(D, \alpha)$, is defined as:

$$Max(D, \alpha) = \min_{\beta < \alpha} \left\{ \beta \mid Mat(D, \beta, \alpha) = \max_{\beta' < \alpha}(Mat(D, \beta', \alpha)) \right\}.$$

3.2 LZ4 Format

As a member of LZ77 compression family, LZ4 also encodes data as an alternating sequence of matched and unmatched subsequences. For an unmatched subsequence U, the original data must be stored explicitly, while for a matched subsequence M, only the match length and the offset indicating the match location are recorded. The LZ4 format is a more efficient, block-based format to represent compressed data, as shown in Figure 3.

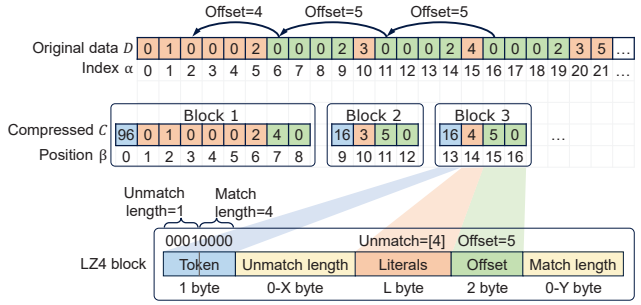


Fig. 3. An example of data compressed in LZ4 format.

Specifically, the LZ4 format encodes each pair of adjacent unmatched subsequence U and matched subsequence M in D as a single block in C. Encoding begins with a single token byte, where the high 4 bits represent the length of U, and the low 4 bits represent the length of M minus 4. If the length of U exceeds 15, additional bytes (denoted as *X bytes*) are used to store this length. Next, L *bytes* of literal data for U are added, followed by a 2-byte offset. Finally, if the length of M exceeds 19, the actual length of M is extended using additional Y *bytes*.

Example 2. *Figure 3 shows the encoding of* $D[15:19]$ *using the LZ4 format. This segment consists of an unmatched subsequence* $U = D[15:15]$ *and a matched subsequence* $M = D[16:19]$*, which matches* $D[11:14]$*. Therefore, the unmatch length is 1, and the match length is 4. First, the token is written. The upper 4 bits* `'0001'` *represent the unmatched length, and the lower 4 bits represent the match length minus 4, i.e.,* `'0000'`*. Thus, the Token byte is* `'00010000'` *in binary, or 16 in decimal. Since the unmatched length is less than 15, no additional bytes are required for its encoding, i.e.,* $X = 0$*. Next, the literal byte representing the unmatched data [4] is written with* $L = 1$*. This is followed by a two-byte offset calculated as* $16 - 11 = 5$*. Finally, because the match length is 4, which is less than 19, no extra bytes are needed to store the match length, i.e.,* $Y = 0$*. Consequently, the compressed representation of* $D[15:19]$ *in the LZ4 format is [16, 4, 5, 0], corresponding to subsequence* $C[13:16]$ *(Block 3) in the compressed data* C*.*

3.3 LZ4 Compressor

LZ4, proposed by Collet in 2011, is a modern, high-speed compression algorithm based on the LZ77 family [7, 10]. Unlike LZ77, LZ4 uses *hash* to accelerate the search for matching subsequences. Algorithm 1 presents a simplified pseudocode of the LZ4 compression process. In practical implementations, various engineering optimizations are applied to achieve higher speed.

Example 3. *Figure 4 illustrates the detailed process of compressing data using the LZ4 algorithm. Assuming the preceding data has already been compressed, we consider compression starting from index 15, i.e.,* $a = p = 15$ *in Algorithm 1. First, possible match of subsequence* $D[15:18]$ *is found by using hash function* $hash(D[15:18]) = hash([4,0,0,0]) = 5$*. Since hash table* $T[5] = 1$*, the value at* $T[5]$ *is updated from 1 to 15. It is checked whether match length* $Mat(D, 1, 15) \geq 4$*, where Mat is defined in Definition 3.4. Unfortunately* $Mat(D, 1, 15) = 0 < 4$*, indicating it is not a real match but hash collision. Thus, line 20 is executed, and* p *is incremented to 16.*

Then, the subsequence $D[16:19]$ *is looked up in the hash table using the same steps. Its hash value is* $hash(D[16:19]) = hash([0,0,0,2]) = 8$*, and there is* $p^* = T[8] = 11$*. The value at* $T[8]$ *is updated to 16. Afterward, the algorithm computes* $Mat(D, 11, 16)$*, and finds a match length of 4, marking the*

Algorithm 1: LZ4 Compression (Simplified)

```
1  Input: The original data D
2  Output: The compressed result C
3  p, a ← 0
4  C, β ← empty buffer, 0
5  T ← new hash table                                  ▷ Initialize all entries to −1
6  while p + 3 < length(D) do
7      s ← D[p : p + 3]
8      p* ← T.get(hash(s))                             ▷ Hash Table Lookup
9      T.put(hash(s), p)                               ▷ Hash Table Update
10     if p* ≠ −1 and D[p* : p* + 3] = s then
11         l ← 4                                       ▷ Match length is 4 now
12         while D[p + l : p + l] = D[p* + l : p* + l] do
13             l ← l + 1
14         Encode literal from a to p into C
15         Encode offset p − p* into C
16         Encode l − 4 into C
17         p ← p + l
18         a ← p
19     else
20         p ← p + 1
21 Encode remaining literals from a to end
22 return C
```

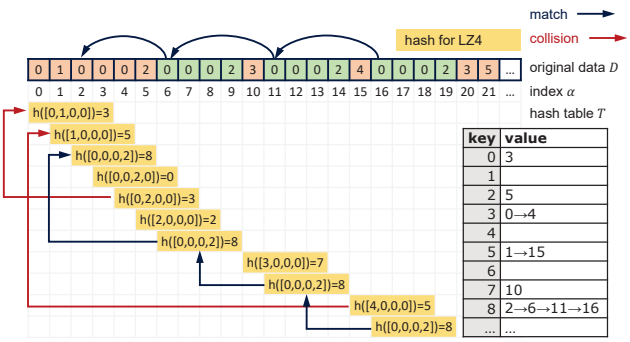


Fig. 4. An example of LZ4 compression algorithm.

completion of a new compressed subsequence. According to lines 14–16 (see Example 2 for details), the compressed data is written accordingly. Finally, both p and a are updated to 16 + 4 = 20 *in lines 17-18.*

LZ4 greedily matches the most recent occurrence of each 4-bytes due to its hash design, which significantly affects match quality. For instance, in Figure 1, the optimal match for the subsequence starting at index 16 is the subsequence starting at index 7, yielding a match length of 5. However, because of the way the hash table is updated, LZ4 instead matches the subsequence at index 16 with the one at index 11, resulting in a shorter match of length 4.

Algorithm 2: Update the hash table using the variable-length hash

```
1  Input: The original data D, hash table T, match position p*, now position p
2  Output: The updated hash table T, the match length l
3  l ← match length start from p and p*                          ▷ Get match length
4  if l < 4 then
5      T[hash(D[p : p + 3])] ← p
6  else
7      p̄, p̄′ ← p + l                          ▷ Get end position of variable-length hash
8      while p < p̄′ do
9          l̄ ← max(4, p̄ − p + 1)                                  ▷ Get length of hash
10         h ← hash(D[p : p + l̄ − 1])
11         isMatch, p* ← false, T[h]
12         if Mat(D, p*, p) ≥ l̄ then
13             isMatch, l ← true, Mat(D, p*, p)
14         if not isMatch then
15             T[h] ← p
16             p ← p + 1
17         else
18             p̄ ← p + l
19 return T, l
```

4 LZV Compressor

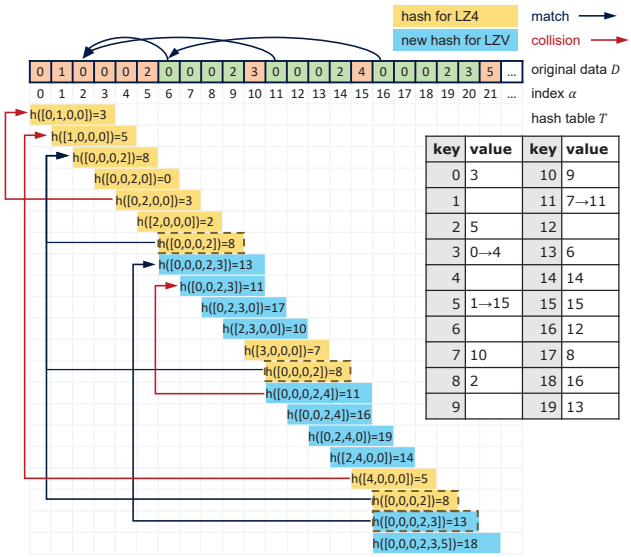
In this section, we propose a novel compression algorithm called LZV, which stands for **LZ**4 using **V**ariable-length hashing. The overall framework of LZV follows the structure of LZ4 as shown in Algorithm 1, while we redesign two key components, the hash lookup and the hash update, using a variable-length hashing scheme.

