LeCo: Lightweight Compression via Learning Serial Correlations

Yihao Liu
Tsinghua University
liuyihao21@mails.tsinghua.edu.cn

Xinyu Zeng
Tsinghua University
zeng-xy21@mails.tsinghua.edu.cn

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT

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 simultaneously. LeCo presents a general approach to this end, making existing (ad-hoc) 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 six 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 3.9× speed up in filter-scanning a Parquet file and a 16% increase in RocksDB 's throughput.

PVLDB Reference Format:
Yihao Liu, Xinyu Zeng, and Huanchen Zhang . LeCo: Lightweight Compression via Learning Serial Correlations. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

Almost all major database vendors today have adopted a column-oriented design for processing analytical queries [30, 32, 42, 49, 57, 68, 70, 85]. 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, 94]. These algorithms, such as Run-Length Encoding (RLE) [28] and Dictionary Encoding, 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, 105]) 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) [18, 58, 64, 71, 81, 84] and real-time analytics [14, 70].

Given a sequence of values of the same data type, there are generally two sources of information redundancy that we can exploit to achieve compression. The first (i.e., Source 1) is embedded in the probability distribution of the values. Here, we assume that the values are generated independently from an information source (i.e., a random variable), and the goal of compression is to approach Shannon's Entropy [88]. All forms of dictionary encoding, including those that exploit substring patterns (e.g., FSST [36], HOPE [102]), belong to this category. An optimal compression ratio is achieved when the code length for each dictionary entry is reversely proportional to its appearance frequency (i.e., by applying Huffman Encoding [59], Arithmetic Encoding [96], etc.). Nevertheless, most column-store engines prefer fixed-length codes. They are willing to sacrifice part of the compression ratio for more efficient query processing with vectorized instructions [27, 106].

Shannon's Entropy, however, is not the lower bound for compressing an existing sequence. The second source of redundancy (i.e., Source 2) lies in the serial correlation between values: the value of the current position depends on its preceding values in the sequence. The insight is that if we can model serial patterns from the sequence concisely, we can use the model to predict individual values and only store error-correction bits if necessary. As opposed to Source 1, which is thoroughly studied, few people have systematically explored Source 2 for lightweight compression (let alone formulating it as a learning problem). Although common algorithms such as RLE, FOR, and Delta Encoding [28, 73] follow this paradigm implicitly (elaborated in Section 2), they are ad-hoc solutions for particular patterns. For example, RLE only models consecutive repetitions, and the "prediction" must be accurate.

As a result, we have missed many opportunities to achieve a compressed size beyond Shannon's Entropy. Figure 2, for example, depicts a column of movie IDs from a real data set [10], where the values exhibit clear serial patterns. After learning a piecewise-linear model that can predict a value based on its position for the data set, we need fewer bits to record the prediction errors (i.e., deltas) because the learned model eliminates the serial redundancy in the original values. Compared to FOR (which is commonly used for unique integer sequences), this approach improves the compression ratio on the data set from 4.84% to 91.05% (refer to Section 4).

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 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, but instead of minimizing the sum of the squared errors, we minimize the maximum error. The reason is that we choose to store the deltas (i.e., prediction errors) in fixed length to support

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

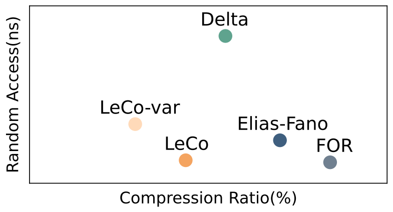


Figure 1: Performance-space trade-offs of LeCo and other lightweight compression algorithms

fast decompression and random access during query processing. A smaller max delta reduces the size of each element in the delta array and, therefore, improves the compression ratio.

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 [65, 103]. 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 and potential of using machine learning to compress columnar data. For each data partition, we store a pre-trained regression model along with an array of fixed-length deltas to record the prediction error at each position. Decompressing a value only involves a model inference plus a random access to the delta array (without the need to decompress the entire partition). The current implementation uses linear regression as the default model with both fixed-length and variable-length partition schemes (other supported models include quadratic and exponential regression). We prefer lightweight models and fixed-length deltas to support fast query processing directly on the compressed values.

We compared LeCo against state-of-the-art lightweight compression algorithms including FOR, Elias-Fano, and Delta Encoding using a microbenchmark consisting of twelve single-column data sets (three synthetic, nine real-world). Figure 1 illustrates their trade-offs between compression ratio and random access speed[1]. LeCo achieves a Pareto improvement over these algorithms. Compared to FOR and Elias-Fano, LeCo improves the compression ratio by up to 81% 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 then extended the microbenchmark with nine multi-column data sets and demonstrated that LeCo is still effective and holds the advantage over existing algorithms on classic relational tables.

We integrated LeCo into two widely-used applications to study its benefit on end-to-end system performance. We first implemented LeCo in Apache Parquet [6]. Experiments show that enabling LeCo in Parquet speeds up a filter-scan query by up to 3.9×, and accelerates bitmap-scan up to 12.6×. We also use LeCo to compress the index blocks in RocksDB [15, 46] and observed a 16% improvement in RocksDB's throughput compared to its default configuration.

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

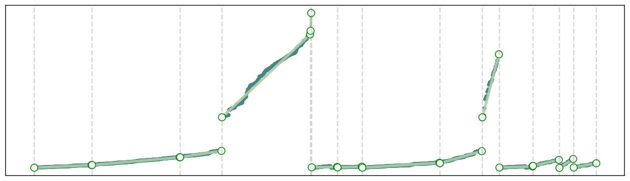


Figure 2: A Motivating Example – Learned piecewise-linear models (light green lines) have little deviations from the original values (dark green dots) on `Movie ID` data set. The dashed grey lines mark partition boundaries.

The paper makes four primary contributions. First, we identify that exploiting the serial correlation between values has a great potential for efficient column compression. Second, we propose the LeCo framework that uses machine learning systematically to remove serial redundancy in a value sequence. Third, we implement a prototype of LeCo that achieves a Pareto improvement on compression ratio and random access speed over the existing algorithms. Finally, we integrate LeCo into example applications and show that it helps improve query performance and save space at the same time.

2 THE CASE FOR LEARNED COMPRESSION

The performance of persistent storage devices has improved by orders of magnitude over the last decade [97]. Modern NVMe SSDs can achieve 7GB/s read throughput and over 500,000 IOPS [17]. 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. 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, the line to separate online analytical processing (OLAP) and online transaction processing (OLTP) is getting blurred in many real-world applications [80]. An analytical workload today is often mixed with OLTP-like queries featuring small range scans or even point accesses [13]. 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 the database management systems (DBMSs). In dictionary encoding, the DBMS maps each unique value in a column to an integer and uses those (smaller) integers to represent the entire column. For fixed-length integer symbols, the DBMS can bit-pack the codes so that each symbol only requires $log_2 N$ bits, where N is the total number of unique records. Most column stores choose to have fixed-length codes so that they can apply SIMD optimizations to speed up filtered scans and perform random accesses efficiently [69]. If variable-length codes are acceptable, however, the

DBMS can apply entropy encoding (e.g., Huffman Encoding [59]) to the integer symbols, where more frequent values are assigned to shorter codes. Then, the compressed size of the column would approach the optimal in this case: Shannon's entropy.

For a mostly unique value sequence (e.g., timestamps), dictionary encoding does not bring compression. The root cause is that dictionary/entropy encoding assumes independence between the values within a column. Under this assumption, the values form a probability distribution instead of a sequence. According to Shannon [33], if the probability distribution is mostly uniform across the value domain, there is not much room for compression.

