LICO: An SIMD-Aware High-Performance Learned Inverted Index Compression Framework

XIANYU ZHU, Renmin University of China, China
QIYU LIU*, Southwest University, China
GUANGYI ZHANG, Shenzhen Technology University, China
ZHIBING SHA, Southwest University, China
JIANWEI LIAO, Southwest University, China
SHA HU, Southwest University, China
LEI CHEN, The Hong Kong University of Science and Technology (Guangzhou), China

Inverted indexes (a.k.a. posting lists) are core data structures in search engines and database systems, where compression is crucial to reduce memory footprint and accelerate query processing. The key challenge lies in encoding a sorted list of integer identifiers in a compact, lossless form while supporting fast decoding. Motivated by recent studies on learned data structures, this work presents **LICO**, a novel Learned Inverted index COmpression framework that encodes sorted integers with error-bounded machine learning models and auxiliary residual arrays for lossless reconstruction. Compared to classical schemes such as P4Delta and Elias-Fano, and recent learning-based approaches such as LA-vector and LeCo, LICO offers three key advantages: (1) a succinct data structure explicitly designed for leveraging parallelism offered by modern hardware; (2) fully automatic adaptation to data distributions without hand-tuned hyperparameters; and (3) rigorous theoretical guarantees on compression ratios. Extensive experiments on web-scale datasets and query workloads show that LICO achieves Pareto-optimal trade-offs between index size and query latency. All code and artifacts are available at: https://github.com/xianyuzhuruc/LICO.

CCS Concepts: • **Information systems → Search index compression**; • **Computing methodologies → Learning linear models**; • **Computer systems organization → Single instruction, multiple data**.

Additional Key Words and Phrases: Learned Data Compression, Data Partitioning, SIMD Acceleration

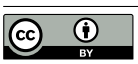
ACM Reference Format:
Xianyu Zhu, Qiyu Liu, Guangyi Zhang, Zhibing Sha, Jianwei Liao, Sha Hu, and Lei Chen. 2026. LICO: An SIMD-Aware High-Performance Learned Inverted Index Compression Framework. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 202 (June 2026), 27 pages. https://doi.org/10.1145/3802079

1 Introduction

Inverted indexes (posting lists) are the backbone of large-scale information retrieval [63] and database systems [46]. A posting list stores a *sorted* sequence of document identifiers (docIDs, hereafter *keys*) for a term, typically in the form of *integers*. Given the vast number of postings

*Corresponding author.

Authors' Contact Information: Xianyu Zhu, Renmin University of China, Beijing, China, xianyuzhu@ruc.edu.cn; Qiyu Liu, Southwest University, Chongqing, China, qyliu.cs@gmail.com; Guangyi Zhang, Shenzhen Technology University, Shenzhen, China, zhangguangyi@sztu.edu.cn; Zhibing Sha, Southwest University, Chongqing, China, zhibing.sha@gmail.com; Jianwei Liao, Southwest University, Chongqing, China, liaotoad@gmail.com; Sha Hu, Southwest University, Chongqing, China, husha@swu.edu.cn; Lei Chen, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, leichen@cse.ust.hk.



ACM 2836-6573/2026/6-ART202
https://doi.org/10.1145/3802079

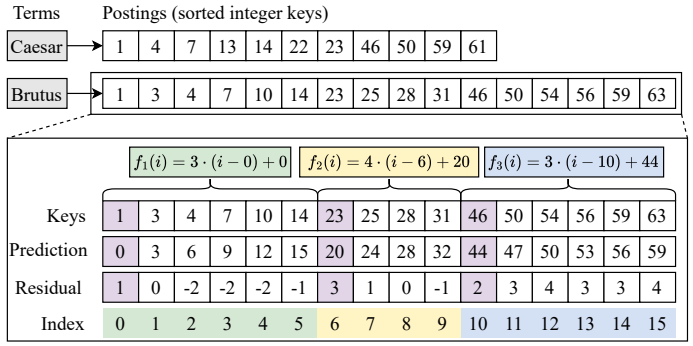


Fig. 1. Lossless learned compression of a sorted key list using a piecewise linear approximation model (PLA). A 3-segment model $f : i \mapsto k_i$ compresses keys with maximum error $\epsilon = 4$; each residual requires $\lceil \log_2(2\epsilon + 1) \rceil = 4$ bits.

that modern search engines and analytical systems can host (in trillions), compression is indispensable: it reduces memory footprint, improves cache locality, and minimizes storage bandwidth requirements [32, 41, 43, 45, 48].

Conventional Index Compression Schemes. A practical compressor for posting lists must satisfy two requirements: **(1) compactness**—store sorted integers in as few bits as possible without loss of information; and **(2) efficiency**—support efficient random access, sequential scans, and list operations such as intersection and union. Decades of research and engineering have produced highly optimized codecs deployed in production systems such as Apache Lucene [1], including delta and bit-packing schemes [8, 26, 60, 64], VByte variants [27, 44], and entropy encoding methods [22, 35, 37]. These approaches typically encode small blocks of gaps between the keys with fixed-width or patched variable-length codes, often optimized for SIMD/GPUs [26] or even custom hardware [21]. A recent experimental survey [45] discusses this design space in detail.

From Handcrafted to Learned Data Compression. It has been shown that lightweight machine learning (ML) models can outperform hand-tuned structures by modeling data distributions directly [11, 13, 17, 18, 24, 30, 31, 33]. Motivated by this shift, recent works like LA-vector [9] and LeCo [34] have explored compression of integer lists using learned models [9]. Given a sorted list $\mathcal{K} = \{k_1, \ldots, k_n\}$, a learned compressor fits a function $f : i \mapsto k_i$, and represents $\mathcal{K}$ as (f, Δ), where $\Delta[i] = k_i - \text{round}(f(i))$ is the residual between the true key and the model prediction. The maximum absolute error $|\Delta[i]|$ is bounded by ϵ. As $|\Delta[i]| \leq \epsilon$, each residual then requires only $\lceil \log_2(2\epsilon+1) \rceil$ bits to store. The motivation behind learning-based compression is that integer distributions in real posting lists are often structurally "simple" (see Section 7.1 for more details), allowing even lightweight models (e.g., piecewise linear models) to approximate the mapping $f : i \mapsto k_i$ with small, bounded errors.

Running Example. Figure 1 illustrates lossless compression of 16 integer keys using a three-segment (f_1, f_2, f_3) piecewise linear approximation (PLA) with $\epsilon = 4$. Each segment predicts k_i from index i in its corresponding domain, and the array $\Delta = \{1, 0, -2, \cdots, 4\}$ stores the residuals for reconstruction, where each residual is encoded in 4 bits. To decode a key, it first evaluates the corresponding segment function and then adds the stored residual to obtain the exact original value.

Limitations of Current Learned Compression Methods. Learned compression is still in its infancy, especially when compared to learned indexing [11, 13, 17, 18, 24]. LA-vector [9] first introduces the concept of "learned compression" and emphasizes theoretical analysis, but neglects practical low-level optimizations, making it unable to outperform mature conventional schemes (see Figure 2). More recently, LeCo [34] proposes a general-purpose learned integer compressor,

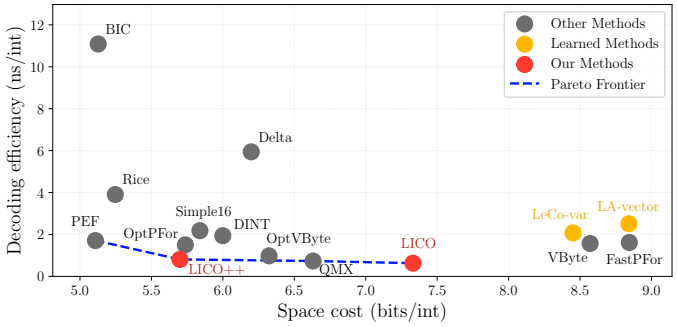


Fig. 2. Trade-offs between space and time tested on real inverted index compression benchmarks [45]. LICO represents our base learned compressor, while LICO++ extends it with enhanced residual encoding. Note that both the x- and y-axes are the smaller, the better.

but it is not tailored to inverted indexes and lacks theoretical guarantees on the compression ratio. Moreover, existing works fail to effectively analyze the relationship between the compression ratio and the error bound ϵ in learned compressors, forcing practitioners to rely on either trial-and-error or heuristic approaches for hyper-parameter configuration.

To advance the line of work on learned compression, we aim to address three fundamental research questions:

RQ1: How can we accurately analyze the storage cost of learned compression for a given model family (e.g., PLAs)?

RQ2: How can the error bound (hyper-parameter) ϵ be automatically configured to maximize compression ratio?

RQ3: How can learned compression be implemented and deployed to meet the latency and throughput demands of practical inverted-index workloads?

Our Solution. To address the aforementioned challenges, we present LICO, an efficient learned inverted index compression framework. LICO comprises two main components: an ML-based compressor for sorted integer lists and a query processing engine.

For **compression**, we develop a rigorous space cost model and identify the key data feature: the variance of key gaps ($g_i = k_i - k_{i-1}$) is the intrinsic data feature that determines the achievable compression ratio. Building on this insight, we formulate the hyper-parameter configuration (for ϵ) as a data partitioning problem that aims to minimize the total space cost, and we propose a linear-time algorithm with provable optimality guarantees.

For **query processing**, LICO supports common operations for inverted indexes, including random access, complete decoding, list intersection, and list union. To ensure high performance on compressed inverted indexes, we design an SIMD-friendly memory layout that maximizes utilization of the hierarchical cache system and fully exploits the parallelism of modern hardware.

In summary, our contributions are threefold:

C1: Theoretical Insights. We develop a space cost model showing that LICO can compress n sorted integer using $0.5n \log_2 \sigma^2 + O(n)$ bits, where σ^2 is the variance of the key gaps, which closely matches the information-theoretic lower bound [42].

C2: Software-Hardware Co-design. We introduce a principled auto-configuration strategy of hyper-parameter ϵ, and query processing algorithms coupled with a cache- and SIMD-friendly memory layout to achieve high query throughput.

C3: Empirical Performance. Through extensive benchmarks on web-scale datasets (>10 billion keys), we show that LICO consistently achieves the Pareto-optimal space-time trade-off among

Fitting Method	Building Time (ms)	Segment Count	Space Cost (bits/int)	Parallelism
OptimalPLA	145,581	133,473,848	10.09	√
GreedyPLA	128,906	165,554,499	10.59	√
SwingPLA	111,559	218,964,753	11.43	√
Spline	65,769	379,054,813	12.19	×

Table 1. Performance of linear models fitted under $\epsilon = 64$ on **CW12B**. OptimalPLA, GreedyPLA, and SwingPLA are ϵ-bounded PLA methods [16, 59]; Spline is a linear-spline baseline fitted using the Greedy Spline Corridor algorithm [38].

highly optimized list compressors (see Figure 2), including the popular delta-based and entropy-based codecs.

Roadmap. Section 2 introduces the background and objectives. Section 3 presents an overview of the LICO framework. Section 4 develops the space cost model and proposes strategies for automatic ϵ configuration. Section 5 details the entire compression pipeline. Section 6 discusses query processing techniques on LICO's compressed format. Section 7 reports the experimental study. Finally, Section 8 reviews related work and Section 9 concludes the paper.

2 Background and Objectives

In this section, we present the foundations of inverted index compression and formulate the problem of learned compression.

2.1 Inverted Index Compression

Compressing an inverted index is equivalent to encoding a sorted integer list. A useful theoretical tool for evaluating list compressors is the **information-theoretic lower bound** (ITLB) [42, 45]. The ITLB states that for n sorted integers drawn uniformly at random from a universe of size $\Gamma \geq n$, the minimum number of bits required to represent the list is:

$$B(\Gamma, n) = \left\lceil \log_2 \binom{\Gamma}{n} \right\rceil \approx n \cdot \left(1.443 + \log_2(\Gamma/n)\right). \tag{1}$$

Any list compressor is called **succinct** if it achieves a total size of $B(\Gamma, n) + o\big(B(\Gamma, n)\big)$ bits, and **quasi-succinct** if it achieves $B(\Gamma, n) + O(n)$ bits. We will show later that the learned compression scheme employed by LICO is quasi-succinct.