4.1 Hash Table Update

Unlike LZ4, which updates the hash table using a fixed length of 4, the main idea of LZV is to update the hash table using a dynamic length. The dynamic length means to add subsequences of a specified length, starting from each position within the matching subsequence, into the hash table, which ensures that the existing contents of the hash table remain unaffected. In contrast, LZ4 only updates the hash table with the first four values of the matching subsequence as a substring, and may lead to collisions, which consequently affect compression as shown in Example 1.

The detailed procedure for updating the hash table with variable-length hashing in LZV is presented in Algorithm 2. Specifically, we use l to denote the match length, $\bar{l}$ to denote the length of the variable-length hash, and $\overline{p}$ to denote the end position of the variable-length hash. For unmatched data, it is directly inserted into the hash table in lines 4–5, consistent with the LZ4. In line 7, it sets $\overline{p}$ and $\overline{p}'$ to the position immediately following the current match and $\overline{p}'$ will remain fixed throughout the subsequent while loop.

The following loop iteratively moves the position p from the start of the matched subsequence to the $\overline{p}'$. At each position p, the algorithm attempts to insert the subsequence $D[p : \overline{p}]$ into the hash table. If the subsequence has not been previously recorded, lines 14–16 insert it as a new



key	value	key	value
0	3	10	9
1		11	7→11
2	5	12	
3	0→4	13	6
4		14	14
5	1→15	15	15
6		16	12
7	10	17	8
8	2	18	16
9		19	13

Fig. 5. An example of LZV compression algorithm, including hash table update and hash table lookup using variable-length hashing.

entry. Otherwise, if the subsequence already exists in the hash table, line 18 extends the $\overline{p}$ position accordingly. This ensures that the extended subsequence starting at p does not overlap with any previously recorded match, thereby avoiding the loss of previously recorded matches.

Example 4. *We use the example in Figure 5 to demonstrate how the hash table is updated in LZV. For unmatched subsequences, the hash table is updated directly—for example, from subsequence $D[0:3]$ to $D[5:8]$. In contrast, for matched subsequences, LZV avoids overwriting existing entries in the hash table. For example, $D[6:9]$ matches $D[2:5]$, and both subsequences produce the same hash value: $hash(D[6:9]) = hash([2,0,0,0]) = 8 = hash(D[2:5])$. LZV does not replace the existing value 2 at key 8 with 6 in the hash table. Instead, it incrementally inserts new entries based on the variable-length hash strategy. Specifically, now position $p = 6$, match length $\overline{l} = 4$, and hash end position $\overline{p} = p + \overline{l} = 10$ in line 7 of Algorithm 2. First, the hash of the subsequence $D[6:10]$ is computed, yielding 13. Since the subsequence has not been previously recorded in the hash table T at key 13, the value is stored to preserve the information subsequence $D[6:10]$ in the hash table and the position p is incremented by 1. This process continues for subsequences $D[7:10]$, $D[8:11]$, and $D[9:12]$. These subsequences, highlighted in blue in the figure, are not hashed or inserted into the table under the standard LZ4.*

Compared to LZ4, we compute additional hashes for the matched portion of the data. In pursuit of extreme speed, LZ4 omits these hash computations, which results in many subsequences not being recorded in the hash table. Consequently, it may fail to find the longest match. In contrast, our hash update strategy records the subsequences starting at each position, ensuring the completeness of information in the hash table, as supported by Proposition 4.1.

4.2 Hash Table Lookup

During the hash table lookup phase, we do not rely solely on hash values computed from fixed-length subsequences. Instead, we increment the subsequence length by one, up to the matched

Algorithm 3: Find the maximal match position using variable-length hash matching

```
1  Input: The original data D, hash table T, now position p
2  Output: The best match position p*
3  p*, l ← T[hash(D, p, 4)], 0
4  if p* == −1 then
5      T[hash(D, p, 4)] ← p
6  else
7      l, nowLen ← Mat(D, p*, p), 4
8      while nowLen <= l do
9          nowLen ← nowLen + 1
10         temp = T[hash(D, p, nowLen)]
11         if temp == −1 & nowLen > l then
12             break
13         else
14             if Mat(D, temp, p) > l then
15                 l, p* ← Mat(D, temp, p), temp
16 return if l ≥ 4 then p* else −1
```

length plus one, and stop when no new match can be found. This approach ensures that the maximal match is identified, as demonstrated in Proposition 4.1. The process is detailed in Algorithm 3.

Specifically, in line 3, we first compute the hash of a 4-byte subsequence, similar to the standard LZ4 approach. If a match is found, we record the current match length as l and initialize the hash window length as *nowLen*. Lines 8– 15 describe an iterative process that incrementally increases the subsequence length used for hash computation in an attempt to discover longer matches. The loop terminates when the hash window length exceeds the longest match length so far, and the algorithm returns the position p^* corresponding to the best match.

Example 5. *We continue using the example in Figure 5 to illustrate how a matched subsequence is found using hash table in LZV. Suppose the current position* $p = 15$. *We get candidate match position* $p^* = T[hash([15 : 18])] = T[5] = 1$ *in line 3 using the hash table* T *and match length* $l = 0$ *in line 7. Since* $nowLen = 4 > l$, *Algorithm 3 return -1, indicating that* $D[15 : 18]$ *is an unmatched subsequence.*

Then, consider position $p = 16$. *We get candidate match position* $p^* = T[hash([16 : 19]] = T[8] = 2$ *in line 3 and* $l = 4$ *in line 7. Since* $nowLen = 4 = l^*$, *we increment nowLen by 1 in line 9 and compute* $temp = T[hash(D[16 : 20])] = T[13] = 6$ *in line 10. As* $Mat(D, 6, 16) = 5 > l$, *we update* $l = 5$ *and* $p^* = 6$ *in line 15. We increment nowLen again in line 9 and compute* $temp = T[hash(D[16 : 22])] = T[18] = -1$ *in line 10. Since* $temp = -1$ *and* $nowLen = 6 > l = 5$, *Algorithm 3 returns* $p^* = 6$, *indicating that the maximal match position of index 16 in data D is index 6.*

4.3 Best Local Match

By comparing the compression results in Figure 4 and 5, we observe that LZV identifies a better match for the subsequence starting at index 16. Assuming the absence of hash collisions, LZV can theoretically identify the longest previous match for the current subsequence, improving compression performance. This property is formally established in Proposition 4.1.

Proposition 4.1. *For any index i in the original data D, its maximal match position $Max(D, i)$, as defined in Definition 3.5, can be found by Algorithm 3.*

Proof. Suppose $Max(D, i) = j$. For clarity, we use u to denote an unmatched element and m to denote a matched one. Note that a pattern like $[u, m, m, u]$ is not possible, as matched elements must appear in runs of at least four consecutive positions. Therefore, for the j-th element in D, if $D[j : j]$ is unmatched, then $D[j : j + 3]$ can only take one of the following forms:

$$[u, u, u, u], \quad [u, u, u, m], \quad [u, u, m, m], \quad [u, m, m, m].$$

For each of these valid cases, the 4-length sequence $D[j : j + 3]$ has not appeared before and thus will be hashed and inserted into the hash table. During the lookup process, we only need to compute the hash of the current 4-length window $D[i : i + 3]$ and query the hash table. Then a match is found, and it corresponds to a candidate position j in the array.