However, the assumption of value independence ignores the position information embedded in the sequence. Values in many real-world columns often exhibit strong serial correlations (e.g., sorted or clustered) where the probability distribution of a particular value in the sequence is determined by (rather than independent of) the values of its preceding positions. A compression algorithm can take advantage of such serial correlations to achieve a compression ratio beyond the entropy[2]. For example, if the DBMS detects N consecutive values of v in a column, it can use RLE to encode the sequence as a tuple (v, N). Unfortunately, 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 succinctly, 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. Existing algorithms such as Frame-of-Reference (FOR), Delta Encoding (Delta), and Run-length Encoding (RLE) are considered special cases under our framework.

FOR divides an integer sequence into frames, and for each frame, it selects the minimum value v_{min} to be the base value of that frame. It then encodes each integer v_0 within that frame as $v_e = v_0 - v_{min}$ which requires fewer (or an equal number of) bits to store. From a LeCo 's point of view, FOR uses a horizontal-line function for the regression task in each frame. Although such a naive model is fast to train and inference, it is usually far from optimal in terms of compression ratio. RLE can be considered a special case of FOR, where the values in a frame must be identical. Therefore, all the "deltas" are zero and thus omitted in the framework.

Delta Encoding achieves compression by only storing the difference between neighboring values. Specifically, for an integer sequence $v_1, v_2, ..., v_n$, Delta encodes the values as $v_1, v_2 - v_1, v_3 - v_2, v_n - v_{n-1}$. Delta is also a special instance of the LeCo framework. 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 [91]. As discussed above, these data mining algorithms can inversely boost the efficiency of compression by extracting away the serial patterns through the LeCo framework. The theoretical foundation of this relationship is previously discussed in [48]. Notice that although we focus on regression in this paper, other data mining techniques, such as anomaly detection, also reveal serial patterns that can improve compression efficiency [29, 38]. The beauty of LeCo is that it aligns the goal of value sequence compression with that of serial pattern extraction. LeCo is an extensible framework: it provides a convenient channel to bring any related advance in data mining to the improvement of sequence compression.

Although designed to solve different problems, LeCo is related to the recent learned indexes [45, 51, 67] 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 $\rightarrow$ value) in classic pattern recognition tasks and thus has a close relationship to data mining, as discussed above.

Moreover, most compression scenarios deal with (relatively) static data sets (e.g., a Parquet file). They often follow the pattern of "compress once and access many times". Unlike indexes where incremental updates are the norm, the learned approach fits well for these common compression tasks that rarely retrain models, thus avoiding the "Achilles' heel" (i.e., inefficiency in handling updates) of learned indexes. Another key difference is that a learned index can overcome prediction errors through the "last-mile" search in the sequence. On the contrary, LeCo must record the exact prediction errors (i.e., the deltas) to guarantee a lossless compression. Because the size of the (fixed-length) delta array dominates the compression ratio and is proportional to the maximum delta size, the objective of the regression task in LeCo is to minimize the maximum delta instead of the sum of the squared deltas.

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.

3 THE LECO FRAMEWORK

As shown in Figure 3, LeCo consists of four modules: `Regressor`, `Partitioner`, `Encoder`, and `Decoder`. Given an uncompressed sequence of values, LeCo first 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

[2]The lower bound for data compression is known as the Kolmogorov Complexity. It is the length of the shortest program that can produce the original data [74]. Kolmogorov Complexity is incomputable.

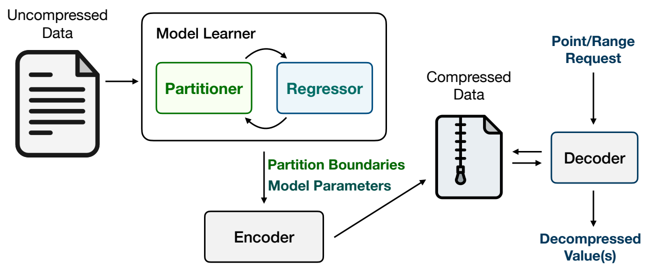


Figure 3: The LeCo Framework – An overview of the modules and the interactions with each other.

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 us to integrate 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 made for each module (Sections 3.1 to 3.3). We then introduce the extension to handling string data in Section 3.4, followed by a brief discussion about generalized learned compression Section 3.5.

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. The default model type in LeCo is the linear regression model $\theta_0 + \theta_1 \cdot i$, where i represents the position in the sequence and θ_0, θ_1 are two floating-point parameters. Classic linear regression minimizes the sum of the squared errors $\sum_i (v_i - (\theta_0 + \theta_1 \cdot i))^2$ (i.e., the l_2 norm of deltas) and 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 in Section 2, real databases rarely choose to have variable-length values because the performance overhead of scanning and filtering on such a sequence is often too large.

LeCo follows the common practice and 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. Therefore, the storage size of the delta array is determined by ϕ rather than the expected value of the deltas, and our regression objective becomes:

$$\begin{aligned} &\text{minimize} \quad \phi \\ &\text{subject to} \quad \lceil log_2(|\theta_0 + \theta_1 i - \mathbf{v}_i|) \rceil \le \phi, \quad i = 0, \dots, n-1 \end{aligned}$$

The constrained optimization problem above can be solved by linear programming, but it is too costly even for a moderate-sized sequence. Here, we propose a simple tweak to the Least Squared Method (LSM) to approximate the optimal solution. The idea is to adjust the interception (i.e., θ_0) of the line obtained from the LSM to reduce ϕ. Specifically, suppose the output of the LSM is $\theta_0 + \theta_1 \cdot i$, and there is a delta sequence $\delta_0, \delta_1, ..., \delta_{n-1}$ where $\delta_i = v_i - \lfloor(\theta_0 + \theta_1 \cdot i)\rfloor$. We use δ_{max} and δ_{min} to denote the maximum and the minimum value in the sequence, respectively, and let $L = |\delta_{max}| + |\delta_{min}|$. Notice that the Least Squared Method guarantees that $\delta_{max} > 0$ and $\delta_{min} < 0$. As long as this condition holds while we move the line vertically, L stays constant. Therefore, ϕ (i.e., $max(\lceil log_2(\delta_{max})\rceil, \lceil log_2(\delta_{min})\rceil)$) achieves minimum (for this particular slope θ_1) when $|\delta_{max}| = |\delta_{min}| = \frac{L}{2}$, which translates to adding $\frac{\delta_{\max}+\delta_{\min}}{2}$ to θ_0.

Besides the interception (θ_0), we can use a more complex algorithm to tweak the slope (θ_1) to reduce ϕ. We choose to omit this optimization because the additional compression gain is negligible. We compared the results of using a linear programming (LP) solver against those of using our "θ_0-tweak" approximation on several real-world data sets[3] (introduced in Section 4.1). We found that the overall space reduction of LP is consistently less than 5% over our approximating method. Such a gain is usually too small to justify the computational overhead of a much more complex algorithm.

The Regressor also supports more sophisticated models, such as quadratic, cubed, exponential, and logarithm. In general, a more complex model might fit some data partitions better to produce a smaller delta array, but the model itself occupies more space and takes longer to train and infer. Our observation suggests that a piece-wise linear model is often good enough for real-world data sets. Unless there is prior knowledge of the value distribution patterns, LeCo sticks with the linear model for a robust performance. A deeper exploration of higher-order models and other sophisticated data mining techniques (e.g., domain transformation) is beyond the scope of this paper.

