Not Small Enough? SegPQ: A Learned Approach to Compress Product Quantization Codebooks

Qiyu Liu
Southwest University
qyliu.cs@gmail.com

Yanlin Qi
HIT Shenzhen
yanlinqi7@gmail.com

Siyuan Han
HKUST
shanaj@connect.ust.hk

Jingshu Peng
ByteDance
jingshu.peng@bytedance.com

Jin Li
Harvard University
jinli@g.harvard.edu

Lei Chen
HKUST & HKUST (GZ)
leichen@cse.ust.hk

ABSTRACT

The rapid advancements of generative artificial intelligence (GenAI) have recently led to renewed attention towards approximate nearest neighbor (ANN) search and vector databases (VectorDB). Among various ANN methodologies, product quantization (PQ) is widely used as a coarser quantizer to generate space-efficient representations for large-scale dense vectors. However, the codebooks generated by PQ often reach sizes of several gigabytes, making them impractical for web-scale, high-dimensional vectors in resource-constrained computing environments.

In this study, we propose **SegPQ**, a simple yet effective vector quantization codebook compression framework for *arbitrary* PQ variants, enabling **in-memory** vector query processing on devices with limited memory. SegPQ is a **lossless** compression framework to any PQ codebook, where the raw codewords are represented by a trained error-bounded piecewise linear approximation model (PLA) and pre-computed low-bit residuals. We theoretically demonstrate that, with high probability, the number of bits per compressed codeword is $1.721 + \lceil \log_2 \epsilon^{\mathrm{OPT}} \rceil$, where ϵ^{OPT} is an error parameter that can be determined by data characteristics. To boost query efficiency, we further design SIMD-aware query processing algorithms on compressed codebooks to fully exploit the hardware parallelism offered by modern architectures. Extensive benchmark results on real datasets showcase that, given **1 billion** vectors, SegPQ can reduce the memory footprint of PQ codebook by up to **4.7** times (approx. **851 MB**) while introducing only **3.3%** extra query processing overhead caused by decompression.

PVLDB Reference Format:
Qiyu Liu, Yanlin Qi, Siyuan Han, Jingshu Peng, Jin Li, and Lei Chen. Not Small Enough? SegPQ: A Learned Approach to Compress Product Quantization Codebooks. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/qyliu-hkust/segpq.

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

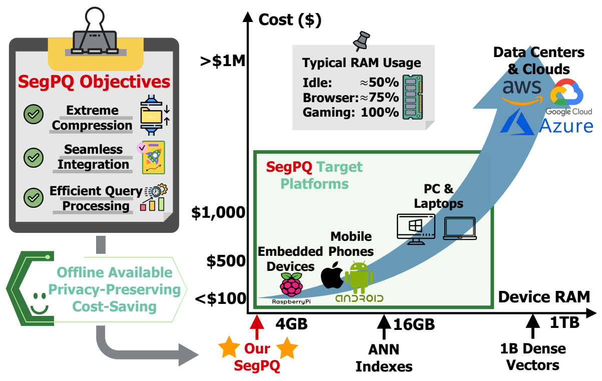
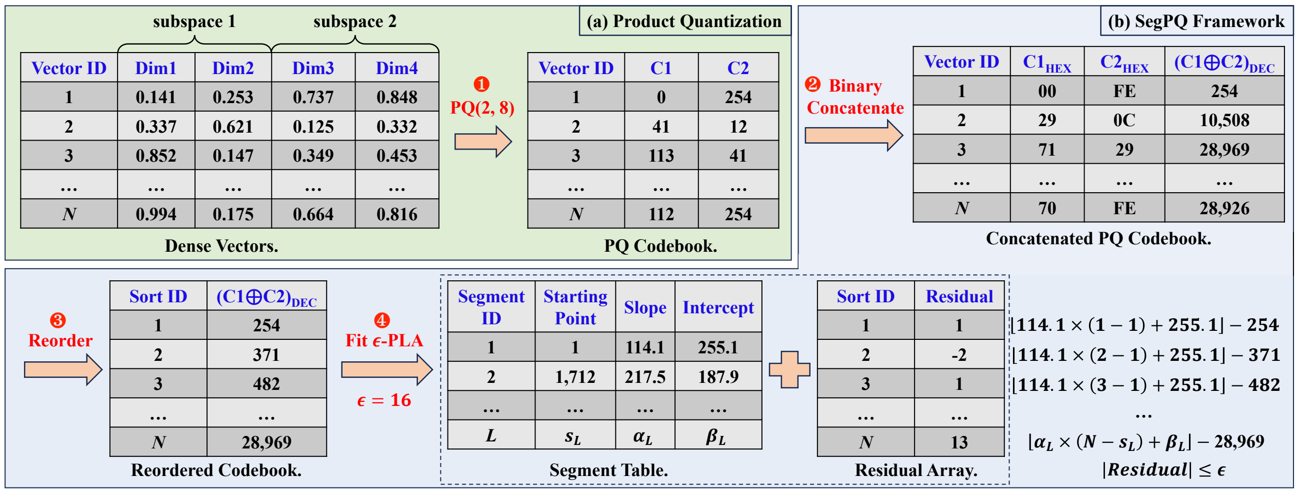


Figure 1: Illustration of SegPQ's design considerations: ❶ compact for resource constrained devices, ❷ compatible with *any* PQ variants, and ❸ neglectable computation overhead.

1 INTRODUCTION

A defining attribute of large language models (LLMs) is their unprecedented scale, challenging their application within *constrained computing environments* [14, 47, 49, 62]. Recent advancements [4, 15, 25, 60], however, have facilitated their deployment on devices with limited resources, such as smartphones and even embedded devices [36, 45, 46]. Nonetheless, these newly proposed systems often come with heavy optimization and complex workflow, with their actual performance to be verified. In this context, techniques like VectorDB and retrieval-augmented generation (RAG) [6, 28, 50] can be the key to advancing LLM applications on such devices. This vector infrastructure of LLMs, integrated with external knowledge, ensures efficient data retrieval through approximate nearest neighbor (ANN) search, allowing LLMs to access private knowledge bases with minimal memory use. In contrast to the large-scale nature of LLMs, ANN engines are designed as small and fast as possible [31, 35]. As LLMs are difficult to condense, refining their already-small integration technologies brings new perspectives, to unlock the potential of on-device AI innovations.

Consider dealing with billion-scale vector sets, which usually come to hundreds of gigabytes. To reduce their sizes, memory-efficient ANN methods like Locality-Sensitive Hashing [2, 17, 34], Proximity Graph [39, 40], and Vector Quantization have been explored [10, 24, 26, 66]. Among these, Product Quantization (PQ) stands out as a simple yet effective way, widely supported by mainstream VectorDBs like Faiss [19], Milvus [43], and Pinecone [52]. PQ



Vector ID	Dim1	Dim2	Dim3	Dim4
1	0.141	0.253	0.737	0.848
2	0.337	0.621	0.125	0.332
3	0.852	0.147	0.349	0.453
…	…	…	…	…
N	0.994	0.175	0.664	0.816

Dense Vectors.

Vector ID	C1	C2
1	0	254
2	41	12
3	113	41
…	…	…
N	112	254

PQ Codebook.

Vector ID	$C1_{HEX}$	$C2_{HEX}$	$(C1\oplus C2)_{DEC}$
1	00	FE	254
2	29	0C	10,508
3	71	29	28,969
…	…	…	…
N	70	FE	28,926

Concatenated PQ Codebook.

Sort ID	$(C1\oplus C2)_{DEC}$
1	254
2	371
3	482
…	…
N	28,969

Reordered Codebook.

Segment ID	Starting Point	Slope	Intercept
1	1	114.1	255.1
2	1,712	217.5	187.9
…	…	…	…
L	s_L	α_L	β_L

Segment Table.

Sort ID	Residual	
1	1	$\lfloor 114.1 \times (1-1) + 255.1 \rfloor - 254$
2	-2	$\lfloor 114.1 \times (2-1) + 255.1 \rfloor - 371$
3	1	$\lfloor 114.1 \times (3-1) + 255.1 \rfloor - 482$
…	…	…
N	13	$\lfloor \alpha_L \times (N - s_L) + \beta_L \rfloor - 28{,}969$

Residual Array.

$|Residual| \leq \epsilon$

Figure 2: Running examples of (a) the classic product quantization (PQ) framework, and (b) our SegPQ framework. In (a), a PQ(2, 8) codebook is constructed on N 4-D dense vectors. PQ(2, 8) means that the sub-space partition number is 2 and each sub-space will be clustered into $2^8 = 256$ parts. In (b), SegPQ first projects each codeword into an integer key using binary concatenation, and then the codebook is reordered to fit an optimal PLA model with error constraint ϵ. Finally, SegPQ computes the differences between PLA predictions and true codewords as residuals, which are materialized together with all line segments as the compressed codebook.

aims to reduce memory usage by dividing vectors into subspaces and compressing them through clustering. As illustrated in Figure 2(a), PQ partitions high-dimensional vectors into m equal-sized subspaces, then uses k-means clustering [38] to find k centroids for each subspace. After clustering, each vector is approximated by concatenating its closest centroids in m sub-spaces, with only $m \cdot \lceil \log_2 k \rceil$ bits.

Example 1.1 (Product Quantization). Figure 2(a) illustrates a toy example of generating a PQ(2, 8) codebook on N 4-D dense vectors. Each partitioned sub-space of 2 dimensions will be clustered into $2^8 = 256$ clusters, and each sub-vector is approximated by its closet k-means centroid. For instance, after applying PQ, a dense vector $\mathbf{x} = [0.141, 0.253, 0.737, 0.848]$ will be encoded as a codeword $[0, 254]$ where 0 and 254 stands for the cluster ID, i.e., the 0-th and 254-th cluster in two partitioned sub-spaces. By using PQ(2, 8), each 4-D dense vector (4 FP32, 128 bits) can be compressed into 2 unsigned bytes (16 bits), achieving an 8× space reduction.

Though achieving a significant compression ratio compared to raw dense vectors, the resulting PQ codebooks, in practical settings (e.g., $m = 4$, $k = 256$), still necessitate 4 GiB for 1 billion vectors. Given a typical *idle* RAM (**≤4 GiB**) [58], as illustrated in Figure 1, this size is still not small enough, thus preventing the deployment to these resource-constrained devices, such as the latest iPhone or personal laptop. A recent work DeltaPQ [65] first attempts to reduce PQ codebook size by reducing redundancies in codewords using delta encoding [67]. However, such a method introduces non-negligible compression and decompression overhead, making it hard to scale to web-scale vector sets.

Motivated by this, we pose the following research question: *"Can we further compress the PQ codebook without compromising its effectiveness in resource-constrained environments?"* Specifically, as depicted in Figure 1, our objectives are threefold: ❶ Compress the codebook generated by PQ to fit in resource-constrained environments; ❷ Maintain *lossless* compression and compatibility to *any* PQ variants, to ensure seamless integration with existing ANN engines; ❸ Ensure efficient construction and query processing of the compressed codebook, with no significant extra overhead.

To achieve these objectives, in this study, we introduce SegPQ, a *lossless and data-driven* PQ codebook compression scheme. SegPQ identifies compression opportunities by carefully investigating the distribution characteristics of PQ codewords and demonstrates that a compact piecewise linear model (PLA) can well approximate the original codebook with controllable errors. Inspired by the recent studies on learned index and learned compression [11, 20, 21, 30], we represent the raw PQ codebook by a small set of error-bounded line segments and an array of pre-computed low-bit residuals (i.e., the differences between PLA predictions and true codewords). With such a compressed codebook, an arbitrary codeword can be *losslessly* recovered from the PLA model output and the pre-computed residual value (i.e., *codewords=segments+error*). Moreover, SegPQ enables *direct* query processing on compressed codebooks and fully exploits SIMD parallelism on modern computing architectures for enhanced codebook traversal efficiency.