Assume $D[j]$ is a matched element, and let $Mat(D, j, i) - 1 = \ell$. Then the subsequence $D[j : j+\ell]$ cannot consist entirely of matched elements. Suppose for contradiction that the entire subsequence $D[j : j + \ell]$ is composed only of matched elements and that it matches with $D[k : k + \ell]$ for some $k < j$. Then, position k also satisfies $Mat(s, k, i) = \ell$, contradicting the assumption that j is the smallest position where the match occurs. Therefore, there must exist at least one unmatched element within the range $D[j : j+\ell]$. Let $j' > j$ be the first position in $D[j : j+\ell]$ such that $D[j']$ is unmatched. According to the construction rules, the sequence $D[j : \max(j', j + 3)]$ must have been inserted into the hash table. Consequently, during the traversal and search process, this sequence can be located through a hash lookup, ensuring the candidate position is found correctly. □

In lines 11-12 of Algorithm 2, if the subsequence $D[i : i']$ is not found in the hash table, and $i' - i + 1$ exceeds the currently known maximum match length, the algorithm terminates the search early. This strategy still guarantees that the best local match position and maximum match length are correctly identified. The correctness of this approach is formally proven in Proposition 4.2.

Proposition 4.2. *Let i be an index of original data D. If none of the subsequences $D[i : i+3]$, $D[i : i + 4]$, $\ldots$, $D[i : i + \ell - 1]$ can find a match with a length of at least ℓ using the hash table, then for any j, $j < i$, there is $Mat(D, i, j) < \ell$.*

Proof. Consider the sequence $D[0 : i + \ell - 1]$. Then the maximum possible match length for a subsequence starting at position i is at most ℓ. From Proposition 4.1, we know that if there exists a match of length ℓ starting from position i, it must be discovered during the traversal process, which checks subsequences of increasing lengths: $D[i : i + 3]$, $D[i : i + 4]$, $\ldots$, $D[i : i + \ell - 1]$.

Therefore, if none of these subsequences produce a match in the hash table, we can conclude that no match of length ℓ or longer exists for the subsequence starting at i. This implies that the length-ℓ subsequence starting at i appears for the first time in the sequence, with no prior occurrence. □

We clarify that the greedy strategy of longest match does not provide a global optimal compression, but locally optimal. The LZ4 format requires a minimum match length of 4, which makes the longest match non-optimal. For example, in data $D = \underbrace{abcda}\overline{baceg}\underbrace{abcda}\overline{ceg}$, the maximal match strategy finds a 5-length match (underline), whereas the optimal result indeed has two 4-length matches of total length 8 (overline). Besides, the longest match does not lead to the smallest number of blocks either. For example, merging any two consecutive phrases leads to an even smaller number of phrases, but ignores the match in-between, i.e., worse bit-wise encoding.

4.4 Time Complexity

Similar to the LZ4 algorithm, the LZV compression divides the data into alternating matched and unmatched subsequences. For a matched subsequence of length l, Proposition 4.3 proves the time

required is $O(l^2)$ in both hash lookup and hash update process. For an unmatched subsequence of length u, we only need to compute its hash values sequentially, taking $O(u)$ time. Therefore, the time complexity of the algorithm is $O(ln)$ as proved in Proposition 4.4.

Proposition 4.3. *The time complexity for finding the matched subsequence of length l in Algorithm 3 is $O(l^2)$.*

Proof. In lines 3–7, we first calculate the hash value of 4 bytes of data starting from the current position and look for a match using the hash table. Since the match length found is smaller than l, this step takes at most $O(l)$ time in $Mat(D, res, p)$. In the loop from lines 8 - 15, each match lookup takes at most $O(l)$ time in $Mat(D, temp, p)$. And the loop will perform at most $O(l)$ iterations. Overall, the time complexity of the algorithm is $O(l^2)$. □

Proposition 4.4. *The time complexity of LZV is $O(ln)$.*

Proof. According to the compression rules of the LZV algorithm, the data is divided into alternating unmatched and matched subsequences. Let the lengths of the i-th unmatched and matched subsequences be denoted by u_i and m_i. For simplicity, consider a case where the original data of length n consists of b unmatched subsequences and b matched subsequences, i.e., $n = \sum_{i=1}^{b} u_i + m_i$, $l = \max m_i$ is the max length of all matched subsequences.

The entire LZV algorithm processes the data sequentially in an almost streaming manner from left to right. According to the Proposition 4.3, the time complexity of finding a matched subsequence of length m_i is $O(m_i^2)$. Updating the hash table for the matched subsequence requires inserting m_i entries into the table, which also takes $O(m_i^2)$ time. Therefore, the total time cost for processing a matched subsequence of length m_i is $O(m_i^2)$. For unmatched data, the algorithm can determine whether a data point is unmatched in $O(1)$ time and insert it into the appropriate hash table entry in $O(1)$ time as well. Therefore, the total time cost for processing an unmatched subsequence of length u_i is $O(u_i)$. So, the overall time complexity of the algorithm is:

$$O(\sum_{i=1}^{b}(m_i^2 + u_i)) \leq O(l \sum_{i=1}^{b}(m_i + u_i)) = O(ln).$$

□

Without leveraging hash-based techniques, finding the maximal match position for each index i requires traversing all preceding positions, leading to a time complexity of $O(ln^2)$. LZ77 introduces a sliding window of size w, restricting the search for matches to only the preceding w characters. This optimization reduces the search range and lowers the time complexity to $O(wln)$ [12]. LZ4 further improves efficiency by employing a simple hash-based lookup strategy, reducing its average-case time complexity to $O(n)$.

The complexity of LZV mainly comes from the repeated checking of data in the recursive iteration and the calculation of the hash function. These steps are primarily designed to ensure the theoretical correctness of the algorithm in the presence of hash collisions. In practice, we can ignore the slight matching effect, avoid redundant checks, and only perform the check at the end. Further optimizations, such as using incremental hash calculation, can further reduce the algorithm's execution time. The effectiveness of these is examined in detail in Section 7.2.5.

4.5 Space Cost

To maintain the integrity of the information stored in the hash table, the LZV computes and updates the hash value for each subsequence starting at every position. Consequently, for a sequence D of length n, the LZV hash table stores n entries. Although LZ4 often skips hash computations after a

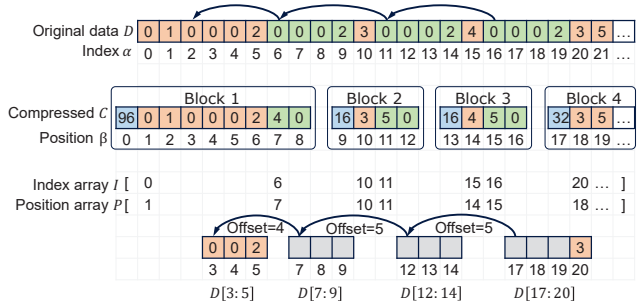


Fig. 6. An example of compressed index search.

match is found, it is still required to compute and record n hash values in the worst case when no matches are present. Our method uses $O(n)$ space to search for the maximal match position, which is already a space-optimized solution. In contrast, a naive implementation would require $O(n^2)$ space, as there are $O(n^2)$ possible subsequences, each of which might represent an maximal match. By adopting an iterative search strategy, we reduce the space requirement from $O(n^2)$ to $O(n)$, achieving a significant improvement in memory efficiency while maintaining effective matching performance.

Prior studies have analyzed the relationship between the hash table size and collision rate. The expected collision rate of a uniform hash function under load factor $\alpha = |S|/|T|$ can be approximated by: $P_c = 1 - e^{-\alpha(1+\alpha)}$ [17], where $|T|$ is the total size of hash table, and $|S|$ is the number of occupied entries. Empirical studies further confirm that maintaining a relatively small load factor significantly reduces collision probability, with $\alpha = 0.75$ commonly adopted as the default threshold for hash table resizing [24].

