LeCo: Lightweight Compression via Learning Serial Correlations

YIHAO LIU, Institute for Interdisciplinary Information Science, Tsinghua University, China
XINYU ZENG, Institute for Interdisciplinary Information Science, Tsinghua University, China
HUANCHEN ZHANG, Institute for Interdisciplinary Information Science, Tsinghua University & Shanghai Qi Zhi Institute, China

Lightweight data compression is a key technique that allows column stores to exhibit superior performance for analytical queries. Despite a comprehensive study on dictionary-based encodings to approach Shannon's entropy, few prior works have systematically exploited the serial correlation in a column for compression. In this paper, we propose LeCo (i.e., Learned Compression), a framework that uses machine learning to remove the serial redundancy in a value sequence automatically to achieve an outstanding compression ratio and decompression performance. LeCo presents a general approach to this end, making existing algorithms such as Frame-of-Reference (FOR), Delta Encoding, and Run-Length Encoding (RLE) special cases under our framework. Our microbenchmark with three synthetic and eight real-world data sets shows that a prototype of LeCo achieves a Pareto improvement on both compression ratio and random access speed over the existing solutions. When integrating LeCo into widely-used applications, we observe up to 5.2× speed up in a data analytical query in the Arrow columnar execution engine, and a 16% increase in RocksDB 's throughput.

CCS Concepts: • **Information systems → Compression strategies**.

Additional Key Words and Phrases: Lightweight Data Compression, Learned, Column Store

ACM Reference Format:
Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 65 (February 2024), 28 pages. https://doi.org/10.1145/3639320

1 INTRODUCTION

Almost all major database vendors today have adopted a column-oriented design for processing analytical queries [30, 40, 49, 57, 70, 71, 73, 91]. One of the key benefits of storing values of the same attribute consecutively is that the system can apply a variety of lightweight compression algorithms to the columns to save space and disk/network bandwidth [27, 28, 101]. These algorithms typically involve a single-pass decompression process (hence, lightweight) to minimize the CPU overhead. A few of them (e.g., Frame-of-Reference or FOR [55, 115]) allow random access to the individual values. This is a much-preferred feature because it allows the DBMS to avoid full-block decompression for highly selective queries, which are increasingly common, especially in hybrid transactional/analytical processing (HTAP) [17, 58, 65, 75, 87, 90] and real-time analytics [13, 73].

Authors' addresses: Yihao Liu, liuyihao21@mails.tsinghua.edu.cn, Institute for Interdisciplinary Information Science, Tsinghua University, Beijing, China; Xinyu Zeng, zeng-xy21@mails.tsinghua.edu.cn, Institute for Interdisciplinary Information Science, Tsinghua University, Beijing, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Institute for Interdisciplinary Information Science, Tsinghua University & Shanghai Qi Zhi Institute, Beijing, China.

ACM 2836-6573/2024/2-ART65
https://doi.org/10.1145/3639320

There are two categories of lightweight compression algorithms that exploit different sources of redundancy in a value sequence. The first are dictionary-based algorithms, including those that encode substring patterns (e.g., FSST [33], HOPE [112]). These algorithms leverage the uneven probability distribution of the values and have a compression ratio limited by Shannon's Entropy [95]. On the other hand, integer compression algorithms such as Run-Length Encoding (RLE) [28], FOR, and Delta Encoding [28, 77] exploit the serial correlation between the values in a sequence: the value of the current position may depend on its preceding values.

However, RLE, FOR, and Delta Encoding are ad-hoc solutions modeling the simplest serial patterns. For example, Delta adopts a model of a basic step function, while RLE only works with consecutive repetitions (elaborated in Section 2). Consequently, we have missed many opportunities to leverage more sophisticated patterns such as the piecewise linearity shown in Figure 1 for better compression in a column store. Prior studies in time-series data storage [46, 47, 60, 68, 84, 105] have proposed to learn the series distribution and minimize the model sizes to achieve a *lossy* compression. These techniques, however, are not applicable to a general analytical system. To the best of our knowledge, none of the existing column stores apply machine learning to improve the efficiency of their lightweight *lossless* compression systematically.

We, thus, propose a framework called LeCo (i.e., Learned Compression) to automatically learn serial patterns from a sequence and use the models for compression. Our key insight is that if we can fit such serial patterns with lightweight machine-learning models, we only need to store the prediction error for each value to achieve a lossless compression. Our framework addresses two subproblems. The first is that given a subsequence of values, how to best fit the data using one model? This is a classic regression problem. However, instead of minimizing the sum of the squared errors, we minimize the maximum error because we store the deltas (i.e., prediction errors) in a fixed-length array to support fast random access during query processing. LeCo also includes a Hyperparameter-Advisor to select the regressor type (e.g., linear vs. higher-order) that would produce the best compression ratios.

The second subproblem is data partitioning: given the type(s) of the regression model, how to partition the sequence to minimize the overall compression ratio? Proactive partitioning is critical to achieving high-prediction accuracy in the regression tasks above because real-world data sets typically have uneven distributions [66, 113]. The partition schemes introduced by *lossy* time-series compression are not efficient to apply. They only target minimizing the total size of the model parameters rather than striking a balance between the model size and the delta array size. Our evaluation (Section 4.8) shows that the state-of-the-art partitioning algorithms [36, 68] are still suboptimal for general *lossless* column compression.

In the *lossless* case, however, having smaller partitions might be beneficial for reducing the local max errors, but it increases the overall model (and metadata) size. Because optimal partitioning is an NP-hard problem, we developed different heuristic-based algorithms for different regression models to obtain approximate solutions in a reasonable amount of time. Another design trade-off is between fixed-length and variable-length partitions. Variable-length partitions produce a higher compression ratio but are slower in random access.

We implemented a prototype of LeCo to show the benefit of using machine learning to compress columnar data losslessly. For each partition, we store a pre-trained regression model along with an array of fixed-length deltas. Decompressing a value only involves a model inference plus a random access to the delta array. LeCo is highly extensible with built-in support for various model types and for both fixed-length and variable-length partition schemes.

We compared LeCo against state-of-the-art lightweight compression algorithms including FOR, Elias-Fano, and Delta Encoding using a microbenchmark consisting of both synthetic and real-world

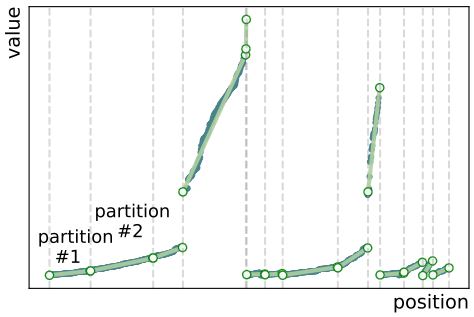


Fig. 1. A motivating example on `movieid` data set.

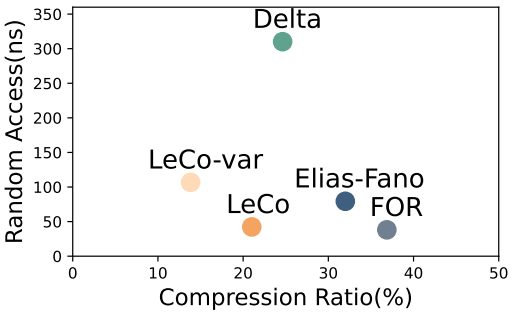


Fig. 2. Performance-space trade-offs.

data sets. As illustrated in Figure 2[1], LeCo achieves a Pareto improvement over these algorithms. Compared to FOR and Elias-Fano, LeCo improves the compression ratio by up to 91% while retaining a comparable decompression and random access performance. Compared to Delta Encoding, LeCo is an order-of-magnitude faster in random access with a competitive or better compression ratio.

We further integrated LeCo into two widely-used applications to study its benefit on end-to-end system performance. We first report LeCo's performance on a columnar execution engine, using Apache Arrow [4] and Parquet [6] as the building blocks. Enabling LeCo in this system speeds up a multi-column filter-groupby-aggregation query by up to 5.2× and accelerates single-column bitmap aggregation query up to 11.8× with a 60.5% reduction in memory footprint. We also use LeCo to compress the index blocks in RocksDB [14, 44] and observed a 16% improvement in RocksDB's throughput compared to its default configuration.

The paper makes three primary contributions. First, we make the case for applying machine learning to lightweight lossless column compression. Second, we propose the Learned Compression (LeCo) framework and implement a prototype that achieves a Pareto improvement on compression ratio and random access speed over existing algorithms. Finally, we integrate LeCo into a columnar execution engine and a key-value store and show that it helps improve the systems' performance and space efficiency simultaneously.

2 THE CASE FOR LEARNED COMPRESSION

The performance of persistent storage devices has improved by orders of magnitude over the last decade [106]. Modern NVMe SSDs can achieve 7GB/s read throughput and over 500,000 IOPS [16]. The speed of processors, on the other hand, remains stagnant as Moore's Law fades [52]. Such a hardware trend is gradually shifting the bottleneck of a data processing system from storage to computation [110]. Hence, pursuing a better compression ratio is no longer the dominating goal when developing a data compression algorithm. Many applications today prefer lightweight compression schemes because decompressing the data is often on the critical path of query execution. Meanwhile, an analytical workload today is often mixed with OLTP-like queries featuring small range scans or even point accesses [12, 86]. To handle such a wide range of selectivity, it is attractive for a data warehouse to adopt compression algorithms that can support fast random access to the original data without decompressing the entire block.

Dictionary encoding is perhaps the most widely used compression scheme in database management systems (DBMSs). Nonetheless, for a sequence where the values are mostly unique, dictionary

[1]Figure 2 is based on the weighted average result of twelve data sets in Section 4.3.

encoding does not bring compression because it assumes independence between the values, and its compression ratio is bounded by Shannon's Entropy [95]. Shannon's Entropy, however, is not the lower bound for compressing an existing sequence[2]. In many real-world columns, values often exhibit strong serial correlations (e.g., sorted or clustered) where the value at a particular position is dependent on the values preceding it. To the best of our knowledge, there is no general solution proposed that can systematically leverage such positional redundancy for compression.

We argue that a learned approach is a natural fit. Extracting serial correlation is essentially a regression task. Once the regression model captures the "common pattern" of the sequence, we can use fewer bits to represent the remaining delta for each value. This Model + Delta framework (a.k.a., LeCo) is fundamental for exploiting serial patterns in a sequence to achieve lossless compression. For example, Boffa et al. attempted to use linear models for storing rank&select dictionaries specifically [32]. In fact, the widely-used FOR, RLE, and Delta Encoding (Delta) can be considered special cases under our framework as well.

FOR divides an integer sequence into frames, and for each value v_i in a frame, it is encoded as $v_i - v_{min}$ where v_{min} is the minimum value of that frame. From a LeCo 's point of view, the regression function for each frame in FOR is a horizontal line. Although such a naive model is fast to train and inference, it is usually suboptimal in terms of compression ratio. RLE can be considered a special case of FOR, where the values in a frame must be identical. Delta Encoding achieves compression by only storing the difference between neighboring values. Similar to FOR, it uses the horizontal-line function as the model, but each partition/frame in Delta only contains one item. The advantage of Delta is that the models can be derived from recovering the previous values rather than stored explicitly. The downside, however, is that accessing any particular value requires a sequential decompression of the entire sequence.

LeCo helps bridge the gap between data compression and data mining. Discovering and extracting patterns are classic data mining tasks. Interestingly, these tasks often benefit from preprocessing the data set with entropy compression tools to reduce "noise" for a more accurate prediction [98]. As discussed above, these data mining algorithms can inversely boost compression efficiency by extracting the serial patterns using the LeCo framework. The theoretical foundation of this relationship is discussed in [48]. The beauty of LeCo is that it aligns the goal of sequence compression with that of serial pattern extraction. LeCo is an extensible framework: it provides a convenient channel to bring related advances in data mining to the improvement of sequence compression.