2.2 Learned Inverted List Compression

Given a sorted integer list $\mathcal{K} = \{k_1, \ldots, k_n\}$ and an error bound ϵ, the objective of learned compression is to construct a compact model $f : i \mapsto k_i$ that approximates the mapping from index space ($\mathcal{I} = \{1, 2, \cdots, n\}$) to key space $\mathcal{K}$, subject to the constraint:

$$|\text{round}(f(i)) - k_i| \leq \epsilon, \quad \forall i \in \{1, \ldots, n\}, \tag{2}$$

where the error bound ϵ guarantees lossless recovery. Intuitively, learning $f(\cdot)$ is equivalent to learning the inverse cumulative distribution function (ICDF, or quantile function) of $\mathcal{K}$ scaled by n.

In latency-sensitive scenarios such as search engines, the overhead of deep learning runtimes (e.g., PyTorch) can be prohibitive. To balance compression ratio and decoding speed, existing works usually avoid heavyweight neural networks in favor of lightweight models that train and infer quickly while retaining sufficient predictive accuracy. In particular, error-bounded piecewise linear models (ϵ-PLA) are popular choices [17, 18] due to their effectiveness and structural simplicity. Table 1 offers a brief preview of how different fitting methods perform for learned data compression,

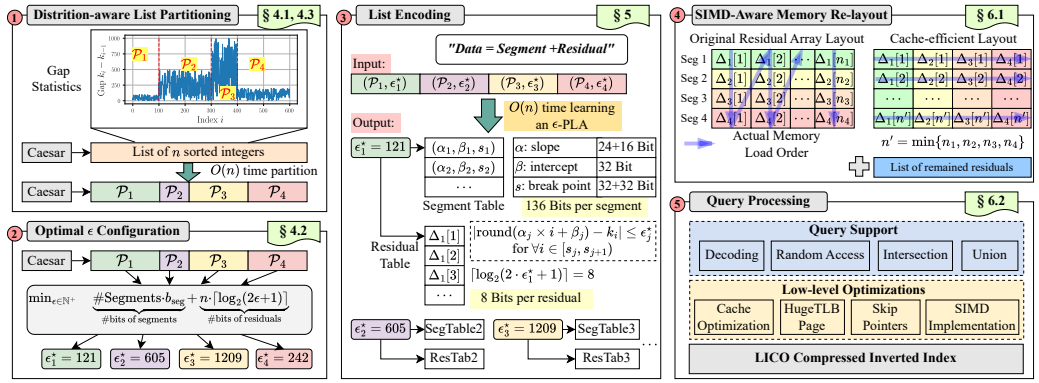


Fig. 3. Architecture of the LICO framework. LICO consists of two key components: an auto-configured list compressor (**Steps ❶❷❸**) and a high-performance query processing engine (**Steps ❹❺**).

and the OptimalPLA [59] algorithm is chosen as the default segment learner in our design for its optimal space cost and support for parallel decoding (see Section 6.2 and Section 7.7 for details).

Definition 2.1 (ϵ-PLA for Compression). Given a list of n sorted integers $\mathcal{K} = \{k_1, \ldots, k_n\}$ and an error bound ϵ, an ϵ-PLA with L segments to the point set $\{(i, k_i)\}_{i=1}^{n}$ is a piecewise linear function

$$f(i) = \begin{cases} \alpha_1 \times (i - s_1) + \beta_1 & i \in [s_1, s_2) \\ \alpha_2 \times (i - s_2) + \beta_2 & i \in [s_2, s_3) \\ \vdots & \vdots \\ \alpha_L \times (i - s_L) + \beta_L & i \in [s_L, +\infty) \end{cases} \tag{3}$$

such that $|\text{round}(f(i)) - k_i| \leq \epsilon$ for all i.

When using ϵ-PLA to represent $\mathcal{K}$, the total storage cost in bits consists of two components - the model parameters and the residuals - which can be expressed as:

$$\#\text{Bits} = L(\mathcal{K} \mid \epsilon) \cdot b_{\text{seg}} + n \cdot \lceil \log_2(2\epsilon + 1) \rceil, \tag{4}$$

where $L(\mathcal{K} \mid \epsilon)$ is the number of segments required to satisfy the error bound, and b_{seg} is the number of bits to encode each linear segment *(break point, slope, intercept)*. In our implementation, $b_{\text{seg}} = 136$ (see Section 5 for more details).

2.3 Design Objectives

From Equation (4), a fundamental trade-off arises: a smaller ϵ reduces the bits per residual but increases the number of segments. Prior work on learned compression [9] has largely focused on theoretical insights, leaving open the question of how to configure the optimal error bound in practice. Moreover, existing learned compressors provide only limited support for query processing directly on compressed lists (e.g., intersection, union) and overlook low-level implementation optimizations, leading to empirical inferiority to mature list compressors such as OptPForDelta [45] and Elias-Fano encoding [54].

In this work, we demonstrate that by addressing these technical challenges, learned compression achieves both **theoretical rigor** through principled compression scheme design and **practical efficiency** through systems-level engineering, making it a promising alternative to state-of-the-art inverted index compressors.

3 LICO Workflow

Figure 3 illustrates the LICO framework, which comprises two major components: a principled machine-learned list compressor (**Steps ❶-❸**) and a query engine that operates directly on the compressed format (**Steps ❹-❺**).

Step ❶: Data-aware Partitioning. We first prove that the effectiveness of piecewise linear approximation for compression is determined only by the variance of key gaps (i.e., the differences between consecutive keys). Empirical evidence further shows that exploiting the intrinsic data locality can substantially enhance compression efficacy. Guided by these insights, LICO employs a linear-time algorithm to partition posting lists into disjoint blocks according to gap variance, thereby aligning compression performance with intrinsic data characteristics (see Section 4.1 and Section 4.3).

Step ❷: Optimal Error Bound Configuration. For each partitioned block $\mathcal{P} \subseteq \mathcal{K}$, we model the expected segment count in Equation (4) as $L(\mathcal{P} \mid \epsilon) = C \cdot |\mathcal{P}| \cdot \sigma^2/\epsilon^2$, where C is a constant that can be estimated empirically and σ^2 represents the variance of gaps within block $\mathcal{P}$. Based on this model, we derive the optimal error bound that minimizes total space cost as $\epsilon^\star = \sqrt{2 \ln 2 \cdot C \cdot b_{\text{seg}} \cdot \sigma^2}$. We further prove that under this configuration, LICO achieves quasi-succinctness, closely matching the ITLB (see Section 4.2).

Step ❸: List Encoding. Given the partitioned blocks and optimal error bound configurations $\{(\mathcal{P}_1, \epsilon_1^\star), \ldots, (\mathcal{P}_m, \epsilon_m^\star)\}$, each block $\mathcal{P}_\ell$ is encoded using an ϵ-PLA with $\epsilon = \epsilon_\ell^\star$, learned via a parallelizable linear-time PLA fitting algorithm [40]. Each block is then represented as a tuple $(\epsilon_\ell^\star, f_\ell, \Delta_\ell)$, where f_ℓ denotes the learned PLA model and $\Delta_\ell = \{\text{round}(f_\ell(i)) - k_i \mid i \in [1, |\mathcal{P}_\ell|]\}$ is the residual array. Residuals are encoded in a compact bit array, and each segment of f_ℓ is stored as a 136-bit tuple (see Section 5).

Step ❹: Memory Re-Layout. To support efficient query execution on modern architectures, LICO reorganizes the compressed representation into cache-friendly memory layouts. Specifically, segment metadata, error bounds, and residual arrays are arranged in contiguous blocks based on accessing order to maximize spatial locality and minimize pointer chasing. This design ensures that subsequent query processing can benefit from high cache-line utilization and reduced memory stalls (see Section 6.1).

Step ❺: Query Processing. LICO's query engine supports random access, full decoding, list intersections, and unions directly over the compressed format. We implement SIMD-based list operators, equipped with optimizations such as skip pointers, incremental model prediction, and huge-page allocation to accelerate query processing. Together, these optimizations enable low-latency query execution while preserving the compression benefits of LICO, allowing LICO to achieve superior decoding efficiency compared with highly optimized SIMD-based compressors (see Section 6.2).

4 Data-Aware Error Bound Configuration

In this section, we analytically model the storage cost of learned compression and derive the optimal error bound ϵ to minimize the space cost. We further introduce a linear-time algorithm that adapts ϵ to local data variance with theoretical guarantees.

4.1 Estimation of Required Segment Count

As indicated by Equation (4), the space cost is a function of the number of segments $L(\mathcal{K} \mid \epsilon)$, which is given in Theorem 4.1. We extend the theoretical analysis of [9] to derive a closed form for $L(\mathcal{K} \mid \epsilon)$ from dataset statistics.

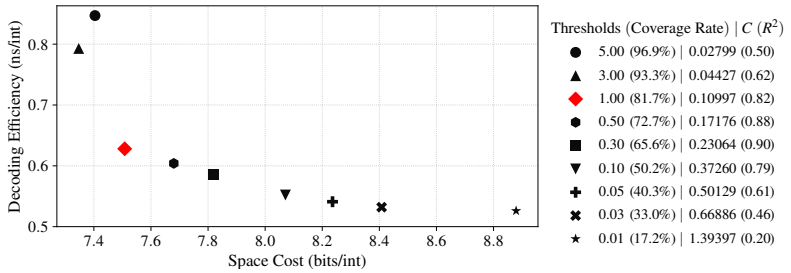


Fig. 4. Space-time trade-off of LICO under varying C on **CW12B**. Given a certain threshold T, all samples whose $\sigma/\epsilon \le T$ are taken into the fitting of C. Red marks the chosen C.

Theorem 4.1 (Expected Segment Count [9]). *Given a list of n sorted integers $\mathcal{K} = \{k_1, \cdots, k_n\}$ and an error bound $\epsilon \in \mathbb{N}^+$ with $\epsilon \gg \sigma$, the expected number of segments $L(\mathcal{K} \mid \epsilon)$ needed to fit an ϵ-PLA on $\{(i, k_i)\}_{i=1}^n$ is $L(\mathcal{K} \mid \epsilon) \propto n\sigma^2/\epsilon^2$, where σ^2 is the variance of key gaps (i.e., $k_i - k_{i-1}$).*

Without loss of generality, we write $L(\mathcal{K} \mid \epsilon) = C \cdot n \cdot \sigma^2/\epsilon^2$, where C is a coefficient to be learned. To estimate C, we randomly sample 5,355 subsets of consecutive keys from real posting lists, physically construct ϵ-PLAs for each sample with $\epsilon \in \{2^2, \ldots, 2^{12}\}$, and record the resulting segment counts. We then apply a threshold $T \in [0.01, 5]$ such that all sampled lists with $\sigma/\epsilon \le T$ are taken to fit C using least square. As Figure 4 shows, when the fitting quality is acceptable ($R^2 \ge 0.6$), smaller C reduces space cost but increases segment count and lowers decoding efficiency; when $R^2 < 0.6$, the fitted C becomes unreliable and may increase space overhead. Based on this trade-off, we select $C = 0.10997$ ($R^2 = 0.82$), which improves decoding speed by 27% with only a 2% space overhead compared to the space-optimal C (see Appendix E of [62] for more details).

4.2 Optimal Error Bound Configuration

Based on Theorem 4.1, Equation (4) can be rewritten as:

$$\#\text{Bits} = C \cdot b_{\text{seg}} \cdot \frac{n\sigma^2}{\epsilon^2} + n \cdot \lceil \log_2(2\epsilon + 1) \rceil, \tag{5}$$

which allows us to derive the optimal configuration of ϵ.

Theorem 4.2 (Optimal Error Bound). *For a sorted list $\mathcal{K}$ with gap variance σ^2, the optimal setting of ϵ to minimize the storage is:*

$$\epsilon^\star = \sqrt{\frac{2 \ln 2 \cdot C \cdot b_{\text{seg}} \cdot n \cdot \sigma^2}{n+1}} \approx \sqrt{2 \ln 2 \cdot C \cdot b_{\text{seg}} \cdot \sigma^2}, \tag{6}$$

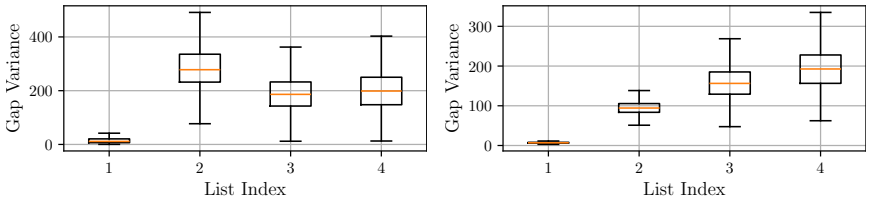
and the corresponding minimum total bits to encode $\mathcal{K}$ is:

$$S(\mathcal{K} \mid \epsilon^\star) \approx n \cdot \left(1.957 + \frac{1}{2} \log_2 \left(C \cdot b_{\text{seg}} \cdot \sigma^2\right)\right). \tag{7}$$

Proof. See Appendix B of [62] for more details. □

Error Scaling. Notably, the results in Theorem 4.2 are derived by relaxing the residual storage term in Equation (5) into a continuous form. This simplification introduces at most a 1-bit error in the encoding cost per residual. In practice, for any $\epsilon^\star$ that falls within the range $[2^k, 2^{k+1})$, we can scale it up to the nearest power of two minus one. This adjustment minimizes the number of required line segments without affecting the bits per residual (see Section 5.1).