Hash collisions may affect the compression ratio. It might seem that compression tasks, especially for long sequences, require very large hash tables as analyzed above. However, empirical evidence suggests otherwise. For example, LZ4's default configuration utilizes a relatively small hash table of only 4,096 entries. Our experimental results in Figure 13a further show that for both LZ4 and LZV, increasing the hash table size beyond this default yields only marginal improvements in compression ratio. This phenomenon occurs because, in compression contexts, a single failed hash lookup only impairs the compression of one byte. For instance, to completely fail to compress a sequence of 100 bytes, approximately 96 consecutive failed lookups would be required, an event whose probability decreases exponentially.

5 LZ4 Query Processor

In a key-value storage system, retrieving a value for a given key typically involves two steps: locating the key's index within the key column, and then using that index to retrieve the corresponding value from the value column. To accelerate this retrieval process on compressed data in LZ4 format, we propose two specialized techniques: **compressed key search** and **compressed index search** to optimize the key lookup and index-based value access, respectively.

5.1 Compressed Index Search

Compressed index search (CIS) is the method to locate the value corresponding to a given index in the original data from the compressed representation. Specifically, given an index α of the original data D and the compressed data C, CIS is used to find the corresponding index β in C, i.e.,

$$\text{CIS}(C, \alpha) = \beta \quad \text{where} \quad C[\beta] = D[\alpha].$$

5.1.1 Operation. CIS consists of the following two main steps.

Step 1: Constructing the Auxiliary Arrays.

Index construction is essential to efficient query on compressed data. In the LZ4 format, compressed data is organized in units of blocks. Each block records its length, but not its absolute position, and later blocks may depend on any earlier ones. Therefore, constructing an index that establishes the relationship of each block within the entire sequence becomes essential.

The auxiliary structure comprises two arrays: the index array I, which records the indices in the original data D, and the position array P, which stores the corresponding indices in the compressed data C. For each LZ4-compressed block, we append the starting indices of both the unmatched and matched parts in D to I, and the starting indices of the corresponding literals and offsets in C to P.

Index construction on compressed data is non-trivial. First, the size of the index to be constructed is linear to the number of blocks in the compressed data. As a result, the construction overhead is much smaller than that of fully decompressing, making it a very lightweight and fast process. The relation between the number of compressed blocks and original data length is given in Proposition 5.3. Second, index construction does not require full decompression; instead, it only fast-decompresses the metadata of each block (e.g., in Figure 6, only the blue and green data are decompressed). Third, multiple queries on the same compressed data can reuse the constructed index, further amortizing the cost. Finally, index construction can be terminated early once the query boundary is reached, avoiding unnecessary computation.

Step 2: Jump-based Querying on Compressed Data.

Using the auxiliary arrays constructed in Step 1, we locate the corresponding blocks in the compressed data. The subsequent query operations over data compressed in the LZ4 format resemble the direct access technique proposed by Sebastian Kreft and Gonzalo Navarro for data compressed by LZ77 in triple format [20].

Specifically, for a query of $D[\alpha]$, we first use the index array I to identify the compressed block x that contains index α, and determine whether α lies within the unmatched or matched subsequence of that block. If α lies within the unmatched subsequence (i.e., $I[2x] \leq \alpha < I[2x+1]$), we directly get the corresponding value as $D[\alpha] = C[P[2x] + (\alpha - I[2x])]$.

If α lies within the matched subsequence (i.e., $I[2x+1] \leq \alpha < I[2x+2]$), we first use the position array P to obtain the corresponding offset o from the compressed data: $o = C[P[2x{+}1] : P[2x{+}1]{+}1]$. We then find the smallest integer k such that $\alpha - ko < I[2x+1]$. At this point, the query of $D[\alpha]$ is transformed into the query of $D[\alpha - ko]$, which signifies that the query has been redirected to a prior compressed block.

Example 6. *Figure 6 illustrates an example of performing compressed index search on compressed data in LZ4 format. When querying $D[17, 20]$, we begin by constructing the auxiliary arrays. According to the index array I, the element at position 20 lies within the unmatched subsequence of compressed block 4, whereas the range $[17, 19]$ falls within the matched subsequence of block 3.*

For position 20, we can directly retrieve its corresponding value the from compressed data at position 18, as indicated by the position array P. For the matched range $[17, 19]$ in compressed block 3, value $C[P[5] : P[5]{+}1] = 5$ indicates that the matched subsequence in this block has an offset of 5. Therefore, the query is redirected to a new range $[17 - 5 : 19 - 5] = [12, 14]$.

This redirection process continues recursively: the query range $[12, 14]$ is transformed into $[7, 9]$, and then further into $[3, 5]$. Since the range $[3, 5]$ lies entirely within an unmatched subsequence, the corresponding values can now be directly accessed from the compressed data. In this way, all original values required by the initial query of $D[17, 20]$ are successfully retrieved.

5.1.2 Time Complexity. Linear time complexity with respect to the number of compressed data blocks is inevitable. In the worst case, the data in each block depends on the data in the preceding

block. As a result, accessing the data in the last block requires sequentially traversing all the previous blocks. We formalize this in the proposition below.

Proposition 5.1. *Let D be an array of length n, and let C be an LZ4 format representation of D consisting of b blocks. In the worst case, retrieving the last element $D[n-1:n-1]$ from C requires at least $O(b)$ time.*

Proof. In the LZ4 compression format, the data in each block may depend on the data in the preceding block. In the worst case, the last element in block B_{b-1} depends on the data in block B_{b-2}, the data in block B_{b-2} depends on the data in block B_{b-3}, and so on, with each block depending on the preceding block until the first block B_0. As a result, accessing the data in the last block requires sequentially traversing all the previous blocks, leading to at least $O(b)$ time complexity. □

The following proposition indicates that the query time of **CIS** is linearly correlated with the number of compressed data blocks.

Proposition 5.2. *Given the compressed data C in LZ4 format consisting of b blocks, the time complexity of compressed-data query using CIS is $O(b)$.*

Proof. For compressed data consisting of b blocks, the each auxiliary array stores two entries per block, resulting in a total of $4b$ entries. Therefore, the time complexity of constructing the auxiliary arrays is $O(b)$. During query execution, only the jump operations involve non-trivial computation, such as boundary checks and offset-based redirection. Each jump leads to a previous block. As a result, the number of possible jumps is bounded by the number of blocks, and the total time complexity of a single jump-based query is also $O(b)$ in the worst case. Hence, the total time complexity of the CIS procedure, including both auxiliary structure construction and jump-based querying, is $O(b)$. □

Furthermore, Proposition 5.3 establishes a theoretical relationship between the number of blocks in the compressed data and its total length, showing that the number of compressed blocks is upper bounded by the length of the compressed data. As a result, the time complexity of CIS is linearly bounded by the size of the compressed data. This also implies that a higher compression ratio leads to a tighter upper bound on query time.

Proposition 5.3. *Given the compressed data C in LZ4 format with b blocks of m bytes, there is $b \leq \frac{m}{3}$.*

Proof. Suppose the length of the original data D is n, and each block in D have a length denoted by n_i. Within this block, let m_i represent the length of the matched subsequence, and u_i represent the length of the unmatched subsequence. Then, there is

$$n = \sum_{i=0}^{b-1} n_i = \sum_{i=0}^{b-1} (m_i + u_i),$$

$$m = \sum_{i=0}^{b-1} \left(3 + u_i + \left\lceil \frac{m_i - 15}{128} \right\rceil + \left\lceil \frac{u_i - 15}{128} \right\rceil\right) \geq \sum_{i=0}^{b-1} (3 + u_i) \geq 3b$$

Therefore, we have $b \leq \frac{m}{3}$. □

5.2 Compressed Key Search

Compressed key search (CKS) is the method used to locate the index of a given key in the original data by searching within the compressed, sorted key column. Specifically, given a key k in the original data D and the compressed data C, CKS returns the index α such that $D[\alpha] = k$, i.e.,

$$\text{CKS}(C, k) = \alpha \text{ where } D[\alpha] = k.$$

It is indeed a more involved operation. The previous solution just first predicts (extrapolates) a search range and then applies binary search within that range. For compressed keys, we notice that the intervals between keys may vary, e.g., with both regular and irregular time intervals in different segments of a time series [11]. CKS leverages this property by alternating between prediction and binary (median) search.