Although designed to solve different problems, LeCo is related to the recent learned indexes [43, 51, 69] in that they both use machine learning (e.g., regression) to model data distributions. A learned index tries to fit the cumulative distribution function (CDF) of a sequence and uses that to predict the quantile (i.e., position) of an input value. Inversely, LeCo takes the position in the sequence as input and tries to predict the actual value. LeCo's approach is consistent with the mapping direction (i.e., position → value) in classic pattern recognition tasks in data mining.

Moreover, LeCo mainly targets immutable columnar formats such as Arrow [4] and Parquet [6]. Updating the content requires a complete reconstruction of the files on which LeCo can piggyback its model retraining. Unlike indexes where incremental updates are the norm, the retraining overhead introduced by LeCo is amortized because the files in an analytical system typically follow the pattern of "compress once and access many times".

We next present the LeCo framework in detail, followed by an extensive microbenchmark evaluation in Section 4. We then integrate LeCo into two real-world applications and demonstrate their end-to-end performance in Section 5.

[2]The lower bound is known as the Kolmogorov Complexity. It is the length of the shortest program that can produce the original data [78]. Kolmogorov Complexity is incomputable.

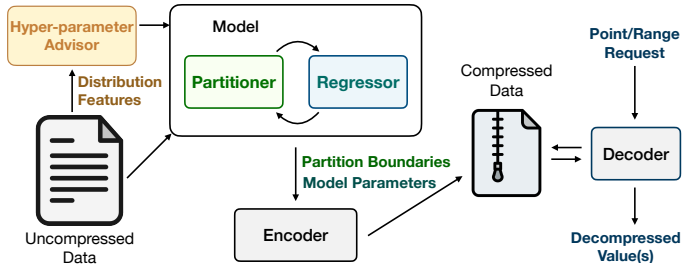


Fig. 3. The LeCo framework – An overview of the modules and their interactions with each other.

3 THE LECO FRAMEWORK

Let us first define the learned compression problem that the LeCo framework targets. Given a data sequence $\vec{v}_{[0,n)} = (v_0, ..., v_{n-1})$, let $P_0 = \vec{v}_{[k_0=0,k_1)}, P_1 = \vec{v}_{[k_1,k_2)}, ..., P_{m-1} = \vec{v}_{[k_{m-1},k_m=n)}$ be a partition assignment $\mathcal{P}$ with m non-overlap segments where each partition j has a model $\mathcal{F}_j$. Let $\delta_i = v_i - \mathcal{F}_j(i)$, where $\mathcal{F}_j(i)$ is the model prediction at position i, for $v_i \in P_j$. The goal of learned compression is to find a partition assignment $\mathcal{P}$ and the associated models $\mathcal{F}$ such that the model size plus the delta-array size are minimized:

$$\sum_{j=0}^{m-1} (\|\mathcal{F}_j\| + (k_{j+1} - k_j)(\max_{i=k_j}^{k_{j+1}-1} \lceil \log_2 \delta_i \rceil))$$

where $\|\mathcal{F}_j\|$ denotes the model size of $\mathcal{F}_j$, and $\max\lceil \log_2 \delta_i \rceil$ is the number of bits required to represent the largest δ_i in the partition.

As shown in Figure 3, LeCo consists of five modules: `Regressor`, `Partitioner`, `Hyperparameter-Advisor`, `Encoder`, and `Decoder`. The Hyper-parameter Advisor trains a Regressor Selector model offline. Given an uncompressed sequence of values at runtime, it extracts features from it for model inference and outputs the recommended Regressor type as well as advises on partitioning strategy. Then, LeCo enters the model learning phase, where the Regressor and the Partitioner work together to produce a set of regression models with associated partition boundaries. The Encoder receives the model parameters as well as the original sequence and then generates a compact representation of the "Model + Delta" (i.e., the compressed sequence) based on a pre-configured format. The compressed sequence is self-explanatory: all the metadata needed for decoding is embedded in the format. When a user issues a query by sending one or a range of positions, the Decoder reads the model of the relevant partition along with the corresponding locations in the delta array to recover the requested values.

A design goal of LeCo is to make the framework extensible. We first decouple model learning (i.e., the logical value encoding) from the physical storage layout because applying common storage-level optimizations such as bit-packing and null-suppression to a delta sequence is orthogonal to the modeling algorithms. We also divide the model learning task into two separate modules. The Regressor focuses on best fitting the data in a single partition, while a Partitioner determines how to split the data set into subsequences to achieve a desirable performance and compression ratio.

Such a modular design facilitates integrating future advances in serial pattern detection and compressed storage format into LeCo. It also allows us to reason the performance-space trade-off for each component independently. We next describe our prototype and the design decisions for each module (Section 3.1 to Section 3.3), followed by the extension to handle string data in Section 3.4.

3.1 Regressor

The Regressor takes in a sequence of values $v_0, v_1, ..., v_{n-1}$ and outputs a single model that "best fits" the sequence. LeCo supports the linear combination of various model types, including constant, linear, polynomial, and more sophisticated models, such as exponential and logarithm. Given a model $\mathcal{F}(i) = \sum_j(\theta_j \cdot \mathcal{M}_j(i))$ where $\mathcal{M}_j$ denotes different model terms with θ_j as its linear combination weight and i represents the position in the sequence, classic regression methods minimize the sum of the squared errors $\sum_i(v_i - \mathcal{F}(i))^2$ (i.e., the l_2 norm of deltas), which has a closed-form solution. If LeCo stores deltas in variable lengths, this solution would produce a delta sequence with minimal size. As we discussed before, real databases usually avoid variable-length values because of the parsing overhead during query execution.

LeCo, therefore, stores each value in the delta array in fixed length. Specifically, LeCo adopts the bit-packing technique. Suppose the maximum absolute value in the delta array is δ_{maxabs}, then each delta occupies a fixed $\phi = \lceil log_2(\delta_{maxabs}) \rceil$ bits. The storage size of the delta array is thus determined by ϕ rather than the expected value of the deltas, and our regression objective becomes:

$$\begin{array}{ll} \text{minimize} & \phi \\ \text{subject to} & \lceil \log_2(|\mathcal{F}(i) - \mathbf{v}_i|) \rceil \leq \phi, i = 0, \ldots, n-1 \\ & \phi \geq 0 \end{array}$$

The constrained optimization problem above can be transformed into a linear programming problem with $2n + 1$ constraints where we can get an approximated optimal solution in $O(n)$ time [94].

We introduce a Regressor Selector (RS) in the Hyperparameter-Advisor to automatically choose the regressor type (e.g., linear vs. higher-order) for a given sequence partition. RS takes in features collected from a single pass of the input data and then feeds them to its classification model (e.g., Classification and Regression Tree or CART). The model is trained offline using the same features from the training data sets. We briefly introduce the main features used in the current RS implementation below.

`Log-scale data range.` Data range gives an upper bound of the size of the delta array. A smaller data range prefers simpler models because the model parameters would take a significant portion of the compressed output.

`Deviation of the` k`th-order deltas.` Given a data sequence $v_0, ..., v_{n-1}$, we define the first-order delta sequence as $d_1^0 = v_1 - v_0, d_1^1 = v_2 - v_1, ..., d_{n-2}^1 = v_{n-1} - v_{n-2}$. Then, the kth-order delta sequence is $\{d_0^k, d_1^k, ..., d_{n-k-1}^k\}$, where $d_{i-1}^k = d_i^{k-1} - d_{i-1}^{k-1}$. Let d_{max}^k, d_{min}^k, and d_{avg}^k be the maximum, minimum, and average delta values, respectively. We then compute the normalized deviation of the kth-order deltas as $\frac{\sum_{i\in[0,n-k)}(d_i^k - d_{avg}^k)}{(n-k)(d_{max}^k - d_{mix}^k)}$. We use this metric to determine the maximum degree of polynomial needed to fit the data. The intuition is that the kth-order delta sequence of a kth-degree polynomial is constant (i.e., with minimum deviation).

`Subrange trend and divergence.` We first split the data into fixed-length subblocks $\{\vec{v}_{[i\cdot s,(i+1)\cdot s)}\}_i$, each containing s records with a data range (i.e., subrange) of r_i. We define the subrange ratio (SR) between adjacent subblocks as $\frac{r_i}{r_{i-1}}$. The metric "subrange trend" $\mathcal{T}$ is the average SR across all subblocks, while "subrange divergence" $\mathcal{D}$ is the difference between the maximum SR and minimum SR. These two metrics provide a rough sketch of the value-sequence distribution: $\mathcal{T}$ depicts how fast the values increase on average, and $\mathcal{D}$ indicates how stable the increasing-trend is.

3.2 Partitioner

Given a Regressor, the Partitioner divides the input sequence $\vec{v}_{[0,n)} = v_0, v_1, ..., v_{n-1}$ into m consecutive subsequences (i.e., partitions) $\vec{v}_{[0,k_1)}, \vec{v}_{[k_1,k_2)}, ..., \vec{v}_{[k_{m-1},k_m)}$ where a regression model is trained on each partition. The goal of the Partitioner is to minimize the total size of the compressed sequences.

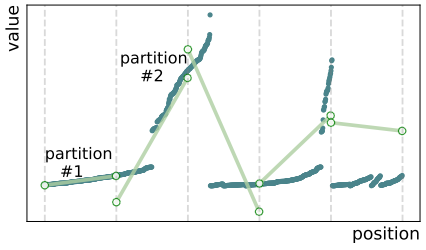


Fig. 4. Fixed-length partitioning example.

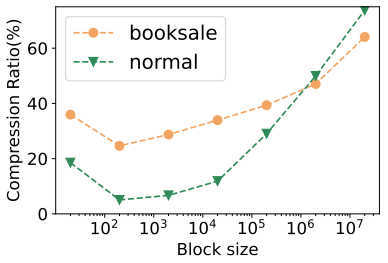


Fig. 5. Compression ratio trend. – Sweeping block size.

Although partitioning increases the number of models to store, it is more likely for the Regressor to produce a smaller delta array when fitting a shorter subsequence. Thus, we require the Partitioner to balance between the model storage overhead and the general model fitting quality. We can find an optimal partition arrangement by computing the compressed size of each possible subsequence through dynamic programming [96]. Such an exhaustive search, however, is forbiddingly expensive with time complexity of $O(n^3)$ and space complexity of $O(n^2)$.

We next propose two practical partitioning schemes developed in LeCo that make different trade-offs between compression ratio and compression/decompression performance.

3.2.1 Fixed-Length Partitioning. The most common strategy is splitting the sequence into fixed-length partitions. This partitioning scheme is easy to implement and is friendly to random accesses. Because each partition contains a fixed number of items, given a position, an application can quickly locate the target partition without the need for a binary search in the metadata. The downside, however, is that fixed-length partitioning is not flexible enough to help the Regressor capture the desired patterns. For example, as shown in Figure 4, if we divide the Movie ID data set into fixed-length partitions, the Regressor would fail to leverage the piecewise linearity in certain ranges. To find an optimal partition size:

(1) Sample $< 1\%$ of the data randomly, consisting of subsequences of length N, where N is the maximum partition length in the search space (e.g., $N = 10k$).
(2) Search the (fixed) partition size between 1 and N that produces the lowest compression ratio on the samples. Because the compression ratio typically has a "U-shape" as we vary the partition size (illustrated in Figure 5), we first perform an exponential search to go past the global minimum. Then, we search back with smaller steps to approach the optimal partition size.
(3) Stop the search process once the compression ratio converges (with $< 0.01\%$ decline between adjacent iterations).

3.2.2 Variable-Length Partitioning. Below, we propose a greedy algorithm for variable-length partitioning for an arbitrary Regressor discussed in Section 3.1 to approximate the optimal solution obtained by the dynamic programming approach.

Our greedy algorithm includes two phases: **split** and **merge**. In the split phase, the algorithm groups consecutive data points into small partitions where the Regressor can predict with small errors. We impose strict constraints to limit the maximum prediction error produced by the Regressor for each partition. Because of our aggressive guarantee of prediction errors, the algorithm tends to generate an excessive number of partitions in the split phase, where the cumulative model size could dominate the final compressed size. To compensate for the over-splitting, the algorithm

enters the merge phase where adjacent partitions are merged if such an action can reduce the final compressed size.