Quasi-Succinctness. We then show that the optimal configuration in Theorem 4.2 leads to a quasi-succinct compressor. Consider n uniformly distributed keys $K_1, \ldots, K_n$ with $K_i \sim \mathcal{U}(0, \Gamma)$.



(a) CW12B (b) CCNews

Fig. 5. Block-wise gap-variance distributions within posting lists (1K keys per block). Tall box plots (e.g., Lists 2—4 in CW12B) indicate larger variance differences across blocks, i.e., stronger locality.

Let $G_i = K_{(i)} - K_{(i-1)}$ be the i-th gap, where $K_{(i)}$ is the i-th order statistics. According to [12], G_i follows a scaled Beta distribution $\Gamma \cdot \text{Beta}(1, n)$ with variance:

$$\text{Var}[G_i] = \frac{\Gamma^2 \cdot n}{(n+1)^2(n+2)} \approx \Gamma^2/n^2. \tag{8}$$

Substituting Equation (8) into Equation (7) yields:

$$n \cdot \left(1.957 + 0.5 \log_2(C \cdot b_{\text{seg}}) + \log_2(\Gamma/n)\right), \tag{9}$$

which simplifies to $B(\Gamma, n) + O(n)$, satisfying the definition of quasi-succinctness (see Equation (1)). It is worth noting that the ITLB [42, 45] characterizes the compression limit under the worst-case assumption of perfectly uniform data. In practice, a list compressor can achieve fewer bits than the ITLB suggests, since real-world datasets often exhibit strong locality. In the subsequent sections, we will show that LICO can achieve better performance by exploiting the local distributional patterns.

Summary of Theoretical Results. Compared with the main theoretical result for *learned indexes* [17] (i.e., for indexing, the segment count L follows $L \propto n\sigma^2/(\mu\epsilon)^2$), our theoretical contributions address the learned compression setting and differ in two key aspects. (1) Theorem 4.1 extends previous results [9, 17], showing that the segment count for *learned compression* depends on the gap variance and is independent of the gap mean, yielding $L \propto n\sigma^2/\epsilon^2$. (2) Theorem 4.2 further shows that the total space cost of a learned compressor using PLA is a convex function of ϵ, which enables selecting an optimal ϵ and leads to quasi-succinctness.

4.3 Data-Aware List Partitioning

Theorem 4.2 highlights a key insight: *the optimal error bound and the minimum space cost depend* ***solely*** *on the* ***variance*** *of key gaps.* As Figure 5 illustrates, the gap distribution of real-world datasets exhibits block-wise heterogeneity, reflecting the strong locality. This observation motivates partitioning a long posting list into multiple blocks with homogeneous gap variance, which enables finer-grained error configuration.

Problem Formulation and Optimal DP Solution. Formally, let $i_1, i_2, \cdots, i_{m+1}$ ($i_1 = 1$ and $i_{m+1} = n$) denote the boundaries for partitioning n sorted keys into m consecutive, disjoint blocks $\mathcal{P}_1, \cdots, \mathcal{P}_m$ where $\mathcal{P}_\ell = \{k_i \mid i \in [i_\ell, i_{\ell+1}]\}$. For block $\mathcal{P}_\ell$ with size n_ℓ and gap variance σ_ℓ^2, the optimal error bound $\epsilon_\ell^\star \approx 4.553 + \sigma_\ell$ and the corresponding minimized space cost is $S(\mathcal{P}_\ell \mid \epsilon_\ell^\star) = n_j \cdot (3.908 + \log_2 \sigma_\ell)$, by setting $C = 0.10997$ and $b_{\text{seg}} = 136$ (see Section 4.1) according to Theorem 4.2. The partitioned compression problem can then be formulated as:

$$\min_{\mathcal{P}_1,\ldots,\mathcal{P}_m} \sum_{\ell=1}^{m} S(\mathcal{P}_\ell \mid \epsilon_\ell^\star). \tag{10}$$

Equation (10) can be optimally solved by a dynamic programming (DP) algorithm based on the following recurrence:

$$\mathrm{OPT}[j][\ell] = \min_{i<j}\{\mathrm{OPT}[i][\ell-1] + S(\mathcal{K}_{i:j} \mid \epsilon^{\star}_{i:j})\}, \quad \text{and} \tag{11}$$

$$\mathrm{OPT}[0][0] = 0, \tag{12}$$

where $\mathrm{OPT}[j][k]$ is the minimum total bits achieved by partitioning the first j keys into ℓ block, $\mathcal{K}_{i:j} = \{k_i, \cdots, k_j\}$, and $\epsilon^{\star}_{i:j}$ is the corresponding optimal error bound configured by Theorem 4.2. To obtain a solution from $\mathrm{OPT}[n][m]$, such a DP algorithm (see Appendix C of [62]) incurs $O(mn^2)$ time and $O(mn)$ space complexity, whose time complexity is prohibitive for billion-scale posting lists.

Linear-time Greedy Algorithm. To scale to large datasets, we further develop a linear-time greedy algorithm, GreedyPartition (Algorithm 1), which iteratively splits the block with the highest space cost into two sub-blocks at the break point that yields the optimal cost reduction. The algorithm runs in $O(m(n + \log m))$ time, as it performs at most $m - 1$ scans of $\mathcal{K}$ and each iteration requires a heap operation in $O(\log m)$. Notably, gap variances can be computed on the fly using prefix sums, leading to $O(1)$ amortized time and space per evaluation. As m is often small in practice, Algorithm 1 is effectively linear in n.

We then show that such a greedy strategy yields a non-trivial constant approximation ratio.

Theorem 4.3 (Approximation Ratio). *Let OPT and ALG denote the total space costs produced by the DP algorithm and the greedy algorithm, respectively. When n is sufficiently large, the greedy algorithm achieves a constant-factor approximation:*

$$\frac{ALG}{OPT} = \frac{m^g}{m} = O(1), \tag{13}$$

where $m^g \le m$ denotes the practical number of partitions generated by the greedy algorithm.

Proof. See Appendix D of [62] for more details. □

Discussion on Partition Number. Algorithm 1 takes the maximum number of partitions m as input but may produce fewer than m blocks if further splitting does not reduce the total space cost. In practice, m can be safely set to a relatively large value (e.g., 2048), allowing the algorithm to automatically determine the appropriate number of partitions. To prevent over-splitting, we further perform error scaling and block merging on the partitioned blocks (see Section 5.1 for more details). Notably, although a larger m increases the worst-case overhead, in practice, Algorithm 1 often terminates early after a few iterations, making the actual runtime negligible, especially compared to the DP algorithm. See Section 7.3 for a detailed evaluation of the impact of m.

Implementation Details. Algorithm 1 requires computing the gap variance to estimate the partition cost (lines 7 and 8). However, sample variance becomes unreliable when the sample size is small, meaning that a partition containing only a few keys may lead to inaccurate cost estimation. To address this, we define the basic partition unit as a fixed-size page of keys, rather than individual keys. Therefore, index i refers to the i-th page instead of the i-th key in the context of list partitioning. Intuitively, the page size should neither be too small nor too large. In our experiments, we vary the page size within the range $[8, 8192]$ and find that setting it to 1K provides a good balance between accuracy and efficiency in practice (see Section 7.3 for details).

5 LICO Compression Workflow

In this section, we detail the complete compression workflow (Algorithm 2) of LICO and introduce its variant LICO++, which majorly differs in the residual encoding strategy.

Algorithm 1 GreedyPartition

Require: List $\mathcal{K}$, maximum partition number m
Ensure: A feasible partition
Initialize an empty max-heap Q
Push $(\mathcal{K}_{1:n}, \epsilon^\star_{1:n})$ with cost $S(\mathcal{K}_{1:n} \mid \epsilon^\star_{1:n})$ to Q
while $|Q| < m$ **do**
  Pop $(\mathcal{K}_{s:t}, \epsilon^\star_{s:t}, S^\star_{s:t})$ with the highest cost from Q
  $i^\star \leftarrow s$, $s^\star \leftarrow S^\star_{s:t}$
  for $i \leftarrow s+1$ **to** $t-1$ **do**
    $S^\star_{s:i} \leftarrow n_{s:i} \cdot \left(3.908 + \log_2 \sigma^2_{s:i}\right)$ ▷ Equation (7)
    $S^\star_{i+1:t} \leftarrow n_{i+1:t} \cdot \left(3.908 + \log_2 \sigma^2_{i+1:t}\right)$ ▷ Equation (7)
    $s \leftarrow S^\star_{s:i} + S^\star_{i+1:t}$ ▷ total cost when splitting at i
    if $s < s^\star$ **then**
      $(i^\star, s^\star) \leftarrow (i, s)$ ▷ greedy split if the total cost is reduced
    end if
  end for
  Push $(\mathcal{K}_{s:i^\star}, \epsilon^\star_{s:i^\star})$ with cost $S^\star_{s:i^\star}$ to Q
  Push $(\mathcal{K}_{i^\star+1:t}, \epsilon^\star_{i^\star+1:t})$ with cost $S^\star_{i^\star+1:t}$ to Q
end while
return Q

5.1 Error Scaling and Block Merging

Recall from Section 4.2 that each optimal $\epsilon^\star$ can be safely scaled to the nearest power of two minus one, which potentially reduces the number of required segments without altering the number of residual bits. In the best case, when $\epsilon^\star = 2^k$, it is scaled to $2^{k+1} - 1$. Since the segment count follows $L \propto n\sigma^2/\epsilon^2$ (Theorem 4.1), this scaling can theoretically reduce the number of segments by up to a factor of four. Furthermore, error scaling substantially decreases the number of distinct ϵ values, thus increasing the likelihood that adjacent partitioned blocks share the same ϵ. Such blocks are subsequently merged to further reduce storage overhead, as each block must otherwise store its own ϵ value. Figure 6 illustrates the processes of error scaling and block merging.

Impact to Cost Model. Notably, although error scaling and block merging further reduce the overall space cost, they introduce a slight deviation from the theoretical model. However, as discussed in Section 4.2, this deviation can be strictly bounded within one bit per integer, ensuring that the model remains a reliable and accurate cost estimator for parameter optimization. Empirically, as shown in Section 7.3, our GreedyPartition algorithm achieves a space cost within 3% of the theoretical optimum.

5.2 Segment Fitting and Encoding

Segment Fitting. For each block $\mathcal{P}_\ell$ with configured error bound ϵ_ℓ, we apply ParaOptimal [40], a linear-time online ϵ-PLA fitting algorithm for monotone datasets that minimizes the number of

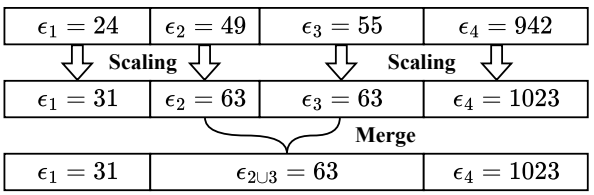


Fig. 6. Illustration of error scaling. After scaling, consecutive blocks with the same ϵ are merged into a single block.