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 overall size of the compressed sequences. Although partitioning increases the number of models to store, it is easier for the Regressor to fit a shorter subsequence to produce a smaller delta array. 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 [89]. 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 straightforward (and most common) strategy is splitting the sequence into fixed-length partitions. This partitioning scheme is easy to implement and is

[3]`movieid`, `house_price` and 300k data sampled from `facebook`

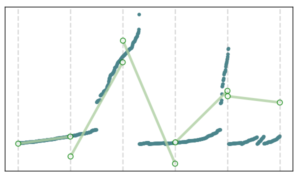


Figure 4: Fixed-length Partitioning Example.

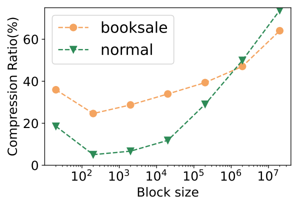


Figure 5: Compression Ratio Trend.

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 (the same one shown in the introduction) into fixed-length partitions, the Regressor and the Partitioner together fail to leverage the piecewise linearity in certain ranges.

We can find a relatively optimal point in the aforementioned trade-off by modifying the partition size. To approach an optimal partition size, we first sample a few subsequences randomly from the original values[4]. We next apply different partition sizes to the sampled data and compute the overall compression ratio. 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. Our evaluation in Section 4 shows that LeCo with the fixed-length partitioning scheme achieves a preferable balance between compression ratio and overall performance for many real-world data sets.

3.2.2 Variable-Length Partitioning. We propose a greedy algorithm for variable-length partitioning to approximate the optimal solution from the dynamic programming approach. Distributions in real-world data sets are typically uneven: Some regions align with the regression models well, while others might have large prediction errors. Intuitively, we want larger partitions for the former to amortize the space for model storage; on the other hand, increasing the number of partitions in a region with irregular local distributions might be beneficial to reduce the size of the delta arrays.

Our greedy algorithm has 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 (i.e, bit-width) 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, each having a more accurate model. If $C \le \tau S_M$, 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.

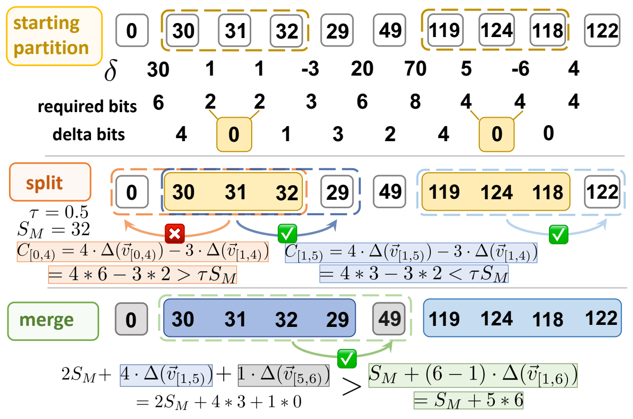
In the **merge** phase, we scan through the list of partitions $\vec{v}_{[0,k_1)}, \vec{v}_{[k_1,k_2)}, ..., \vec{v}_{[k_{m-1},k_m)}$ produced in the split phase and merge the adjacent ones if the size of the merged partition is smaller than the total size of the individual ones. Suppose the algorithm 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 $2S_M + (k_i - k_{i-1})\cdot\Delta(\vec{v}_{[k_{i-1},k_i)}) + (k_{i+1}-k_i)\cdot\Delta(\vec{v}_{[k_i,k_{i+1})})$. We accept this merge if it results in a size reduction. We iterate the partition list multiple times until no qualified merge exists.

Notice that the algorithms used in both the split and merge phases are greedy. Thus, 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 the bit-width of the max prediction error $\Delta(\vec{v}_{[i,j)})$ of this partition is already large, there is a high probability that it stays the same when including an extra data point to the partition (i.e., $\Delta(\vec{v}_{[i,j+1)}) = \Delta(\vec{v}_{[i,j)})$). Therefore, the space cost of adding this point becomes $C = \Delta(\vec{v}_{[i,j)})$. As long as $\Delta(\vec{v}_{[i,j)}) \le \tau S_M$, this "bad" partition would keep absorbing data points, which is destructive to the overall compression.

We next introduce two examples to show how to integrate this general partitioning algorithm into specific compression schemes (i.e., regression models). In each case, we discuss (1) how to compute/approximate $\Delta(\vec{v}_{[i,j)})$ (to speed up the process if necessary) without invoking the Regressor, and (2) how to find reasonably good starting partitions for our greedy process.

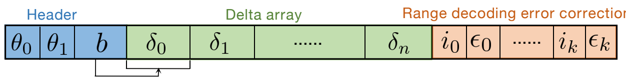
Delta Encoding As discussed in Section 2, Delta Encoding is considered a specific design point under the LeCo framework. The model in each Delta partition is an implicit step function, and only the first value in the partition is explicitly stored as the model. The prediction errors (i.e., the $\delta's$) of Delta are the differences between each pair of the adjacent values. Let $v_0, v_1, ..., v_{n-1}$ be the original sequence. Let $d_1 = v_1 - v_0, d_2 = v_2 - v_1, ..., d_{n-1} = v_{n-1} - v_{n-2}$. Then $\Delta(\vec{v}_{[0,n)}) = \lceil log_2(max_{i=1}^{n-1} d_i)\rceil$. After adding the next data

[4]We found that a sampling rate < 1% is often good enough empirically



starting partition
0 30 31 32 29 49 119 124 118 122
δ 30 1 1 -3 20 70 5 -6 4
required bits 6 2 2 3 6 8 4 4 4
delta bits 4 0 1 3 2 4 0 0
split
$\tau = 0.5$
$S_M = 32$
$C_{[0,4)} = 4 \cdot \Delta(\vec{v}_{[0,4)}) - 3 \cdot \Delta(\vec{v}_{[1,4)})$
$= 4*6-3*2 > \tau S_M$
$C_{[1,5)} = 4 \cdot \Delta(\vec{v}_{[1,5)}) - 3 \cdot \Delta(\vec{v}_{[1,4)})$
$= 4*3-3*2 < \tau S_M$
merge
$2S_M + 4 \cdot \Delta(\vec{v}_{[1,5)}) + 1 \cdot \Delta(\vec{v}_{[5,6)}) > S_M + (6-1) \cdot \Delta(\vec{v}_{[1,6)})$
$= 2S_M + 4*3+1*0$
$= S_M + 5*6$

Figure 6: Variable-length Partitioning on Delta – For each attempt to add points or merge segments (marked by arrows), we present the "evidence" (cost calculations), deciding whether this action succeeds below the arrows.



Header
Delta array
Range decoding error correction
θ_0 θ_1 b δ_0 δ_1 ...... δ_n i_0 ϵ_0 ...... i_k ϵ_k

Figure 7: LeCo's Storage Format for One Partition

point v_n to this partition, we can directly compute $\Delta(\vec{v}_{[0,n+1)}) = \max\{\Delta(\vec{v}_{[0,n)}), d_n\}$.

A good starting partition for Delta Encoding 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.

According to Figure 6, we choose to first grow the segment $\{30, 31, 32\}$. In the dark blue dashed frame, we try adding the record 29 to the existing segment, which includes a new delta of 3 bits. Thus, we update the model prediction error of the newly-formed segment $\Delta(\vec{v}_{[1,5)}) = \max(\Delta(\vec{v}_{[1,4)}), 3) = 3$. The cost enhancement of adding this point is calculated to be 6, which is smaller than the pre-defined threshold $\tau S_M = 0.5 * 32 = 16$, and thus the original partition successfully grows. In the merge phase, we try to merge the partition $\{30, 31, 32, 29\}$ with its adjacent segment $\{49\}$ presented in the green dashed frame. This attempt succeeded because the space consumption of the segment formed is smaller than the summation of the two original segments.