Specifically, in the **split** phase, we first pick a few starting partitions. A starting partition contains at least a minimum number of consecutive values for the Regressor to function meaningfully (e.g., three for a linear Regressor). Then, we examine the adjacent data point to determine whether to include this point into the partition. The intuition is that if the space cost of incorporating this data point is less than a pre-defined threshold, the point is added to the partition; otherwise, a new partition is created.

The splitting threshold is related to the model size S_M of the Regressor. Suppose the current partition spans from position i to $j-1$: $\vec{v}_{[i,j)}$. Let $\Delta(\vec{v})$ be a function that takes in a value sequence and outputs the number of bits required to represent the maximum absolute prediction error from the Regressor (i.e., $\lceil log_2(\delta_{maxabs}) \rceil$). Then, the space cost of adding the next data point v_j is

$$C = (j+1-i) \cdot \Delta(\vec{v}_{[i,j+1)}) - (j-i) \cdot \Delta(\vec{v}_{[i,j)})$$

We compare C against τS_M, where τ is a pre-defined coefficient between 0 and 1 to reflect the "aggressiveness" of the split phase: a smaller τ leads to more fine-grained partitions with more accurate models. If $C \leq \tau S_M$, v_j is included to the current partition $\vec{v}_{[i,j)}$. Otherwise, we create a new partition with v_j as the first value.

In the **merge** phase, we scan through the list of partitions $\vec{v}_{[0,k_1)}, \vec{v}_{[k_1,k_2)}, ..., \vec{v}_{[k_{m-1},k_m)}$ produced in the split phase and merge the adjacent ones if the size of the merged partition is smaller than the total size of the individual ones. Suppose the algorithm proceeds at partition $\vec{v}_{[k_{i-1},k_i)}$. At each step, we try to merge the partition to its right neighbor $\vec{v}_{[k_i,k_{i+1})}$. We run the Regressor on the merged partition $\vec{v}_{[k_{i-1},k_{i+1})}$ and compare its size $S_M + (k_{i+1} - k_{i-1}) \cdot \Delta(\vec{v}_{[k_{i-1},k_{i+1})})$ to the combined size of the original partitions $2S_M + (k_i - k_{i-1}) \cdot \Delta(\vec{v}_{[k_{i-1},k_i)}) + (k_{i+1} - k_i) \cdot \Delta(\vec{v}_{[k_i,k_{i+1})})$. If it results in a size reduction, we would accept this merge. We iterate the partition list multiple times until no qualified merge exists.

We summarize our vari-length partitioning algorithm as follows:

[Init Phase] Scan all data points once. Pick a few "good" initial positions to form starting partitions.
[Split Phase] Scan the starting partition set once.
- Try "growing" each starting partition by adding adjacent points.
- Calculate the inclusion cost and approve the inclusion if it is below the predefined threshold related to the model size. Otherwise, start a new partition with a single point.
- Stops after each point belongs to a partition.

[Merge Phase] Scan the partition sets multiple times.
- Merge a partition to its right neighbor if the combined one achieves a lower compression ratio.
- Stops when no merge can reduce the total space.

We next discuss two aspects that largely determine the efficiency of the above algorithm.

Computing $\Delta(\vec{v}_{[i,j)})$ Efficiently. The computational complexity of $\Delta(\vec{v}_{[i,j)})$ dominates the overall algorithm complexity because the function is invoked at every data point inclusion in the split phase. For a general k-degree *polynomial model* $\sum_{i\in[0,k]} \theta_i \cdot x^i$, we can use the method introduced in [94] to compute $\Delta(\vec{v}_{[i,j)})$ in linear time. To further speed up the process for the *linear Regressor* (which is most commonly used), we propose a much simpler metric $\widetilde{\Delta}(\vec{v}_{[i,j)}) = \log_2(\max_{k=i+1}^{j-1}(d_k) - \min_{k=i+1}^{j-1}(d_k))$, where $d_k = v_k - v_{k-1}$ to approximate the functionality of $\Delta(\vec{v}_{[i,j)})$ The intuition is that the proposed metric $\widetilde{\Delta}(\vec{v}_{[i,j)})$ indicates the difficulty of the linear regression task and has a positive correlation to max bit-width measure $\Delta(\vec{v}_{[i,j)})$.

As discussed in Section 2, Delta Encoding is considered a specific design point under the LeCo framework. The model in each Delta partition is an implicit step function, and only the

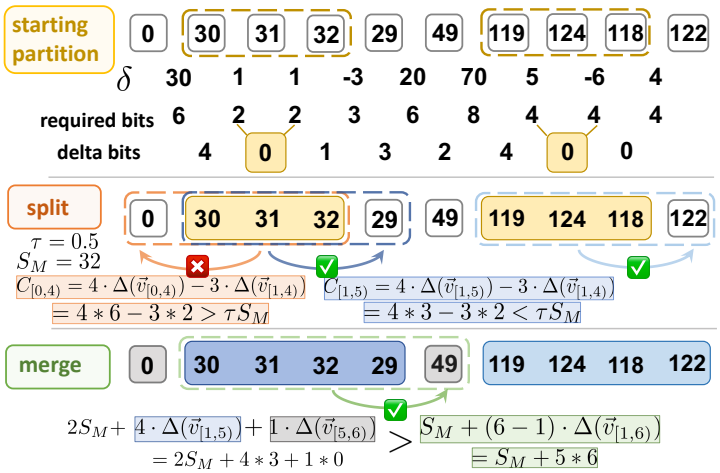


Fig. 6. Variable-length partitioning on Delta encoding – Value 29 is successfully included into segment $\{30, 31, 32\}$ in the split phase because its inclusion cost $C_{[1,5)} = 6$ is less than the pre-defined threshold $\tau \cdot S_M = 16$. In the merge phase, the attempt to merge segment $\{30, 31, 32, 29\}$ and $\{49\}$ succeeds because the space consumption of the segment formed is smaller than the summation of the two original segments.

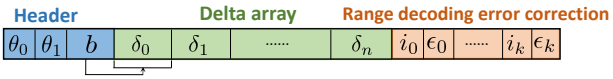


Fig. 7. LeCo's storage format for one partition

first value in the partition is explicitly stored as the model. The prediction errors (i.e., the $\delta's$) of Delta Encoding are the differences between each pair of the adjacent values. Therefore, $\Delta(\vec{v}_{[i,j)}) = \lceil \log_2(\max_{k=i+1}^{j-1} d_k) \rceil$, where $d_k = v_k - v_{k-1}$. After adding the next data point v_j to this partition, we can directly compute $\Delta(\vec{v}_{[0,j+1)}) = \max\{\Delta(\vec{v}_{[0,j)}), d_j\}$.

Selecting Good Starting Positions. Because the algorithms used in both the split and merge phases are greedy, the quality of the algorithms' starting partitions can significantly impact the partition results, especially for the split phase. Suppose we start at a "bumpy" region $\vec{v}_{[i,j)}$ during splitting. Because $\Delta(\vec{v}_{[i,j)})$ of this partition is already large, there is a high probability that it stays the same when including an extra data point in the partition (i.e., $\Delta(\vec{v}_{[i,j+1)}) = \Delta(\vec{v}_{[i,j)})$). Therefore, the space cost of adding this point becomes a constant $C = \Delta(\vec{v}_{[i,j)})$. As long as $C \leq \tau S_M$, this "bad" partition would keep absorbing data points, which is destructive to the overall compression.

For a general *polynomial model* of degree k, we select segments where the $(k+1)$th-order deltas (refer to the definition in Section 3.1) are minimized as the positions to initiate the partitioning algorithm. The intuition is that the discrete $(k+1)$th-order deltas approximate the $(k+1)$th-order derivatives of a continuous function of degree k. If a segment has small $(k+1)$th-order deltas, the underlying function to be learned is less likely to contain terms with a degree much higher than k.

For Delta Encoding, a good starting partition is when the differences between the neighboring values are small (i.e., a small model prediction error) and when the neighboring points form roughly an arithmetic progression (i.e., the partition has the potential to grow larger). We, therefore, compute the bit-width for each delta in the sequence first ("required bits" in Figure 6). We then compute the second-order "delta bits" based on those "required bits" and pick the positions with the minimum

value (the yellow-boxed zeros in Figure 6) as the initial partitions. The required bits are used as the tie-breaker to determine the partition growth precedence.

To summarize, we compared the split-merge partitioning algorithm with the linear Regressor against the optimal partitioning obtained via dynamic programming on real-world data sets introduced in Section 4.1 and found that our greedy algorithm imposes less than 3% overhead on the final compressed size.

3.2.3 Partitioning Strategy Advising. Compared to fixed-length partitions, variable-length partitions could produce a higher compression ratio with a cost of slower random access and compression speed. The choice of the partitioning strategies depends largely on the application's needs. To facilitate estimating the trade-offs, our Hyperparameter-Advisor provides two scores to indicate the potential space benefit of adopting the variable-length strategy.

The two scores are inspired by the definitions of "local hardness" ($\mathcal{H}_l$) and "global hardness" ($\mathcal{H}_g$) of a data set introduced in [104]. $\mathcal{H}_l$ captures the local unevenness in the values distribution, while $\mathcal{H}_g$ depicts the degree of variation of the distribution at a global scale. Intuitively, if the data set is locally hard (i.e., $\mathcal{H}_l$ is high), no Regressor would fit the data well regardless of the partitioning strategy. On the other hand, if the data set is locally easy but globally hard (i.e., $\mathcal{H}_g$ is high), applying variable-length partitioning could improve the compression ratio significantly because it is able to catch the "sharp turns" in the global trend of the value distribution.

Similar to [104], we compute $\mathcal{H}_l$ by running the piece-wise linear approximation (PLA) algorithm with a small error bound (e.g., $\epsilon = 7$) on the data set and count the number of segments generated. The count is then divided by the data set size to normalize the $\mathcal{H}_l$ score. For $\mathcal{H}_g$, we run the same PLA algorithm with a much larger error bound (e.g., $\epsilon = 4096$). Instead of counting the number of segments, we use the the average gap[3] between adjacent segments and the variance of the segment lengths to estimate the "global hardness" of the value distribution. $\mathcal{H}_g$ is the summation of these two numbers, with each normalized.

3.3 Encoder and Decoder

The **Encoder** is responsible for generating the final compressed sequences. The input to the Encoder is a list of value partitions produced by the Partitioner, where each partition is associated with a model. The Encoder computes the delta for each value through model inference and then stores it in the delta array.

The storage format is shown in Figure 7. There is a `header` and a `delta array` for each partition. In the header, we first store the model parameters. For the default linear Regressor, the parameters are two 64-bit floating-point numbers: intercept θ_0 and slope θ_1. Because we bit pack the delta array according to the maximum delta, we must record the bit-length b for an array item in the header.

For fixed-length partitions, the Encoder stores the partition size L in the metadata. If the partitions are variable-length, the Encoder keeps the start index (in the overall sequence) for each partition so that a random access can quickly locate the target partition. We use ALEX [43] (a learned index) to record those start positions to speed up the binary search.

To decompress a value given a position i, the **Decoder** first determines which partition contains the requested value. If the partitions are fixed-length, the value is located in the $\lfloor \frac{i}{L} \rfloor$th partition. Otherwise, the Decoder conducts a "lower-bound" search in the metadata to find the partition with the largest start index $\leq i$.