Algorithm 2 ListCompression

Require: List $\mathcal{K}$, maximum partition number m
Ensure: Compressed representation of $\mathcal{K}$
1: Init bit arrays S, Δ to store segments and residuals
2: $(\mathcal{P}, \mathcal{E}) \leftarrow \text{GreedyPartition}(\mathcal{K}, m)$ ▷ invoke Algorithm 1
3: $(\mathcal{P}', \mathcal{E}') \leftarrow \text{ErrorScaling}(\mathcal{P}, \mathcal{E})$ ▷ ϵ scaling and block merging
4: **for** $\ell = 1, \cdots, |\mathcal{P}'|$ **do**
5: $\quad \mathcal{S}_\ell \leftarrow \text{PLAFit}(\mathcal{P}'_\ell, \mathcal{E}'_\ell)$ ▷ fit an ϵ-PLA using algo. in [40]
6: $\quad \Delta_i \leftarrow \text{PLAErrEval}(\mathcal{S}_\ell)$ ▷ compute residuals for keys in $\mathcal{P}'_\ell$
7: $\quad$ Encode $\mathcal{S}_\ell$ and store to S ▷ segment encoding
8: $\quad$ Encode Δ_ℓ and store to Δ ▷ residual encoding
9: **end for**
10: **return** (S, Δ)

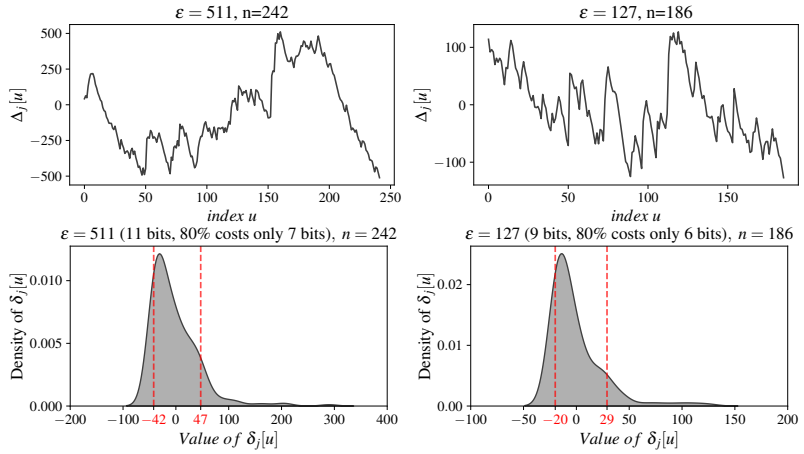


Fig. 7. Distribution of residual gaps ($\delta_j[u] = \Delta_j[u] - \Delta_j[u-1]$) for two typical segments fitted on **CW12B**.

line segments. This algorithm has also been commonly employed in PLA-based learned indexes such as the PGM-Index [18] and learned compressors such as LA-vector [9].

Segment Encoding. ParaOptimal represents each line segment in an ϵ-PLA as the diagonal of a bounding box, which is defined by its bottom-left corner $(x^{\text{lo}}, y^{\text{lo}})$ and top-right corner $(x^{\text{hi}}, y^{\text{hi}})$. Since both the input and output of learned compression are integers, the resulting coordinates $x^{\text{lo}}, y^{\text{lo}}, x^{\text{hi}}, y^{\text{hi}}$ are also guaranteed to be integers. Thus, each segment can be compactly encoded as a 5-tuple of integers $(\Delta y, \Delta x, y^{\text{lo}}, x^{\text{lo}}, s)$, where s is the starting index of this segment, and $\Delta y = y^{\text{hi}} - y^{\text{lo}}$, and $\Delta x = x^{\text{hi}} - x^{\text{lo}}$.

Recall that each segment of an ϵ-PLA is represented as $f(i) = \alpha \cdot (i - s) + \beta$, which can be rewritten as:

$$\begin{aligned} f(i) &= \frac{\Delta y}{\Delta x} \cdot (i - s) + \frac{\Delta y}{\Delta x} \cdot (s - x^{\text{lo}}) + y^{\text{lo}} \\ &= \frac{\Delta y}{\Delta x} \cdot (i - x^{\text{lo}}) + y^{\text{lo}}. \end{aligned} \tag{14}$$

However, directly computing Equation (14) may lead to numerical issues and potentially violate the error bound ϵ. To address this, we compute $f(i)$ using integer arithmetic as follows:

$$\tilde{f}(i) = \left\lfloor \frac{\Delta y \cdot (i - x^{\text{lo}}) + \text{Sign}(i - x^{\text{lo}}) \cdot \frac{\Delta x}{2}}{\Delta x} \right\rfloor + y^{\text{lo}}, \tag{15}$$

where $\lfloor \cdot / \cdot \rfloor$ denotes integer division. It can be shown that $|f(i) - \tilde{f}(i)| \leq \frac{1}{2}$, meaning that $\tilde{f}(i)$ is the nearest integer to $f(i)$. Since the optimal ϵ-PLA fitting algorithm guarantees that $|f(i) - k_i| \leq \epsilon$, it always holds that:

$$|\tilde{f}(i) - k_i| \leq |\tilde{f}(i) - f(i)| + |f(i) - k_i| \leq \epsilon + \frac{1}{2}, \tag{16}$$

thus preserving the error bound as both ϵ and $\tilde{f}(i) - k_i$ are integers.

Physical Storage. The values Δy, Δx, y^{lo}, x^{lo}, and s share the same data type as the input keys. When the key type is uint32_t, each segment occupies $32 \times 5 = 160$ bits. However, Δy and Δx typically fall well below the maximum range of $2^{32} - 1$, allowing the use of more compact binary representations to store. For practical web-scale inverted index applications [45], it is sufficient to encode Δy and Δx using 24-bit and 16-bit integers, respectively, resulting in a total segment size of $b_{\text{seg}} = 136$ bits.

5.3 Residuals Encoding

For the j-th segment $(\Delta y_j, \Delta x_j, y_j^{\text{lo}}, x_j^{\text{lo}}, s_j)$ of an ϵ-PLA model covering indexes $i \in [s_j, s_{j+1})$, we define the corresponding residual array as Δ_j ($|\Delta_j| = s_{j+1} - s_j$), where each residual $\Delta_j[u]$ represents the deviation between the predicted value and the true key:

$$\Delta_j[u] = k_{u+s_j} - \tilde{f}(u + s_j), \quad \text{for } u \in [0, s_{j+1} - s_j). \tag{17}$$

Since $-\epsilon \leq \Delta_j[u] \leq \epsilon$, LICO directly encodes each residual compactly by storing the sign in a 1-bit vector and the magnitude in a $\lceil \log_2(\epsilon + 1) \rceil$-bit vector. For example, in a block of $\epsilon = 127$, each $\Delta_j[u]$ requires 1 sign bit and 7 magnitude bits.

LICO++: Further Compression on Residuals. In most cases, residual arrays contribute the majority (over 70%) of the total compressed data size. As shown in Figure 7, residuals in real datasets exhibit strong locality. By analyzing the distribution of the differences between consecutive residuals, i.e., $\delta_j[u] = \Delta_j[u] - \Delta_j[u-1]$, we find that over 80% of $\delta_j[u]$ values lie within only 5~10% of its full range (i.e., $[-2\epsilon, 2\epsilon]$), revealing the opportunity for further compression. Moreover, the local patterns observed in Figure 7 arise fundamentally from the behavior of existing PLA fitting algorithms like ParaOptimial [40]. These algorithms typically add points to a temporary segment in an online manner until the maximum error constraint is violated, resulting in local substructures where the absolute fitting error first decreases and then increases.

Based on these findings, LICO++ improves the compression ratio by further compressing the residuals via a delta-based encoding scheme. Notably, since residuals are unsorted, our learned compression methods cannot be applied again. LICO++ contains three steps. **(1) Residual Gap Computation.** For each block $\mathcal{P}_\ell$ fitted with an ϵ_ℓ-PLA containing L segments, LICO++ computes the residual gaps $\delta_j[u]$ ($j \in [1, L]$, $u \in [0, s_{j+1} - s_j)$) as:

$$\delta_j[u] = \begin{cases} \Delta_j[0], & \text{if } u = 0, \\ \Delta_j[u] - \Delta_j[u-1], & \text{if } u \in (0, s_{i+1} - s_i). \end{cases} \tag{18}$$

As segments fitted on block $\mathcal{P}_\ell$ share a common ϵ_ℓ, their delta arrays δ_j can be concatenated into a single contiguous delta array for compression. **(2) Zigzag Transformation.** Delta encoding methods [60, 64] typically require non-negative integer inputs. To ensure this, LICO++ maps each $\delta_j[u]$ to a positive value using the following Zigzag transformation:

$$\text{Zigzag}(\delta_j[u]) = (\delta_j[u] \ll 1) \oplus (\delta_j[u] \gg 31). \tag{19}$$

Unlike offset-based transformation, the Zigzag trick requires no additional metadata and can be efficiently implemented using bitwise intrinsics. It also ensures that small-magnitude residuals are mapped to small positive integers, thus preserving the locality. **(3) Residual Gap Compression.**

Algorithm 3 MemoryRelayout

Require: Residual arrays $\Delta_1, \Delta_2, \ldots, \Delta_L$, SIMD batch size s_{batch}
Ensure: Δ^{simd}, Δ^{tail}
$\Delta^{\text{simd}} \leftarrow \{\}$, $\Delta^{\text{tail}} \leftarrow \{\}$
Sort $\{\Delta_j\}_{j=1,\cdots,L}$ in descending order by length
for $b = 1$ to $\lceil L/s_{\text{batch}} \rceil$ **do**
  Select next s_{batch} arrays as batch $\mathcal{B}$
  $c_{\min} \leftarrow \min_{\Delta \in \mathcal{B}} |\Delta|$
  Truncate each $\Delta \in \mathcal{B}$ to length $c_{\min}$
  Form matrix $M \in \mathbb{N}^{s_{\text{batch}} \times c_{\min}}$ by stacking truncated arrays
  Append rows of transpose M^{T} to Δ^{simd}
  Append remaining residuals $\{\Delta[c_{\min}:] \mid \Delta \in \mathcal{B}\}$ to Δ^{tail}
end for
Append all remaining residual arrays (if any) to Δ^{tail}
return Δ^{simd}, Δ^{tail}

After transformation, the resulting deltas are compressed using the PFor scheme [64] and stored as a bit array. In implementation, LICO++ adopts the SIMD-optimized simd256fastpfor codec from the FastPFor library [5].

LICO v.s. LICO++. Compared to LICO, which directly encodes the residual arrays, LICO++ exploits the distributional characteristics of residuals induced by the PLA fitting process and applies delta-based encoding for further compression. Extensive experiments on real-world datasets show that LICO++ can further improve the compression ratio of LICO by up to 1.29×, with less than 30% additional decoding time. Despite this overhead, LICO++ still outperforms popular compression schemes in overall efficiency. See Section 7.4 for more evaluation details.

5.4 Discussion of LICO's Extension

Handling Dynamic Updates. LICO can be extended with an LSM-tree-style design to support dynamic operations (insertion, update, and deletion). In particular, recent updates are temporarily buffered in a small mutable block and fitted with PLA. Incoming queries first probe this buffer and then fall back to the immutable segments to ensure data freshness. When the buffer reaches its capacity, LICO triggers a compaction-like process that merges buffered updates into the corresponding static segments and refits the affected segments. Deletions do not modify the original segment (via inserting tombstones), insertions possibly incur a segment-level refit, and updates are handled as a deletion followed by an insertion. This design effectively amortizes refitting cost while preserving sorted order and efficient query processing.

Extension to Unsorted Lists. LICO targets sorted lists, where monotonicity enables efficient error-bounded segment fitting (e.g., via OptimalPLA) and compact residuals. For unsorted lists, this structure is absent, so the fitting stage must be modified. Inspired by learned secondary indexing [23], one practical approach is to reorganize an unsorted list into locally ordered blocks (e.g., via partitioning or mapping) and then compress each block using the same LICO pipeline. This extension incurs additional space and time overhead for block metadata and, if original order must be preserved, an order-restoration structure.

6 LICO Query Processing

We next describe LICO's SIMD-friendly memory layout and query processing techniques on the compressed data format.

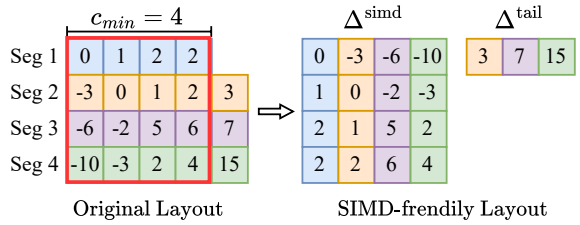


Fig. 8. An example of memory re-layout for the four-way SIMD-based decoding pipeline.