Linear Regressor Computing $\Delta(\vec{v}_{[i,j)})$ by invoking the linear Regressor takes $O(N)$ time. Again, let $v_0, v_1, ..., v_{n-1}$ be the original sequence and $d_k = v_k - v_{k-1}$ for $k = 1, 2, ..., n-1$. To speed up the partitioning process, we propose a metric $\widetilde{\Delta}(\vec{v}_{[i,j)}) = log_2(\max_{k=i+1}^{j-1}(d_k) - \min_{k=i+1}^{j-1}(d_k))$ to approximate the functionality of $\Delta(\vec{v}_{[i,j)})$ in the split/merge cost computations. 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 $\Delta(\vec{v}_{[i,j)})$.

The best starting partition for the linear Regressor is when the included points form an arithmetic progression. Thus, similar to Delta Encoding, we select positions with small second-order deltas (computed from the original sequence) as the start points to initiate the partitioning algorithm.

We evaluated our variable-length partitioning algorithm with the linear Regressor on the real-world data sets (introduced in Section 4.1). Compared to the optimal partitioning obtained via dynamic programming, our greedy algorithm imposes less than 3% overhead on the final compressed size.

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 into 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 θ_0 (the intercept) and θ_1 (the slope). Because we apply bit-packing to 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 [45] (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 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 propose an optimization to speed up range decompression by saving one floating-point operation per value. For position i, the model prediction is $\hat{v}_i = \theta_0 + \theta_1 \cdot i$. But if the previous prediction

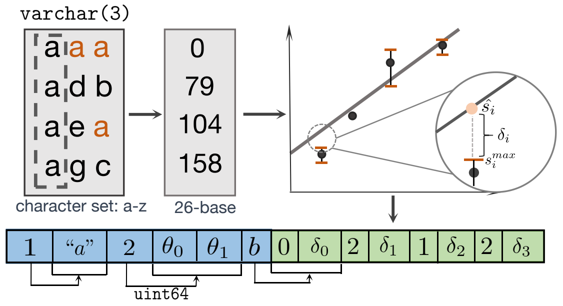


Figure 8: LeCo String Compression – An example of extending LeCo to support strings, including algorithm optimizations and storage format modifications.

$\hat{v}_{i-1} = \theta_0 + \theta_1 \cdot (i-1)$ is available (which is true in range decompression), we can obtain $\hat{v}_i$ by computing $\hat{v}_{i-1} + \theta_1$, thus saving the floating-point multiplication. According to our experiments, this optimization improves the throughput of range decompression throughput by $10 - 20\%$.

There is an additional issue to fix for this optimization. Because of the limited precision in the floating-point representation, the value decoded through the θ_1-accumulation approach could be different from the one decoded via direct model inference at certain position i (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. Notice that the space cost of this error-correction list is negligible because the miscomputation caused by insufficient precision is rare.

3.4 Extension to Handling Strings

So far, our discussion has focused on compressing integer values. In this section, we extend our support to 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. The basic mapping is to treat a string as a 128-based integer. For example, a length-2 string "db" is equivalent to $100 \times 128 + 98 = 12898$ in base 10. To represent large integers, we use the built-in uint64 for a value less than 8 bytes. If the value is between 8 and 16 bytes, we use the built-in uint128. For values larger than 16 bytes, we rely on the multiprecision support from the Boost library [7]. We next present several optimizations to reduce the sizes of the mapped integers to speed up computation and facilitate regression.

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. This effectively shortens the string lengths and thus potentially reduces the sizes of the mapped integers. The next optimization is to 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. In Figure 8, the data set only involves lower-case letters, we can treat them as 26-based integers, and string "db" is mapped to $3 \times 26 + 1 = 79$ instead of 12898 in the previous 128-based conversion.

Notice that for an arbitrary M-based mapping, the computation required to recover each character from the integer is expensive. Given the mapped integer v, it requires an integer modulo $v\%M$ to decode the current character and an integer division v/M to prepare for decoding the next one. Both operations take tens of CPU cycles. To speed up decoding, we set M to its closest power of two (2^m) so that the modulo becomes a left-shift followed by a bit-wise AND ($v\&((1 << m) - 1)$), and the division becomes a right-shift ($v >> m$). For example, for strings that only consist of lower-case characters, we set $M = 32$.

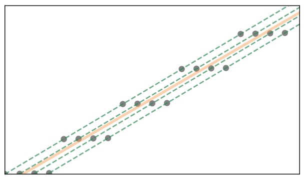


Figure 9: A Case Study – Distribution with a clear "step" pattern.

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 prefix of the partition. We also record the length of each varchar value in the delta array. These lengths can be omitted for fixed-length strings (i.e., without padding).

3.5 Discussion

We believe that the prototype introduced in this section shows only the "tip of the iceberg" of the potential of learned compression. As discussed in Section 2, learned compression is closely related to data mining, and a plain regression is just one of the most common approaches in pattern discovery. Other (auxiliary) methods, such as domain transformation, can largely improve the model-fitting accuracy on certain data sets. For example, as shown in Figure 9, if we break the domain into four according to function $T(i) = i \bmod 4$ and apply a linear Regressor on each of the subdomains (the green dashed lines), we can obtain more accurate predictions than the original solid yellow line, thus achieving a better compression ratio. A deeper integration of data mining techniques into the LeCo framework remains an interesting future direction.

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 [79, 93], Frame-of-Reference (FOR) [55, 105], Delta Encoding (Delta) [28], and rANS [47]. FOR and Delta are introduced in Section 2. rANS is a variant of arithmetic encoding [96] 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, with m being the difference between the maximum and minimum value of the sequence. 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 values.

We evaluate LeCo and the baseline solutions extensively on nine integer data sets and three string data sets:

- **linear, normal**: synthetic data sets with 200M 32-bit integers following a clean linear (or normal) distribution.
- **poisson**: 87M 64-bit timestamps modeling events collected by distributed sensors from the SuRF paper [101]. The timestamps follow a Poisson distribution.
- **ml**: 14M 64-bit timestamps in milliseconds derived from the UCI-ML data set [19].
- **booksale**: 200M 32-bit integers from the SOSD benchmark [65] representing book sale popularity data at Amazon.
- **facebook**: 200M 64-bit integers from the SOSD benchmark representing Facebook user IDs.
- **wiki**: 200M 64-bit integers from the SOSD benchmark representing edit timestamps of Wikipedia articles.
- **movieid**: 20M 32-bit movie IDs in MovieLens that users "liked" (sorted by user) [10].
- **house_price**: 100K 32-bit sorted integers representing the distribution of house prices in the US [11].
- **email**: 30K email addresses (host reversed) with an average string length of 15 bytes [2].
- **hex**: 100K sorted hexadecimal strings (up to 8 bytes) from the FSST paper [36].
- **word**: 222K English words with an average length of 9 bytes [3].

Figure 10 shows a visualization of the nine integer data sets. We observe noticeable unevenness in the zoom-in distributions of the 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 code [5] is compiled using gcc version 9.4.0 with `-O3` optimization. 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, as described in Section 3.2. `LeCo-fix` and `LeCo-var` are linear-Regressor LeCo prototypes that adopt fixed-length and variable-length partitioning, respectively. For all fixed-length partitioning methods, including LeCo-fix, Delta, FOR, and Elias-Fano, the partition size is obtained through a quick sampling-based parameter search described in Section 3.2. For `Delta-var` and `LeCo-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*) as well as the compression throughput. Then the algorithm performs N uniformly random accesses, where N equals to the number of items in the data set, and reports the average latency per access. 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.