In summary, our technical contributions are threefold. ❶ **Problem Exploration.** We share a new perspective to further compress PQ codebooks by directly learning the codewords' distribution information. ❷ **Theoretical Results.** SegPQ is a principled approach where the simple idea delivers non-trivial results. Our theoretical analysis reveals that SegPQ achieves an encoding efficacy of $1.721 + \lceil \log_2 \epsilon^{OPT} \rceil$ bits per PQ codeword, where ϵ^{OPT} is the optimal PLA error constraint that can be determined by data distribution characteristics. ❸ **Prominent Performance.** Extensive benchmark results on 4 real datasets demonstrate that SegPQ significantly reduces the memory footprint of the original PQ codebook by up to **4.7** times (to ≈ **851 MB** for **1 billion** vectors) with only a **3.3%** increase in computation overhead.

The remainder of this paper is structured as follows. Section 2 introduces the basis of product quantization and formulates the codebook compression problem. Section 3 presents our SegPQ framework, detailing the compression and decompression algorithms. Section 4 theoretically analyzes the compression efficacy of SegPQ. Section 5 presents the experimental evaluation and results. Section 6 reviews and discusses related works. Finally, Section 7 concludes the paper and highlights future directions.

2 PRELIMINARIES

In this section, we overview the general PQ framework. Then, we formulate the problem of lossless codebook compression. Table 1 summarizes major notations used hereafter.

2.1 Product Quantization

Product Quantization (PQ) is a simple yet effective approach to compress high-dimensional dense vectors into compact discrete forms for efficient ANN search [24, 26, 66].

For a dense vector $\mathbf{x} \in \mathbb{R}^d$, let $\mathbf{x} = [\mathbf{x}^1, \mathbf{x}^2, \cdots, \mathbf{x}^m]$ denote the concatenation of m sub-vectors of equal sizes. W.l.o.g., we assume that d is divisible by m, implying that each $\mathbf{x}^i \in \mathbb{R}^{d/m}$. Given N vectors $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$, PQ learns an encoding scheme $C(\cdot)$ that projects $\mathbf{x} \in \mathbb{R}^d$ onto a codeword formed by concatenating m sub-codes $C(\mathbf{x}) = [c_1(\mathbf{x}^1), \cdots, c_m(\mathbf{x}^m)]$, where $c_i : \mathbb{R}^{d/m} \mapsto \Sigma_i$ and Σ_i is a discrete set of cardinality k. Let decoder $D(\cdot)$ represent the process of recovering the original vector from $C(\mathbf{x})$. PQ aims to minimize the following quantization error on $\mathbf{X}$:

$$\begin{aligned} &\min_{c_1,\cdots,c_m} \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{x}_i - D(C(\mathbf{x}_i))\|^2 \\ &\text{s.t.} \quad C(\mathbf{x}) \in \Sigma^1 \times \cdots \times \Sigma^m, \end{aligned} \tag{1}$$

where $\|\cdot\|$ is the Euclidean distance to depict the reconstruction distortion for each vector $\mathbf{x}_i$.

When $m = 1$, minimizing Eq. (1) is equivalent to the classical k-means problem, and Lloyd's heuristics can be applied to find a local optimum [38]. For $m > 1$, PQ adopts k-means to assign $\mathbf{x}_i$ to its nearest centroid in the i-th sub-space, corresponding to the i-th encoder function $c_i(\cdot)$. Then, the decoder function $D(\cdot)$ concatenates the m k-means centroids to reconstruct the original dense vector $\mathbf{x}$. We use PQ(m, $\lceil \log_2 k \rceil$) to denote a PQ quantizer of partition number m and k-means cluster number k.

The encoder function $C(\mathbf{x})$ will generate a codeword from the Cartesian product $\Sigma^1 \times \cdots \times \Sigma^m$, meaning that PQ can represent exponentially large vector sets using $m \cdot \lceil \log_2 k \rceil$ bits per codeword. In practice, k is usually set to 256 such that each sub-code occupies one byte. Given a query vector $\mathbf{y} \in \mathbb{R}^d$, the Euclidean distance $\|\mathbf{y} - \mathbf{x}\|$ for $\mathbf{x} \in \mathbf{X}$ can be approximated by either symmetric distance computation $\text{SDC}(\mathbf{y}, \mathbf{x}) = \|D(C(\mathbf{y})) - D(C(\mathbf{x}))\|$ or asymmetric distance computation $\text{ADC}(\mathbf{y}, \mathbf{x}) = \|\mathbf{y} - D(C(\mathbf{x}))\|$. Both SDC and ADC can be efficiently processed by pre-computing a distance lookup table of size $O(k^2)$ [26]. In practice, SDC is generally less accurate but more efficient than ADC due to the greater loss of information in the query vector.

Table 1: Summary of major notations.

Notation	Explanation
m, k	the partition number and clustering number
$\mathbf{x}^1, \cdots, \mathbf{x}^m$	m partitioned sub-vectors of $\mathbf{x} \in \mathbb{R}^d$
$C(\cdot), D(\cdot)$	the PQ encoder function and decoder function
$\mathbf{q}_i$	the PQ codeword representation of vector $\mathbf{x}_i$
$\mathbf{Q}$	the PQ codebook of a vector database $\mathbf{X}$
ϵ	the maximum error constraint of a PLA model
(s, α, β)	breakpoint, slope, and intercept of a segment
δ	the residual value
b	the number of bits required to encode δ

2.2 Lossless Codebook Compression

When dealing with web-scale vector databases, the PQ codebook size can escalate to dozens of gigabytes, hindering the deployment on devices with constrained memory budgets. Motivated by this challenge, we formulate the problem of PQ codebook compression.

Definition 2.1 (Lossless Codebook Compression). Let $\mathbf{Q}$ denote the codebook of N codewords $\mathbf{q}_1, \cdots, \mathbf{q}_N$, learned by a PQ quantizer with parameters m and k. The problem of *lossless codebook compression* is to find a *compact representation* $\mathbf{Q}^c$ that can efficiently support operations: ❶ *Random Access*, denoted by $\mathsf{AT}(\mathbf{Q}^c, i)$, which returns $\mathbf{q}_i$ given an index i; and ❷ *Complete Decoding*, denoted by $\mathsf{DC}(\mathbf{Q}^c)$, which returns the entire raw codebook $\mathbf{Q}$.

The compression ratio r is defined as the relative ratio between the memory footprints of $\mathbf{Q}$ and $\mathbf{Q}^c$. Ideally, r should be as large as possible. However, according to the *information-theoretic lower bound* [57, 64], encoding N monotone elements drawn from a universe of size U requires at least $N \cdot \lceil \log_2 \binom{U+N}{N} \rceil \approx N \cdot \log_2 \frac{U+N}{N}$ bits. Thus, the design principle of the compressed codebook $\mathbf{Q}^c$ is to try to match the information-theoretic lower bound while incurring acceptable additional overheads.

3 METHODOLOGIES

We overview the SegPQ framework in Section 3.1 and elaborate on the compression and decompression algorithm details in Section 3.2 and Section 3.3, respectively.

3.1 SegPQ Framework Overview

The core idea of SegPQ is rooted in the observation that an arbitrary PQ codebook $\mathbf{Q}$ can be *reorganized* in a manner that allows it to be well-fitted by even simple machine learning models. Specifically, we adopt *piecewise linear approximation* to learn $\mathbf{Q}$ and materialize all *residuals* (i.e., the difference between the prediction and the true codeword), thereby ensuring that the raw codewords can be efficiently recovered without any loss of information.

The SegPQ workflow, as shown in Figure 2(b), is outlined as follows.

❶ **Projection.** For a PQ codebook $\mathbf{Q} = \{\mathbf{q}_1, \cdots, \mathbf{q}_N\}$, we construct a *bijection* to map each $\mathbf{q}$ to a 1-D *integer* sorting key, meanwhile enabling the reverse recovery from the given sorting key to $\mathbf{q}$.

❷ **Reordering.** Let $\mathcal{K}$ denote the set of projected 1-D sorting keys. We then sort $\mathcal{K}$ in *increasing order* and reorder the raw $\mathbf{Q}$ according to the corresponding sorting key.

Algorithm 1: SegPQ Codebook Compression

```
Input: the original PQ codebook $\mathbf{Q}$
Output: the compressed codebook $\mathbf{Q}^c$
1 $\mathcal{K} \leftarrow \{\mathsf{BinCat}(\mathbf{q}) | \mathbf{q} \in \mathbf{Q}\}$ // Project to 1-D key
2 $sort(\mathcal{K})$ // Reorder by sorting key
3 $\epsilon^{\mathsf{OPT}} \leftarrow \sqrt{\frac{2C \cdot \ln 2 \cdot (m \log_2 k + 2F)}{N}}$ // Configure by Theorem 4.7
4 $f \leftarrow \mathsf{OptPLA}(X = \{1, \cdots, N\}, Y = \mathcal{K}, \epsilon^{\mathsf{OPT}})$ // Fit PLA
5 $\Delta \leftarrow \{\lfloor f(i) \rfloor - \mathsf{Q}[i] | i \in \{1, \cdots, N\}\}$ // Compute residuals
6 return $(f.segments, \Delta)$ // Return segments and residuals
```

❸ **Model Fitting.** With a *monotonic* key list $\mathcal{K}$ and a pre-specified error parameter ϵ, we learn an error-bounded piecewise linear approximation model (ϵ-PLA) $f(i)$ such that $|\lfloor f(i) \rfloor - \mathcal{K}_i| \leq \epsilon$. Intrinsically, $f(\cdot)$ learns the *inverse cumulative distribution function* (ICDF) of $\mathcal{K}$.
❹ **Index Construction.** For $i \in \{1, \cdots, N\}$, we compute the *residual* for each codeword $\mathbf{q}_i$ as $\delta_i = \lfloor f(i) \rfloor - \mathcal{K}_i$. Given that $\delta_i \in [-\epsilon, \epsilon]$, we require $b = \lceil 1 + \log_2 \epsilon \rceil$ bits to encode each residual into binary strings. Let $\Delta = \{\delta_1, \cdots, \delta_N\}$ denote the residual array. The SegPQ compression of the raw codebook $\mathbf{Q}$ is a tuple $\mathbf{Q}^c = (f, \Delta)$ of learned line segments and the pre-computed residual array.
❺ **Query Processing.** Given a compressed codebook $\mathbf{Q}^c = (f, \Delta)$, each codeword $\mathbf{q}_i$ can be recovered using $\mathcal{K}_i = \lfloor f(i) \rfloor + \Delta[i]$, considering that the projection from $\mathbf{Q}$ to $\mathcal{K}$ is a *bijection* in Step ❶. Since this recovery process is lossless, the subsequent ANN query processing remains *unchanged*.

3.2 Lossless Learned Compression

We then introduce how SegPQ constructs the compressed codebook in detail. The pseudo-code is given in Algorithm 1.