After identifying the partition, the Decoder reads the model parameters from the partition header and then performs a model inference using $i' = i -$ `start_index` to get a predicted value $\hat{v}$. Then, the Decoder fetches the corresponding $\delta_{i'}$ in the delta array by accessing from the $(b \cdot i')$th bit to

[3]first value of the latter segment - last value of the former segment

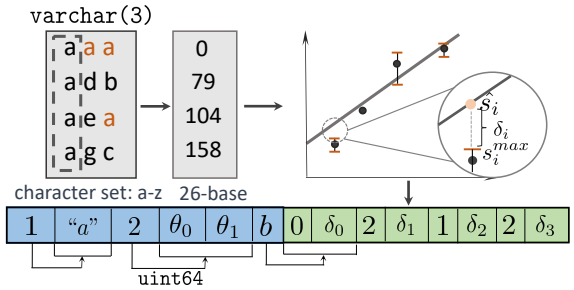


Fig. 8. LeCo string compression – An example including algorithm and storage format modifications.

the $(b \cdot (i'+1)-1)$th bit. Finally, the Decoder returns the decompressed value $\lfloor \hat{v} \rfloor + \delta_{i'}$. Decoding a value involves at most two memory accesses, one for fetching the model (often cached) and the other for fetching the delta.

The basic algorithm for **range decompression** is to invoke the above decoding process for each position in the range. Because of the sequential access pattern, most cache misses are eliminated. For the default linear regression, the Decoder performs two floating-point calculations for model inference (one multiplication and one addition) and an integer addition for delta correction.

We carry out an optimization to increase the range decompression throughput by $10-20\%$. For position i, the model prediction is $\hat{v}_i = \theta_0 + \theta_1 \cdot i$. We can obtain $\hat{v}_i$ by computing $\hat{v}_{i-1} + \theta_1$, thus saving the floating-point multiplication. However, because of the limited precision in the floating-point representation, the θ_1-accumulation result at certain position i is incorrect (i.e., $\lfloor \theta_0 + \sum_1^i \theta_1 \rfloor + \delta_i \neq \lfloor \theta_0 + \theta_1 \cdot i \rfloor + \delta_i$). Therefore, we append an extra list to the delta array to correct the deviation at those positions.

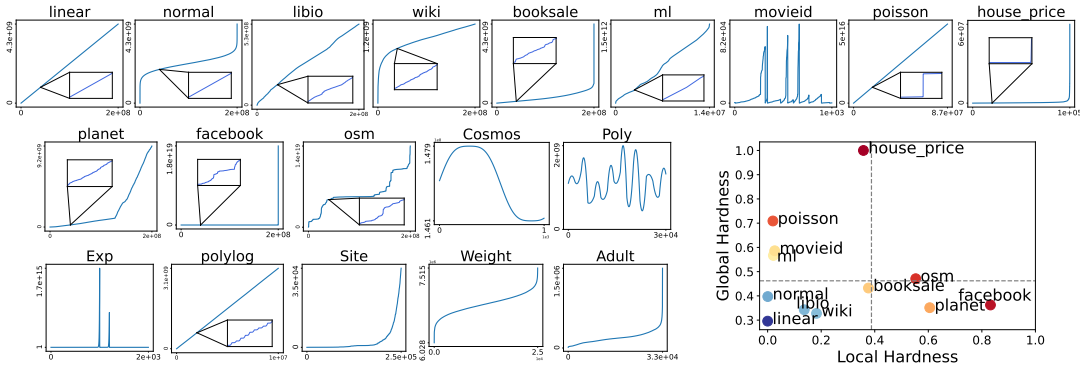
3.4 Extension to Handling Strings

The (integer-based) algorithms discussed so far can already benefit a subset of the string columns in a relational table where the values are dictionary-encoded. In this section, we extend our support to mostly unique string values under the LeCo framework. The idea is to create an order-preserving mapping between the strings and large integers so that they can be fed to the Regressor.

Given a partition of string values, we first extract their common prefix (marked in dashed box in Figure 8) and store it separately in the partition `header`. Then, we shrink the size of the character set if possible. Because many string data sets refer to a portion of the ASCII table, we can use a smaller base to perform the string-integer mapping. For example, we adopt 26-based integers in Figure 8 with only lower-case letters presenting.

LeCo requires strings to be fixed-length. For a column of `varchar(3)`, we pad every string to 3 bytes (padding bytes marked with orange "a" in Figure 8). An interesting observation is that we can leverage the flexibility in choosing the padding characters to minimize the stored deltas. Suppose the string at position i is s_i, and the smallest/largest valid string after padding is s_i^{min}/s_i^{max} (i.e., pad each bit position with the smallest/largest character in the character set). We then choose the padding adaptively based on the predicted value $\hat{s}_i$ from the Regressor to minimize the absolute value of the prediction error. If $\hat{s}_i < s_i^{min}$, we adopt the minimum padding and store $\delta_i = s_i^{min} - \hat{s}_i$ in the delta array; if $\hat{s}_i > s_i^{max}$, we use the maximum padding and produce $\delta_i = s_i^{max} - \hat{s}_i$; if $s_i^{min} \leq \hat{s}_i \leq s_i^{max}$, we choose $\hat{s}_i$ as the padded string directly and obtain $\delta_i = 0$.

The lower part of Figure 8 shows the updated storage format to accommodate varchars. Additionally, the `header` includes the maximum padding length (without prefix) along with the common



(a) Data Distribution Plot. – The first row presents the nine data sets classified as "local easy".

(b) Dataset Hardness.

Fig. 9. Data distribution with hardness evaluation.

prefix of the partition. We also record the length of each varchar value in the delta array (the slot before each delta value) to mark the boundary of the valid bytes from padded bytes in order to decode correctly. These lengths can be omitted for fixed-length strings.

4 MICROBENCHMARK EVALUATION

We evaluate LeCo in two steps. In this section, we compare LeCo against state-of-the-art lightweight compression schemes through a set of microbenchmarks. We analyze LeCo's gains and trade-offs in compression ratio, random access speed, and range decompression throughput. In Section 5, we integrate LeCo into two widely-used applications to show the end-to-end performance.

4.1 Compression Schemes and Data Sets

The baseline compression schemes under evaluation are Elias-Fano [85, 100], Frame-of-Reference (FOR) [55, 115], Delta Encoding (Delta) [28], and rANS [45]. FOR and Delta are introduced in Section 2. rANS is a variant of arithmetic encoding [103] with a decoding speed similar to Huffman [59]. Elias-Fano is an encoding mechanism to compress a sorted list of integers. Suppose the list has n integers and the difference between the maximum and minimum value of the sequence is m. Elias-Fano stores the lower $\lceil log_2(\frac{m}{n}) \rceil$ bits for each value explicitly with bit packing. For the remaining higher bits, Elias-Fano uses unary coding to record the number of appearances for each possible higher-bit value. For example, the binary sequence 00000, 00011, 01101, 10000, 10010, 10011, 11010, 11101 is encoded as "00 11 01 00 10 11 10 01" for the lower bits and "110 0 0 10 1110 0 10 10" for the higher bits. Elias-Fano is quasi-succinct [100] in that it only requires $(2 + \lceil log_2(\frac{m}{n}) \rceil)$ bits per element.

We evaluate LeCo and the baseline solutions extensively on thirteen integer data sets:

- **linear, normal**: synthetic data sets with 200M 32-bit sorted integers following a clean linear (or normal) distribution.
- **poisson**: 87M 64-bit timestamps following a Poisson distribution that models events collected by distributed sensors [111].
- **ml**: 14M 64-bit sorted timestamps from the UCI-ML data set [18].
- **booksale, facebook, wiki, osm**: each with 200M 32-bit or 64-bit sorted integers from the SOSD benchmark [66].

- **movieid**: 20M 32-bit "liked" movie IDs from MovieLens [11].
- **house_price**: 100K 32-bit sorted integers representing house prices in the US [9].
- **planet**: 200M 64-bit sorted planet ID from OpenStreetMap [38].
- **libio**: 200M 64-bit sorted repository ID from libraries.io [82].
- **medicare**: (used in Section 4.5) 1.5 billion augmented 64-bit integers (without order) exported from the public BI benchmark [24].

seven additional non-linear data sets (used in Section 4.4):

- **cosmos**: 100M 32-bit data simulating a cosmic ray signal[4].
- **polylog**: 10M 64-bit synthetic data of a biological population growth curve[5].
- **exp, poly**: 200M 64-bit synthetic data, each block follows the exponential or polynomial distribution of different parameters.
- **site, weight, adult**: 250k, 25k and 30k sorted 32-bit integer column exported from the websites_train_sessions, weights_heights, and adult_train data sets in mlcourse.ai [23].

nine tabular data sets, each sorted by its primary key column:

- **lineitem, partsupp, orders**: TPC-H [26] tables, scale factor = 1.
- **inventory, catalog_sales, date_dim**: TPC-DS [25] tables, scale factor = 1.
- **geo, stock, course_info**: real-world tables extracted from geonames [20], GRXEUR price [21], and Udemy course [22].

and three string data sets:

- **email**: 30K email addresses (host reversed) with an average string length of 15 bytes [2].
- **hex**: 100K sorted hexadecimal strings (up to 8 bytes) [33].
- **word**: 222K English words with an average length of 9 bytes [3].

Figure 9a visualizes the eighteen integer data sets where noticeable unevenness is observed frequently in real-world data sets.

4.2 Experiment Setup

We run the microbenchmark on a machine with Intel®Xeon®(Ice Lake) Platinum 8369B CPU @ 2.70GHz and 32GB DRAM. The three baselines are labeled as `Elias-Fano`, `FOR`, and `Delta-fix`. `Delta-var` represents our improved version of Delta Encoding that uses the variable-length Partitioner in LeCo. `LeCo-fix` and `LeCo-var` are *linear-Regressor* LeCo prototypes that adopt fixed-length and variable-length partitioning, respectively. The corresponding LeCo variants with *polynomial Regressor* are labeled `LeCo-Poly-fix` and `LeCo-Poly-var`. For all the fixed-length partitioning methods, including LeCo-fix, LeCo-Poly-fix, Delta, FOR, and Elias-Fano, the partition size is obtained through a quick sampling-based parameter search described in Section 3.2.1. For Delta-var, LeCo-var, and LeCo-Poly-var, we set the split-parameter τ to be small (in the range $[0, 0.15]$) in favor of the compression ratio over the compression throughput.

Given a data set, an algorithm under test first compresses the whole data set and reports the compression ratio (i.e., *compressed_size / uncompressed_size*) and compression throughput. Then the algorithm performs N uniformly-random accesses (N is the size of the data set) and reports the average latency. Finally, the algorithm decodes the entire data set and measures the decompression throughput. All experiments run on a single thread in the main memory. We repeat each experiment three times and report the average result for each measurement.

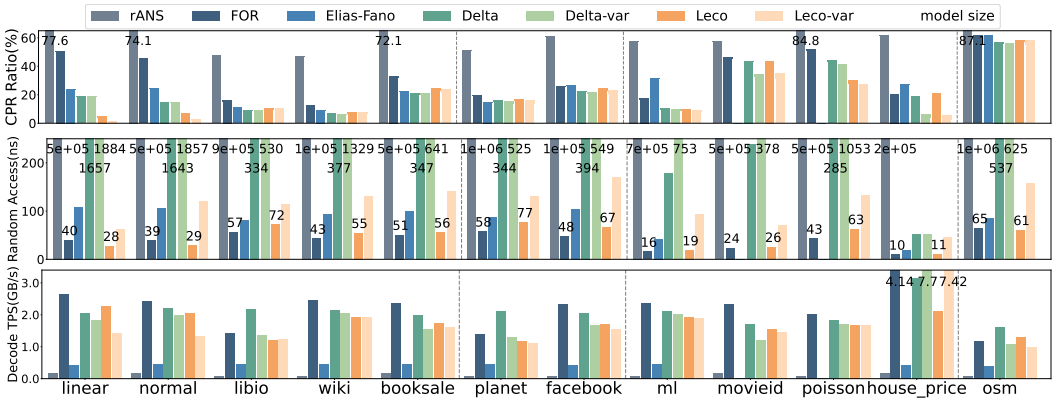


Fig. 10. Compression microbenchmark – Measurement of seven compression schemes on twelve integer data sets from three aspects: Compression Ratio, Random Access Latency, and Full Decompression Throughput. We break down the compression ratio into model size (marked with the cross pattern) and delta size in the first row. The dashed lines split these data sets into four groups in the order of locally easy - globally easy, locally hard - globally easy, locally easy - globally hard, and locally hard - globally hard according to Figure 9b.