4.3 Integer Benchmark

Figure 11 shows the experiment results for compression ratio, random access latency, and decompression throughput on the nine 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 accesses.

4.3.1 *Compression Ratio.*

As shown in the first row of Figure 11, 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 (refer to Section 2), where the output of the Regressor is fixed to a horizontal line (θ_0 equals the minimum value in the partition).

The compressed sizes from LeCo are also smaller than those from Elias-Fano across (almost) all data sets. Although Elias-Fano is proved to be quasi-succinct (relative to the information-theoretic lower bound), it is ignorant of the embedded serial correlation between the values, thus missing the opportunity to further compress the data sets. 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-fix and Delta-var, LeCo-fix and LeCo-var shows a remarkable improvement on compression ratio for "smooth" (synthetic) data sets: `linear`, `normal`, and `poisson`. For the remaining (real-world) data sets, however, LeCo remains competitive. This is because many real-world data sets exhibit local unevenness, as shown in Figure 10. The degree of such irregularity is often at the same level as the difference between adjacent values. Hence, the size of the delta array (i.e., to record prediction errors) in LeCo is similar to that of the delta array (i.e., differences between adjacent values) in Delta Encoding for these data sets.

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

[5] https://gitfront.io/r/Leco2023/Hk2zGFeQUSVw/Learn-to-Compress/

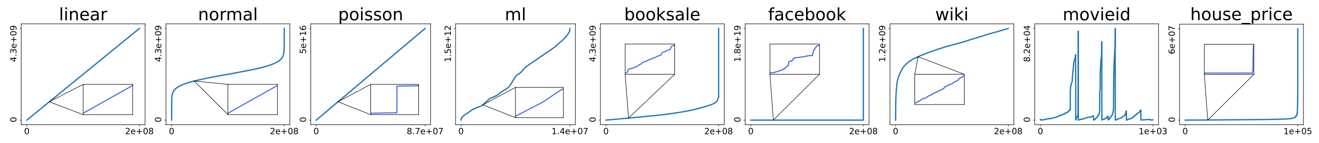


Figure 10: Data Distribution.

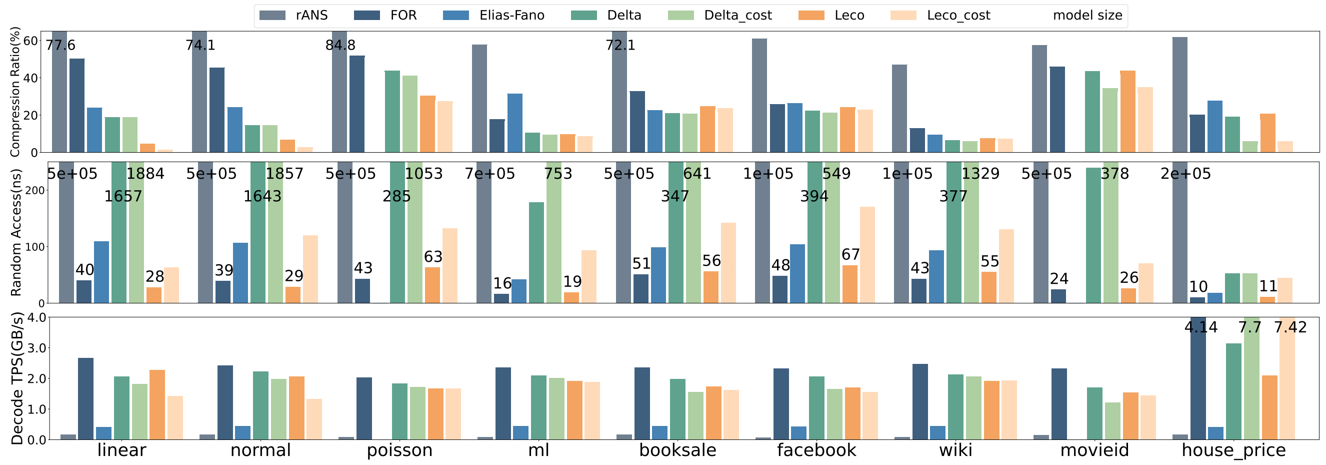


Figure 11: Compression Microbenchmark – Measurement of seven compression schemes on nine integer data sets from three aspects: Compression Ratio, Random Access Latency, and Full Decompression Throughput. We break down the compression ratio into model size (marked with the cross pattern) and delta size in the first row.

partitions accordingly to avoid oversized partitions caused by unfriendly patterns to the Regressor.

We also implement the partitioning algorithm used by FITing tree [53] in LeCo and compare it with LeCo-var. FITing tree predefines a fixed prediction-error bound (ϵ) for each partition and greedily determines the partition boundaries of data in one pass.

Figure 12 shows a comparison of the compression ratios, where both LeCo-FITing and LeCo-var use their default hyper-parameters (i.e., ϵ for LeCo-FITing, τ for LeCo-var). LeCo-FITing is consistently worse than LeCo-var. In particular, LeCo-FITing produces a compression ratio that is even higher than LeCo-fix on the `normal` data set. This is because a globally fixed error bound does not adapt well to a data set with a rapidly changing slope. We observe that LeCo-FITing is much more sensitive to its hyperparameter. As shown in Figure 13, when varying LeCo-FITing's hyper-parameter ϵ from 3 to 13 bit on the `books` data set, the compression ratio exhibits a big swing between 0.3 to 0.6. In contrast, the output compression ratio of LeCo-var is relatively stable when we sweep its hyper-parameter τ from 0 to 0.2.

4.3.2 ***Random Access.*** The second row of Figure 11 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 first 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 a random access.

4.3.3 ***Full Decompression.*** The third row in Figure 11 shows the throughput of each compression algorithm for decompressing an entire data set. In general, LeCo-fix is 14% – 34%[6] slower than its fastest competitor FOR. Decoding a value sequentially in FOR only requires an integer addition, while LeCo-fix must include an extra floating-point operation. 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. Once LeCo's Partitioner detects them, these sequences are likely to form their own partitions, making the decompression task trivial for these partitions.

Notice that the benefit of having a smaller compressed size is barely reflected in the above throughput measurements when everything is already in memory. We show in Section 5 that a better compression ratio (by LeCo) can improve the query performance in real-world applications.

4.3.4 ***Compression throughput.*** Figure 14 shows the compression throughput for each algorithm weighted averaged across all the nine data sets with error bars[7]. The variance is mainly because of the different compression ratios across data sets. LeCo-fix has a similar compression speed to the baselines because our linear Regressor has a low computational overhead. Algorithms that adopt

[6] except for `house_price` where the enhancement of FOR over LeCo-fix is 49%
[7] The standard deviation of Delta-var and LeCo-var is 0.011 and 0.008.

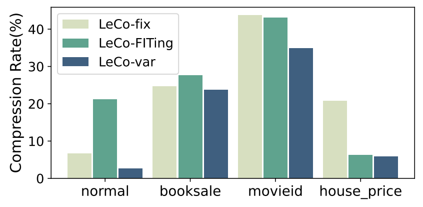


Figure 12: LeCo-var vs. LeCo-FITing

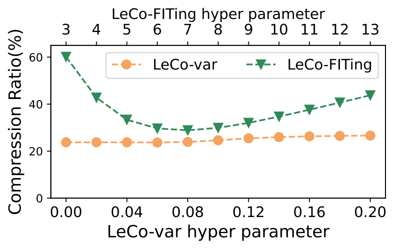


Figure 13: Hyper Parameter Sweep.