3.2.1 Codebook Projection. Directly learning a codeword composed of m sub-codes is challenging. Instead, during the preprocessing steps (❶ Projection and ❷ Reordering), we construct a projection $\mathcal{M} : \mathbb{N}^m \mapsto \mathbb{N}$ that maps a codeword $\mathbf{q}$ (with sub-space number m) to an integer key k. We require that $\mathcal{M}$ is a *bijection* to ensure the original codeword can be recovered *losslessly* from its projected value. In SegPQ, we choose the binary concatenation operation $\mathsf{BinCat}(\cdot)$ as the mapping function.

Definition 3.1 (Binary Concatenation). Given a PQ codeword $\mathbf{q} = \{\mathbf{q}^1, \mathbf{q}^2, \cdots, \mathbf{q}^m\}$, we define the binary concatenation operation as

$$\mathsf{BinCat}(\mathbf{q}) = \mathsf{dec}(\mathsf{bin}(\mathbf{q}^1) \oplus \cdots \oplus \mathsf{bin}(\mathbf{q}^m)), \tag{2}$$

where $\mathsf{bin}(\cdot)$ retrieves the binary representation of $\mathbf{q}^i$, $\oplus$ denotes the concatenation of two binary strings, and $\mathsf{dec}(\cdot)$ converts the concatenated binary string to its corresponding decimal value.

Example 3.2 (Binary Concatenation Example). In Figure 2, each PQ codeword is composed of two 8-bit sub-codes (i.e., four hexadecimal digits). For codeword $\mathbf{q}_1 = [00_{\mathsf{hex}}, \mathsf{FE}_{\mathsf{hex}}]$, $\mathsf{BinCat}(\mathbf{q}_1) = \mathsf{dec}(00\mathsf{FE}_{\mathsf{hex}}) = 254$; similarly, for codeword $\mathbf{q}_2 = [29_{\mathsf{hex}}, 0\mathsf{C}_{\mathsf{hex}}]$, $\mathsf{BinCat}(\mathbf{q}_2) = \mathsf{dec}(290\mathsf{C}_{\mathsf{hex}}) = 10,508$.

It is easy to verify that $\mathsf{BinCat}(\cdot)$ is a bijection as each binary string corresponds to a unique integer. The recovery process is also straightforward by *evenly* partitioning the binary representation

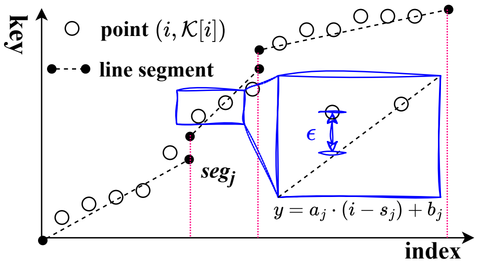


Figure 3: Illustration of an ϵ-PLA model.

of the projected codeword $\mathsf{bin}(\mathsf{BinCat}(\mathbf{q}))$ into m parts. Since the projected set $\mathcal{K} = \{\mathsf{BinCat}(\mathbf{q}) | \mathbf{q} \in \mathbf{Q}\}$ is logically equivalent to the raw codebook $\mathbf{Q}$, the problem of compressing $\mathbf{Q}$ can be then transformed to finding a succinct representation of $\mathcal{K}$.

Not limited to PQ, the $\mathsf{BinCat}(\cdot)$ projection is applicable to codewords learned by a wide range of vector quantizers, such as Optimized Product Quantization (OPQ) [24], Additive Quantization (AQ) [8], and Residual Quantization (RQ) [41]. This makes our SegPQ a general framework to take any quantization methodology as a pluggable component. From our experimental studies in Section 5, SegPQ+**X** can robustly compress the sizes of codebooks learned by quantizer **X**, where **X** can be PQ, OPQ, AQ, etc.

3.2.2 Reordering and Model Fitting. We then sort the input codewords in $\mathbf{Q}$ based on the set of binary concatenation values $\mathcal{K}$ in increasing order. With a sorted set $\mathcal{K}$, SegPQ finds a projection function $f : \mathcal{I} \mapsto \mathcal{K}$ satisfying a pre-specified error constraint ϵ, where $\mathcal{I} = \{1, 2, \cdots, |\mathcal{K}|\}$ is the index set. Such mapping f together with residuals (i.e., $\lfloor f(i) \rfloor - \mathcal{K}_i$) can serve as a compressed lossless representation of $\mathbf{Q}$ and enable direct query processing on the compressed data.

Intuitively, learning f is equivalent to learning the inverse cumulative distribution function (ICDF, a.k.a. quantile function) of $\mathcal{K}$. To balance the model's expressivity with inference efficiency, existing learned indexes often employ simple models like piecewise linear functions to approximate this mapping [12, 21]. An error-bounded piecewise linear approximation (ϵ-PLA) is illustrated in Figure 3 and defined as follows.

Definition 3.3 (ϵ-PLA). Given a set of points in Cartesian space $\{(i, \mathcal{K}[i])\}_{i=1,\cdots,N} \subseteq \mathcal{I} \times \mathcal{K}$, an ϵ-PLA is defined as a piecewise linear function of L line segments,

$$f(i) = \begin{cases} \alpha_1 \cdot (i - s_1) + \beta_1 & \text{if } s_1 \leq i < s_2 \\ \alpha_2 \cdot (i - s_2) + \beta_2 & \text{if } s_2 \leq i < s_3 \\ \quad \cdots & \quad \cdots \\ \alpha_L \cdot (i - s_L) + \beta_L & \text{if } s_L \leq i \end{cases} \tag{3}$$

such that $|\mathcal{K}[i] - \lfloor f(i) \rfloor| \leq \epsilon$ holds for $\forall i = 1, \cdots, N$. Each segment in f is a tuple (s_j, α_j, β_j), where s_j is the starting index, α_j is the slope, and β_j is the intercept.

The rationale for choosing ϵ-PLA is twofold. ❶ Unlike deep learning models requiring heavy runtimes like Pytorch, PLA is both space- and time-efficient in training and inference. ❷ The theoretical results presented in Section 4 will demonstrate that, due

Algorithm 2: SegPQ Codebook Decompression

```
Function AT($\mathbf{Q}^c = (Segs, \Delta), i$):
    // Find the proper segment for idx $i$
    $\ell^* \leftarrow$ minimum $\ell \in \{1, \cdots, |Segs|\}$ s.t. $Segs[\ell].s \geq i$
    // Compute the decimal codeword approximation
    $\mathbf{q}^* \leftarrow \lfloor Segs[\ell^*].\alpha \times (i - Segs[\ell^*].s) + Segs[\ell^*].\beta \rfloor$
    // Recover the codeword by adding residual
    return Split($\mathbf{q}^* + \Delta[i]$)
// Traverse the whole codebook $\mathbf{Q}^c$
Function DC($\mathbf{Q}^c = (Segs, \Delta)$):
    $\mathbf{Q} \leftarrow [\,]$ // Initialize an empty codebook
    // Optimize the loop using SIMD
    for $i \in \{1, 2, \cdots, |Segs|\}$ do
        for $j \in \{1, 2, \cdots, Segs[i].coverage\}$ do
            $\mathbf{q}^* \leftarrow \lfloor Segs[i].\alpha \times j + Segs[i].\beta \rfloor$
            $\mathbf{Q}.add$(Split($\mathbf{q}^* + \Delta[Segs[i].s + j]$))
    return $\mathbf{Q}$
```

to the uniform nature of $\mathcal{K}$'s distribution, we can use just a few line segments to fit a PLA with small error rates.

In implementation, we adopt the online algorithm [48] to fit an *optimal* ϵ-PLA such that the required number of line segments is minimized to satisfy the error constraint ϵ. Notably, although the optimal ϵ-PLA algorithm minimizes L under a given error constraint, it remains *unknown* how large L should be. We theoretically answer this question in Section 4, based on which we quantitatively derive the space overhead and the compression ratio of SegPQ.

Example 3.4 (SegPQ Compression). Continuing to Example 3.2, SegPQ reorders the original codewords based on their binary concatenation values (Step ❸ in Figure 2). Then, a piecewise linear model $f(\cdot)$ with error constraint 16 is fitted to depict the relationship between the sorting ID and the binary concatenation value (Step ❹ in Figure 2). For instance, segment 1 with slope 114.1 and intercept 255.1 covers codewords (reordered) with ID from 1 to 1,711. The predicted key for the 3-rd codeword 482 is $f(3) = \lfloor 114.1 \times (3-1) + 255.1 \rfloor = 483$. Thus, the residual for this codeword can be computed as $\delta_3 = 483 - 482 = 1$. Finally, the trained PLA model and pre-computed residuals are materialized as the lossless compressed version of the original codebook.

3.2.3 Discussion of Missing Details. We then discuss some missing details of SegPQ's codebook compression.

Reordering. To fit an optimal ϵ-PLA model, a key step of SegPQ's compression is to reorder all codewords based on their binary concatenation values. As such an extra sorting phase modifies the original vector ID information, here we assume a generic scenario where the original vector IDs can be safely updated, meaning that the IDs of associated objects can be re-assigned based on their new sorting indexes. The following example illustrates such scenarios.

Example 3.5 (RAG with SegPQ). In a RAG-powered domain-specific dialog system, an external knowledge database can be represented as a set of tuples (id, d, e_d), where id is the *original* document ID, d is the raw document to be retrieved, and e_d is the learned embedding for d. After training a PQ quantizer, we can compute the PQ codeword for each embedding e_d as c_d and reorder the entire database based on the binary concatenation of c_d (id will be reset in this step). Regarding physical storage, the original documents to be retrieved are usually stored on disk and possibly managed by some external index like B+-tree for fast access. For embedding vectors, only the compressed codewords (obtained using SegPQ) are kept in memory, enabling efficient in-memory kNN or semantic search, especially for resource-constrained devices. Since the documents on disk have already been reordered, we can correctly retrieve them based on the vector search result.

Connection and Difference to Learned Index. Let $\mathcal{K}$ denote a list of sorted keys and let $\mathcal{I} = \{1, 2, \cdots, |\mathcal{K}|\}$ be the corresponding index set. The indexing problem aim to find a mapping $g : \mathcal{K} \mapsto \mathcal{I}$ with controllable error ϵ, ensuring the exact location of any $k \in \mathcal{K}$ can be found correctly. Finding g is equivalent to fitting the CDF of $\mathcal{K}$. Conversely, the compression problem is to fit $f : \mathcal{I} \mapsto \mathcal{K}$ within error constraint ϵ, such that the original data $\mathcal{K}[i]$ can be losslessly recovered. Fitting f is equivalent to learning the inverse CDF (a.k.a., the quantile function) of $\mathcal{K}$.

Error Parameter. Intuitively, the value of error constraint ϵ can be neither too large nor too small. A large ϵ reduces the required segment count but increases the bits per residual; conversely, a small ϵ saves bits per residual at the cost of introducing more segments to meet the error constraint. Our theoretical analysis in Section 4 reveals that the relationship between compression ratio and ϵ is convex. An analytical optimal solution, as used in Line 3 of Algorithm 1, exists for ϵ to maximize SegPQ's compression efficacy.

Time Complexity. Reordering the key set (Line 2) takes time $O(N \log N)$ by invoking a standard sorting algorithm. Configuring ϵ^{OPT} in Line 3 can be done in constant time. Additionally, Line 4 fits an optimal ϵ-PLA in a single pass over the data (i.e., $O(N)$) by adopting the online algorithm [48]. Therefore, the total time complexity of Algorithm 1 is $O(N \log N + N) = O(N \log N)$.