4.3 Integer Benchmark

Figure 10 shows the experiment results for compression ratio, random access latency, and decompression throughput on the twelve integer data sets. Elias-Fano does not apply to `poisson` and `movieid` because these two data sets are not fully-sorted.

Overall, LeCo achieves a Pareto improvement over the existing algorithms. Compared to Elias-Fano and FOR, the LeCo variants obtain a significantly better compression ratio while retaining a comparable decompression and random access speed. When compared to Delta Encoding, LeCo remains competitive in the compression ratio while outperforming the Delta variants by an order of magnitude in random access.

4.3.1 ***Compression Ratio.*** As shown in the first row of Figure 10, the compression ratios from the LeCo variants are strictly better than the corresponding ones from FOR. This is because FOR is a special case of LeCo: the output of its Regressor is fixed to a horizontal line (refer to Section 2).

We further plot the local hardness $\mathcal{H}_l$ and the global hardness $\mathcal{H}_g$ (defined in Section 3.2.3) of the different data sets in Figure 9b. The horizontal/vertical dashed line marks the average global/local hardness among the data sets. we observe that LeCo's compression-ratio advantage over FOR is larger on locally-easy data sets (40.9% improvement on average) than the three locally-hard data sets (9.3% improvement on average). This is because local unevenness in the distribution makes it difficult for a regression algorithm to fit well.

LeCo also compresses better than Elias-Fano across (almost) all data sets. Although Elias-Fano is proved to be quasi-succinct, it fails to leverage the embedded serial correlation between the values for further compression. rANS remains the worst, which indicates that the redundancy embedded in an integer sequence often comes more from the serial correlation rather than the entropy.

Compared to Delta Encoding, LeCo shows a remarkable improvement in compression ratio for "smooth" (synthetic) data sets: `linear`, `normal`, and `poisson`. For the remaining (real-world) data

[4]We use $(\sin \frac{x+10}{60\pi} + \frac{1}{10} \sin \frac{3(x+10)}{60\pi}) \times 10^6 + \mathcal{N}(0, 100)$ to construct it.
[5]Constructed by concatenating the polynomial and logarithm distribution, in turn, every 500 records.

FOR	Elias-Fano	Delta-fix	Delta-var	LeCo-fix	LeCo-var
0.81±0.28	0.58±0.17	1.04±0.14	0.04±0.01	0.78±0.11	0.02±0.01

Table 1. Compression throughput (GB/s).

sets, however, LeCo remains competitive. This is because many real-world data sets exhibit local unevenness, as shown in Figure 9a. The degree of such irregularity is often at the same level as the difference between adjacent values.

Another observation is that variable-length partitioning is effective in reducing the compression ratio on real-world data sets that have rapid slope changes or irregular value gaps (e.g., `movieid`, `house_price`). Our variable-length partitioning algorithm proposed in Section 3.2 is able to detect those situations and create partitions accordingly to avoid oversized partitions caused by unfriendly patterns to the Regressor. We also notice that LeCo-var achieves an additional 28.2% compression compared to LeCo-fix on the four locally-easy and globally-hard data sets, while the improvement drops to $< 10\%$ for the remaining data sets[6]. This indicates that the two metrics used for the partitioning strategy advising (refer to Section 3.2.3) is effective in identifying data sets that can potentially benefit from variable-length partitions.

4.3.2 ***Random Access.*** The second row of Figure 10 presents the average latency of decoding a single value in memory for each compression scheme. The random access speed of LeCo-fix is comparable to that of FOR because they both require only two memory accesses per operation. FOR is often considered the lower bound of the random access latency for lightweight compression because it involves minimal computation (i.e., an integer addition). Compared to FOR, LeCo-fix requires an additional floating-point multiplication. This overhead, however, is mostly offset by a better cache hit ratio because LeCo-fix produces a smaller compressed sequence.

LeCo-var is slower because it has to search the metadata to determine the corresponding partition for a given position. This index search takes an extra $35 - 90$ ns depending on the total number of partitions. The Delta variants are an order of magnitude slower than the others in most data sets because they must decompress the entire partition sequentially to perform random access.

4.3.3 ***Full Decompression.*** The third row in Figure 10 shows the throughput of each compression algorithm for decompressing an entire data set. In general, LeCo-fix is $14\% - 34\%$[7] slower than its fastest competitor FOR because LeCo-fix involves an extra floating-point operation upon decoding each record. Delta-var and LeCo-var perform exceptionally well on `house_price`. The reason is that part of the data set contains sequences of repetitive values. LeCo's Partitioner would detect them and put them into the same segment, making the decompression task trivial for these partitions.

4.3.4 ***Compression throughput.*** Table 1 shows the compression throughput for each algorithm weighted averaged across all the twelve data sets with error bars. LeCo-fix has a similar compression speed to the baselines because our linear Regressor has a low computational overhead. Algorithms that adopt variable-length partitioning (i.e., Delta-var and LeCo-var), however, are an order of magnitude slower because the Partitioner needs to perform multiple scans through the data set and invokes the Regressor (or an approximate function) frequently along the way. Such a classic trade-off between compression ratio and throughput is often beneficial to applications that do not allow in-place updates.

[6]Except for the ideal cases in `linear` and `normal`

[7]except for `house_price` where the enhancement of FOR over LeCo-fix is 49%

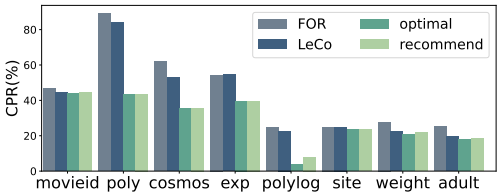


Fig. 11. Regressor selection result.

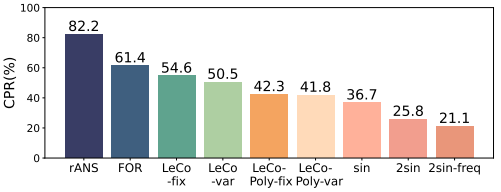


Fig. 12. Compression ratio on cosmos.

4.4 Cases for higher-order models

Although linear models perform sufficiently well in the above integer benchmark[8], there are cases where higher-order models shine. Because our setting is mostly read-only, it is usually worthwhile to spend more computation to compress the data once and then benefit from long-term space and query efficiency.

We first verify the effectiveness of our Regressor Selector in the Hyperparameter Advisor (refer to Section 3.2.3). In this experiment, we consider the following six Regressor types: constant (FOR), linear, polynomial up to a degree of three, exponential, and logarithm. We create synthetic data sets (with random noise) for each Regressor type and extract the features introduced in Section 3.2.3 to train the classification model (i.e., CART) offline.

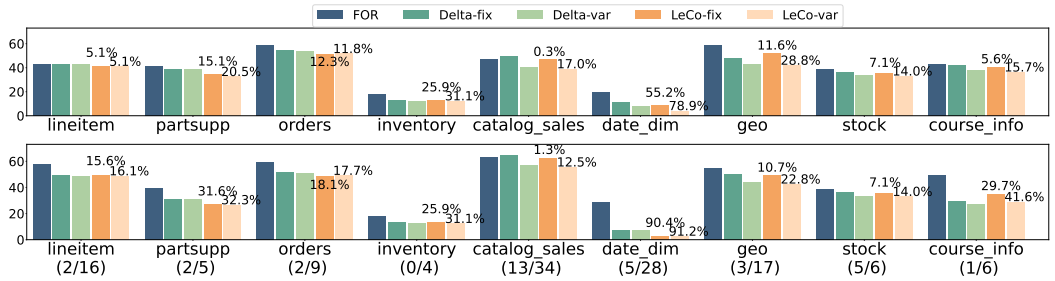
In Figure 11, we compare the compression ratios obtained by using our recommended Regressor per partition (labeled `recommend`) to those obtained by FOR, LeCo-fix, and the `optimal` (i.e., exhaustively search in the candidate Regressor types and pick the one with the best compression ratio). Note that none of the eight tested data sets were used for training. We observe that `recommend` achieves a compression ratio close to the `optimal`, with up to 64.7% improvement over LeCo-fix (with linear regression only) on data sets that exhibit higher-order patterns. For data sets that are mostly linear (e.g. `movieid`), the benefit of applying higher-order models is limited, as expected.

One can even extend the LeCo framework to leverage domain knowledge easily. For example, the `cosmos` data set contains a mixture of two signals (i.e., sine function) with random noise. As shown in Figure 12, if we include a sine term in the Regressor (labeled `sin`), we are able to achieve a better compression ratio (36.7%) compared to the recommended polynomial model (42.3%). If we include two sine terms (labeled `2sin`), we are able to extract an additional 29.7% compression out of the LeCo framework compared to `sin`. If we further know the approximate frequencies of the two sine terms (labeled `2sin-freq`), LeCo produces an even better compression ratio, as presented in Figure 12.

4.5 Compressing Dictionaries

Building dictionaries that preserve the key ordering is a common technique to achieve compression and speed up query processing [31, 83, 112]. Reducing the memory footprint of such dictionaries is an important use case of LeCo. In the following experiment, we perform a hash join with the probe side being dictionary encoded. Specifically, we use the `medicare` dataset as the probe-side column, and we pre-build a hash table of size 84MB in memory, which contains 50% of the unique values (i.e., 50% hash table hit ratio during the join). The probe side first goes through a filter of selectivity of 1% and then probes the hash table for the join. The probe-side values are encoded using an order-preserving dictionary compressed by LeCo (i.e., LeCo-fix), FOR, and Raw (i.e., no

[8]Many data sets in the integer benchmark come from the SOSD benchmark [66], which favors linear models.



name	lineitem	partsupp	orders	inventory	catalog_sales	data_dim	geo	stock	course_info
size	725MB	114MB	164MB	226MB	283MB	10MB	1.5GB	9MB	73MB
sortedness	0.24	0.32	0.51	0.81	0.07	0.78	0.45	0.98	0.19
# total columns	16	5	9	4	34	28	17	6	6
# numeric columns	8	4	3	4	33	16	4	5	6

Fig. 13. Multiple column – Compression ratio of five methods on nine tabular data sets. The second row of the result only considers columns with cardinality ≥ 10%. We report the size in bytes, average sortedness (in the range [0, 1]), total column number, and integer/numerical column number of each table. We mark the enhancement ratio of LeCo variants over FOR above the bars.

compression). We vary the memory budget from 3GB to 500MB and report the throughput (defined as the raw data size of the probe side divided by the query execution time) of executing this query.

Figure 14 shows that applying LeCo improves the throughput up to 95.7× compared to FOR when the memory budget for this query is limited. This is because LeCo compresses the probe-side dictionary from 2.4GB to 5.5MB (cpr ratio = 0.23%) so that it constantly fits in memory. For comparison, the dictionary size compressed using FOR is still 400MB (cpr ratio = 17%). When the available memory is limited, this larger dictionary causes a significant number of buffer pool misses, thus hurting the overall query performance.

4.6 Multi-Column Benchmark

In this section, we evaluate the effectiveness of LeCo on nine multi-column tabular data sets[9]. As shown in Figure 13 (bottom right), we compute the "sortedness" of a table (in the range [0, 1]) by averaging the sortedness of each column using the portion of inverse pairs [35] as the metric.

From Figure 13 (the top row), we observe that LeCo achieves a better compression ratio than FOR in all nine tables. This is because columns in a table are often correlated [54, 61, 92]. Our "sortedness" metric indicates that non-primary-key columns have different degrees of correlation with the primary-key (i.e., sorting) column across tables, thus partially inheriting the serial patterns. Tables with high sortedness such as `inventory` and `data_dim` are more likely to achieve better compression ratios with the LeCo variants.