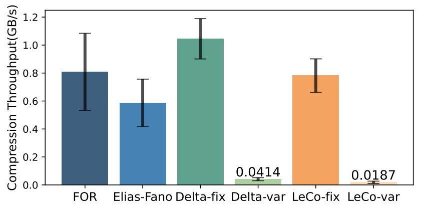


Figure 14: Compression Throughput.

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.

Variable-length partitioning presents a classic trade-off between compression ratio and throughput (note that LeCo-var is competitive in random access and range decompression). For data processing applications that do not allow in-place updates (which is common), it might be beneficial to apply compression schemes with variable-length partitioning to trade the expensive one-time compression for smaller file sizes.

4.4 Multi-Column Benchmark

In this section, we evaluate the effectiveness of LeCo on multi-column tabular data sets[8]. As shown in Figure 15 (bottom right), the benchmark includes three tables from TPC-H [26] (i.e., `lineitem`, `partsupp`, and `orders`), three tables from TPC-DS [25] (i.e., `inventory`, `catalog_sales`, and `date_dim`), and three real-world tables (i.e., `geo` [21], `stock` [22], and `course_info`[23]). The benchmark considers numerical columns only. Each table is sorted by its primary-key column. We compute the “sortedness” of a table (in the range [0, 1]) by averaging the sortedness of each column with respect to the primary-key column using the portion of inverse pairs[39] as the metric.

From Figure 15 (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, 60, 86]. 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 linear patterns. Tables with high sortedness measures such as `inventory` and `data_dim`, therefore, are more likely to achieve outstanding compression ratios with the LeCo variants.

The bottom left of Figure 15 presents the compression ratios of the TPC-H tables[9] 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.

4.5 String Benchmark

We compare LeCo (i.e., LeCo-fix) against the state-of-the-art lightweight string compression algorithm FSST [36] on the three string data sets `email`, `hex`, and `word` (introduced in Section 4.1). Figure 16 shows the experiment results with random access latency on the x-axis and compression ratio on the y-axis.

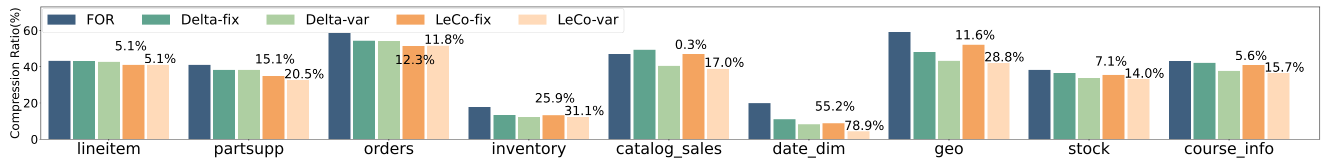
FSST adopts a dictionary-based approach by building a fine-grained static symbol table to map a partial string to a 1-byte code. Because each compressed string has a variable length, FSST must store a byte-offset array to support random access. An optimization (not mentioned in the FSST paper) is to 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 in each figure: one that uses the exact character-set size and the other that extends the character-set size to the closest power of two (refer to Section 3.4).

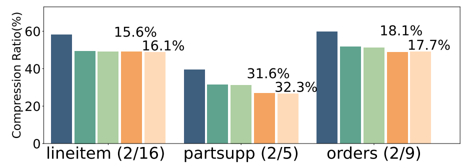
As shown in Figure 16, LeCo has a competitive compression ratio and a superior random access speed compared to optimized FSST on the `email` and `hex` data sets. A random decode in LeCo only involves two memory accesses and a few light numerical computations. For the `word` data set, however, the compression ratio of LeCo is worse than that of FSST. This is because the two algorithms focus on reducing different types of redundancies, as explained in Section 2. FSST targets at detecting repeated patterns to reduce the entropy, while LeCo is mainly designed to exploit the serial correlation between values for a more compact sequence representation. In the case of English words, prefixes, roots, and suffixes are common (which are friendly to FSST), while there is a large variance in the words’ serial patterns (which is hard to fully capture using the current LeCo prototype). An interesting future work is to combine these two (partially orthogonal) approaches to pursue an even better lightweight compression scheme.

5 SYSTEM EVALUATION

To show how LeCo can benefit real-world systems, we integrated LeCo into two widely-used applications: Parquet [6] and RocksDB [15, 46]. For each application, we first introduce the implementations. Then, we present a set of experiments to demonstrate how LeCo improves query peformance and storage cost simultaneously. 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 I/OPS. This setting is similar to RocksDB’s recommendation [16]. We use Apache Arrow 8.0.0, Parquet version 2.6.0 and RocksDB Release version 6.26.1 in the following experiments.

[8]Elias-Fano is not included as baseline because most columns are not strictly sorted.
[9]Due to space limitations, we only present the results of TPC-H. Results of the other six data sets can be found in our technique report at [24].





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

Figure 15: 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.

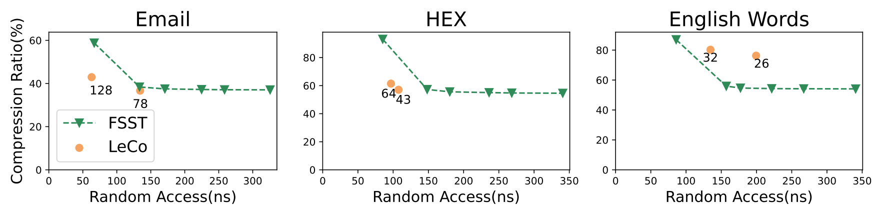


Figure 16: LeCo String Evaluation – Comparison between LeCo and FSST on three string data sets. We mark the base value used to convert strings above each point of LeCo.

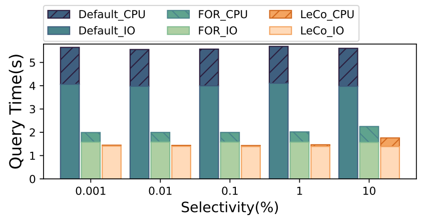


Figure 17: Parquet Filter Scan Query.

5.1 Integration to Parquet

Apache Parquet [6] has become a standard open format for columnar data storage. In this section, we study how lightweight compression schemes such as LeCo can benefit the format in both query performance and storage cost. We implemented LeCo and FOR in Apache Parquet using its C++ implementation [4]. 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`. For the experiments in Section 5.1.1 and Section 5.1.2, We set Parquet's row group size to 10M rows and disabled the block compression in Parquet to remove the common interference to our result presentation.

5.1.1 *Filter Scan.*

Filter scan is one of the most common operations performed at database storage. It refers to the process of applying a filter predicate on one column and then use the result bitmap or selection vector to filter another column (for engines that supports late materialization [37]). We created a table consisting of 200M rows and two columns: (1) `ts` is an almost-sorted timestamp column (in seconds) extracted from the `ml` [19] data set, and (2) `id` contains (randomly shuffled) Facebook user IDs [65]. The query returns a list of users who log in to the application in a specific time range per day: SELECT `id` FROM table WHERE *time_1* < `ts` % $(24 * 60 * 60) <$ *time_2*. We adjust the time range (i.e., *time_2* - *time_1*) to control the query selectivity. We generated three Parquet files with `Default`, FOR, and LeCo-fix as the encoding algorithm separately. For FOR and LeCo-fix, we set the partition size to 10K items. The resulting file size is 3.0GB, 1.3GB, and 1.1GB, respectively. We repeat each query three times and report the average execution time.