3.3 Querying Compressed Codebooks

Due to the bijective nature of BinCat-based projection and error-bounded property of ϵ-PLA, the i-th codeword can be losslessly reconstructed by simply adding the PLA prediction to the i-th residual. The following running example illustrates the SegPQ's decompression process.

Example 3.6 (SegPQ Decompression). Continuing to Example 3.4, for the 1-st and 2-nd codewords C_1 and C_2, the corresponding PLA predictions are $\lfloor f(1) \rfloor = \lfloor 114.1 \times (1-1) + 255.1 \rfloor = 255$ and $\lfloor f(2) \rfloor = \lfloor 114.1 \times (2-1) + 255.1 \rfloor = 369$. The corresponding residuals are 1 and -2. Thus, C_1 and C_2 can be losslessly recovered by $C_1 = \mathrm{Split}(\mathrm{bin}(255-1)) = \mathrm{Split}(\mathtt{00FE}_{\mathrm{hex}}) = [0, 254]$ and $C_2 = \mathrm{Split}(\mathrm{bin}(369+2)) = \mathrm{Split}(\mathtt{0173}_{\mathrm{hex}}) = [1, 115]$.

Since SegPQ's decompression is *lossless*, the ANN query processing procedure remains *unchanged* on the decompressed codewords, where either SDC or ADC can be invoked. In this section, we introduce two SegPQ decompression utilities, random access (Section 3.3.1) and SIMD-aware full traversal (Section 3.3.2).

3.3.1 Random Access. The pseudo-code is given in Lines 1–7 of Algorithm 2. Specifically, Line 3 searches for the corresponding line segment given an index i. Line 5 computes the approximate key

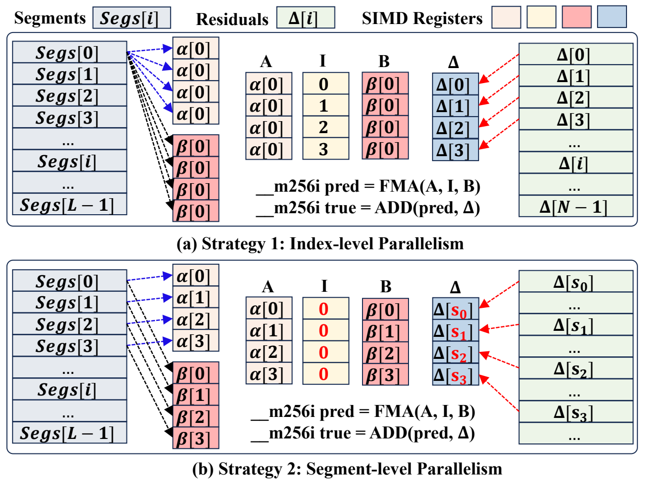


Figure 4: Illustration of SIMD-aware traversal: (a) Index-level parallelism, and (b) Segment-level parallelism. FMA(A, I, B)=A×I+B and (s_i, α_i, β_i) refers to the i-th segment $Segs[i]$.

predicted by the linear function. Finally, the corresponding residual $\Delta[i]$ is added to reconstruct the original codeword.

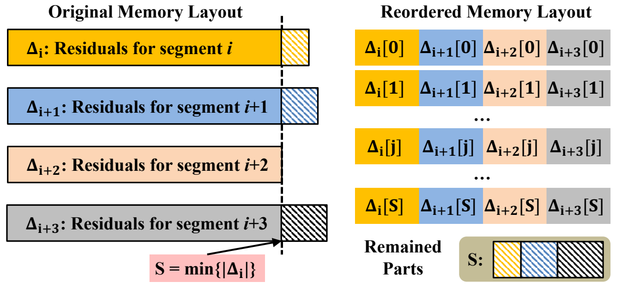
The segment search operation in Line 3 dominates the total decompressing cost, which takes $O(\log_2 L)$ by using a standard binary search implementation (e.g., std::lower_bound in C++), where L is the number of line segments. To speed up random access, one may further construct an in-memory index like B+-tree on the breakpoints of L segments. However, we do not consider constructing auxiliary index structures as it introduces extra storage and computation overhead, and random access is not frequently used in PQ-based ANN processing.

3.3.2 *SIMD-Aware Traversal.*

In practice, every codeword needs to be accessed to find the nearest neighbors to a query vector, either using ADC or SDC as discussed in Section 2. Therefore, traversing the entire codebook is more prevalent than random access. To accelerate traversal, we harness the hardware parallelism offered by *single instruction multiple data* (SIMD), a feature available on most modern CPUs[1]. SIMD vectorizes the input and executes *multiple* arithmetic operations within a single instruction. Specifically, each SIMD register of width 256 (or 512) can simultaneously process 8 (or 16) single-precision floats.

Depending on how SIMD parallelism is applied, we consider two strategies as depicted in Figure 4: ❶ the index-level parallelism where multiple indexes are processed simultaneously for each segment, and ❷ the segment-level parallelism where multiple segments are processed simultaneously for each index. Strategy ❷ generally saves computation because, for the same segment, after computing $f(i) = \alpha \cdot i + \beta$, the next prediction $f(i+1)$ can be made by reusing the result of $f(i)$ as $f(i+1) = f(i) + \alpha$. Due to the page limits, we put the code snippets for SIMD-aware codebook traversal in Appendix B.

[1]Customer-grade CPUs like Intel© Core™ and AMD© Ryzen™ typically provide 256-bit SIMD registers (i.e., AVX2), while enterprise CPUs like Intel© Xeon™ are equipped with wider 512-bit SIMD registers (i.e., AVX512). Besides SIMD parallelism on X86 platforms, we also plan to release an ARM-based version in the future.



Original Memory Layout
Δ_i: Residuals for segment i
Δ_{i+1}: Residuals for segment i+1
Δ_{i+2}: Residuals for segment i+2
Δ_{i+3}: Residuals for segment i+3
S = min{|Δ_i|}
Reordered Memory Layout
Δ_i[0] Δ_{i+1}[0] Δ_{i+2}[0] Δ_{i+3}[0]
Δ_i[1] Δ_{i+1}[1] Δ_{i+2}[1] Δ_{i+3}[1]
Δ_i[j] Δ_{i+1}[j] Δ_{i+2}[j] Δ_{i+3}[j]
Δ_i[S] Δ_{i+1}[S] Δ_{i+2}[S] Δ_{i+3}[S]
Remained Parts
S:

Figure 5: Illustration of the memory re-layout for segment-level parallelism (i.e., Strategy ❷ in Section 3.3). W.l.o.g., we assume that four segments are processed in parallel for each index starting from 0.

Unlike random access, traversing the entire codebook requires a total time of $O(N)$ and an amortized time of $O(1)$ per codeword, as no segment search operation is required.

3.3.3 *Details of Memory Re-layout.*

However, as shown in Figure 4(b), the residual array access pattern for Strategy ❷ is non-contiguous, resulting in additional overhead due to the more cache misses when loading data from memory to SIMD registers[2]. To improve spatial locality, we reorganize the residual array's memory layout based on the actual memory access order of Strategy ❷, such that the memory access latency can be well hidden.

To improve the cache utilization, as illustrated in Figure 5, the basic idea is to align the memory layout with the actual access order of residual array Δ. Let $\Delta_i, \cdots, \Delta_{i+3}$ denote the corresponding sub-arrays of residuals for segment $i, \cdots, i+3$. In segment-level parallelism, at each time, we process multiple segments for the same index starting from 0. Thus, the actual access order to $[\Delta_i, \cdots, \Delta_{i+3}]$ should be $\Delta_i[0], \cdots, \Delta_{i+3}[0]$, $\Delta_i[1], \cdots, \Delta_{i+3}[1], \cdots, \Delta_i[s], \cdots, \Delta_{i+3}[s]$, where $s = \min\{|\Delta_i|, \cdots, |\Delta_{i+3}|\}$. By reorganizing the memory layout of Δ in such an order, the latency of loading residuals from memory into SIMD register can be significantly reduced, given that modern CPUs prefetch consecutive data of a cache-line size (typically 64 bytes on mainstream architectures) with a single memory loading instruction. Notably, as the size of each sub-array $|\Delta_i|$ differs for each segment, there are inevitably some residuals left. These remaining residuals are materialized and processed sequentially after finalizing all indexes $0, \cdots, s$.

4 THEORETICAL ANALYSIS

In this section, we theoretically answer the question: *How much space can our SegPQ framework reduce, and at what cost?* The roadmap of our theoretical results is given below.

❶ Proposition 4.1 introduces our core observations on the distribution of PQ codewords;

❷ Lemma 4.3 derives the variance of the *codeword gap* distribution, serving as the building block of subsequent analysis;

❸ Theorem 4.4 presents our central theoretical results regarding the *expected* line segment coverage;

[2]Typical access latencies of L1 cache, last level cache (LLC), and main memory on modern architectures are 1 ns, 20 ns, and 100 ns, respectively.

❹ Based on Theorem 4.4, we further derive the space overhead and compression ratio of SegPQ in Corollary 4.6;
❺ Finally, Theorem 4.7 derives the optimal error parameter ϵ^{OPT} in a closed form, and Corollary 4.8 gives the corresponding bits per compressed codeword as $1.721 + \lceil \log_2 \epsilon^{\mathrm{OPT}} \rceil$.

4.1 Expected Segment Coverage

As the total storage overhead of SegPQ depends on the number of line segments required to satisfy the error constraint ϵ, we begin with the first theorem showing the *expected* number of codewords covered by *each* line segment.

Proposition 4.1 (Codeword Distribution). *Considering a PQ quantizer of parameters m (sub-space number) and k (cluster number), let* $\mathbf{q}_i$ *denote the quantized codeword for an arbitrary dense vector* $\mathbf{x}_i$. *Then, the binary concatenation* $\mathrm{BinCat}(\mathbf{q}_i)$ *follows a uniform distribution* $U(0, k^m)$.

Proposition 4.1 naturally holds as k-means is independently invoked on each partitioned sub-space when training PQ, leading to an equal likelihood for each bit of $\mathrm{BinCat}(\mathbf{q}_i)$ to be set. Section 5.2 reports the distributions of $\mathrm{BinCat}(\mathbf{q})$ on real datasets to validate this statement. Let $x_1, \cdots, x_N$ be N i.i.d. random samples drawn from a uniform distribution $U(0, k^m)$. We then define an important data feature named *codeword gap* and derive its mean and variance in Lemma 4.3. Notably, in the subsequent analysis, we interchangeably use x_i and $\mathrm{BinCat}(\mathbf{q}_i)$ if the context is clear.

Definition 4.2 (Gap). Given $X = \{x_1, \cdots, x_N\}$, the i-th gap for $i \in \{2, \cdots, N\}$ is defined as a random variable $g_i = x_{(i)} - x_{(i-1)}$ where $x_{(i)}$ and $x_{(i-1)}$ denote the i-th and $(i-1)$-th order statistics of X, respectively (i.e., the i-th and $(i-1)$-th smallest values of X).

Lemma 4.3 (Gap Distribution Characteristics). *Under Proposition 4.1, for an arbitrary* $i \in 2, \cdots, N$, *the mean and variance of gap* g_i *are given by,*

$$\begin{aligned} \mathbf{E}[g_i] &= \frac{k^m}{N+1}, \\ \mathbf{Var}[g_i] &= \frac{(k^m)^2 \cdot N}{(N+1)^2 \cdot (N+2)} = \Theta\left(\frac{k^{2m}}{N^2}\right), \end{aligned} \tag{4}$$

where m is the partition number, k is the k-means cluster number, and N is the total number of vectors.