The bottom left of Figure 13 presents the compression ratios of the TPC-H tables with high-cardinality columns only (i.e., NDV > 10% #row). LeCo's has a more noticeable advantage over FOR on columns that are likely to select FOR as the compression method.

4.7 String Benchmark

We compare LeCo (i.e., LeCo-fix) against the state-of-the-art lightweight string compression algorithm FSST [33] using three string data sets `email`, `hex` and `words`. FSST adopts a dictionary-based

[9]Elias-Fano is not included as a baseline because most columns are not strictly sorted.

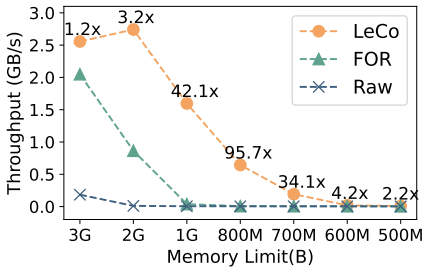


Fig. 14. Hash probe TPS.

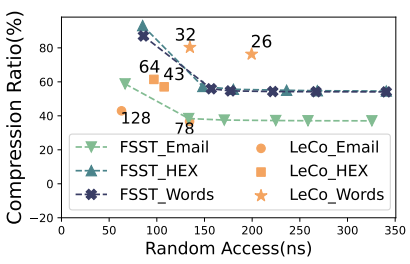


Fig. 15. String evaluation.

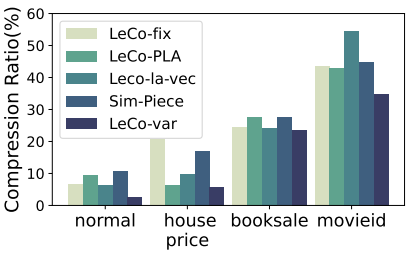


Fig. 16. Partition efficiency.

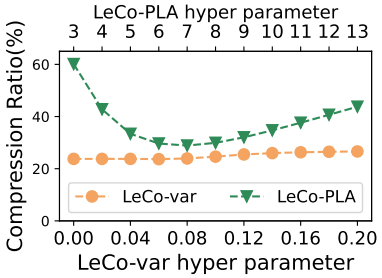


Fig. 17. Robustness test.

approach by building a fine-grained static symbol table to map a partial string to a 1-byte code. Because each compressed string has a variable length, FSST must store a byte-offset array to support random access. An optimization (not mentioned in the FSST paper) is to delta-encode this offset array to trade its random access speed for a better compression ratio. To perform a fair comparison, we tested six different block sizes of the delta encoding: 0 (i.e., no delta compression), 20, 40, 60, 80, and 100. For LeCo, we present two data points with different character-set sizes.

Figure 15 shows the random access latencies and compression ratios for different algorithm configurations. Each LeCo point is marked with the base value used to convert strings. We observed that LeCo's string extension provides a higher random access speed while retaining a competitive compression ratio, compared to FSST on both `email` and `hex` data sets. The compression ratio of LeCo, however, is slightly worse than that of FSST on `word`. This is because dictionary-based algorithms are more suitable for human-readable strings that contain repeating patterns such as common prefixes, roots, and suffixes, while learned compression is better at leveraging serial patterns between values.

4.8 Partitioner Efficiency

In this section, we compare LeCo's default Partitioner (as described in Section 3.2) to state-of-the-art partitioning algorithms, including the PLA algorithm adopted by time-series compression [84], as well as FITing tree [53], Sim-Piece introduced in [68], and the la_vector algorithm proposed in [32]. The angle-based PLA predefines a fixed global prediction error bound (ϵ) and determines the partition boundaries greedily in one pass. Sim-Piece adopts the angle-based PLA as its partitioner and compactly stores linear models with the same intercept together to reduce the overall space. They sacrifice model parameter precisions to create more segments with the same intercept. On the other hand, la_vector translates each data point v_i into a vertex i, where the weight of edge (i, j)

is defined as the compression ratio of segment $[v_i, v_j]$. The optimal partitioning problem is thus converted into finding the shortest path in the above graph $\mathcal{G}$. la_vector approximates $\mathcal{G}$ with $\mathcal{G}'$ with fewer edges and proofs that the best compression ratio achieved on $\mathcal{G}'$ is at most $k \cdot l$ larger than that on $\mathcal{G}$ where k is a constant and l is the shortest path length.

We integrated PLA, Sim-Piece, and la_vector into the LeCo framework (with the linear Regressor denoted by LeCo-PLA, Sim-Piece and LeCo-la-vec, respectively) and repeated the experiments in Section 4.3 on four representative data sets. As shown in Figure 16, all three candidate methods exhibit significantly worse compression ratios compared to LeCo-var. The globally-fixed error bound in LeCo-PLA fails to adapt to data segments with rapidly changing slopes. We also found that LeCo-PLA is more sensitive to its hyperparameter compared to LeCo-var, as shown in Figure 17 where we sweep the hyperparameters for LeCo-PLA (ϵ) and LeCo-var (τ) on the `books` data set. The model compaction in Sim-Piece doesn't take effect because, on mostly sorted data sets, the intercept of each linear model is also increasing. The precision sacrifice in their implementation results in an even worse compression ratio on `house_price` compared to LeCo-PLA. For LeCo-la-vec, although it finds the shortest path in the approximate "compression-ratio graph", it overlooked the length of the shortest path, resulting in an excessive number of models that dominate the compressed size on data sets such as `movieid`.

5 SYSTEM EVALUATION

To show how LeCo can benefit real-world systems, we integrated LeCo into two widely-used applications: (1) a columnar execution engine implemented using Arrow [4] and Parquet [6] and (2) RocksDB [14]. All experiments are conducted on a machine with 4× Intel®Xeon® (Cascade Lake) Platinum 8269CY CPU @ 2.50GHz, 32GB DRAM, and a local NVMe SSD of 447GB with 250k maximum read IOPS. We use Apache Arrow 8.0.0, Parquet version 2.6.0, and RocksDB Release version 6.26.1 in the following experiments.

5.1 Integration to Arrow and Parquet

We first integrated LeCo (as well as FOR and Delta for comparison) into Apache Arrow (the most widely-used columnar *in-memory* format) and Apache Parquet (the most widely-used columnar *storage* format), and built an execution engine prototype using their C++ libraries to demonstrate how LeCo can benefit query processing.

Parquet uses dictionary encoding as the default compression method. It falls back to plain encoding if the dictionary grows too large. We refer to this mechanism as `Default`. In the following experiments, we set Parquet's row group size to 10M rows and disable block compression unless specified otherwise.

The primary component of the Arrow format is the Arrow Array that represents a sequence of values of the same type. Except for basic dictionary encoding, no compression is applied to Arrow arrays to guarantee maximum query-processing performance. We re-implemented the Arrow Array structure using lightweight compression methods (i.e., LeCo, FOR, and Delta) without changing its interface. We use a consistent compressed format for the Arrow Array and Parquet Column Chunk so that no additional decoding is required when scanning the data from disk to memory.

The Arrow Compute library implements various basic database operators (e.g., Take, Filter, GroupBy) on Arrow arrays as compute functions. Our execution engine uses these compute functions as building blocks. The engine is implemented using late materialization [34] where intermediate results are passed between operators as position bitmaps. We also push down the filters to the storage layer (i.e., Parquet).

5.1.1 Filter-Groupby-Aggregation. We create a query template of a typical filter-groupby-aggregation as follows. Suppose we have 10k sensors recording measurements. The table `T` has

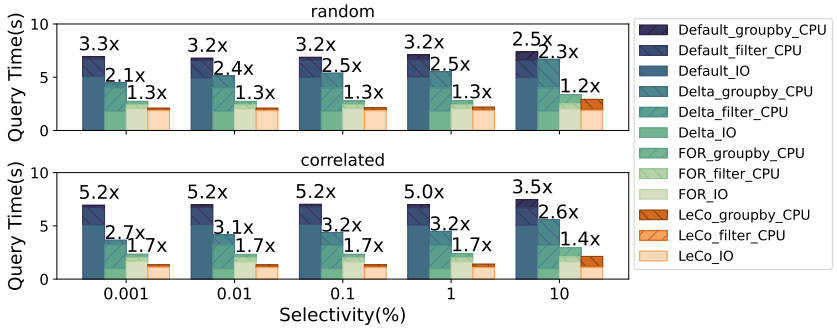


Fig. 18. Filter groupby aggregation

three columns: (1) `ts`, timestamps (in seconds, almost sorted) extracted from the `ml` [18] data set, (2) `id`, 16-bit sensor IDs ranging from 1 to 10k, and (3) `val`, 64-bit-integer sensor readings. To vary the compressibility of the table, we generate two different data distributions for the `id` and the `val` columns: (1) `random`: both `id` and `val` are randomly generated and are difficult to compress no matter which algorithm, and (2) `correlated`: `id`s are clustered in groups of 100, and `val`s are monotonically increasing across groups (but random within a group). There are serial patterns in this setting for lightweight compression algorithms to leverage.

We construct the following query that outputs the average reading for each sensor within a given time range per day: `SELECT AVG(val) FROM T WHERE ts_begin < ts % val_2 < ts_end GROUP BY id`. We adjust the time range (i.e., `ts_end - ts_begin`) to control the query's selectivity. When executing this query, our execution engine first pushes down the filter predicate to Parquet, which outputs a bitmap representing the filtering results. The engine then scans the `id` and the `val` column from Parquet into Arrow arrays and performs the groupby-aggregation. Both groupby and aggregation only decode entries that are still valid according to the filter-bitmap, which involves random accesses to the corresponding Arrow arrays.

We generated four Parquet files with Default, Delta, FOR, and LeCo as the encoding algorithms (with a partition size of 10k entries). In the case of `random` distribution, the resulting file sizes are 3.8GB, 1.3GB, 1.5GB, and 1.4GB, respectively. The `correlated` distribution setting have better compression ratios. The corresponding file sizes are 3.8GB, 706MB, 1.2GB, and 785MB. We execute the above query template and repeat each query instance three times with its average execution time reported.

As shown in Figure 18, all three lightweight compression algorithms outperform the Default because of the significant I/O savings proportional to the file size reduction. Compared to Delta, LeCo is much more CPU-efficient because Delta requires to decode the entire partition to random-access particular entries during the groupby-aggregation. Compared to FOR, LeCo mainly gains its advantage through the I/O reduction due to a better compression ratio. This I/O advantage becomes larger with a more compressible data set (i.e., `correlated`).

Interestingly, LeCo is up to 10.5× faster than FOR when performing the filter operation. Suppose that the model of a partition is $\theta_0 + \theta_1 \cdot i$, and the bit-length of the delta array is b. For a less-than predicate $v < \alpha$, for example, once LeCo decodes the partition up to position k, where $\theta_0 + \theta_1 \cdot k - 2^{b-1} > \alpha$ (assume $\theta_1 \geq 0$), we can safely skip the values in the rest of the sequence because they are guaranteed to be out of range. FOR cannot perform such a computation pruning because the `ts` column is not *strictly* sorted.

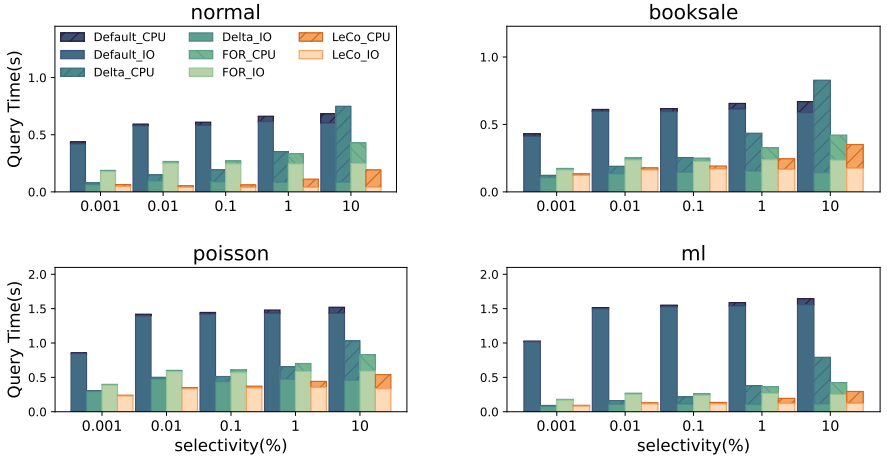


Fig. 19. Bitmap aggregation.