As shown in Figure 17, LeCo reduces the query execution time by 3.7× compared to Parquet `Default` and by 1.4× compared to FOR on average with varying selectivities. The I/O times across the algorithms are proportional to the corresponding file sizes because the I/O skipping is performed at the row-group level (via zone maps)[10]. Both FOR and LeCo support partition-level skipping in memory by reading the partition headers only. But interestingly, LeCo saves up to 9.2× CPU times compared to FOR in this filter-scan template. This is because LeCo can further skip decoding items *within* a partition. Suppose that the model of the 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.

5.1.2 *Bitmap Selection.*

We then zoom in to examine the bitmap selection operation only. Specifically, when Parquet takes a bitmap as input, it applies the filter to the column and outputs a sequence of qualified values. For the experiments in this subsection, we use a subset of the data sets introduced in Section 4.1: two synthetic data sets (`normal` and `poisson`) and two real-world data sets (`booksale` and `ml`[11]). Given a data set, we create one Parquet file for each compression option (i.e., Default, FOR, and LeCo-fix) to store the data. A row group is skipped if it does not contain any requested value. We then feed each Parquet file with bitmaps having different ratios

[10]Zone maps at a finer granularity is currently unsupported Parquet's C++ version.
[11]Because the `ml` data set is too small compared to the others, we scale it to 200M rows while preserving its value distribution.

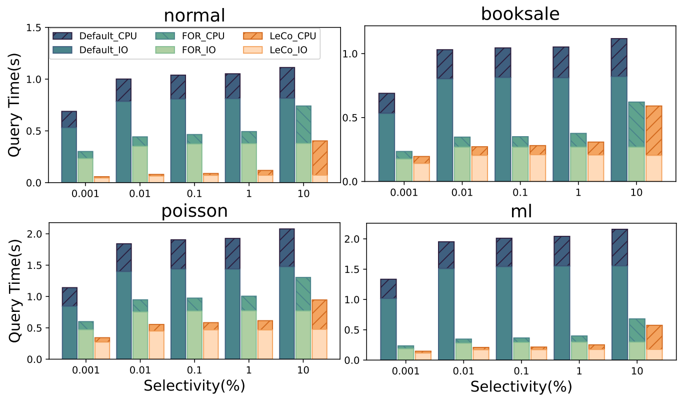


Figure 18: Parquet Bitmap Selection – The overall speedup ratio of LeCo over FOR is marked above each LeCo's bar.

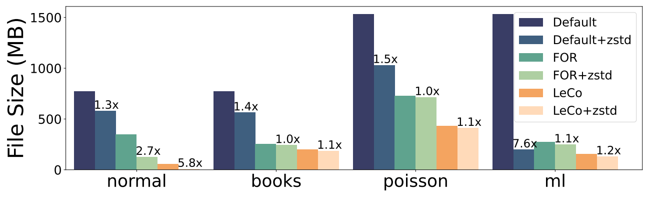


Figure 19: Parquet With zstd Compression – Numbers on bars indicating additional improvement introduced by zstd.

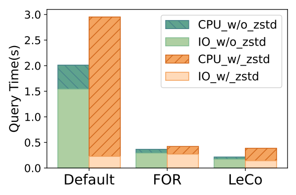


Figure 20: Time breakdown of zstd on Parquet.

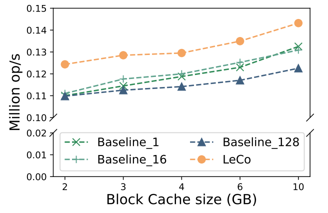


Figure 21: RocksDB Seek Query Throughput.

of "ones" (to represent a varying filter selectivity). Each bitmap includes ten set-bit clusters following a Zipf-like distribution.

Figure 18 shows the time consumed to perform a bitmap-scan on Parquet files with different compression options. We vary the filter selectivity on the x-axis. LeCo consistently outperforms Default (by up to 12.6×) and FOR (by up to 5.6×). Compared to FOR with a selectivity less than 1%, LeCo's speedup comes from both the I/O reduction (due to a better compression ratio) and the CPU saving (due to better caching). As the selectivity increases to 10%, the size advantage of LeCo is generally diluted by an increased computational cost for decoding[12].

5.1.3 Enabling Block Compression. Usually, people also enable block compression on columnar storage format such as Parquet and ORC [5] to further reduce the storage overhead. We repeat the Parquet loading phase of the above experiments with zstd [20] enabled to show how block compression algorithms affect the final file sizes with different lightweight compression techniques, including LeCo.

As shown in Figure 19, applying zstd on top of the lightweight encoding schemes in Parquet can further reduce the file sizes. LeCo + zstd is the most space-efficient configuration in all cases with an exceptionally low compression ratio on the `normal` data set. The relative improvement of LeCo + zstd over LeCo is higher than that in the case of FOR. These results show that LeCo's ability to remove serial redundancy is complementary to general-purpose block compression algorithms based on dictionary encoding.

The decompression overhead of zstd, however, can be significant. We repeated the bitmap selection experiments in Section 5.1.2 with zstd turned on for Parquet. Figure 20 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. Each SSTable is divided into multiple data blocks (4KB by default). RocksDB builds an index on top of the data blocks. For each pair of adjacent data blocks B_{i-1} in front of 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 offest and the size of B_i.

To locate a particular key k, RocksDB performs a 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 simply stores the offset of each block in a delta-encoded sequence.

Our second system application is to use LeCo to compress the keys and values separately in a RocksDB index block to shrink its index size and to improve the lookup performance at the same time. We use 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[13] 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). These three baselines represent different trade-offs between the index block size and lookup performance. We configured RocksDB according to the settings in its

[12]Because the "zone" skipping granularity in the current C++ implementation of Parquet is a row group, a 10% selectivity requires almost an entire file scan + decoding.

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

Performance Benchmark [16][14]. 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 followed by 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 [41] 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 21 shows the system throughputs for LeCo, Elias-Fano, and the baselines with a varying block cache size (block cache in RocksDB is used to cache both index blocks and data blocks). 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 (because 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, as shown in Figure 21.

Second, compared to `Baseline_16` and `Baseline_128` where the index blocks are compressed using Delta Encoding. Although LeCo no longer exhibits a index-size advantage over these baselines, it supports faster random accesses and thus saves a significant amount of computations. For `Baseline_128`, for example, it has to decompress the entire 128-entry unit each time before it accesses a single entry. On the contrary, LeCo only requires two memory probes to perform a random access in the index block.

To summarize, in this application, we use LeCo to compress the plain indexes (or "zone maps") to speed up the binary search. Such a small change noticeably improved the overall performance of a complex system (RocksDB). 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 [28] omits the leading zeros in the bit representation of an integer and records the byte length of each value, such as in 4-Wise NS [87], Masked-VByte [83], Google varint [1] and varint-G8IU [90]. Dictionary [31, 34, 36, 76, 78, 85, 102] and entropy-based compression algorithms [59, 96] build a bijective map between the original values and the code words. As discussed in Section 2, they only perform well on columns with low/moderate cardinalities. Block compression algorithms such as LZ77 [104], Gzip [8], Snappy [9], LZ4 [12], and zstd [20] interpret data as bit streams. They achieve compression by replacing repeated bit patterns with shorter dictionary codes. All of the above approaches miss the opportunity to exploit the serial correlation between values to achieve a compressed size beyond Shannon's Entropy.

Antonio et al. [35] applies a similar model used in FITing-Tree [53] and PGM-Index [51] on an index structure called the rank/select dictionary to reduce its size. As opposed to a specific data structure, LeCo is a general and extensible framework that bridges data compression with data mining and is proven beneficial to various real-world applications. Semantic compression [54, 60, 61] aims to compress tabular data using complicated models like Bayesian networks. Their focus is to exploit correlations **between** columns to achieve better compression.