Specifically, we first construct a model based on the metadata of the compressed data. This model is used to predict the index α_p corresponding to the target key k. We then retrieve the key k_p at α_p from the compressed data by compressed index search (CIS). If $k_p = k$, the correct index has been found. Otherwise, the model is updated using the observed value k_p at α_p like the binary search, and the prediction process is repeated.

As explained in Section 5.1, CIS needs to construct an index at first. However, a single CKS may involve multiple CIS calls, but these calls can share the same index. In each CIS process, it starts to search at the index α_p and jump forward until the exact text is found. Experimental result in Table 4 shows that, for most datasets, the number of CIS calls is relatively small, enabling compressed queries to achieve substantial time savings.

Regarding the prediction model used in CKS, we employed a linear model. This model has a key property that its construction requires only the values at the two endpoints, and it relies on the assumption that keys are nearly evenly distributed. Its construction does not require training. Instead, it only makes use of the endpoint values obtained from each query.

We acknowledge that the prediction used in CKS is related to the learned index [18], as it also uses a predictive model to estimate key positions in sorted data. For compressed keys, we notice that the intervals between keys may vary, e.g., with both regular and irregular time intervals in different segments of a time series. Our approach of alternating prediction with median (binary) search is highly efficient for compressed queries, without being overly dependent on the specific distribution of the intervals. The learned index needs to learn, construct, and store the index structure based on the data, which incurs additional space overhead. It contradicts the purpose of compressing data for lower storage cost, while our proposal processes directly on the compressed data without extra storage cost.

Example 7. *Figure 7 illustrates the specific procedure of performing CKS on a sorted key column. Suppose the original key column is given by* $\{k_i = i^2 \mid i = 1, 2, \ldots, 10\}$*, the index range is* $[1 : 10]$ *and we aim to query the index corresponding to* $k = 16$.

First, CKS does a prediction-based search by constructing a linear model using the metadata $k_1 = 1$ *and* $k_{10} = 100$*, according to the following formula:* $\frac{k-k_1}{k_{10}-k_1} = \frac{index-1}{10-1}$*. Using this linear model with* $k = 16$*, we obtain the first predicted index* α_p *is 2. We then query the compressed time column at index 2 and retrieve* $t_2 = 4$*. Since* $t_2 < t$*, we update the search index range from*$[1 : 10]$ *to* $[2 : 10]$.

Second, CKS performs a binary search step. Given the search range $[2 : 10]$*, it selects the midpoint index* $\alpha_b = 6$*, and retrieves the key value* $k_6 = 36$*. Since* $k_6 > k$*, the search range is updated to* $[2 : 6]$.

Next, CKS conducts a prediction-based query within the updated range $[2 : 6]$*. Given that* $k_2 = 4$ *and* $k_6 = 36$*, a linear model is constructed, as previously described, to predict the next probe index* $\alpha'_p = 4$*. Querying index* 4 *returns* $k_4 = 16$*, which matches the target key. Therefore, CKS successfully returns the result* 4.

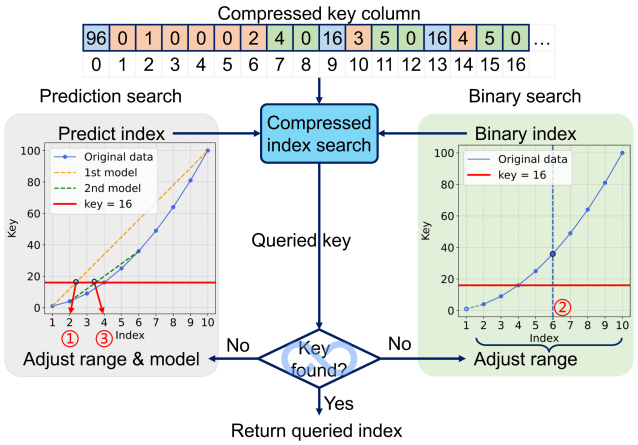


Fig. 7. Compressed key search, querying the index of key $k = 16$. The first step ① uses prediction search, yielding index 2, then ② binary search gives index 6, and finally prediction search ③ returns index 4.

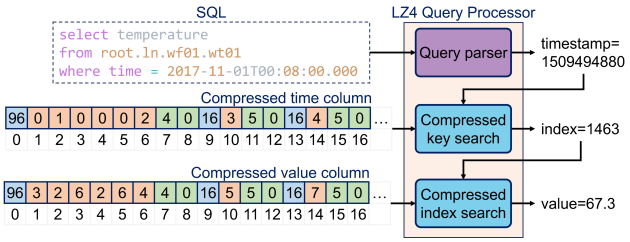


Fig. 8. System implementation in Apache TsFile.

Regarding time complexity, the integrated approach achieves $O(1)$ complexity for time columns with stable growth, and maintains $O(\log n)$ complexity for irregularly growing data. As a result, the method attains optimal performance in both scenarios.

6 System Implementation

We implement a compressed-data query pipeline in Apache TsFile, a new open-source time series storage system. The query processing workflow is illustrated in Figure 8. Specifically, given an input SQL query with a temporal constraint, we first extract the time condition and execute a *compressed key search* on the compressed timestamp column. This yields the index positions of the relevant time range. Subsequently, we use the resulting index to perform a compressed key search on the compressed value column to retrieve the corresponding data points.

As a compression algorithm, `LZV` is implemented in the class `LzVCompressor`, which inherits from the abstract base class `ICompressor`. The encoding scheme strictly follows the standard LZ4 format, ensuring that the resulting compressed data remains compatible with any standard LZ4 decoder.

Since the time column in time series data is strictly increasing with no duplicate values, LZ77-based algorithms do not achieve good compression on it. So, we choose to compress the residuals between the original timestamps and their predicted values—this represents the simplest case of the LeCo [21]. This transformation is implemented in the class `LongPredictEncoder`, which

Table 2. Dataset Properties.

Dataset	Equally	Mode Ratio	Length
Electricity [31]	Yes	100%	26303
GPS	No	11.7%	100000
Samsung [35]	No	4.9%	100000
P12 [29]	No	39.4%	12000
AirQuality [33]	Yes	100%	9357
CS-Sensors	Yes	100%	99999
Cyber-Vehicle	No	15.9%	7278
EPM-Education [32]	No	67.7%	99998
FANYP-Sensors	No	97.5%	100000
GW-Magnetic [6]	No	44.4%	18352
Metro-Traffic [14]	No	78.8%	48202
Nifty-Stocks	No	75.6%	2047
TH-Climate	No	76.8%	100000
TRAJET-Transport	No	53.2%	100000
TY-Fuel	No	88.4%	29729
TY-Transport	No	63.8%	51221
USGS-Earthquakes	No	0.8%	75808

inherits from the `Encoder` interface. The corresponding decoding logic is provided by the class `LongPredictDecoder`.

The mechanism for querying compressed data is implemented in the class `LZ4CompressedQuery`. In particular, for timestamp columns, we design a specialized prediction-based query function, `compressedPredictBinaryDeltaSearch`, which combines prediction with binary search. This method leverages the regularity of timestamp sequences to narrow down the search range efficiently, improving query performance compared to pure binary search.

7 Experiment

In this section, we present comprehensive experiments. First, Section 7.1 introduces the experimental setup and datasets. In Section 7.2, we evaluate the performance of LZV. Section 7.3 examines the effectiveness of the two compressed-data query methods: CIS and CKS. In Section 7.4, we implement these methods into the TsFile system and evaluate the overall performance of the compressed-data query pipeline from a system-level perspective.