Algorithm 4 NextGeq

Require: Candidate key k^{cand}, segment count L
Ensure: The first key k^{res} satisfying $k^{\text{res}} \geq k^{\text{cand}}$

```
K^skip ← {k_{s_1}, ··· , k_{s_L}}                ▷ get first keys for each segment
j* ← upper_bound(K^skip, k^cand)                 ▷ find segment with k_{s_j} ≥ k^cand
for j ← j* − 1 to L do
    for i ← 1 to s_{j+1} − s_j do
        Decode the i-th key in the j-th segment as k'    ▷ Equation (15)
        if k' ≥ k then
            return k'
        end if
    end for
end for
```

6.1 Memory Re-layout

As introduced before, LICO's query processing engine is designed to fully leverage SIMD units, which offer powerful parallel execution and are widely supported on modern CPUs. However, in the original memory layout (see Figure 8), the SIMD-based decoding pipeline accesses the same offset across multiple segments in each round, leading to inefficiently scattered memory accesses. To mitigate this memory bottleneck, LICO transforms the original memory layout into a cache- and SIMD-friendly layout by truncating segments to the minimum batch length and transposing the residuals into Δ^{simd} for sequential loading during decoding. As specified in Algorithm 3, it first sorts all residual arrays by length, then groups them into batches of size s_{batch}, where s_{batch} is determined by the register width of SIMD (e.g., $s_{\text{batch}} = 8$ for AVX-512). Within each batch, residual arrays are truncated to the minimum batch length c_{min}, stacked into a matrix M, and rows of the transposed matrix M^{T} are appended to Δ^{simd}. The remaining unaligned elements beyond c_{min} are appended to Δ^{tail} for secondary processing (see Section 6.2). This relayout ensures that residuals belonging to the same SIMD lane are stored and accessed contiguously in memory, thereby improving cache locality and parallel decoding efficiency.

Implementation Details. For LICO, the relayout algorithm only needs to be executed once during compression. In contrast, for LICO++, since the residuals undergo an additional round of encoding, they must be decompressed before relayout, making it hard to perform relayout in advance as in LICO. However, as query terms in real-world document retrieval exhibit strong locality [25], for LICO++, we can maintain a buffer that decodes and relayouts the residual arrays corresponding to frequently accessed (hot) lists.

6.2 List Operator Implementation

Random Access. Random access fetches a key at any position in the list, which is naturally supported by LICO. Given a query position i, LICO first locates the segment s whose index interval

covers i (e.g., via binary search). The linear model of s then predicts the estimated key $\tilde{f}(i)$ by Equation (15), and recovers the exact docID through adding the residual: $k_i = \tilde{f}(i) + \Delta[i]$, where $\Delta[i]$ is the residual of position i.

Full List Decoding. LICO simultaneously decodes keys covered by s_{batch} segments, where s_{batch} depends on the width of SIMD registers supported by the system. When computing segment predictions according to Equation (15), LICO adopts an incremental computation strategy to reuse intermediate results from the previous step. For the j-th segment, let $\tilde{g}_j(i) = \Delta y_j \cdot (i - x_j^{\text{lo}}) + \text{Sign}(i - x_j^{\text{lo}}) \cdot \frac{\Delta x_j}{2}$ denote the numerator of the first term in Equation (15). Then, for this segment, the $(i+1)$-th predicted key $p_j[i+1]$ can be computed incrementally by reusing $\tilde{g}(i)$ as follows:

$$\begin{aligned} \tilde{g}_j(i+1) &= \tilde{g}_j(i) + \Delta y_j + \frac{\Delta x_j}{2} \cdot \left(\mathbf{1}_{\{i = x_j^{\text{lo}} - 1\}} + \mathbf{1}_{\{i = x_j^{\text{lo}}\}} \right), \\ p_j[i+1] &= \lfloor \tilde{g}_j(i+1) / \Delta x_j \rfloor + y_j^{\text{lo}}. \end{aligned} \tag{20}$$

All operations in Equation (20) can be fully vectorized and pipelined, eliminating the conditional branching required by direct computation from Equation (15). W.l.o.g., let $s_{\text{batch}} = 8$ for AVX512. At each time, LICO computes 8 predictions in parallel, i.e., $p_j[i], \cdots, p_{j+7}[i]$. The original keys are then reconstructed by adding the corresponding residuals. To align 64-bit intermediate predictions ($\tilde{g}_j(i)$ in Equation (20)) with 32-bit keys, LICO employs an SIMD pipeline consisting of two Decode Units and one Store Unit. As illustrated in Figure 9, the two Decode Units simultaneously add residuals to the predictions at positions i and i+1 across eight segments. Then, 16 scattered 32-bit keys are produced, each requiring a separate memory write. To reduce memory access overhead, LICO rearranges these keys so that two consecutive keys from the same segment become adjacent and packs each pair into a single 64-bit word, halving the number of random memory writes. Notably, the above SIMD computation applies to the truncated and aligned portion (Δ^{simd}) produced by Algorithm 3, while the remaining portion (Δ^{tail}) is processed sequentially.

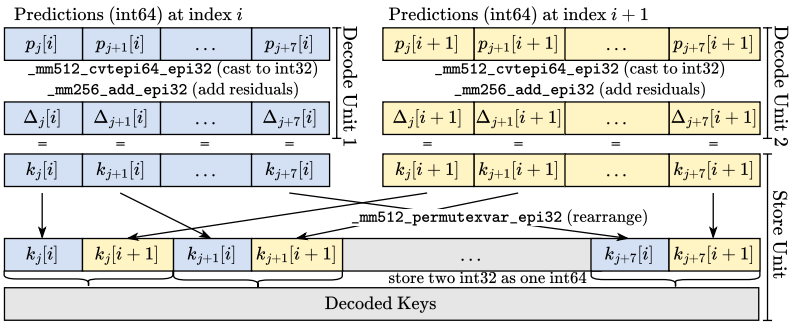


Fig. 9. Overview of the SIMD-based decoding pipeline. Each SIMD iteration executes two Decode Units in parallel followed by one Store Unit.

List Intersection. List intersection retrieves common elements from two posting lists. To support fast intersection, traditional compressors usually employ a NextGeq (Next Greater or Equal) operator to quickly skip non-matching keys [45]. Typical implementations partition a list into blocks and store skip pointers to the maximum key of each block to enable fast pruning. In contrast, LICO natively supports NextGeq without any additional metadata. The reason is that LICO inherently partitions each list into segments, each containing sufficient information to perform the skip operations. Algorithm 4 implements LICO's NextGeq operator.

While NextGeq efficiently filters non-matching elements, each pass of NextGeq produces only one result. To enable parallelism, we extend an SIMD-based list intersection algorithm [53] that

	Vbyte	OptVbyte	BIC	Delta	Rice	PEF	DINT	OptPFor	FastPFor	Simple16	QMX	LA-vector	LeCo-var	LICO	LICO++
Decode	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
Intersection	√	√	√	√	√	√	√	√	×	√	√	×	×	√	√
Union	√	√	√	√	√	√	√	√	×	√	√	×	×	√	√
SIMD	√	√	×	×	×	×	×	×	√	×	√	×	×	√	√

Table 2. Query support of evaluated inverted index compression methods.

treats every 16 keys (for AVX512) as a sliding window. Two windows w_a and w_b slide across the lists to be intersected, skipping non-overlapping regions based on their maximum values. For windows that may overlap, the vp2intersect instruction [10] efficiently identifies matching positions and produces a bitmask for batch output. During sliding, each window skips all elements less than or equal to the other's maximum. This vectorized approach achieves substantially higher throughput than the element-wise NextGeq method. For intersecting more than two lists, we first perform one or two SIMD intersection passes on smaller lists to shrink the intermediate result, then employ NextGeq (Algorithm 4) to rapidly skip irrelevant elements in the remaining list(s).

List Union. The list union operation returns unique keys from multiple posting lists. Unlike decoding and intersection, the main cost of list union stems from key deduplication and result output. Following existing benchmarks [45], LICO first fully decodes the input lists, and then applies an SIMD-based union algorithm [51] that uses bitonic merge and predecessor comparison to deduplicate and produce the result (see Appendix F of [62] for more details).

7 Evaluation and Discussions

In this section, we report the experimental evaluation results of the LICO framework. Specifically, we would like to answer the following questions through the experimental study:

Q1: Does the derived optimal space cost model (Section 4.2) align with empirical observations?
Q2: Can LICO's partition-based automatic error bound configuration strategy effectively capture data locality?
Q3: As an end-to-end framework for compressing sorted integers, does LICO achieve a better space-time trade-off compared with state-of-the-art baselines?
Q4: Can LICO efficiently support common list operators, such as intersection and union?
Q5: Can SIMD significantly improve LICO's decoding efficiency compared to the scalar implementation?
Q6: Can LICO generalize across different fitting methods and synthetic datasets of varying sizes?

7.1 Experimental Setups

Compared Baselines. We implement and compare a comprehensive set of conventional inverted index compressors, including VByte [27], OptVByte [44], BIC [36], Delta [15], Rice [47], PEF [41], DINT [43], FastPFor [26], OptPFor [60], Simple16 [61], and QMX [52]. These baselines cover most mainstream codecs adopted in practical systems [1, 4], and are configured according to a recent benchmark [45]. For learned compressors, we evaluate LA-vector [9], the first learned compressor, and LeCo-var [34], a recent method designed for compressing generic datasets beyond sorted integers. For both LA-vector and LeCo-var, we follow their respective ϵ configuration strategies to determine the error bounds. Table 2 summarizes the major technical features of all evaluated methods.

Datasets. We evaluate LICO on three large-scale web document collections used to generate posting lists. (1) CW12B is the ClueWeb12 B13 corpus, containing roughly 50 million web pages [3]. (2) CCNews consists of about 40 million news articles published between September 2016 and March 2018, publicly available from CommonCrawl [2]. (3) WITD includes 5 million English documents from the Wikipedia-based Image Text Dataset [49]. Following the setups of a recent inverted index

Dataset	CW12B	CCNews	WITD
#Lists	2,289	3,066	899
Universe Size	43.5M	52.2M	5.4M
#Integers (docIDs)	8.7B	13.9B	0.31B
Gap Mean ± SD	24.31 ± 14.22	18.74 ± 12.08	28.58 ± 14.15
Gap Var. ± SD	3.4e4 ± 5.7e5	592.2 ± 690.0	990.2 ± 806.6
V ($\epsilon = 31$)	27.3	44.3	23.4
V ($\epsilon = 63$)	63.8	112.3	59.8
V ($\epsilon = 127$)	161.5	307.2	182.0
#Queries	1,756	2,661	1,326
2 terms	789 (44.9%)	1,092 (41.0%)	356 (26.9%)
3 terms	606 (34.5%)	921 (34.6%)	399 (30.1%)
4 terms	257 (14.6%)	434 (16.3%)	282 (21.3%)
5+ terms	104 (5.9%)	214 (8.0%)	289 (21.8%)

Table 3. Statistics of datasets and query workloads. V is the average OptimalPLA segment length under a certain ϵ; larger V implies stronger local linearity and higher predictability.

benchmark [45], docIDs (i.e., keys) are assigned to documents according to the lexicographic order of their URLs. Table 3 summarizes major statistics of the datasets.

Dataset Hardness. Prior work [55] introduces dataset hardness to quantify the position-to-value predictability in learned indexes. We adapt this notion to inverted-index compression by defining the hardness H of a posting list as the number of OptimalPLA segments needed to approximate the positions to docIDs mapping. To enable comparison across lists of varying lengths, we adopt the average segment length $V = n/H$, where n is the list length. As Table 3 shows, real posting lists exhibit long linear regions even at moderate ϵ, supporting the motivation for learned compression.

Query Workloads. For random access, we generate 500 queries per dataset, each consisting of 100,000 positions sampled uniformly from a posting list. For intersection and union queries that involve multiple posting lists, we construct query workloads from the TREC 2005 and TREC 2006 Efficiency Track topics [6]. We select query terms whose constituent words appear in the corresponding test collections. Workloads statistics are also summarized in Table 3.

Experimental Environment. All experiments are conducted on an Ubuntu 22.04 LTS server equipped with an Intel(R) Xeon(R) Gold 6430 CPU and 512 GB RAM. All methods are written in C++ and compiled by GCC with the `-O3` flag. Each experiment is repeated five times, and we report the average performance.