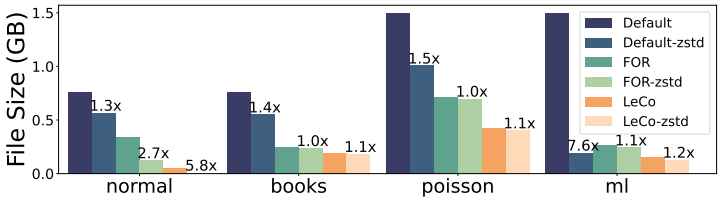


Fig. 20. Parquet with zstd compression – Additional improvement introduced by zstd is marked on bars.

5.1.2 Bitmap Aggregation. In this experiment, we zoom in on the critical bitmap aggregation operation of the above end-to-end query and further verify LeCo's performance and space benefits on four different data sets introduced in Section 4.1: `normal`, `poisson`, `booksale`, and `ml`[10]. For each data set, we create four Parquet files with different lightweight compression algorithms (i.e., Default, Delta, FOR, and LeCo) enabled as above. The bitmaps used in the experiments include ten set-bit clusters following a Zipf-like distribution with a varying ratio of "ones" (to represent different filter selectivities). Data is scanned directly into Arrow arrays in a row-group granularity, where a row-group is skipped if the bits in the corresponding area in the bitmap are all zeros. We then feed the arrays and the bitmap to the Arrow Compute function to perform the summation.

As shown in Figure 19, LeCo consistently outperforms Default (by up to 11.8×), Delta (by up to 3.9×), and FOR (by up to 5.0×). LeCo's speedup comes from both the I/O reduction (due to a better compression ratio) and the CPU saving (due to fast random access and better caching). Moreover, we found that LeCo consumes less memory during the execution. LeCo's peak memory usage (for processing a Parquet row group) is 60.5%, 35.3%, and 10.0% less compared to Default, FOR, and Delta, respectively on average. This is preferred for systems with constrained memory budgets.

5.1.3 Enabling Block Compression. People often enable block compression on columnar storage formats such as Parquet and ORC [5] to further reduce the storage overhead. We repeat the Parquet

[10]we scale `ml` to 200M rows while preserving its value distribution.

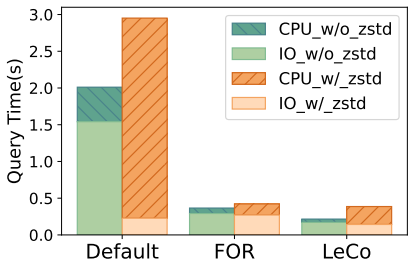


Fig. 21. Time breakdown of zstd on Parquet.

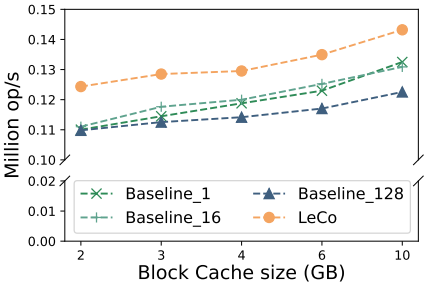


Fig. 22. RocksDB seek query throughput.

loading phase of the above experiments with zstd [19] enabled to show how block compression algorithms affect the final file sizes.

As shown in Figure 20, the additional improvement introduced by zstd is marked above each bar. Applying zstd on top of the lightweight encoding schemes in Parquet can further reduce the file sizes. The relative improvement of LeCo + zstd over LeCo is higher than that in the case of FOR. This shows that LeCo's ability to remove serial redundancy is complementary to some degree to the general-purpose block compression algorithms.

The decompression overhead of zstd, however, can be significant. We perform the bitmap selection experiment with zstd turned on for Parquet. Figure 21 shows an example result (`ml` data set, selectivity = 0.01). We observe that the I/O savings from zstd are outweighed by its CPU overhead, leading to an increase in the overall query time. The result confirms our motivation in Section 2 that heavyweight compression algorithms are likely to cause CPU bottlenecks in modern data processing systems.

5.2 RocksDB Index Block Compression

RocksDB is a key-value store based on log-structured merge trees. Each level consists of a sorted run of key-value pairs stored in a sequence of SSTables. RocksDB divides each SSTable into multiple data blocks (4KB by default) and builds an index block upon the data blocks. For each pair of adjacent data blocks B_{i-1} and B_i, an index entry is created where the key is the shortest string greater than the last key in B_{i-1} and smaller than the first key in B_i. The value of the index entry is a "block handle" that records the byte offset and the size of B_i. To locate a particular key k, RocksDB performs a "lower-bound" binary search in the index block and obtains the entry with the smallest key $\geq k$. It then reads the associated "block handle" and fetches the corresponding data block that (potentially) contains k.

RocksDB offers a native compression scheme for the index blocks. It includes a hyper-parameter called "restart interval" (RI) to make trade-offs between the lookup performance and the index size. The value of RI determines the size of a compression unit in an index block. Within each compression unit, RocksDB applies a variation of Delta Encoding to both the keys and values. For the index keys, suppose k_{i-1} proceeds k_i in the compressed sequence. Then k_i is encoded as (m_i, k_i') where m_i denotes the length of the shared prefix between k_{i-1} and k_i, and k_i' is the remaining suffix. For the "block handles", RocksDB stores the offset of each block in a delta-encoded sequence.

We use LeCo to compress the keys and values separately in a RocksDB index block to shrink its size and to improve the lookup performance at the same time. We adopt LeCo-fix for both key and value sequences. Because all internal keys in RocksDB are strings, we use LeCo with the string extension to compress the keys.

We compare RocksDB with LeCo against[11] three baseline configurations: `Baseline_1`, `Baseline_16`, and `Baseline_128`. The number at the end of each label denotes the value of the RI parameter (1 is RocksDB's default). We configured RocksDB according to the settings in its Performance Benchmark [15][12]. We turned on direct I/O to bypass the large OS page cache.

In each experiment, we first load the RocksDB with 900 million record generated from the above RocksDB Performance Benchmark. Each record has a 20-byte key and a 400-byte value. The resulting RocksDB is around 110 GB. LeCo, `Baseline_1`, `Baseline_16`, and `Baseline_128` achieve a compression ratio of 28.1%, 71.3%, 18.9% and 15.9%, respectively on the index blocks in RocksDB. We then perform 200M non-empty `Seek` queries using 64 threads. The query keys are generated using YCSB [39] with a skewed configuration where 80% of the queries access 20% of the total keys. We repeat each experiment three times and report the average measurement.

Figure 22 shows the system throughputs for LeCo, and the baselines with a varying block cache size. RocksDB with LeCo consistently outperforms the three baseline configurations by up to 16% compared to the second-best configuration. The reasons are two-fold. First, compared to `Baseline_1` where no compression for the index blocks are carried out (each compression unit only contains one entry), LeCo produces smaller index blocks so that more data blocks can fit in the block cache to save I/Os. Such a performance improvement is more recognizable with a smaller block cache.

Second, compared to `Baseline_16` and `Baseline_128` where the index blocks are compressed using Delta Encoding. Although LeCo no longer exhibits an index-size advantage over these baselines, it saves a significant amount of computations. Compared to `Baseline_128` which need to decompress the entire 128-entry unit before it accesses a single entry, LeCo only requires two memory probes to perform a random access in the index block.

To sum up, applying LeCo speeds up binary search in the index blocks. Such a small change improved the performance of a complex system (RocksDB) noticeably. We believe that other systems with similar "zone-map" structures can benefit from LeCo as well.

6 RELATED WORK

Many prior compression algorithms leverage repetitions in a data sequence. Null suppression omits the leading zeros in the bit representation of an integer and records the byte length of each value [1, 28, 89, 93, 97]. Dictionary [29, 31, 33, 80, 83, 91, 112] and entropy-based compression algorithms [59, 103] build a bijective map between the original values and the code words. Block compression algorithms such as LZ77 [114], Gzip [7], Snappy [8], LZ4 [10], and zstd [19] achieve compression by replacing repeated bit patterns with shorter dictionary codes. These approaches, however, miss the opportunity to exploit the serial correlation between values to achieve a compressed size beyond Shannon's Entropy.

A pioneer work by Boffa et al. [32] proposed to use a similar linear model as in the PGM-Index [51] with a customized partitioning algorithm (i.e., la_vector) to compress a specific data structure called the rank&select dictionaries. Their approach represents a specific design point in the LeCo framework that is much more general and extensible in model types and partitioning algorithms. Also, LeCo's default variable-length partitioning algorithm is shown to be more efficient than la_vector for compressing columnar data.

Semantic compression [54, 61, 62] aims to compress tabular data by exploiting correlations between columns using complex models like Bayesian networks. LFR[108] and DFR[107] use linear

[11]The fixed partition size are set to 64 entries for LeCo.

[12]block_size = 4096B; pin_l0_filter_and_index_blocks_in _cache is enabled.

model or Delta-like model to compress data without partitioning. Because their model parameters vary at each data point, they do not support quick random access.

Data partitioning plays an essential role in achieving a good compression ratio for various algorithms. Several prior work [85, 88] targeting inverted indexes proposed partitioning algorithms for specific compression schemes like Elias-Fano [100] and VByte [99, 102]. The partitioning algorithms introduced in Section 3.2 are applicable to an arbitrary linear combination of regression models. In terms of storage format, FastPFOR [115] and NewPFD [109] stores outlier values separately in a different format to improve the overall storage and query efficiency.

Time-series/IoT data compression field adopts a similar idea with LeCo of approximating data distribution with models, but they target keeping the prediction error within a predetermined threshold and achieve **lossy** compression. Their optimization goal is to minimize the total space of model parameters. Partitioning algorithms for linear models [47, 84, 105] and constant value models [74] are designed to minimize the segment number. Sim-Piece[68] introduces a more compact format to keep the output models. Eichinger et al. [46] consider utilizing higher order models but require additional computation effort in the approximation process.

Codec selection is critical in improving data compression performances. A common practice is to define a feature set and use machine learning classifiers for selection. Abadi et al. [28] empirically analyzed the performance of different codecs and manually built a decision tree for selection. While the features introduced by CodecDB [64] overlook the chance to utilize distribution patterns, in contrast to our Regressor Selector.

Both learned indexes and learned compression use regression to model data distributions. RMI [69] and RS [67] apply hierarchical machine learning models to fit the CDFs, while PGM-Index [51], FITing-Tree [53], and CARMI [113] put more effort into the partitioning strategies to reduce model prediction errors. ALEX [43] and Finedex [79] proposed techniques such as a gapped array and non-blocking retraining to improve the indexes' update efficiency.

Previous work [27, 110, 116] have shown that heavyweight compression algorithms [7, 8, 59] designed for disk-oriented systems could incur notable computational overhead to the overall system performance. Algorithms such as FSST [33] and PIDS [63], therefore, emphasize low CPU usage besides a competitive compression ratio. Other related work reduces the computational overhead by enabling direct query execution on compressed formats [28, 41, 64], including filter and aggregation/join pushdowns [37, 42, 50, 56, 72, 76, 81].

7 CONCLUSION

This paper introduces LeCo, a lightweight compression framework that uses machine learning techniques to exploit serial correlation between the values in a column. We provide a complementary perspective besides Shannon's entropy to the general data compression problem. The LeCo framework bridges data mining and data compression with a highly modular design. Both our micro-benchmark and system evaluation show that LeCo is able to achieve better storage efficiency and faster query processing simultaneously.