7.1 Experimental Setup

All the experiments are run on a machine equipped with Intel(R) Core(TM) i9-13900H CPU@2.60GHz and 32GB memory. The LZ4 algorithm used in our evaluation is based on the standard implementation provided by the net.jpountz.lz4 library. For comparison, we implement the naive LZ77 algorithm, using a window size of 65536. To ensure consistency, all compression results are encoded using the standard LZ4 format. Decompression for all algorithms is performed using the standard LZ4 decoder. The source code and datasets used in the experiments are available at [1].

Table 2 summarizes the datasets used in our experiments. For datasets with excessive length, we use the first 100,000 points for consistency. Since the timestamps are unique, we use the value by subtracting the linear regression to improve the performance of all compression algorithms on the time column. In addition, the table indicates whether the time column is strictly equally spaced and reports the proportion of the most frequent interval to characterize the regularity of

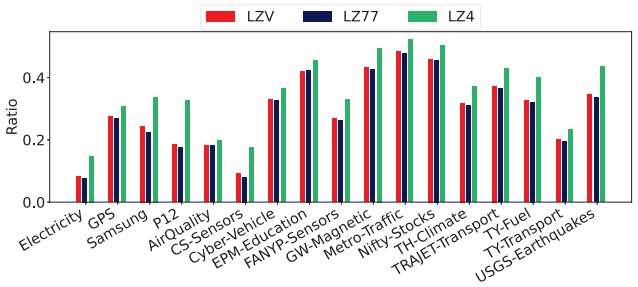


Fig. 9. Compression ratio.

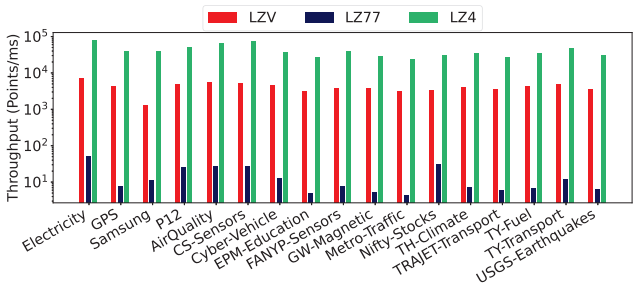


Fig. 10. Compression speed.

time spacing. These properties validate our observations about the distributional characteristics of the key column in Section 5.2. Overall, our datasets feature both equally and unequally spaced, and value types including integers and floats. More details are available at [1].

7.2 Compression Performance

We evaluate the compression performance of LZV in terms of compression ratio, compression time, and decompression time.

7.2.1 Compression Ratio. Figure 9 presents the compression ratios of various compression algorithms. The compression ratio is defined as: $ratio = \frac{compressedSize}{uncompressedSize}$. A lower $ratio$ indicates better compression performance. Overall, LZV demonstrates a significant improvement over LZ4 across all datasets, highlighting its effectiveness. Although it performs slightly worse than the LZ77, this is partly due to unavoidable hash collisions and partly due to engineering optimizations made to accelerate the algorithm. For a detailed analysis and ablation study regarding this, please refer to Sections 7.2.4 and 7.2.5.

7.2.2 Compression Speed. Figure 10 presents the compression throughput of various compression algorithms across all datasets. Overall , LZV is approximately one order of magnitude slower than LZ4, but it is 2 to 3 orders of magnitude faster than the naive LZ77 implementation. The performance gap with LZ4 mainly stems from the more complex hash lookup and update mechanisms we introduced to improve match quality. However, LZV still significantly outperforms LZ77 in speed, demonstrating the advantage of using hash-based matching strategies.

7.2.3 Decompression Speed. Figure 11 presents the decompression throughput of various compression algorithms across all datasets. Unlike the variations observed during compression, the

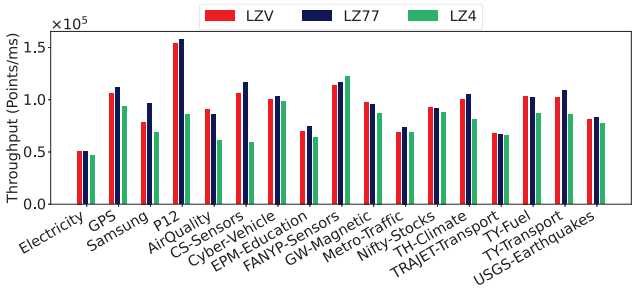


Fig. 11. Decompression speed.

(a) Compression ratio

(b) Compression speed

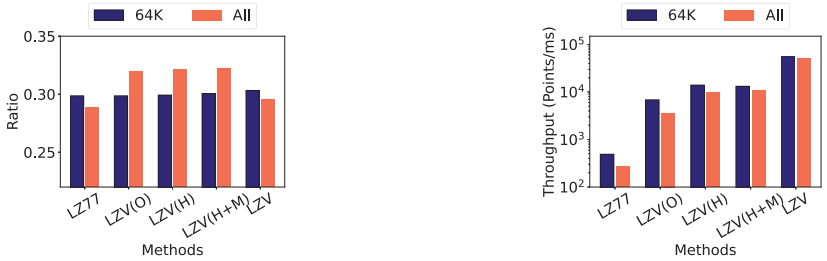
Fig. 12. Ablation study of hash table size.

decompression speeds across different algorithms are relatively similar. This is because all compressed data are stored in the LZ4 format, allowing the use of a standard LZ4 decompressor for decompression. Overall, LZ77 exhibits slightly faster decompression speed compared to LZV, while LZ4 demonstrates the slowest decompression performance. In general, a better compression ratio tends to result in higher decompression throughput.

7.2.4 Hash Table Size. Overall, the difference in compression ratio shown in Figure 9 between LZV and LZ77 is small, indicating that the impact of hash collisions is likely limited. We include experiments analyzing the impact of hash collisions on the compression ratio and speed in Figure 12, considering different hash table sizes. To provide a clearer presentation, we present the average results across all datasets here. Overall, increasing the size of the hash table brings positive gains in compression ratio for both LZ4 and LZV. In addition, LZV consistently achieves a better compression ratio than LZ4 under all tested hash table sizes. Furthermore, the benefit of enlarging the hash table is more pronounced for LZV.

However, the improvement gradually diminishes and eventually becomes negligible as the hash table continues to grow. Specifically, for the LZV, when the hash table size is large, the difference in compression ratio between LZV and LZ77, as shown in Figure 9, is small, indicating that the impact of hash collisions is limited.

7.2.5 Ablation Study. Apart from hash collisions, there are other factors that may affect the compression ratio. In these experiments, LZV (O) is implemented strictly according to Section 4, excluding hash collisions, while LZV (H) includes their effects. LZV (H+M) further constrains the maximum match length to 256. LZV also includes optimizations like periodic hash table updates for better compression on long sequences and reduced iterative lookups for faster compression.



(a) Compression ratio. (b) Compression speed.

Fig. 13. Ablation study of LZV.

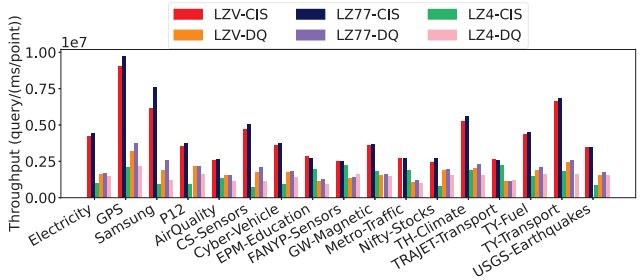


Fig. 14. Compressed index search throughput

Since the LZ4 format requires matched positions to stay within a 64K window, we tested datasets of both 64K and longer lengths. For simplicity, we present the average results across all datasets.

Figure 13a shows the compression ratio results. For datasets with a size of 64K, the compression ratio differences are minimal, with LZV (O) and LZ77 performing identically. Hashing has little effect, as the 2^{20} hash table is large enough, as shown in Figure 12a. In fact, LZV achieves up to 99% of the compression performance of LZ77 in most cases. However, for datasets longer than 64K, standard LZV discards matches that exceed the 64K window, resulting in poor performance for LZV (O, H, H+M). Although the optimizations on LZV slightly compromise the theoretical matching results, they help LZV handle long datasets better. The optimized LZV achieves about 95% of the compression performance of LZ77 on most datasets.