7.2 Validation of Theoretical Results (Q1)

We first empirically validate the correctness of our theoretical results on the optimal ϵ configuration. We randomly sample three lists of distinct gap variances from each dataset and synthesize three additional uniform lists, each of length one million. We then compress all lists using LICO with $\epsilon \in \{2^2 - 1, \cdots, 2^{12} - 1\}$ and compare the theoretically predicted $\epsilon^\star$ (Equation (6)) with the empirically observed optimal ϵ. As shown in Figure 10, a clear U-shaped relationship between ϵ and compression effectiveness (bits/int) can be observed. When ϵ is small, the ϵ-PLA produces an excessive number of segments, leading to higher space cost. As ϵ increases, the space cost decreases until residual storage begins to dominate, after which space cost rises again. This trend aligns closely with the prediction of Equation (5). Moreover, after error scaling, the theoretically derived $\epsilon^\star$ from Equation (6) aligns precisely with the empirically optimum, demonstrating that our cost model accurately guides the selection of $\epsilon^\star$ to achieve near-optimal space cost.

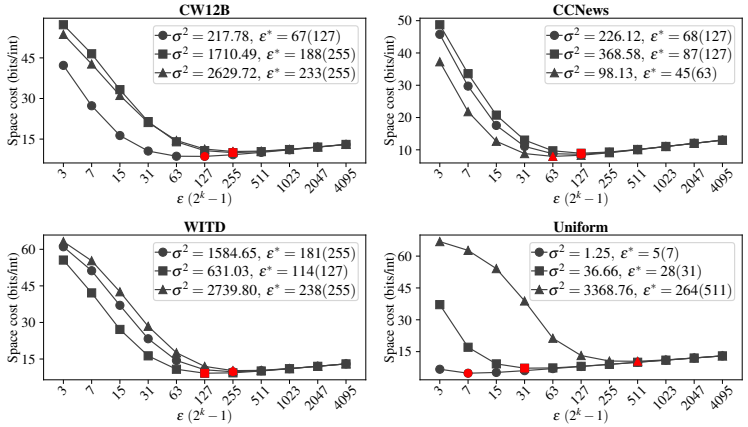


Fig. 10. Space cost under different error configurations. For each list, the empirically observed optimal ϵ is marked in red. The legends report the theoretically optimal $\epsilon^\star$ estimated by Equation (6) and its error-scaled value.

7.3 Partition Algorithm Evaluation (Q2)

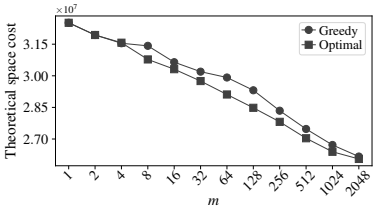
We next evaluate the effectiveness and efficiency of the partitioning algorithm introduced in Section 4.3, and empirically study how partition number m and page size affect performance. We randomly select two posting lists from CW12B with low variance ($\sigma^2 < 10^4$) and two with high variance ($\sigma^2 > 10^6$). Each list is partitioned using both the proposed greedy algorithm (Algorithm 1) and the optimal DP-based partitioning for comparison. After partitioning, we physically compress each list with LICO and record the resulting space cost to validate the accuracy of the cost model and the effectiveness of the greedy partitioning algorithm.

Theoretical Effectiveness. Figure 11a compares the theoretical space cost (Equation (10)) achieved by the greedy and optimal partitioning algorithms as the number of partitions m varies. For small m, the greedy algorithm produces slightly larger space costs than the optimal solution. As m increases (e.g., $m = 512$ in Figure 11a), Greedy gains more flexibility in partitioning, and its space cost quickly converges to that of the optimal algorithm.

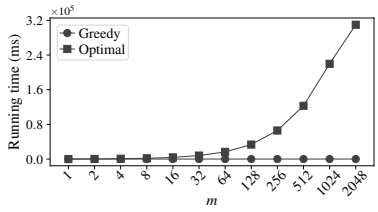
Practical Effectiveness. Figure 11c reports LICO's actual space cost (bits/int) under different partition numbers m, comparing greedy and optimal algorithms with and without error scaling. The greedy partitioning achieves space costs nearly identical to the optimal solution across all m, consistent with theoretical results in Figure 11a. Moreover, the error-scaling technique (Section 5.1) further reduces space cost by cutting segment count without increasing residual bits, demonstrating its practical effectiveness. Another finding is that, for a dataset with high variance (e.g., $\sigma^2 > 10^6$), partitioning effectively captures local distribution patterns to set an appropriate ϵ and substantially reduces total space cost compared with no partitioning ($m = 1$). In contrast, when variance is low, fine-grained partitioning offers limited additional benefit.

Partition Efficiency. From the above results, when the partition number $m = 1024$, the resulting space cost approaches the optimal value, and further increasing m yields only marginal gains. At this point, the space cost difference between the greedy and DP algorithms is less than 1.5%, while the greedy algorithm is 4 orders of magnitude faster than DP.

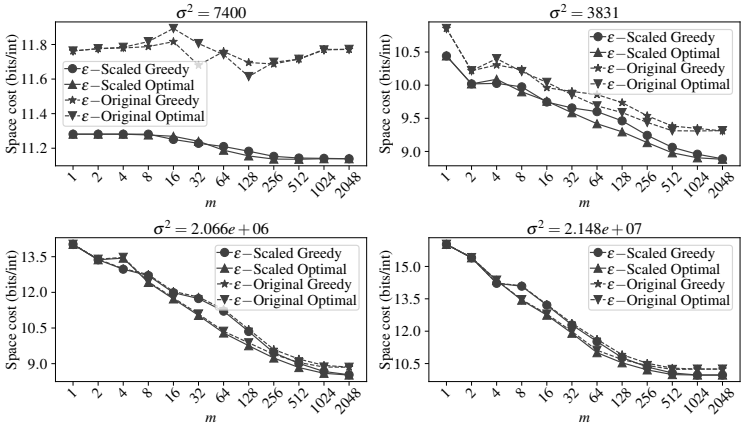
Effect of Page Size. LICO partitions keys using fixed-size pages (Section 4.3); here we study how page size affects greedy partitioning. As Figure 12 presents, space cost exhibits a U-shaped trend: very small pages capture the distribution poorly, while very large pages are too coarse for fine-grained partitioning. Larger pages also cut the partition number and thus shorten build time.



(a) Theoretical space cost of two partitioning algorithms.



(b) Running time of two partitioning algorithms.



(c) Actual space cost of partition algorithms w/ and w/o ϵ-scaling.

Fig. 11. Evaluation of the greedy and optimal partitioning algorithms under varying m (page size = 1K).

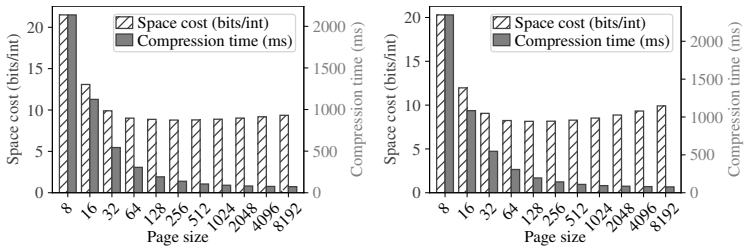
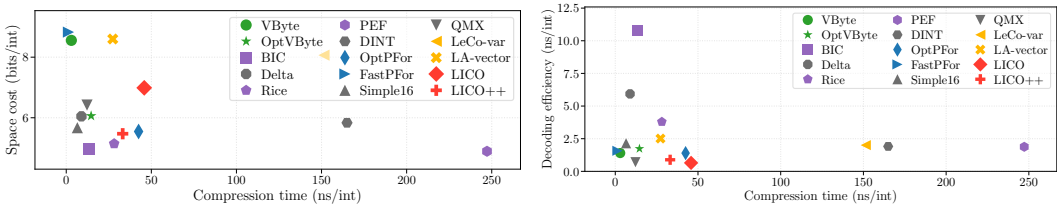


Fig. 12. Impact of page size on the performance of the greedy partitioning algorithm.

Empirically, a 1K page raises average space by up to 3% while reducing build time by ~70%, thus we use 1K as the default in subsequent experiments.

7.4 Overall Performance Evaluation (Q3)

Compression Time. Figure 13 illustrates the trade-offs among compression time, space cost, and decoding efficiency across methods. Overall, LICO and LICO++ outperform PEF, DINT, and LeCo-var, as these baselines spend substantial time searching or tuning their internal structures. Compared with simpler codecs, LICO++ trades higher compression time for lower space cost (Figure 13a), while this overhead is largely offset by the high decoding efficiency of LICO and LICO++ (Figure 14) in inverted index workloads, where decoding dominates execution. When compared with LA-vector, LICO's extra overhead stems primarily from its partitioning algorithm. Interestingly, despite requiring an extra residual encoding step, LICO++ achieves even shorter compression time. This is because LICO++'s residual compression reduces sensitivity to extreme residuals, allowing it to select larger ϵ values and thus reduce both the number of fitted segments and branch mispredictions.



(a) Compression time vs. Space cost (b) Compression time vs. Decoding efficiency

Fig. 13. Average performance among compression time, decoding efficiency, and space cost on three datasets.

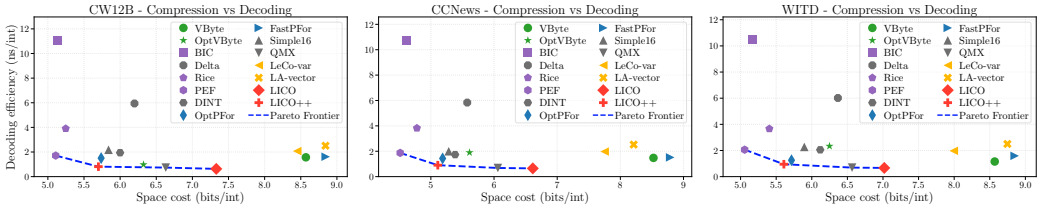


Fig. 14. Trade-off between space cost (bits per integer) and decoding efficiency (nanoseconds per integer).

Space-Time Trade-off. Figure 14 shows the trade-off between space cost and list decoding time for different compression methods. Both LICO and LICO++ lie on the Pareto frontier: LICO prioritizes decoding efficiency, while LICO++ achieves better overall balance by further reducing space cost with only a minor overhead.

For space cost, variable-length codecs (purple), such as PEF and Rice, achieve the highest compression ratios, followed by LICO++ and OptPFor. Compared with LICO, LICO++ achieves up to 1.29× higher compression ratio, confirming the effectiveness of residual encoding proposed in Section 5.3. Other learned compressors (yellow) are less space-efficient. The reason is that LA-vector does not capture fine-grained locality, and LeCo is not optimized for inverted index compression, instead emphasizing generality across diverse data types such as unordered data and strings.

In terms of efficiency, LICO achieves the highest decoding throughput, benefiting from SIMD parallelism and memory access optimizations. LICO++ remains close, only about 0.3 ns/int slower, revealing that the effect of second-stage residual decoding is minor. Other SIMD-optimized codecs, such as VByte, QMX, and FastPFor, also deliver relatively high decoding speeds but consume much more space than LICO++. Variable-length codecs with excellent space efficiency, such as PEF and Rice, are difficult to align in memory and vectorize, resulting in significantly slower decoding times compared with the SIMD-optimized LICO family.

7.5 Query Processing Evaluation (Q4)

We next evaluate the performance of random access, list intersection, and union across different methods. Notably, some compressors, such as LA-vector and LeCo, do not natively support list query operations. While a naïve implementation could be added for comparison, it would lack the necessary optimizations and lead to unfair results; therefore, we exclude them from this experiment.

Random Access. Table 4 presents the latency per query of random access. PEF achieves the best average latency per query, benefiting from its dedicated indexing structure and fast block decoding. LICO and LICO++ follow very closely and outperform the other gap-based baselines. Intuitively, gap-based compressors typically require recovering docIDs via prefix sums of gaps, which incurs additional latency for random access; in contrast, LICO/LICO++ performs direct reconstruction within a segment and only pays for locating the corresponding segment via binary search. Compared with LICO, LICO++ may incur extra work due to its residual decoding strategy;

Method	CW12B	CCNews	WITD	Average Latency
VByte	20.020	18.890	17.261	18.724
PEF	10.574	**7.029**	3.898	**7.167**
OptPFor	21.552	20.702	17.045	19.766
Simple16	37.741	37.185	32.568	35.831
QMX	13.265	14.210	9.234	12.236
LICO	9.705	10.156	6.776	8.879
LICO++	**9.661**	11.028	**2.118**	7.602

Table 4. Random Access latency (milliseconds per query).