Proof. Consider $U_1, \cdots, U_N$ are N i.i.d. random variables on range $[0, 1]$ and $U_{(1)}, \cdots, U_{(N)}$ are the corresponding order statistics. According to [18], $U_{(i)} - U_{(i-1)} \sim \mathbf{Beta}(1, N)$. Under Proposition 4.1, $g_i = x_{(i)} - x_{(i-1)} = k^m \cdot (U_{(i)} - U_{(i-1)})$, meaning that g_i also follows a beta distribution $g_i \sim k^m \cdot \mathbf{Beta}(1, N)$. Thus, the results in Eq. (4) can be easily obtained. □

Since $\mathbf{E}[g_i]$ and $\mathbf{Var}[g_i]$ are independent of i, we use $\mathbf{E}[g]$ and $\mathbf{Var}[g]$ to denote the mean and variance for an arbitrary g_i. Based on Lemma 4.3, we then analyze the expected line segment coverage.

Theorem 4.4 (Expected Segment Coverage). *Given a set of sorted values* $X^* = \{x_{(1)}, \cdots, x_{(N)}\}$ *and an error parameter satisfying* $\epsilon \gg \sqrt{\mathbf{Var}[g]}$, *the expected number of values in* X^* *covered by a line segment* $\ell(i) = \mathbf{E}[g] \cdot (i-1)$ *is,*

$$\mathbf{E}\left[\min\left\{i \in \mathbb{N}^+ \mid |\ell(i) - x_{(i)}| > \epsilon\right\}\right] = \Theta\left(\frac{\epsilon^2 \cdot N^2}{k^{2m}}\right). \tag{5}$$

Proof. We first obtain the result that the expected segment coverage is $\frac{\epsilon^2}{\mathbf{Var}[g]}$ by extending the theoretical results in [20]. Then, by combining Lemma 4.3, we have the result stated in Theorem 4.4. Please refer to Appendix C for detailed proof. □

4.2 Memory Footprint Analysis

Intuitively, Theorem 4.4 supports the rationale behind SegPQ, which is exemplified below.

Example 4.5 (SegPQ's Compression Efficacy). Considering a vector set of 1 million, we construct a PQ codebook with $m = 4$ and $k = 16$, where the gap variance is measured as 0.477. By allocating each residual 6 bits, i.e., $\epsilon = 2^{6-1} = 32$, the expected segment coverage can be computed as 1,024 using Eq. (5) (note that $32 \gg \sqrt{0.477} = 0.69$ satisfies the condition $\epsilon \gg \sqrt{\mathbf{Var}[g]}$ required by Theorem 4.4). Therefore, representing 1 million vectors requires approximately 1,000 segments and an array of 1 million 6-bit residuals. The compression ratio r can be roughly computed as,

$$r = \frac{16 \text{ Bit} \times 10^6}{(16 + 32 \times 2) \text{ Bit} \times 10^3 + 6 \text{ Bit} \times 10^6} \approx 2.6,$$

which is a significant memory reduction.

We then formally analyze the memory footprint and compression ratio of SegPQ based on Theorem 4.4.

Corollary 4.6 (SegPQ Space Cost). *Considering a PQ quantizer of parameters m and k,* $\mathbf{Q}$ *is the corresponding codebook of size* $N \times m$ *for N dense vectors. Then, SegPQ uses in total*

$$M(\epsilon) = \underbrace{N \cdot \lceil 1 + \log_2 \epsilon \rceil}_{\textit{residuals}} + \underbrace{L(\epsilon) \cdot (m \cdot \lceil \log_2 k \rceil + 2F)}_{\textit{segments}} \tag{6}$$

bits to compress the codebook $\mathbf{Q}$ *without any loss of information, where* ϵ *is the error parameter, F is the number of bits to represent a floating number (typically 32), and* $L(\epsilon)$ *is the number of required line segments, which is essentially a* monotonically decreasing function *of* ϵ. *The compression ratio between SegPQ's compressed codebook and the raw codebook* $\mathbf{Q}$, *denoted by* $r(\epsilon)$, *can be then derived as,*

$$r(\epsilon) = \frac{m \cdot \lceil \log_2 k \rceil}{\lceil 1 + \log_2 \epsilon \rceil + L(\epsilon) \cdot (m \cdot \lceil \log_2 k \rceil + 2F)/N}. \tag{7}$$

According to Theorem 4.4, in expectation[3], by setting the slope of a line segment to $\mathbf{E}[g]$, the number of line segments $L(\epsilon)$ can be derived as follows,

$$L(\epsilon) \propto \frac{k^{2m}}{N \cdot \epsilon^2}. \tag{8}$$

In practice, the *optimal* piecewise linear fitting technique [21, 48] is adopted in SegPQ to learn segments where the number of segments is guaranteed to be minimized under the error constraint ϵ. Thus, $L(\epsilon) = O(k^{2m}/N\epsilon^2)$.

[3]The conclusion is drawn hastily as $1/\mathbf{E}[Z] \neq \mathbf{E}[1/Z]$ for an arbitrary random variable Z. However, we can still attain the result $L(\epsilon) \propto N/\epsilon^2$ by following the procedures discussed in [20].

4.3 Optimal Error Parameter Configuration

We then discuss how to configure ϵ to minimize the space overhead described in Eq. (6). In practice, we collect some probe data to fit $\widetilde{L}(\epsilon) = C/\epsilon^2$ as an estimation for $L(\epsilon)$. For example, on the `SIFT` dataset, $\widetilde{L}(\epsilon) \approx 1.84 \times 10^{10} \cdot 1/\epsilon^2$. Further details on fitting $\widetilde{L}$ are provided in Section 5.2. By taking $\widetilde{L}(\epsilon) = C/\epsilon^2$ into Eq. (6), we can then derive an optimal error constraint ϵ that minimizes the space cost $M(\epsilon)$.

Theorem 4.7 (Optimal Parameter Setting). *The SegPQ space overhead $M(\epsilon)$ given in Corollary 4.6 is minimized by setting*

$$\epsilon^{\mathrm{OPT}} = \sqrt{\frac{2C \cdot \ln 2 \cdot (m \log_2 k + 2F)}{N}}, \tag{9}$$

and the corresponding residual bits $b^{\mathrm{OPT}} = \lceil 1 + \log_2 \epsilon^{\mathrm{OPT}} \rceil$.

Proof. Taking $\widetilde{L}(\epsilon) = C/\epsilon^2$ into $M(\epsilon)$ (i.e., Eq. (6)), we have,

$$M(\epsilon) = N \cdot \lceil 1 + \log_2 \epsilon \rceil + \frac{C \cdot (m \cdot \lceil \log_2 k \rceil + 2F)}{\epsilon^2}. \tag{10}$$

We then define two auxiliary functions $g(\epsilon)$ and $h(\epsilon)$:

$$\begin{aligned} g(\epsilon) &= N \cdot (1 + \log_2 \epsilon) + \frac{C \cdot (m \cdot \lceil \log_2 k \rceil + 2F)}{\epsilon^2}, \\ h(\epsilon) &= N \cdot (2 + \log_2 \epsilon) + \frac{C \cdot (m \cdot \lceil \log_2 k \rceil + 2F)}{\epsilon^2}. \end{aligned} \tag{11}$$

Clearly, $g(\epsilon) \le M(\epsilon) \le h(\epsilon)$. We then have the derivatives of $g(\epsilon)$ and $h(\epsilon)$ as follows,

$$\frac{\partial g}{\partial \epsilon} = \frac{\partial h}{\partial \epsilon} = \frac{N}{\epsilon \cdot \ln 2} - 2C \cdot \frac{m \cdot \lceil \log_2 k \rceil + 2F}{\epsilon^3}. \tag{12}$$

By setting Eq. (12) to zero, we have,

$$\frac{N \cdot \epsilon^2}{\ln 2} - 2C \cdot \left(m \cdot \lceil \log_2 k \rceil + 2F\right) = 0. \tag{13}$$

By solving Eq. (13), we have,

$$\epsilon^{\mathrm{OPT}} = \sqrt{\frac{2C \cdot \ln 2 \cdot \left(m \cdot \lceil \log_2 k \rceil + 2F\right)}{N}}. \tag{14}$$

For $\epsilon \in (0, \epsilon^{\mathrm{OPT}}]$, $\frac{\partial g}{\partial \epsilon} = \frac{\partial h}{\partial \epsilon} \le 0$, and for $\epsilon \in [\epsilon^{\mathrm{OPT}}, +\infty)$, $\frac{\partial g}{\partial \epsilon} = \frac{\partial h}{\partial \epsilon} \ge 0$. Thus, ϵ^{OPT} is the minimum point of both $g(\epsilon)$ and $h(\epsilon)$. As $M(\epsilon)$ is sandwiched by $g(\epsilon)$ and $h(\epsilon)$, ϵ^{OPT} is also the minimum point of $M(\epsilon)$, and we have the results shown in Theorem 4.7. □

Corollary 4.8 (Optimal Bits Per Codeword). *By setting ϵ^{OPT} as shown in Eq. (9), the optimal number of bits required to compress each PQ codeword is given by,*

$$\begin{aligned} \frac{M(\epsilon^{\mathrm{OPT}})}{N} &= \lceil 1 + \log_2 \epsilon^{\mathrm{OPT}} \rceil + \frac{1}{2 \ln 2} \\ &\approx 1.721 + \lceil \log_2 \epsilon^{\mathrm{OPT}} \rceil. \end{aligned} \tag{15}$$

By further expanding Eq. (15), the optimal bits per compressed codeword can be derived as $\log_2 \frac{k^m}{N} + O(\log_2 \frac{k^m}{N})$, nearly matching the information-theoretical lower bound [57].

Table 2: Statistics of evaluation datasets.

Dataset	#Vectors	#Dims	Raw Size	PQ Config
GIST	1,000,000	960	3.7 GB	PQ(6, 4)
Audio	438,229,156	128	211 GB	PQ(4, 8)
SIFT	1,000,000,000	128	123 GB	PQ(4, 8)
Deep1B	1,000,000,000	128	361 GB	PQ(4, 8)

5 EXPERIMENTAL STUDY

In this section, we report the experimental results to demonstrate the effectiveness and efficiency of SegPQ. Specifically, we aim to answer the following questions:
❶ whether the core propositions on PQ codewords' distribution hold and whether the experimental results support our major theoretical findings (▷ Section 5.2);
❷ whether the SegPQ framework can effectively reduce the space overhead of raw PQ codebooks (▷ Section 5.3);
❸ whether the SegPQ compressed codebook can be constructed and queried efficiently (▷ Section 5.3);
❹ whether the SegPQ framework can well adapt to other vector quantizers like AQ and OPQ (▷ Section 5.4).

5.1 Datasets and Setups

5.1.1 Benchmark Datasets. We adopt four large-scale datasets that are commonly evaluated in previous ANN studies. **Audio** is a set of 128-D audio features extracted from YouTube video frames [69]. **SIFT** and **GIST** are sets of 128-D SIFT/GIST image descriptors [26, 27]. **Deep1B** is a billion-scale dataset of pre-trained embeddings from a deep neural network [9], where each embedding vector is reduced to 96-D by PCA. Table 2 summarizes the statistics and default PQ configurations of the benchmark datasets.