Figure 13b shows compression speed results. Overall, the LZV-based method with hashing significantly outperforms LZ77 in speed. For longer datasets, the throughput decreases for the first three methods. For LZ77, this is because the average search window size approaches 64K, compared to an average 32K window size for 64K datasets. In LZV (O, H), longer matches in larger datasets reduce their throughput. In contrast, the optimized LZV improves throughput and performs well on both short and long datasets.

7.3 Compressed-Data Query Performance

We evaluate the query performance of CIS and CKS in this Section.

7.3.1 Compressed Index Search Performance. For each dataset, we sequentially measured the average query time for querying the value at each index position. Since the query time is proportional

Table 3. Jump times in compressed index search (avg/max)

Dataset	LZ77	LZ4	LZV (our)
Electricity	17/35	1155/12928	26/71
GPS	32/76	2004/5654	45/112
Samsung	29/91	4365/11477	51/170
P12	14/25	532/2134	19/43
AirQuality	12/20	253/1450	14/29
CS-Sensors	38/103	5749/37011	51/197
Cyber-Vehicle	14/28	421/4181	23/67
EPM-Education	28/79	560/2913	32/165
FANYP-Sensors	24/57	408/3106	28/75
GW-Magnetic	15/43	274/902	17/42
Metro-Traffic	19/45	499/2855	21/60
Nifty-Stocks	12/25	151/790	14/29
TH-Climate	34/91	1404/5021	41/142
TRAJET-Transport	28/83	413/3446	29/130
TY-Fuel	17/39	709/3664	23/55
TY-Transport	21/58	1188/5543	31/94
USGS-Earthquakes	24/60	3412/31567	31/101

to the sequence length, we further normalized the results by dividing the average time by the sequence length. The detailed experimental results are shown in Figure 14.

Overall, compressed-data queries on data compressed by LZ77 and LZV achieve the highest throughput. In contrast, for data compressed by LZ4, the throughput of compressed-data queries is, in some cases, even lower than that of decompressed queries.

The longer query time for LZ4-compressed data is due to its hash design. LZ4 uses a simple hash mechanism, causing frequent hash replacements and long dependency chains. This leads to more jumps during queries. As shown in Table 3, both the average and maximum number of jumps for LZ4 are about two orders of magnitude higher than for LZ77 and LZV. Therefore, querying LZ4-compressed data is more time-consuming than querying the data after decompression in many cases.

7.3.2 Compressed Key Search Performance. Table 4 reports the average number of probing steps required to locate a target key using different query strategies on the key column. As shown, the pure prediction-based approach performs well on datasets with perfectly equidistant keys. However, its performance degrades significantly on datasets with non-uniform key intervals. In contrast, binary search is insensitive to key spacing and consistently requires approximately log steps on average. The hybrid strategies preserve the efficiency of prediction on equidistant datasets while retaining the robustness of binary search on irregular datasets, maintaining log probing steps even under non-uniform key distributions.

The choice of prediction model directly affects the number of searches in CKS. In addition to using a simple linear model for fitting, we also attempt to use a quadratic function model. Compared with the simple linear fitting, the quadratic fitting requires a higher computational cost, while the improvement in fitting accuracy (in terms of the number of searches required, as shown in Table 4) is not significant, and in some cases even worse than the linear fitting. Therefore, we adopt the combination of linear fitting and binary search as the default search method used in CKS.

Table 4. Search times in compressed key search (avg)

Dataset	Lin.	Bin.	Quad.+Bin.	Lin.+Bin.
Electricity	1.0	13.8	1.1	1.1
GPS	25.6	15.7	15.4	15.7
Samsung	3.4	15.7	5.7	5.7
P12	3.5	12.6	5.6	5.8
AirQuality	1.0	12.2	1.1	1.1
CS-Sensors	1.1	15.7	1.1	1.1
Cyber-Vehicle	10.9	11.9	12.3	12.3
EPM-Education	477.0	13.7	13.9	12.7
FANYP-Sensors	7.0	10.2	5.6	4.2
GW-Magnetic	538.5	13.2	11.6	10.7
Metro-Traffic	8.0	14.3	11.0	10.7
Nifty-Stocks	2.2	10.0	3.5	3.5
TH-Climate	29.3	15.7	9.9	9.0
TRAJET-Transport	12.1	14.3	9.7	9.2
TY-Fuel	12.5	13.9	14.3	14.3
TY-Transport	16.9	14.2	9.0	8.9
USGS-Earthquakes	5.4	15.3	9.0	9.0

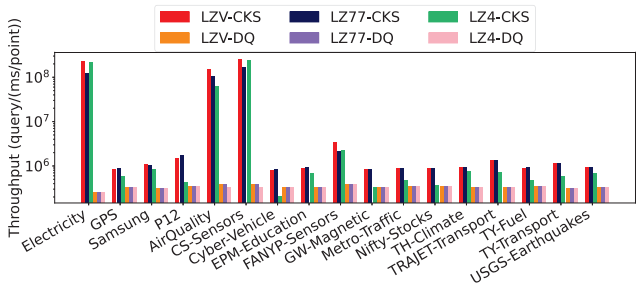


Fig. 15. Compressed key search throughput.

Figure 15 shows the performance of compressed key search. For each dataset, we sequentially measured the average query time for retrieving the corresponding index for each timestamp in the compressed time column. The query time inherently contains a linear factor with respect to the data size as analyzed in Section 5.1.2, and the data size used in the experiments vary significantly. In order to present the results on the same scale, we normalized the results via dividing the average time by the data size.

Overall, compressed-data query performs better, particularly on datasets Electricity, AirQuality, and CS-Sensors, where the key column is equally spaced. Moreover, CKS on data compressed by LZ77 and LZV achieves the highest throughput. For data compressed by LZ4, compressed-data queries achieve higher throughput than decompressed queries in the vast majority of cases.

7.4 System Evaluation

Figure 16 presents the experimental results of performing compressed-data queries within a specific key-value store system, Apache TsFile. We evaluated the throughput of both decompression queries

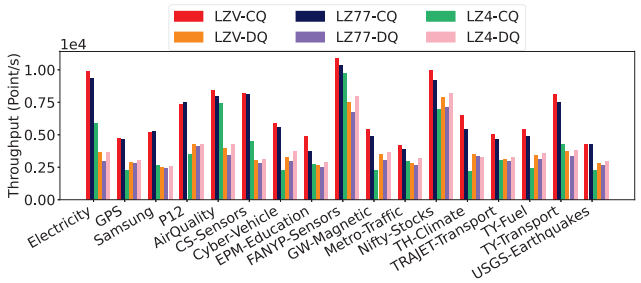


Fig. 16. Query throughput in Apache TsFile.

(now) and compressed-data queries. Specifically, using the SQL statement shown in Figure 8, we queried the value at a given timestamp. The experimental results are presented in Figure 16.

Overall, the queries CQ on data compressed by LZV and LZ77 achieve the highest throughput. For data compressed by LZ4, compressed-data queries outperform decompression queries in some scenarios, while the opposite holds in others. This verifies the necessity of improving LZ4 by enhancing its match quality and reducing data dependency. For different datasets, a better compression ratio generally leads to improved query performance, supporting the linear complexity analysis with respect to compressed data size presented in Proposition 5.2. In summary, the system-level experiments confirm the benefits of performing compressed-data queries on data in LZ4 format, particularly on data compressed by LZV and LZ77.