Method	CW12B					CCNews					WITD				
	2	3	4	5+	Avg.	2	3	4	5+	Avg.	2	3	4	5+	Avg.
VByte	50.03	58.37	62.80	70.41	60.40	62.55	68.36	68.07	65.46	66.11	3.29	4.12	4.55	3.92	3.97
OptVByte	56.37	60.19	60.46	**65.75**	60.69	70.35	69.13	63.13	**56.69**	64.82	3.59	3.87	**4.03**	3.71	**3.80**
BIC	161.27	238.32	292.76	348.75	260.28	193.58	272.42	305.71	323.37	273.77	10.31	17.15	20.10	17.62	16.29
Delta	91.20	123.25	145.45	170.84	132.68	109.96	141.68	153.80	157.26	140.68	6.19	9.50	11.21	9.79	9.17
Rice	79.42	105.53	123.47	143.87	113.07	96.10	121.22	130.63	132.57	120.13	5.30	8.01	9.40	8.05	7.69
PEF	66.30	70.96	71.16	76.25	71.17	78.83	76.63	70.67	64.55	72.67	4.27	4.56	4.57	4.01	4.35
DINT	53.03	63.67	71.79	81.45	67.49	63.53	70.76	72.21	73.03	69.88	3.93	4.70	5.05	4.46	4.53
OptPFor	59.53	69.88	76.26	85.90	72.89	71.86	78.34	78.86	76.87	76.48	3.76	4.88	5.41	4.83	4.72
Simple16	66.47	81.44	90.49	102.74	85.29	78.24	89.13	91.25	89.78	87.10	4.32	5.63	6.18	5.51	5.41
QMX	53.86	58.03	60.28	66.09	59.57	65.98	66.61	63.16	59.61	63.84	3.52	3.97	4.20	**3.67**	3.84
LICO	**19.67**	**31.84**	**37.78**	**45.38**	**33.67**	**24.32**	**39.17**	**42.31**	**40.89**	**36.67**	**0.99**	**2.07**	**2.48**	**2.32**	**1.97**
LICO++	**18.98**	**32.27**	**47.49**	70.00	**42.19**	**23.37**	**38.73**	**52.08**	70.31	**46.12**	**1.22**	**2.62**	4.80	7.71	4.09

Table 5. Intersection query latency (milliseconds per query) under varying numbers of query terms. The top-2 results are shown in bold, and the best results among non-LICO methods are underlined.

Method	CW12B					CCNews					WITD				
	2	3	4	5+	Avg.	2	3	4	5+	Avg.	2	3	4	5+	Avg.
VByte	66.68	146.98	248.41	424.08	221.54	82.67	177.23	286.61	470.10	254.15	4.64	13.75	26.70	54.28	24.84
OptVByte	71.69	158.02	266.54	454.94	237.80	88.54	190.56	306.66	501.79	271.89	4.89	13.73	26.47	54.69	24.95
BIC	174.66	333.24	529.16	860.05	474.28	205.36	382.03	579.65	920.98	522.01	11.69	29.90	54.38	108.04	51.00
Delta	109.08	214.38	348.19	572.17	310.96	131.59	251.27	393.56	622.03	349.61	7.55	21.44	40.84	81.42	37.81
Rice	97.92	194.86	317.88	523.87	283.63	117.93	228.17	358.39	568.68	318.29	6.91	19.29	36.72	71.64	33.64
PEF	82.52	174.82	293.02	498.83	262.30	98.54	204.20	328.07	531.12	290.48	5.54	16.07	31.32	62.73	28.92
DINT	65.30	144.84	246.28	417.67	218.52	78.66	168.56	273.69	448.76	242.42	4.64	12.44	23.81	50.63	22.88
OptPFor	76.18	166.31	287.05	501.45	257.75	91.57	194.34	319.97	533.74	284.91	5.32	16.05	31.62	67.21	30.05
Simple16	85.65	182.60	314.80	553.61	284.16	100.76	210.66	347.31	586.50	311.31	5.89	17.48	34.50	73.40	32.82
QMX	72.05	159.48	279.80	497.28	252.15	85.45	185.15	311.90	536.56	279.77	4.82	15.24	31.50	66.71	29.57
LICO	**18.25**	**34.16**	**53.27**	**88.98**	**48.67**	**21.37**	**42.61**	**64.11**	**97.60**	**56.42**	**0.84**	**2.69**	**5.65**	**12.88**	**5.52**
LICO++	**17.16**	**33.29**	**54.00**	**90.49**	**48.74**	**20.88**	**39.59**	**61.68**	**98.62**	**55.19**	**1.09**	**3.18**	**6.39**	**13.94**	**6.15**

Table 6. Average latency (milliseconds) per union operation by varying the number of query terms. The best results are highlighted in the same way as in the intersection experiment.

however, this overhead is amortized when multiple accesses hit the same posting list. We also observe that LICO++ can outperform LICO on smaller datasets (e.g., WITD), because its larger ϵ (discussed in Section 7.4) reduces the number of segments substantially (about only 10% of LICO's segments on WITD), which lowers the segment-location cost and accelerates random access.

List Intersection. Table 5 reports the average processing time of intersection queries under different numbers of query terms. The top two results are highlighted in bold, while the best-performing non-LICO method is underlined. From the results, LICO and LICO++ achieve the best intersection performance in most cases, outperforming the fastest non-learned SIMD-based methods by up to 2.64×. This advantage comes from their stronger pruning capability and SIMD-aware memory

Method	Building Time (ns/int)	Space cost (bits/int)	Decoding Time (ns/int)
$\text{LICO}_{OptimalPLA}$	45.83	7.15	0.61
$\text{LICO}_{SwingPLA}$	38.46	7.32	0.92
LICO_{Spline}	39.03	8.47	3.06

Table 7. Performance of LICO fitting by different methods (averaged over three datasets). For decoding, OptimalPLA and SwingPLA use the SIMD-aware pipeline, while Spline uses the scalar decoder.

layout. An interesting finding is that, when the number of query terms is small (≤ 4), LICO and LICO++ perform similarly; as the number of terms increases, LICO++ quickly gets slower. Further analysis reveals that LICO++ typically adopts larger ϵ values, which weakens the pruning power of the NextGeq operator (potentially causing up to twice as many integers to be decompressed), thereby reducing performance.

List Union. Table 6 presents the performance results for list union queries. Since union operations require a full scan of all lists, their throughput largely depends on the sequential decoding speed. Similar to the intersection case, the SIMD-optimized LICO and LICO++ equipped with low-level optimizations achieve the highest performance. Among the remaining codecs, DINT performs the best, likely due to its cache-friendly dictionary design and the fact that decoding involves only a single dictionary lookup, enabling fast sequential processing [43]. Other fast-decoding methods, such as the VByte family and QMX, also show competitive union performance.

7.6 SIMD Effectiveness Evaluation (Q5)

We next conduct an in-depth analysis of the performance across AVX-512 (default), AVX2 (256-bit registers), and a scalar baseline. The scalar baseline performs element-wise key reconstruction but uses the same integer arithmetic optimizations. To adapt the default LICO pipeline to AVX2, we halve the number of keys decoded in each Decode Unit and adjust the Store Unit accordingly (Figure 9).

Evaluation results. Figure 15 reports LICO's performance under different decoding implementations. SIMD decoders (AVX2/AVX-512) execute 2.5~3.5× fewer instructions, incur 6.3~8.9× fewer branch mispredictions, and perform 2× fewer L1D store operations than the scalar baseline, yielding a 2~2.5× reduction in CPU cycles. These improvements arise from three factors: SIMD's ability to process multiple data elements per instruction, branch elimination via masking techniques, and store merging in the Store Unit (i.e., combining two 32-bit stores into a single 64-bit store). Compared to AVX2, AVX-512 further reduces instructions by 50% and branch mispredictions by 10%, resulting in a 10%~20% improvement in decoding throughput. This advantage stems from its wider vector registers, which support processing more data per instruction.

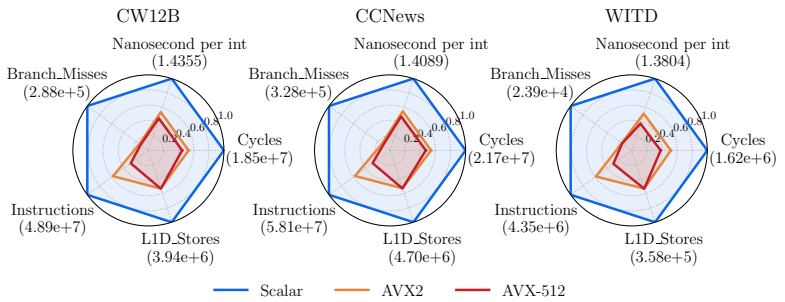


Fig. 15. In-depth evaluation of LICO decoding implementations: scalar, AVX2, and AVX-512. Parentheses report the maximum value for each metric; lower is better.

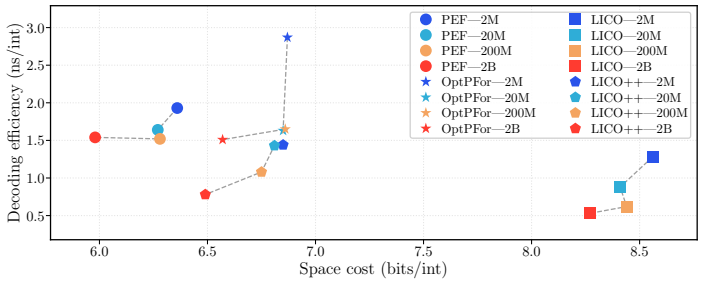


Fig. 16. Performance of LICO, PEF, and OptPFor across synthetic datasets of varying sizes

7.7 Sensitivity Evaluation (Q6)

We next evaluate the sensitivity of our framework on different fitting methods and dataset scales.

Fitting Method Influence. Table 7 reports LICO's end-to-end performance under different fitting methods. LICO$_{OptimalPLA}$ performs the best in both space cost and decoding efficiency, while LICO$_{SwingPLA}$ builds faster at the cost of slightly worse space usage and decoding speed. LICO$_{Spline}$ is substantially worse, consistent with its higher segment count and non–SIMD-friendly decoding (i.e., splines exhibit a predecessor dependency).

Data Size Influence. To further evaluate LICO's robustness across dataset scales, we generate four synthetic datasets by mixing posting lists sampled from three real datasets. Each synthetic dataset contains 2,000 lists, with list length capped at 1K, 10K, 100K, and 1M docIDs, yielding total sizes of 2M, 20M, 200M, and 2B integers, respectively. As shown in Figure 16, LICO and LICO++ remain robust as scale increases (list length $\geq$ 100K), while performance degrades slightly on the smallest dataset because short lists cannot effectively amortize SIMD pipeline overhead.

7.8 Summary of Results

Extensive experimental results demonstrate that our LICO framework effectively balances compression ratio and query processing efficiency. In terms of compression effectiveness, LICO++ achieves compression ratios comparable to theoretically optimal variable-length codecs such as PEF and Rice. In terms of efficiency, both LICO and LICO++ exhibit competitive random access performance and consistently outperform traditional, highly optimized compression methods across list decoding, intersection, and union operations.

8 Related Work

In this section, we review related studies on learned indexes, learned data compression, and inverted-index compression techniques.

Learned Index. The concept of learned index was first introduced by Kraska et al. [24], providing a data-driven alternative to conventional B$^+$-tree indexes. Subsequent studies have extended this paradigm in multiple directions. ALEX [13] supports dynamic workloads through gap arrays, while LIPP [56] further reduces lookup latencies by eliminating last-mile search errors. PGM-Index [17, 29] introduces a recursive ϵ-PLA construction with provable error bounds, achieving a theoretically optimal trade-off between space and time [29]. To enhance scalability and concurrency, XIndex [50] introduces two-phase compaction for multi-core environments, and FINEdex [28] minimizes data dependencies to support scalable in-memory operations.

Learned Compression. As the dual problem of learned indexing, learned compression models the mapping from indexes to keys using lightweight ML models. Oosterhuis et al. [39] first explore the potential of learned index compression by training a *lossy* classifier using a neural network to predict whether a term appears in a document. As an early *lossless* list compressor, LA-vector [9]

primarily establishes the conceptual foundations of learned compression, while LeCo [34] extends its applicability from sorted key compression to generic data types. In contrast to these existing efforts, our LICO framework targets principled parameter tuning and high-performance optimization for inverted index compression, achieving significantly superior performance.