5.1.2 Vector Quantization Methods. We adopt three different vector quantizers to construct the codebook: ❶ **PQ**, the classic product quantization [26]; ❷ **OPQ**, the optimized product quantization that minimizes the vector quantization distortions w.r.t. space partitioning and quantization codebooks [24]; ❸ **AQ**, the additive quantization that encodes dense vectors by the *sum* of multiple codebooks [8]. By default, PQ is used as the quantizer to generate codebooks, with dataset-specific configurations provided in Table 2. For OPQ and AQ, the hyper-parameters are adjusted to ensure the resulting codebook sizes match those of PQ. The vector quantization implementations are chosen from the commonly used Facebook's faiss library [19].

5.1.3 Compared Baselines. We implement and evaluate five compression methods: ❶ **SegPQ**, the full-fledged SegPQ framework with SIMD-based traversal acceleration (by default, the segment-level parallelism is enabled); ❷ **DeltaPQ** [65], the state-of-the-art PQ codebook compression method based on indexing similar codewords' difference; ❸ **LZ4** [37], a widely used lossless compressor for byte stream, and we configure LZ4 to achieve the highest compression ratio (i.e., `LZ4 -9`); ❹ **LZMA** [51], the Lempel-Ziv-Markov chain algorithm, which is another lossless compression scheme optimized for compression ratio; ❺ **BUFF** [33], a generic lossless

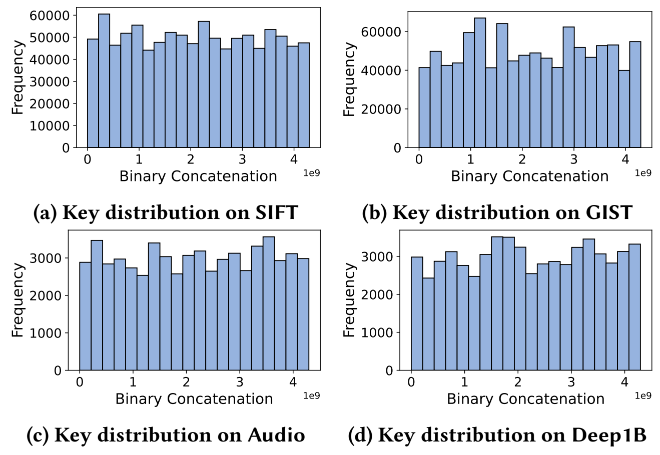


Figure 6: Validation of Proposition 4.1 that the binary concatenations of raw PQ codewords are uniformly distributed.

compressor designed for bounded floats. Additionally, we also compare ❻ **Bolt** [10], another vector quantizer that tradeoffs space overhead and accuracy, and ❼ **PQFS** [3], an efficient PQ codebook traversal algorithm for fast ANN processing.

Notably, in this experimental study, we focus on the compression efficacy of our proposed methods over a wide range of vector quantization codebooks. We do not choose to compare other ANN indexes like LSH [2] or HNSW [39] variants as they are orthogonal to our work and have been intensively evaluated in previous ANN benchmarks such as [7, 32].

5.1.4 Environment Setups. All compared methods are implemented in C++ and compiled by g++ 11 with optimization level O2. All experiments are conducted on a Linux server with an AMD EPYC 7413 CPU and 512 GB of main memory. In the subsequent evaluations, we primarily focus on memory footprint (compression ratio), compression time, and query processing time as the key metrics. Commonly used ANN performance metrics like Recall@K are not reported, as our SegPQ is a *lossless* compression technique, meaning that there is no influence on the ANN search results.

5.2 Validation of Theoretical Results

5.2.1 Binary Codeword Distribution. We first validate Proposition 4.1 on codeword distribution. Figure 6 shows the histograms of PQ codewords after binary concatenation (i.e., the projection step of SegPQ) on various datasets. The results reveal that the projected codewords exhibit *approximate* uniform distributions across multiple datasets. Despite not being perfectly uniform, the subsequent evaluations indicate that slight differences in distribution *do not* affect the correctness of the theoretical results. We reasonably postulate that Proposition 4.1 can be further relaxed to an *arbitrary* distribution with finite variance, which is left for our future work.

To quantitatively validate Proposition 4.1, we further compute the Wasserstein distances [1] between the observed distributions and a uniform distribution $U(0, k^m)$. The results are 0.0075, 0.0185, 0.0176, 0.0119 for SIFT, GIST, Audio, and Deep1B, respectively, demonstrating the closeness between the projected codewords' distribution and the uniform distribution $U(0, k^m)$.

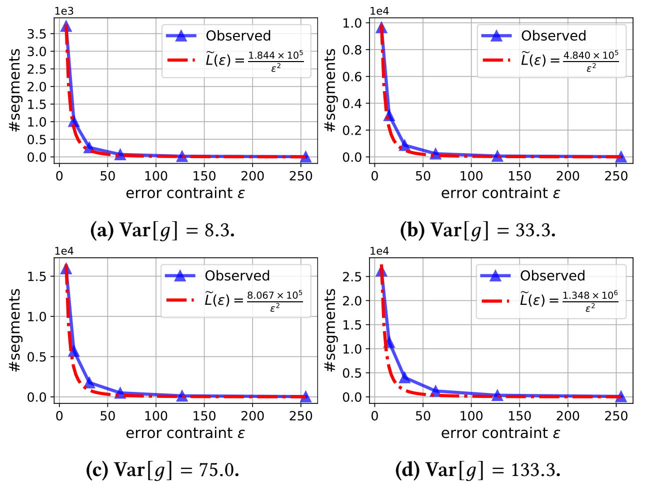


Figure 7: The number of required line segments w.r.t. error constraint ϵ on SIFT with different levels of variance. Blue solid lines refer to the observed results, and red dashed lines refer to the estimated curve $\widetilde{L}(\epsilon) = C/\epsilon^2$.

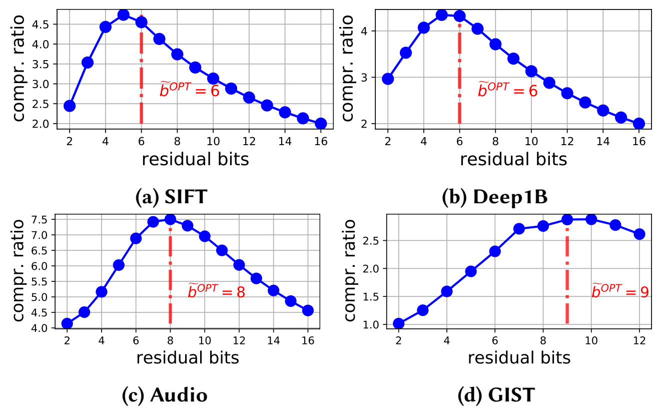
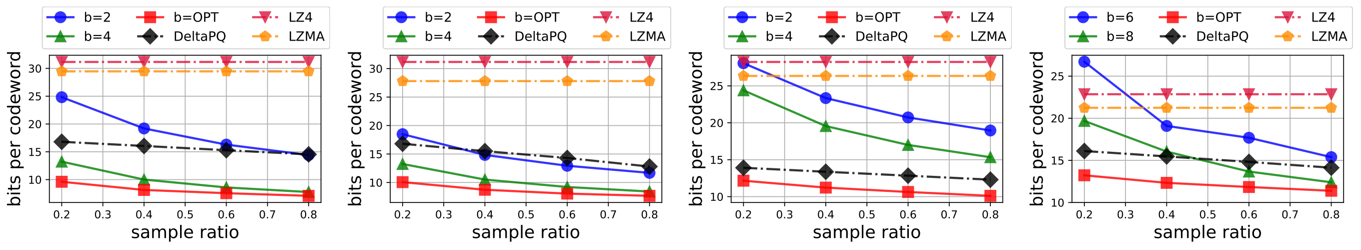


Figure 8: Compression ratios w.r.t. residual bits for storing each residual (b) on 4 datasets SIFT, Deep1B, Audio, and GIST.

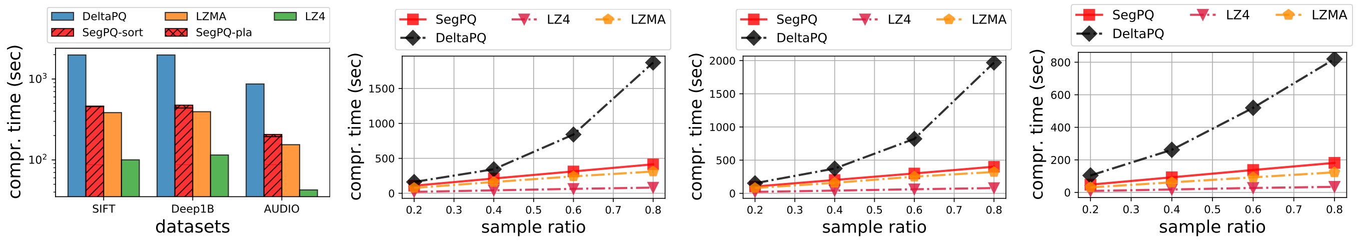
5.2.2 Average Segment Coverage. We then empirically verify our core theoretical results on the expected number of codewords covered by a line segment (i.e., Theorem 4.4). To do this, we construct four datasets, each consisting of 10^5 vectors sampled from SIFT, with varying gap variances $\textbf{Var}[g] \in \{8.3, 33.3, 75.0, 133.3\}$. Figure 7 illustrates the number of line segments $L(\epsilon)$ w.r.t. the error parameter ϵ, where an inverse relationship can be consistently observed. We then fit $\widetilde{L}(\epsilon) = \frac{C}{\epsilon^2}$ using the observed points. The results indicate that $C \propto N\textbf{Var}[g]$, which is in alignment with Theorem 4.4 and Corollary 4.6. As the optimal error configuration (i.e., Eq. (9) in Theorem 4.7) requires C as input, by extensive testing, we adopt $\widetilde{C} \approx 1.734N\textbf{Var}[g] = 1.734 \cdot \frac{k^{2m}}{N}$ as a robust estimation. Experimental results presented in Section 5.2.3 demonstrate the effectiveness of the optimal parameter setting strategy based on this estimation.

5.2.3 Effectiveness of Optimal Parameter Setting. Figure 8 shows the results of compression ratio w.r.t. different bits (b) allocated for



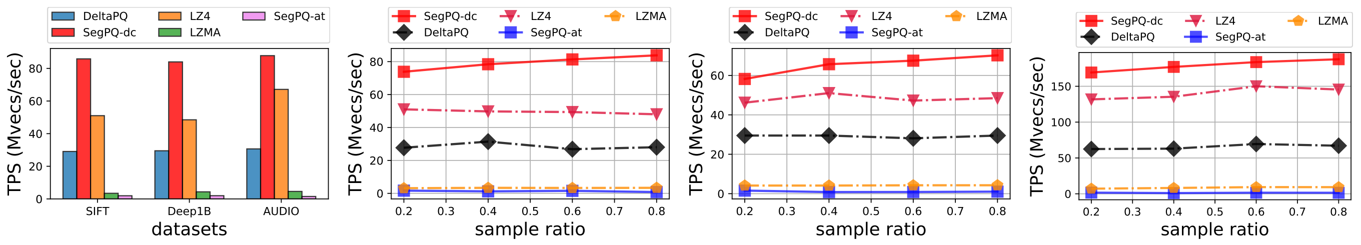
(a) SIFT: memory v.s. s (b) Deep1B: memory v.s. s (c) Audio: memory v.s. s (d) GIST: memory v.s. s

Figure 9: Bits per codeword (BPC) w.r.t. different sample ratios s on four datasets. Solid lines refer to our SegPQ with different error bits settings, and b=OPT refers to the optimized error parameter selection strategy using Eq. (9).