LFR[99] and DFR[98] attempts to use linear model or Delta-like model to compress data without partitioning, but their model parameter vary at each data point, thus they do not support random access on the compressed sequence.

Data partitioning plays an essential role in achieving a good compression ratio for various algorithms. Several prior work [79, 82] targeting inverted indexes proposed partitioning algorithms for specific compression schemes like Elias-Fano [93] and VByte [92, 95]. The partitioning algorithms introduced in Section 3.2 are general under the LeCo framework (although not universal).

In terms of storage format, FastPFOR [105] and NewPFD [100] apply an optimization where the outlier values (detected using a threshold) are stored separately in a different format to improve the overall storage and query efficiency.

Both learned indexes and learned compression use regression to model data distributions. Existing learned indexes such as RMI [67] and RS [66] apply hierarchical machine learning models to depict the distributions. PGM-Index [51], FITing-Tree [53], and CARMI [103] put more effort into the data partitioning strategies to reduce model prediction errors. ALEX [45] and Finedex [75] proposed techniques such as a gapped array and non-blocking retraining to alleviate the learned indexes' intrinsic inefficiency in handling updates. The relationship between LeCo and learned indexes has been described in detail in Section 2.

Previous work [27, 106] have shown that heavyweight compression algorithms [8, 9, 59] designed for disk-oriented systems could incur notable computational overhead to the overall system performance, especially with a narrowing gap between the processor speed and the storage bandwidth. Algorithms such as FSST [36] and PIDS [62], therefore, emphasize on low CPU usage besides a competitive compression ratio. Other related work reduces the computational overhead by enabling direct query execution on compressed formats [28, 43, 63], including filter and aggregation/join pushdowns [40, 44, 50, 56, 69, 72, 77].

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 an 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, LeCo is extensible to incorporate future advances in relevant data mining techniques. Both our microbenmark and system evaluation show that LeCo is able to achieve better storage efficiency and faster query processing simultaneously.

[14]block_size = 4096B; block format_version = 5; pin_l0_filter_and_index_blocks_in _cache is enabled.

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. Boost multiprecision library. https://www.boost.org/doc/libs/1_55_0/libs/multiprecision/doc/html/boost_multiprecision/intro.html.

[8] 2022. GNU GZip. https://www.gnu.org/software/gzip/.

[9] 2022. Google snappy. http://google.github.io/snappy/.

[10] 2022. Kaggle Movie ID dataset. https://www.kaggle.com/datasets/grouplens/movielens-20m-dataset?select=rating.csv.

[11] 2022. Kaggle USA Real Estate Dataset. https://www.kaggle.com/datasets/ahmedshahriarsakib/usa-real-estate-dataset?select=realtor-dataset-100k.csv.

[12] 2022. Lz4. https://github.com/lz4/lz4.

[13] 2022. Personal communication, anonymized for review. .

[14] 2022. Real-time Analytics for MySQL Database Service. https://www.oracle.com/mysql/.

[15] 2022. Rocksdb Github. https://github.com/facebook/rocksdb.

[16] 2022. Rocksdb Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks.

[17] 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/.

[18] 2022. SingleStore. https://www.singlestore.com/.

[19] 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.

[20] 2022. Zstandard. https://github.com/facebook/zstd.

[21] 2023. GeoNames Data. https://www.geonames.org/export/.

[22] 2023. HistData GRXEUR. https://www.histdata.com/.

[23] 2023. Kaggle Udemy Courses. https://www.kaggle.com/datasets/hossaingh/udemy-courses.

[24] 2023. Technical Report. https://gitfront.io/r/Leco2023/Hk2zGFeQUSVw/Learn-to-Compress/blob/technical_report.pdf.

[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] Shikha Agrawal and Jitendra Agrawal. 2015. Survey on anomaly detection using data mining techniques. *Procedia Computer Science* 60 (2015), 708–713.

[30] Ramakrishna Varadarajan Nga Tran Ben Vandiver Lyric Doshi Chuck Bear Andrew Lamb, Matt Fuller. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1790–1801.

[31] Gennady Antoshenkov, David Lomet, and James Murray. 1996. Order preserving string compression. In *Proceedings of the Twelfth International Conference on Data Engineering.* IEEE, 655–663.

[32] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data.* 2205–2217.

[33] Andrew Barron, Jorma Rissanen, and Bin Yu. 1998. The minimum description length principle in coding and modeling. *IEEE transactions on information theory* 44, 6 (1998), 2743–2760.

[34] 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.

[35] 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.

[36] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[37] 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.

[38] Paul Boniol, John Paparrizos, Themis Palpanas, and Michael J Franklin. 2021. SAND: streaming subsequence anomaly detection. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1717–1729.

[39] C. G. Borroni. 2013. A new rank correlation measure. *Statistical Papers* 54, 2 (2013), 255–270.

[40] Lemke Christian, Sattler Kai-Uwe, Faerber Franz, and Zeier Alexander. 2010. Speeding up queries in column stores: a case for compression. *DaWaK (2010)* (2010), 117–129.

[41] 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.* 143–154.

[42] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data.* 215–226.

[43] 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. *arXiv preprint arXiv:2004.09350* (2020).

[44] 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.

[45] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2019. ALEX: An Updatable Adaptive Learned Index. (2019). https://doi.org/10.1145/3318464.3389711 arXiv:arXiv:1905.08898

[46] 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.

[47] 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).

[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 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.* 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, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, 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] 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.

[61] HV Jagadish, Jason Madar, and Raymond T Ng. 1999. Semantic compression and pattern extraction with fascicles. In *VLDB*, Vol. 99. 186–97.

[62] 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.

[63] 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 International Conference on Management of Data*. 843–856.

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

[65] A. Kipf, R Marcus, A Van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. 2019. SOSD: A Benchmark for Learned Indexes. (2019).

[66] 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.

[67] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[68] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

[69] 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 International Conference on Management of Data*. 311–326.

[70] 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.

[71] 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.

[72] Jae-Gil Lee, Gopi Attaluri, Ronald Barber, Naresh Chainani, Oliver Draese, Frederick Ho, Stratos Idreos, Min-Soo Kim, Sam Lightstone, Guy Lohman, et al. 2014. Joins on encoded and partitioned data. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1355–1366.

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

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

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

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

[81] 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.

[82] 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.

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

[84] 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.

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

[86] 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.

[87] 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.

[88] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[89] 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.

[90] 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.

[91] Phillip Taylor, Nathan Griffiths, Zhou Xu, and Alex Mouzakitis. 2018. Data mining and compression: where to apply it and what are the effects? (2018).

[92] 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.

[93] Sebastiano Vigna. 2013. Quasi-succinct indices. In *Proceedings of the sixth ACM international conference on Web search and data mining*. 83–92.

[94] 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.

[95] Hugh E Williams and Justin Zobel. 1999. Compressing integers for fast file access. *Comput. J.* 42, 3 (1999), 193–201.

[96] Ian H Witten, Radford M Neal, and John G Cleary. 1987. Arithmetic coding for data compression. *Commun. ACM* 30, 6 (1987), 520–540.

[97] 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.

[98] 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. https://doi.org/10.1109/ICCSN.2011.6013638

[99] 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 '18)*. Association for Computing Machinery, New York, NY, USA, 62–66. https://doi.org/10.1145/3177457.3177482

[100] 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.

[101] 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 International Conference on Management of Data*. 323–336.

[102] 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.

[103] 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.

[104] J. Ziv and A. Lempel. 1977. A universal algorithm for data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.

[105] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 59–59.

[106] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. 2012. Vectorwise: A vectorized analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1349–1350.