Inverted Index Compression. Unlike lossy compression methods prevalent in scientific computing [57, 58], the compression of inverted index (sorted integers) constitutes a lossless sequence compression task grounded in information theory. Early works such as Delta and Gamma codes [15], Golomb [20], and Rice [47] established universal coding schemes based on quotient–remainder or unary encodings. Byte-aligned methods like VByte [27] later traded slight compression loss for much faster decoding. To improve the space-time balance, block-wise schemes were proposed: FOR [19] and the Simple family [7] use fixed-size packing, BIC [36] applies interpolation, and PForDelta [64] encodes exceptions separately. Subsequent designs, such as Elias-Fano/PEF [14, 41], achieve quasi-succinct compression with guaranteed compression ratio. Unlike traditional codecs that rely on hand-tuned parameters (e.g., PForDelta), our LICO framework automatically adapts to data distributions. Moreover, LICO achieves quasi-succinct compression comparable to Elias-Fano, while its memory-aligned design fully exploits SIMD parallelism for high-performance query processing.

9 Conclusion

This paper presents LICO, a learned compression framework for inverted indexes with both theoretical rigor and practical efficiency. LICO models sorted integer sequences using piecewise linear predictors and encodes residuals compactly to guarantee lossless reconstruction. To automatically configure error bounds, LICO designs a principled cost model that analytically derives the optimal ϵ for input dataset. At the system level, LICO adopts an SIMD-aware memory layout and accordingly designs query processing techniques to fully exploit hardware parallelism for high-performance query processing. Extensive experiments on web-scale inverted index compression benchmarks demonstrate that LICO consistently achieves Pareto-optimal space-time trade-offs and improves query performance by up to 5.52× compared with both highly optimized conventional codecs and recent learned compressors. Together, these results make our LICO framework not only a theoretically grounded compression scheme but also a practical, DBMS-ready solution for real-world applications.

10 Acknowledgments

We sincerely thank the anonymous reviewers for their constructive feedback. This work is partly supported by the National Natural Science Foundation of China (Grant No. 61872299 and No. 62441230), the A3 Foresight Program No. 62461146205, the Fundamental Research Funds for the Central Universities (SWU-KR24043), the Guangdong Provincial College Youth Innovative Talent Project (Grant No. 2025KQNCX075), the Natural Science Foundation of Top Talent of SZTU (Grant No. GDRC202520), and the gift from Sichuan JiuXin Technologies Co., Ltd.

References

[1] [n. d.]. Apache Lucene. https://lucene.apache.org/. Accessed: 2025-10-10.
[2] [n. d.]. CC-News Dataset. https://huggingface.co/datasets/stanford-oval/ccnews. Accessed: 2025-10-10.
[3] [n. d.]. Clueweb12. https://lemurproject.org/clueweb12/. Accessed: 2025-10-10.
[4] [n. d.]. ElasticSearch. https://www.elastic.co/elasticsearch. Accessed: 2025-10-10.
[5] [n. d.]. The FastPFOR C++ library : Fast integer compression. https://github.com/fast-pack/FastPFOR. Accessed: 2025-10-10.
[6] [n. d.]. Text REtrieval Conference (TREC) 2006 Terabyte Track. https://trec.nist.gov/data/terabyte06.html. Accessed: 2025-10-10.

[7] Vo Ngoc Anh and Alistair Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Inf. Retr.* 8, 1 (2005), 151–166. doi:10.1023/B:INRT.0000048490.99518.5C

[8] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units. *Proc. VLDB Endow.* 4, 8 (2011), 470–481.

[9] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2021. A "Learned" Approach to Quicken and Compress Rank/Select Dictionaries. In *ALENEX*. SIAM, 46–59.

[10] Guille D. Canas. 2021. Faster-Than-Native Alternatives for x86 VP2INTERSECT Instructions. *CoRR* abs/2112.06342 (2021). arXiv:2112.06342 https://arxiv.org/abs/2112.06342

[11] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *OSDI*. USENIX Association, 155–171.

[12] Herbert A David and Haikady N Nagaraja. 2004. *Order statistics*. John Wiley & Sons.

[13] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD Conference*. ACM, 969–984.

[14] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (1974), 246–260. doi:10.1145/321812.321820

[15] Peter Elias. 1975. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory* 21, 2 (1975), 194–203. doi:10.1109/TIT.1975.1055349

[16] Hazem Elmeleegy, Ahmed K Elmagarmid, Emmanuel Cecchet, Walid G Aref, and Willy Zwaenepoel. 2009. Online piece-wise linear approximation of numerical streams with precision guarantees. *Proceedings of the VLDB Endowment* 2, 1 (2009), 145–156.

[17] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *ICML (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 3123–3132.

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

[19] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, Susan Darling Urban and Elisa Bertino (Eds.). IEEE Computer Society, 370–379. doi:10.1109/ICDE.1998.655800

[20] Solomon W. Golomb. 1966. Run-length encodings (Corresp.). *IEEE Trans. Inf. Theory* 12, 3 (1966), 399–401. doi:10.1109/TIT.1966.1053907

[21] Jun Heo, Jaeyeon Won, Yejin Lee, Shivam Bharuka, Jaeyoung Jang, Tae Jun Ham, and Jae W. Lee. 2020. IIU: Specialized Architecture for Inverted Index Search. In *ASPLOS*. ACM, 1233–1245.

[22] David A Huffman. 2007. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (2007), 1098–1101.

[23] Andreas Kipf, Dominik Horn, Pascal Pfeil, Ryan Marcus, and Tim Kraska. 2022. LSI: a learned secondary index structure. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Philadelphia, Pennsylvania) *(aiDM '22)*. Association for Computing Machinery, New York, NY, USA, Article 4, 5 pages. doi:10.1145/3533702.3534912

[24] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD Conference*. ACM, 489–504.

[25] John D. Lafferty and ChengXiang Zhai. 2001. Document Language Models, Query Models, and Risk Minimization for Information Retrieval. In *SIGIR*. ACM, 111–119.

[26] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.* 45, 1 (2015), 1–29.

[27] Daniel Lemire, Nathan Kurz, and Christoph Rupp. 2018. Stream VByte: Faster byte-oriented integer compression. *Inf. Process. Lett.* 130 (2018), 1–6.

[28] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proc. VLDB Endow.* 15, 2 (2021), 321–334. doi:10.14778/3489496.3489512

[29] Qiyu Liu, Siyuan Han, Yanlin Qi, Jingshu Peng, Jin Li, Longlong Lin, and Lei Chen. 2025. Why Are Learned Indexes So Effective but Sometimes Ineffective? *Proc. VLDB Endow.* 18, 9 (2025), 2886–2898. https://www.vldb.org/pvldb/vol18/p2886-liu.pdf

[30] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. 2025. How good are multi-dimensional learned indexes? An experimental survey. *The VLDB Journal* 34, 2 (2025), 17.

[31] Qiyu Liu, Yanyan Shen, and Lei Chen. 2021. Lhist: Towards learning multi-dimensional histogram for massive spatial data. In *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 1188–1199.

[32] Qiyu Liu, Yanyan Shen, and Lei Chen. 2022. HAP: an efficient hamming space index based on augmented pigeonhole principle. In *Proceedings of the 2022 International Conference on Management of Data*. 917–930.
[33] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable learned bloom filters for data streams. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2355–2367.
[34] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1 (2024), 65:1–65:28.
[35] Alistair Moffat, Radford M. Neal, and Ian H. Witten. 1998. Arithmetic Coding Revisited. *ACM Trans. Inf. Syst.* 16, 3 (1998), 256–294.
[36] Alistair Moffat and Lang Stuiver. 1996. Exploiting Clustering in Inverted File Compression. In *Data Compression Conference*. IEEE Computer Society, 82–91.
[37] Alistair Moffat and Justin Zobel. 1992. Parameterised Compression for Sparse Bitmaps. In *SIGIR*. ACM, 274–285.
[38] Thomas Neumann and Sebastian Michel. 2008. Smooth Interpolating Histograms with Error Guarantees. In *Sharing Data, Information and Knowledge, 25th British National Conference on Databases, BNCOD 25, Cardiff, UK, July 7-10, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5071)*, W. Alex Gray, Keith G. Jeffery, and Jianhua Shao (Eds.). Springer, 126–138. doi:10.1007/978-3-540-70504-8_12
[39] Harrie Oosterhuis, J. Shane Culpepper, and Maarten de Rijke. 2018. The Potential of Learned Index Structures for Index Compression. In *Proceedings of the 23rd Australasian Document Computing Symposium* (Dunedin, New Zealand) *(ADCS '18)*. Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. doi:10.1145/3291992.3291993
[40] Joseph O'Rourke. 1981. An On-Line Algorithm for Fitting Straight Lines Between Data Ranges. *Commun. ACM* 24, 9 (1981), 574–578.
[41] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano indexes. In *SIGIR*. ACM, 273–282.
[42] Rasmus Pagh. 2001. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM J. Comput.* 31, 2 (2001), 353–363.
[43] Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. 2019. Fast Dictionary-Based Compression for Inverted Indexes. In *WSDM*. ACM, 6–14.
[44] Giulio Ermanno Pibiri and Rossano Venturini. 2020. On Optimally Partitioning Variable-Byte Codes. *IEEE Trans. Knowl. Data Eng.* 32, 9 (2020), 1812–1823.
[45] Giulio Ermanno Pibiri and Rossano Venturini. 2021. Techniques for Inverted Index Compression. *ACM Comput. Surv.* 53, 6 (2021), 125:1–125:36.
[46] Ian Rae, Alan Halverson, and Jeffrey F. Naughton. 2014. In-RDBMS inverted indexes revisited. In *ICDE*. IEEE Computer Society, 352–363.
[47] Robert Rice and James Plaunt. 1971. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology* 19, 6 (1971), 889–897.
[48] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. 2002. Compression of inverted indexes for fast query evaluation. In *SIGIR*. ACM, 222–229.
[49] Krishna Srinivasan, Karthik Raman, Jiecao Chen, Michael Bendersky, and Marc Najork. 2021. WIT: Wikipedia-based Image Text Dataset for Multimodal Multilingual Machine Learning. In *SIGIR*. ACM, 2443–2449.
[50] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 308–320. doi:10.1145/3332466.3374547
[51] Tetzank. 2015. SIMDSetOperations Library. GitHub repository. https://github.com/tetzank/SIMDSetOperations. Accessed: 2025-10-10.
[52] Andrew Trotman. 2014. Compression, SIMD, and Postings Lists. In *Proceedings of the 2014 Australasian Document Computing Symposium, ADCS 2014, Melbourne, VIC, Australia, November 27-28, 2014*, J. Shane Culpepper, Laurence Anthony F. Park, and Guido Zuccon (Eds.). ACM, 50.
[53] Ash Vardanian. 2025. SimSIMD Library. GitHub repository. https://github.com/ashvardanian/SimSIMD. Accessed: 2025-10-10.
[54] Sebastiano Vigna. 2013. Quasi-succinct indices. In *WSDM*. ACM, 83–92.
[55] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (2022), 3004–3017.
[56] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288. doi:10.14778/3457390.3457393
[57] Mingze Xia, Sheng Di, Franck Cappello, Pu Jiao, Kai Zhao, Jinyang Liu, Xuan Wu, Xin Liang, and Hanqi Guo. 2024. Preserving topological feature with sign-of-determinant predicates in lossy compression: A case study of vector field critical points. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4979–4992.

[58] Mingze Xia, Bei Wang, Yuxiao Li, Pu Jiao, Xin Liang, and Hanqi Guo. 2025. Tspsz: An efficient parallel error-bounded lossy compressor for topological skeleton preservation. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 3682–3695.
[59] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded Piecewise Linear Representation for online stream approximation. *VLDB J.* 23, 6 (2014), 915–937.
[60] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *WWW*. ACM, 401–410.
[61] Jiangong Zhang, Xiaohui Long, and Torsten Suel. 2008. Performance of compressed inverted list caching in search engines. In *WWW*. ACM, 387–396.
[62] Xianyu Zhu, Qiyu Liu, Guangyi Zhang, Zhibing Sha, Jianwei Liao, Sha Hu, and Lei Chen. 2026. LICO: An SIMD-Aware High-Performance Learned Inverted Index Compression Framework (Technical Report). https://github.com/xianyuzhuruc/LICO.
[63] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* 38, 2 (2006), 6.
[64] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. IEEE Computer Society, 59.

Received October 2025; revised January 2026; accepted February 2026