(a) Overall comparison (b) SIFT: compression time (c) Deep1B: compression time (d) Audio: compression time

Figure 10: Evaluation results on codebook compression time. Note the logarithm scale in Figure 10a.



(a) Overall comparison (b) SIFT: throughput v.s. s (c) Deep1B: throughput v.s. s (d) Audio: throughput v.s. s

Figure 11: Evaluation results on codebook decompression throughput (unit: million vectors per second).

each residual. Note that, maximizing the compression ratio is equivalent to minimizing $M(\epsilon)$ as in Theorem 4.7. The red dashed lines mark the optimal error bits $\widetilde{b}^{\mathrm{OPT}}$ estimated by Theorem 4.7. Our estimation $\widetilde{b}^{\mathrm{OPT}}$ consistently approximates the observed optimum across all datasets in Figure 8, which validates ❶ the correctness of our optimal parameter setting strategy as described in Theorem 4.7 and ❷ the effectiveness of constant C configured previously.

5.3 SegPQ Evaluation Results

5.3.1 Memory Footprint. To demonstrate the power of memory reduction for SegPQ, we report the average bits per codeword (BPC) by varying different data sizes (controlled by a sample ratio $s \in \{0.2, 0.4, 0.6, 0.8\}$ for each dataset). According to the PQ configuration detailed in Table 2, without compression (i.e., the raw codewords), BPC for SIFT, Deep1B, and Audio is 32, and BPC for GIST is 24. The results presented in Figure 9 reveal that our SegPQ with b^{OPT} consistently outperforms all compared methods. For the two billion-scale datasets SIFT and Deep1B, SegPQ with optimal configuration can robustly compress each codeword to **less than 10 bits**, which is up to **1.78**×, **3.62**×, **4.07**×, and **4.18**× smaller compared to DeltaPQ, LZMA, LZ4, and the original PQ, respectively. The reason for SegPQ's superior performance over the current SOTA method, DeltaPQ, is that our SegPQ can fully utilize the information hidden behind codewords' distribution. With such a high compression ratio, the PQ codebooks for billion-scale vector sets SIFT, Deep1B, and Audio are reduced to **838 MB**, **882 MB**, and **508 MB**, respectively. Given a typical idle RAM for edge devices (**≤2 GiB**), as illustrated in Figure 1, these resulting codebooks can easily fit into most mainstream mobile and even embedded devices. Thus, the challenging objectives from Section 1 are achieved via our SegPQ.

Besides, from Figure 9, BPC for our SegPQ decreases as s (or data size N) grows. This is because according to Theorem 4.4 and Corollary 4.8, when other parameters are fixed, the optimal BPC is inversely proportional to $\sqrt{N}$.

Table 3: Evaluation of ANN query processing efficiency on the Deep1B dataset. The bits per raw codeword is configured to 32. The query time refers to the total time of running 1,000 top-50 queries.

Method	Memory Unit: GiB	Index Time Unit: sec	Query Time Unit: sec
PQ32	3.8	1199	60.73
SegPQ+PQ32	**0.91 (↓322%)**	1299 (↑8.3%)	64.52 (↑6.2%)
OPQ32	3.8	2220	85.95
SegPQ+OPQ32	**0.84 (↓352%)**	2322 (↑4.6%)	90.04 (↑4.8%)
Bolt	3.0	8344	789.84
BUFF+PQ32	2.4	1441	149.56
DeltaPQ	1.4	3941	132.27
PQFS32	3.8	1635	10.93

5.3.2 Compression Efficiency. Figure 10 reports the scalability of codebook compression across various datasets. From the results, LZ4 is the most efficient algorithm that can process all billion-scale data within 100 seconds, despite having the worst compression ratio according to Section 5.3.1. SegPQ is comparable to LZMA and slightly worse than LZ4, where compressing 1 billion codewords takes **414 seconds**. Such a construction cost is acceptable given that training a PQ(4, 8) codebook using faiss takes 12,390 seconds, which is about **30×** larger than SegPQ compression time. Overall, SegPQ, LZ4, and LZMA can well scale to billion-scale datasets; however, DeltaPQ, the SOTA codebook compression technique, takes more than half an hour to compress 1 billion vectors and shows super-linear growth as the data size increases. This is because, DeltaPQ is based on indexing differences between codewords, which grows much faster as data size scales.

Additionally, according to Figure 10a, the dominant part of SegPQ construction is reordering all codewords based on binary concatenation results (i.e., Step ❷ in Section 3.1). In this work, we simply adopt std::sort, which can be improved by using some distribution-aware sorting algorithms as the codewords distribution is known.

5.3.3 Decompression Efficiency. We then investigate the decompression efficiency of compressed codebooks. Figure 11 illustrates the processing throughput (measured in million vectors per second) w.r.t. different data sizes. Among all datasets, SegPQ's SIMD-aware traversal (i.e., SegPQ-dc in Figure 11) can decompress 1 billion codewords in **11.65 seconds**, which consistently outperforms LZ4, DeltaPQ, and LZMA by up to **1.75×**, **3.03×**, and **24.98×**, respectively. This performance gap can be further widened by leveraging multi-threading, which is ignored in this work as all the other baselines are not optimized for multiple threads. Besides, though as slow as LZMA, SegPQ is the only method that supports random access (i.e., SegPQ-at in Figure 11) to any location within the compressed codebook.

An interesting observation is that the throughput of SegPQ slightly **increases** as the data size grows. This can be explained as a larger data size results in fewer average line segments, based on Theorem 4.4, ultimately requiring less time to access the segments.

Table 4: Evaluation of SegPQ+X (X can be PQ, OPQ, and AQ) with various configurations on the Deep1B dataset. PQ16, PQ32, and PQ64 (similar for OPQ and AQ) indicate that the quantized vectors are using 16, 32, and 64 bits, respectively. The query time refers to the total time required to execute 1,000 top-50 queries. AQ32 and AQ64 fail to construct due to runtime memory overflow.

Method	Memory Unit: GiB	Index Time Unit: sec	Query Time Unit: sec
PQ16	1.9	436	53.92
OPQ16	1.9	691	56.26
AQ16	1.9	78242	121.52
SegPQ+PQ16	**0.23 (↓715%)**	494 (↑12.5%)	56.17 (↑4.2%)
SegPQ+OPQ16	**0.23 (↓715%)**	749 (↑8.4%)	58.51 (↑4.0%)
SegPQ+AQ16	**0.24 (↓692%)**	78351 (↑0.1%)	126.48 (↑4.1%)
PQ32	3.8	1199	60.73
OPQ32	3.8	2220	85.95
AQ32	-	-	-
SegPQ+PQ32	**0.91 (↓322%)**	1299 (↑8.3%)	64.52 (↑6.2%)
SegPQ+OPQ32	**0.84 (↓352%)**	2322 (↑4.6%)	90.04 (↑4.8%)
SegPQ+AQ32	-	-	-
PQ64	7.5	543	115.31
OPQ64	7.5	579	208.14
AQ64	-	-	-
SegPQ+PQ64	**2.95 (↓154%)**	672 (↑23.8%)	119.81 (↑3.9%)
SegPQ+OPQ64	**3.39 (↓121%)**	716 (↑23.6)	213.14 (↑2.4%)
SegPQ+AQ64	-	-	-

5.3.4 ANN Query Efficiency. We further evaluate the overall ANN query processing efficiency using SegPQ. Compared to PQ and OPQ, SegPQ achieves a substantial space reduction of up to **352%**, with only minor increases in index and query costs, capped at **8.3%** and **6.2%**, respectively. More results on integrating SegPQ with other vector quantization codebooks are discussed in Section 5.4. PQFS demonstrates the highest query efficiency, as its codebook structure is specifically designed for fast ANN query processing. However, PQFS incurs the same high memory overhead as standard PQ. We further compare with other vector compression techniques, including Bolt, BUFF, and DeltaPQ. The results show that SegPQ achieves the best trade-off between index memory overhead and ANN query efficiency.

5.4 Evaluation on SegPQ+X

We perform supplementary experiments to showcase the query efficiency of SegPQ when integrated with PQ, OPQ, and AQ across different quantization bit settings (16/32/64). The results on Deep1B are given in Table 4. In summary, the results indicate that SegPQ is lightweight, with a minor impact (typically **less than 10%**) on both PQ index construction and ANN query processing efficiency. This is because ❶ our SegPQ decompression algorithm fully utilizes hardware resources for acceleration, and ❷ PQ kNN query processing does not require decompressing **all** codewords, meaning that PQ distance computation can be well pipelined with SegPQ

Table 5: Evaluation of SegPQ+X under various error bit configurations, where X can be PQ32 or OPQ32.

Error Bits	Method	Memory Unit: GiB	Query Time Unit: sec
N.A.	PQ	3.8	60.73
	OPQ	3.8	80.95
4	SegPQ+PQ	0.92 (↓313%)	64.01 (↑5.4%)
($\epsilon = 8$)	SegPQ+OPQ	1.09 (↓249%)	84.42 (↑4.3%)
8	SegPQ+PQ	1.00 (↓280%)	63.57 (↑4.7%)
($\epsilon = 128$)	SegPQ+OPQ	1.01 (↓276%)	83.80 (↑3.5%)
12	SegPQ+PQ	1.40 (↓171%)	64.02 (↑5.4%)
($\epsilon = 2048$)	SegPQ+OPQ	1.41 (↓170%)	84.18 (↑4.0%)
16	SegPQ+PQ	1.86 (↓104%)	63.83 (↑5.1%)
($\epsilon = 32768$)	SegPQ+OPQ	1.86 (↓104%)	83.99 (↑3.8%)

decompression. Besides, such extra overhead caused by decompression becomes more negligible when processing a large batch of queries (e.g., 1K in the experiments) as multiple queries can re-use the already decompressed codes. Results on datasets other than Deep1B are similar and thus are omitted here due to page limits.

We further evaluate SegPQ+PQ and SegPQ+OPQ under different error bit settings. Results for SegPQ+AQ are similar and omitted due to space constraints. The results are given in Table 5. In summary, integrating SegPQ with PQ and OPQ saves up to **313%** space while introducing **less than 6%** extra query overhead as ϵ varies across a wide range, supporting our above discussions.

5.5 Summary of Results

Finally, we summarize the major experimental observations. ❶ SegPQ robustly achieves the highest compression ratio among all baselines and is notably the only method that consumes memory **less than 1 GiB** for 1 billion vectors. ❷ The construction of SegPQ is scalable, and its overhead is negligible (**less than 3.4%**) compared to PQ training time. ❸ Querying SegPQ's compressed codebook is highly efficient, with a minor effect on the overall ANN query processing (**less than 6%**). Especially, when compared to the SOTA PQ codebook compression method, DeltaPQ [65], our SegPQ demonstrates superiority in all three dimensions: memory footprint, construction efficiency, and query efficiency.

6 RELATED WORK

In this section, we review related works from three perspectives: ❶ LLMs, RAG and vector databases, ❷ ANN indexing, and ❸ learned data indexing.