REFERENCES

[1] 2009. Google Varint. https://static.googleusercontent.com/media/research.google.com/en//people/jeff/WSDM09-keynote.pdf.
[2] 2018. 300 Million Email Database. https://archive.org/details/300MillionEmailDatabase.
[3] 2020. English Word Dataset in HOPE. https://github.com/efficient/HOPE/blob/master/datasets/words.txt.
[4] 2022. Apache Arrow. https://arrow.apache.org/.
[5] 2022. Apache ORC. https://orc.apache.org/.
[6] 2022. Apache Parquet. https://parquet.apache.org/.
[7] 2022. GNU GZip. https://www.gnu.org/software/gzip/.

[8] 2022. Google snappy. http://google.github.io/snappy/.
[9] 2022. Kaggle USA Real Estate Dataset. https://www.kaggle.com/datasets/ahmedshahriarsakib/usa-real-estate-dataset?select=realtor-dataset-100k.csv.
[10] 2022. Lz4. https://github.com/lz4/lz4.
[11] 2022. Movie ID dataset. https://www.kaggle.com/datasets/grouplens/movielens-20m-dataset?select=rating.csv.
[12] 2022. Personal communication, anonymized for review. .
[13] 2022. Real-time Analytics for MySQL Database Service. https://www.oracle.com/mysql/.
[14] 2022. Rocksdb Github. https://github.com/facebook/rocksdb.
[15] 2022. Rocksdb Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks.
[16] 2022. Samsung 980 PRO 4.0 NVMe SSD. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/.
[17] 2022. SingleStore. https://www.singlestore.com/.
[18] 2022. UCI Machine Learning Repository: Timestamp in Bar Crawl: Detecting Heavy Drinking Data Set. https://archive.ics.uci.edu/ml/datasets/Bar+Crawl%3A+Detecting+Heavy+Drinking.
[19] 2022. Zstandard. https://github.com/facebook/zstd.
[20] 2023. GeoNames Data. https://www.geonames.org/export/.
[21] 2023. HistData GRXEUR. https://www.histdata.com/.
[22] 2023. Kaggle Udemy Courses. https://www.kaggle.com/datasets/hossaingh/udemy-courses.
[23] 2023. mlcourse.ai. https://github.com/Yorko/mlcourse.ai/tree/main/data.
[24] 2023. Public BI Benchmark. https://homepages.cwi.nl/~boncz/PublicBIbenchmark/.
[25] 2023. TPC-DS Benchmark Standard Specification. https://www.tpc.org/tpcds/.
[26] 2023. TPC-H Benchmark Standard Specification. https://www.tpc.org/tpch/.
[27] Daniel Abadi, Peter Boncz, Stavros Harizopoulos Amiato, Stratos Idreos, and Samuel Madden. 2013. *The design and implementation of modern column-oriented database systems.* Now Hanover, Mass.
[28] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 671–682.
[29] Gennady Antoshenkov, David Lomet, and James Murray. 1996. Order preserving string compression. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*. IEEE, 655–663.
[30] Nikos Armenatzoglou, Sanuj Basu, Bhanoori Naga, et al. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*. 2205–2217.
[31] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 283–296.
[32] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2021. A "Learned" Approach to Quicken and Compress Rank/Select Dictionaries. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 46–59.
[33] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.
[34] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research (CIDR)*. 225–237.
[35] C. G. Borroni. 2013. A new rank correlation measure. *Statistical Papers* 54, 2 (2013), 255–270.
[36] Scott H Cameron. 1966. *Piece-wise linear approximations.* Technical Report. IIT RESEARCH INST CHICAGO IL COMPUTER SCIENCES DIV.
[37] Lemke Christian, Sattler Kai-Uwe, Faerber Franz, and Zeier Alexander. 2010. Speeding up queries in column stores: a case for compression. *12th International Conference on Data Warehousing and Knowledge Discovery (DaWaK)* (2010), 117–129.
[38] Google Cloud. 2017. OpenStreetMap(2017). https://console.cloud.google.com/marketplace/details/openstreetmap/geo-openstreetmap.
[39] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*. 143–154.
[40] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.
[41] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. Morphstore: Analytical query engine with a holistic compression-enabled processing model. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2396–2410.
[42] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. 2015. Query optimization in Oracle 12c database in-memory. *Proceedings*

of the VLDB Endowment 8, 12 (2015), 1770–1781.

[43] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

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

[45] Jarek Duda. 2013. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540* (2013).

[46] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. 2015. A time-series compression technique and its application to the smart grid. *The VLDB Journal* 24 (2015), 193–218.

[47] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. 2009. Online Piece-Wise Linear Approximation of Numerical Streams with Precision Guarantees. 2, 1 (2009).

[48] Christos Faloutsos and Vasileios Megalooikonomou. 2007. On data mining, compression, and kolmogorov complexity. *Data mining and knowledge discovery* 15, 1 (2007), 3–20.

[49] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.

[50] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 31–46.

[51] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.

[52] Kenneth Flamm. 2019. Measuring Moore's law: evidence from price, cost, and quality indexes. In *Measuring and Accounting for Innovation in the 21st Century*. University of Chicago Press.

[53] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1189–1206.

[54] Yihan Gao and Aditya Parameswaran. 2016. Squish: Near-optimal compression for archival of relational datasets. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1575–1584.

[55] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering (ICDE)*. IEEE, 370–379.

[56] Goetz Graefe and Leonard D Shapiro. 1990. *Data compression and database performance*. University of Colorado, Boulder, Department of Computer Science.

[57] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[58] Dongxu Huang, Qi Liu, Qiu Cui, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[59] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[60] Nguyen Quoc Viet Hung, Hoyoung Jeung, and Karl Aberer. 2012. An evaluation of model-based approaches to sensor data compression. *IEEE Transactions on Knowledge and Data Engineering* 25, 11 (2012), 2434–2447.

[61] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. 2020. DeepSqueeze: deep semantic compression for tabular data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1733–1746.

[62] H. V. Jagadish, J. Madar, and Raymond T. Ng. 1999. Semantic Compression and Pattern Extraction with Fascicles. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*. 186–198.

[63] Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, and Aaron J Elmore. 2020. PIDS: attribute decomposition for improved compression and query performance in columnar storage. *Proceedings of the VLDB Endowment* 13, 6 (2020), 925–938.

[64] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A Chien, Jihong Ma, and Aaron J Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 843–856.

[65] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*. IEEE, 195–206.

[66] A. Kipf, R Marcus, A Van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. 2019. SOSD: A Benchmark for Learned Indexes. In *33rd Conference on Neural Information Processing Systems (NeurIPS)*.

[67] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–5.

[68] Xenophon Kitsios, Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2023. Sim-piece: highly accurate piecewise linear approximation through similar segment merging. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1910–1922.

[69] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. 489–504.

[70] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *Proceedings of the 31st International Conference on Data Engineering (ICDE)*. IEEE, 1253–1258.

[71] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (2012).

[72] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 311–326.

[73] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1740–1751.

[74] Iosif Lazaridis and Sharad Mehrotra. 2003. Capturing sensor-generated time series with quality guarantees. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*. IEEE, 429–440.

[75] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. 2017. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1598–1609.

[76] Jae-Gil Lee, Gopi Attaluri, Ronald Barber, et al. 2014. Joins on encoded and partitioned data. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1355–1366.

[77] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

[78] Ming Li, Paul Vitányi, et al. 2008. *An introduction to Kolmogorov complexity and its applications*. Vol. 3. Springer.

[79] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.

[80] Yinan Li, Craig Chasseur, and Jignesh M Patel. 2015. A padded encoding scheme to accelerate scans by leveraging skew. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1509–1524.

[81] Yinan Li and Jignesh M Patel. 2013. Bitweaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 289–300.

[82] Libraries.io. 2017. Repository ID in Libraries.io. https://libraries.io/data.

[83] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J Elmore. 2019. Mostly order preserving dictionaries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1214–1225.

[84] Ge Luo, Ke Yi, Siu-Wing Cheng, Zhenguo Li, Wei Fan, Cheng He, and Yadong Mu. 2015. Piecewise linear approximation of streaming time series data with max-error guarantees. In *Proceedings of the 31st International Conference on Data Engineering (ICDE)*. IEEE, 173–184.

[85] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned elias-fano indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*. 273–282.

[86] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 1771–1775.

[87] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner (2014, January 28) Available at https://www. gartner. com/doc/2657815/hybrid-transactionanalyticalprocessing-foster-opportunities* (2014), 4–20.

[88] Giulio Ermanno Pibiri and Rossano Venturini. 2019. On optimally partitioning variable-byte codes. *IEEE Transactions on Knowledge and Data Engineering* 32, 9 (2019), 1812–1823.

[89] J. Plaisance, N. Kurz, and D Lemire. 2016. Vectorized VByte Decoding. *Computerence* (2016).

[90] Hasso Plattner. 2009. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 1–2.

[91] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, 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.

[92] Vijayshankar Raman and Garret Swart. 2006. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 858–869.

[93] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2010. Fast integer compression using SIMD instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. 34–40.

[94] Raimund Seidel. 1991. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry* 6 (1991), 423–434.
[95] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.
[96] Fabrizio Silvestri and Rossano Venturini. 2010. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th ACM international conference on Information and knowledge management*. 1219–1228.
[97] Alexander A Stepanov, Anil R Gangolli, Daniel E Rose, Ryan J Ernst, and Paramjit S Oberoi. 2011. SIMD-based decoding of posting lists. In *Proceedings of the 20th ACM International conference on Information and knowledge management*. 317–326.
[98] Phillip M Taylor, Nathan Griffiths, Zhou Xu, and Alexandros Mouzakitis. 2019. Data mining and compression: where to apply it and what are the effects?. In *Proceedings of the 8th ACM SIGKDD International Workshop on Urban Computing*.
[99] Larry H Thiel and HS Heaps. 1972. Program design for retrospective searches on large data bases. *Information Storage and Retrieval* 8, 1 (1972), 1–20.
[100] Sebastiano Vigna. 2013. Quasi-succinct indices. In *Proceedings of the sixth ACM international conference on Web search and data mining*. 83–92.
[101] Benjamin Welton, Dries Kimpe, Jason Cope, Christina M Patrick, Kamil Iskra, and Robert Ross. 2011. Improving i/o forwarding throughput with data compression. In *2011 IEEE International Conference on Cluster Computing*. IEEE, 438–445.
[102] Hugh E Williams and Justin Zobel. 1999. Compressing integers for fast file access. *Comput. J.* 42, 3 (1999), 193–201.
[103] Ian H Witten, Radford M Neal, and John G Cleary. 1987. Arithmetic coding for data compression. *Commun. ACM* 30, 6 (1987), 520–540.
[104] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proceedings of the VLDB Endowment* 15, 11 (2022), 3004–3017.
[105] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded piecewise linear representation for online stream approximation. *The VLDB journal* 23 (2014), 915–937.
[106] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*. 1–11.
[107] Ren Xuejun, Fang Dingyi, and Chen Xiaojiang. 2011. A Difference Fitting Residuals algorithm for lossless data compression in wireless sensor nodes. In *2011 IEEE 3rd International Conference on Communication Software and Networks*. 481–485.
[108] Ren Xuejun and Ren Zhongyuan. 2018. A Sensor Node Lossless Compression Algorithm Based on Linear Fitting Residuals Coding. In *Proceedings of the 10th International Conference on Computer Modeling and Simulation (ICCMS)*. 62–66.
[109] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*. 401–410.
[110] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment* 17, 2 (2023), 148–161.
[111] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. 323–336.
[112] Huanchen Zhang, Xiaoxuan Liu, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-preserving key compression for in-memory search trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1601–1615.
[113] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2679 – 2691.
[114] J. Ziv and A. Lempel. 1977. A universal algorithm for data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.
[115] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE, 59–59.
[116] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. 2012. Vectorwise: A vectorized analytical DBMS. In *Proceedings of the 28th International Conference on Data Engineering (ICDE)*. IEEE, 1349–1350.

Received July 2023; revised October 2023; accepted November 2023