8 Conclusion

Our paper tackles three core data management problems: effective data compression, compressed data indexing and efficient compressed data querying. (1) For compression algorithm, LZV provides an innovative optimization over both LZ4 and LZ77. Compared with LZ4, LZV can identify longer as well as earlier matches, which not only improves the compression ratio but also enhances performance of compressed queries. Compared with LZ77, LZV uses hashing to significantly reduce the compression time. (2) For compressed data indexing, it is essential for compressed query and not simplistic for its construction cost and other optimizations. (3) For compressed queries, we support direct querying over LZ4-format compressed data, which to the best of our knowledge is the first such attempt. To accelerate query processing, we propose a method that alternates between prediction and binary search. In compressed keys, the intervals between keys may change dynamically over time. Our alternating approach is robust to such variations and achieves better performance than binary search in experiments. Additionally, we implement a query pipeline over compressed data within Apache TsFile to evaluate the effectiveness and efficiency of our compression and query methods on real-world systems. The experimental results demonstrate that LZV incurs higher compression time in exchange for a better compression ratio and improved query performance, making it well-suited for write-once-read-many (WORM) database scenarios.

Acknowledgments

This work is supported in part by the National Key Research and Development Plan (2025ZD1601701, 2024YFB3311901, 2021YFB3300500), the National Natural Science Foundation of China (62232005, 92267203, 62021002), State Grid Ningxia Electric Power Co. Science and Technology Project (SGNXYX00SCJS2400058), Beijing National Research Center for Information Science and Technology (BNR2025RC01011), and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

References

[1] 2025. Source Code and Datasets. https://github.com/liuzhiheng20/LZV. Accessed: 2025-07-12.

[2] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, Jason Tsong-Li Wang (Ed.). ACM, 967–980. https://doi.org/10.1145/1376616.1376712

[3] Apache Cassandra Documentation. 2023. Compression. https://cassandra.apache.org/doc/latest/cassandra/managing/operating/compression.html. Accessed: 2025-07-03.

[4] Apache IoTDB Documentation. 2025. Encoding and Compression — Apache IoTDB. https://iotdb.apache.org/UserGuide/latest-Table/Technical-Insider/Encoding-and-Compression.html. Accessed: 2025-07-03.

[5] Apache TsFile. n.d.. Apache TsFile GitHub Repository. https://github.com/apache/tsfile. Accessed: 2025-07-07.

[6] Barsocchi, Paolo, Crivello, Antonino, Rosa, Davide, Palumbo, and Filippo. 2016. Geo-Magnetic field and WLAN dataset for indoor localisation from wristband and smartphone. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5DW43.

[7] Matej Bartik, Sven Ubik, and Pavel Kubalík. 2015. LZ4 compression algorithm on FPGA. In 2015 IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2015, Cairo, Egypt, December 6-9, 2015. IEEE, 179–182. https://doi.org/10.1109/ICECS.2015.7440278

[8] Tomás Benes, Matej Bartík, and Pavel Kubalík. 2019. High Throughput and Low Latency LZ4 Compressor on FPGA. In 2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019, David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner (Eds.). IEEE, 1–5. https://doi.org/10.1109/RECONFIG48160.2019.8994794

[9] ClickHouse Documentation. 2024. Compression Modes — ClickHouse Documentation. https://clickhouse.com/docs/en/data-compression/compression-modes. Accessed: 2025-07-03.

[10] Yann Collet. 2011. Real Time Data Compression: LZ4 Explained. https://fastcompression.blogspot.com/2011/05/lz4-explained.html [Online; accessed 28-June-2025].

[11] Andreas Eckner. 2012. A framework for the analysis of unevenly spaced time series data. Preprint. Available at: http://www. eckner. com/papers/unevenly_spaced_time_series_analysis (2012), 93.

[12] Keisuke Goto and Hideo Bannai. 2013. Simpler and Faster Lempel Ziv Factorization. In 2013 Data Compression Conference, DCC 2013, Snowbird, UT, USA, March 20-22, 2013, Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer (Eds.). IEEE, 133–142. https://doi.org/10.1109/DCC.2013.21

[13] Yutong Han, Bin Wang, Xiaochun Yang, Tao Qiu, and Huaijie Zhu. 2019. Efficient regular expression matching on LZ77 compressed strings using negative factors. World Wide Web 22, 6 (2019), 2519–2543. https://doi.org/10.1007/S11280-019-00667-Z

[14] Hogue and John. 2019. Metro Interstate Traffic Volume. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5X60B.

[15] Google Inc. 2023. Snappy: A Fast Compression Format. https://github.com/google/snappy. Accessed: 2025-04-28.

[16] Hao Jiang and Sian-Jheng Lin. 2020. A Rolling Hash Algorithm and the Implementation to LZ4 Data Compression. IEEE Access 8 (2020), 35529–35534. https://doi.org/10.1109/ACCESS.2020.2974489

[17] Donald E. Knuth. 1998. The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.

[18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 489–504. https://doi.org/10.1145/3183713.3196909

[19] Sebastian Kreft and Gonzalo Navarro. 2010. LZ77-Like Compression with Fast Random Access. In 2010 Data Compression Conference (DCC 2010), 24-26 March 2010, Snowbird, UT, USA, James A. Storer and Michael W. Marcellin (Eds.). IEEE Computer Society, 239–248. https://doi.org/10.1109/DCC.2010.29

[20] Sebastian Kreft and Gonzalo Navarro. 2011. Self-indexing Based on LZ77. In Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6661), Raffaele Giancarlo and Giovanni Manzini (Eds.). Springer, 41–54. https://doi.org/10.1007/978-3-642-21458-5_6

[21] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. Proc. ACM Manag. Data 2, 1 (2024), 65:1–65:28. https://doi.org/10.1145/3639320

[22] LZ4. n.d.. LZ4 - Extremely Fast Compression. https://lz4.org/ Accessed: 2025-07-16.

[23] Mark R. Nelson. 1989. LZW data compression. Dr. Dobb's J. 14, 10 (Oct. 1989), 29–36.

[24] Oracle Corporation. 2024. Class HashMap (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. Accessed: 2025-10-14.

[25] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. Proc. VLDB Endow. 8, 12 (2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[26] Feifan Pu and Jianqiu Xu. 2025. TSQ: An Optimized Framework for Efficiently Answering Time Series Queries. Data Sci. Eng. 10, 2 (2025), 296–313. https://doi.org/10.1007/S41019-024-00273-8

[27] RocksDB Documentation. 2022. Compression (RocksDB Wiki). https://github.com/facebook/rocksdb/wiki/Compression. Accessed: 2025-07-03.

[28] David Salomon. 2004. Data Compression: The Complete Reference, 3rd Edition. Springer. http://www.davidsalomon.name/DC3advertis/DComp3Ad.html

[29] Ikaro Silva, George Moody, Daniel J Scott, Leo A Celi, and Roger G Mark. 2012. Predicting in-hospital mortality of icu patients: The physionet/computing in cardiology challenge 2012. In 2012 computing in cardiology. IEEE, 245–248.

[30] LanceDB Team. 2025. Columnar File Readers In-Depth: Compression Transparency. https://www.lancedb.com/blog/blog/columnar-file-readers-in-depth-compression-transparency/. Accessed: 2025-07-06.

[31] Artur Trindade. 2015. ElectricityLoadDiagrams20112014. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58C86.

[32] Vahdat, Mehrnoosh, Oneto, Luca, Anguita, Davide, Funk, Mathias, Rauterberg, and Matthias. 2015. Educational Process Mining (EPM): A Learning Analytics Data Set. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5NP5K.

[33] Vito and Saverio. 2008. Air Quality. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C59K5F.

[34] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. Proc. ACM Manag. Data 1, 2 (2023), 195:1–195:27. https://doi.org/10.1145/3589775

[35] Wolfgang Weiss, Víctor Juan Expósito Jiménez, and Herwig Zeiner. 2017. A Dataset and a Comparison of Out-of-Order Event Compensation Algorithms. In International Conference on Internet of Things, Big Data and Security. https://api.semanticscholar.org/CorpusID:43274009

[36] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. Proc. VLDB Endow. 17, 2 (2023), 148–161. https://doi.org/10.14778/3626292.3626298

[37] Xin Zhao, Jialin Qiao, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. 2024. Apache TsFile: An IoT-native Time Series File Format. Proc. VLDB Endow. 17, 12 (2024), 4064–4076. https://doi.org/10.14778/3685800.3685827

[38] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[39] J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934

Received July 2025; revised October 2025; accepted November 2025