LLM and Edge AI. Recent advancements in LLMs [16, 47, 61] have revolutionized NLP tasks, demonstrating transformative power of Transformer architectures [63] with billion-scale parameters. Further highlighted by scaling laws [29], these models have grown to unprecedented sizes, exhibiting unique behaviors and emergent abilities across various tasks. However, a paradigm shift has recently occurred, with these previously unwieldy models being deployed on *constrained computing environments* such as mobile phones [36, 45, 46]. This hybrid mobile-cloud deployment has spurred both academia and industry [4, 15, 25, 60] to explore methods for harnessing the model's power. To facilitate responsible on-device AI that prioritizes personalization and privacy protection, Apple's foundational model [4] leverages adapters alongside a suite of optimization techniques. Meanwhile, Google's Gemma 2B [25] emphasizes broad hardware compatibility, optimized for deployment across various edge AI environments. Nonetheless, these are newly proposed systems that come with heavy optimization and complex workflow.

RAG and VectorDB. In parallel, integral techniques like VectorDB and RAG with the ANN search engine [6, 22, 28, 50] are investigated to enhance LLM's performance. VectorDB enables efficient storage and retrieval of high-dimensional data, while ANN facilitates similarity searches. RAG [22] represents a significant advancement by combining parametric LLMs with non-parametric datastores, leveraging the strengths of VectorDB and ANN to provide efficiency, dynamic knowledge updating, and enhanced explainability. Compared to LLMs, these techniques provide a lightweight alternative for innovation on resource-constrained devices.

ANN Indexing. Nearest neighbor (NN) problems on high dimensional data have been studied for decades [5, 32, 35]. Exact NN solutions typically include *k*d-tree variants [53, 54] and metric tree variants [13, 68]. These methods are hard to scale to billion-scale vector databases due to the prohibitive space and time complexities. To handle web-scale vectors, ANN techniques have been intensively studied, which can be categorized into locality-sensitive hashing (LSH) based [2, 17, 34], proximity graph based [39, 40], and vector quantization based [24, 26, 66] approaches. Among the various ANN techniques, PQ variants are generally adopted by mainstream VectorDB products [19, 43, 52] as coarse quantizers and jointly used with other indexes. Our SegPQ is technically orthogonal to existing ANN methods and can be *seamlessly* applied to *any* PQ variants to obtain a lossless memory reduction.

Learned Models as Indexes. A recent tendency across machine learning, database, and system communities is to integrate machine learning models with conventional data structures to improve space/time efficiency, such as B+-tree [21, 30], hash tables [56], and Bloom filters [44], leading to the concept named *learned index*. Recent theoretical studies [20] demonstrate the effectiveness of piecewise linear models in fitting distributions with controllable error. In our work, we extend the results in [20] to the case of learned PQ codebook compression by fully utilizing the distribution information behind PQ codewords. In our technical report [59], we discuss more details on the connections to learned indexes.

7 CONCLUSION AND FUTURE WORK

With an outlook to the broader context of edge-AI, this work proposes SegPQ to *further losslessly* compress the PQ codebook. Simple yet non-trivial theoretical results are established to demonstrate SegPQ's efficacy. Extensive evaluation on billion-scale vector search scenarios reveals that SegPQ can reduce the raw codebook size to **less than 1 GiB** with **negligible** computation overhead introduced for 1B vectors. Our future work aims to embed SegPQ into prevalent VectorDB engines, facilitating seamless integration of vector search engines by custom choice, right on edge devices.

A SIMD-AWARE TRAVERSAL WITH STRATEGY 1

The codebook traversal with index-level parallelism is given below. For each line segment, we parallelize the computation of multiple input indices using SIMD. In the subsequent code snippets, we adopt AVX2 where 8 float32 or 4 float64 can be processed in parallel.

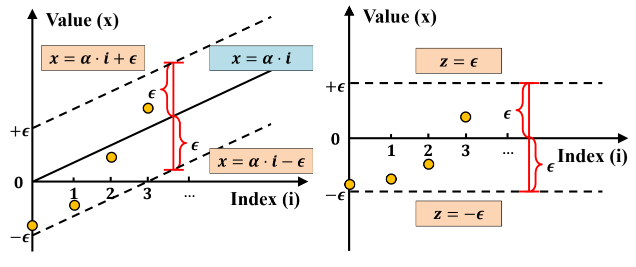
```
#include <x86intrin.h>
// ...
std::vector<Segment> segments;
std::vector<err_t> residuals;
// ...
void decodec_s1(CodeType *codes) {
  for (auto i=0; i<segments.size()-1; ++i) {
    Segment seg = segments[i];
    auto cover = segments[i+1].s - seg.s;
    __mm256 a = _mm256_set1_ps(seg.a);
    __mm256 b = _mm256_set1_ps(seg.b);
    for (auto j=0; j<cover; j+=8) {
      __mm256 idx = _mm256_set_ps(j+7, j+6, j+5, j+4, j+3, j+2, j+1, j);
      __mm256 approx = _mm256_fmadd_ps(a, idx, b);
      __mm256 r = _mm256_loadu_ps(&residuals[seg.s+j]);
      __mm256 q = _mm256_add_ps(_mm256_floor_ps(approx), r);
      collect(q);
    }
    // process the remaining indices
    // ...
  }
}
```

B SIMD-AWARE TRAVERSAL WITH STRATEGY 2

The codebook traversal with index-level parallelism is given below. For each index, we parallelize the computation for multiple line segments using SIMD. Notably, the residual array is reorganized as shown in Appendix 3.3.3 to reduce the memory access latency.

```
#include <x86intrin.h>
// ...
std::vector<float> slopes;
std::vector<float> covers;
std::vector<float> intercepts;
std::vector<err_t> residuals; // after re-layout
// ...
void decodec_s2(CodeType *codes) {
  for (auto i=0; i<segments.size()-1; i+=8) {
    // compute 1st linear approximation f(0)
    __mm256 a = _mm256_loadu_ps(&slopes[i]);
    __mm256 b = _mm256_loadu_ps(&intercepts[i]);
    __mm256 idx = _mm256_set1_ps(0);
    __mm256 approx = _mm256_fmadd_ps(a, idx, b);
    __mm256 r = _mm256_loadu_ps(&residuals[0]);
    __mm256 q = _mm256_add_ps(_mm256_floor_ps(approx), r);
    collect(q);

    // compute the min segment coverage
    auto mc = std::min_element(&covers[i], &covers[i+7]);
    // compute f(1), f(2), ...
    for (auto j=1; j<mc; ++j) {
      // f(idx+1) = f(idx) + a
      approx = _mm256_add_pd(approx, a);
      r = _mm256_loadu_ps(&residuals[j*8]);
      __mm256 q = _mm256_add_ps(_mm256_floor_ps(approx), r);
      collect(q);
    }
    // process the remaining indices
    // ...
  }
  // process the remaining segments
  // ...
}
```



(a) Relationship $x = \alpha \cdot i$. **(b) Transformation $z = x - \alpha \cdot i$.**

Figure 12: Illustration of the mean exit time (MET) problem [20]. Without loss of generality, we consider a line segment without intercept and consider indices starting from 0. After transformation $z = x - \alpha \cdot i$, the line segment coverage problem will be transformed to the Mean Exit Time estimation out of the error interval $[-\epsilon, \epsilon]$.

C PROOF OF THEOREM 4.4

We obtain Theorem 4.4 on the top of the results presented in the first theoretical result for *learned index* [20]. To investigate the expected segment coverage, w.l.o.g, we consider monotone points $(i, x_i) \in \mathbb{R}^2$ with indices i starting from 0, i.e., $\mathcal{X} = \{(0, x_0), (1, x_1), (2, x_2), \cdots\}$. We aim at finding the number of points between two lines $x = \alpha \cdot i + \epsilon$ and $x = \alpha \cdot i - \epsilon$ where α is the slope and ϵ is the pre-specified error parameter. Similar to [20], as illustrated in Figure 12, a linear transformation $z = x - \alpha \cdot i$ is performed to project $\mathcal{X}$ to $\mathcal{Z} = \{(i, z_i) | i = 0, 1, 2, \cdots\}$ where

$$\begin{aligned} z_i = x_i - \alpha \cdot i &= \sum\nolimits_{j=0}^{i} (x_j - x_{j-1}) - \alpha \cdot i \\ &= \sum\nolimits_{j=1}^{i} (g_j - \alpha) \\ &= \sum\nolimits_{j=1}^{i} h_j. \end{aligned} \quad (16)$$

Then, $\mathcal{Z}$ can be regarded as a realized random process with increments $h_j = g_j - \alpha$ where g_j is the *gap* defined in Definition 4.2. The expected coverage can be modeled as the following *mean exit* *time* [20, 23, 42, 55] estimation problem,

$$\mathbf{E}[i^*] = \mathbf{E}\left[\min\left\{i \in \mathbb{N} | z_i > \epsilon \vee z_i < -\epsilon\right\}\right]. \tag{17}$$

By setting $\alpha = \mathbf{E}[g_j]$, the expectation of $h_j = g_j - \alpha$ in Eq. (16) is 0 and the variance $\mathbf{Var}[h_j] = \mathbf{Var}[g_j - \alpha] = \mathbf{Var}[g_j]$. According to the theoretical results on MET [20, 23, 42, 55], when $\epsilon \gg \sqrt{\mathbf{Var}[h_j]}$, with high probability, the MET of $\mathcal{Z}$ out of interval $[-\epsilon, \epsilon]$ is

$$\mathbf{E}[i^*] = \frac{\epsilon^2}{\mathbf{Var}[g_j]}. \tag{18}$$

According to Lemma 4.3, gaps g_j follow a Beta distribution $k^m \cdot \mathbf{Beta}(1, N)$, and by taking $\mathbf{Var}[g_j] = \Theta\left(\frac{k^{2m}}{N^2}\right)$ into Eq. (18), we have

$$\mathbf{E}[i^*] = \frac{\epsilon^2}{\Theta\left(\frac{k^{2m}}{N^2}\right)} = \Theta\left(\frac{\epsilon^2 \cdot N^2}{k^{2m}}\right). \tag{19}$$

Thus, we achieve the final result in Theorem 4.4. □

Notably, in Theorem 4.4, we analyze the expected segment coverage on a dataset $\mathcal{X} = \{(0, x_0), (1, x_1), (2, x_2), \cdots\}$, which is the inverse version of the problem studied in [20] on a dataset $\mathcal{X}' = \{(x_0, 0), (x_1, 1), (x_2, 2), \cdots\}$. Intuitively, fitting $\mathcal{X}'$ is equivalent to fitting the cumulative function for $\{x_0, x_1, \cdots\}$, whereas fitting $\mathcal{X}$ is equivalent to learning the inverse cumulative function (a.k.a. quantile function). Such an inverse problem results in totally different theoretical results. On $\mathcal{X}'$, according to [20], the expected coverage is $\mathbf{E}[i^*] = (\epsilon^2 \cdot \mathbf{E}[g])/\mathbf{Var}[g]$; however, on $\mathcal{X}$, according to Eq. (18), the expected segment coverage is $\mathbf{E}[i^*] = \epsilon^2/\mathbf{Var}[g]$, which is independent of gap mean $\mathbf{E}[g]$. Another recent theoretical analysis [11] also studies the potential of PLA in data compression. However, they directly adopt the results in [20], where the term $\mathbf{E}[g]$ should not be included. Based on the theoretical results tailored to PQ codewords' distribution characteristics, we have the opportunity to further optimize the PLA configuration such that the compression ratio can be maximized.

REFERENCES

[1] Luigi Ambrosio, Nicola Gigli, and Giuseppe Savaré. 2008. *Gradient flows: in metric spaces and in the space of probability measures*. Springer Science & Business Media.

[2] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *FOCS*. IEEE Computer Society, 459–468.

[3] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In *42nd International Conference on Very Large Data Bases*, Vol. 9. 12.

[4] apple-flm [n.d.]. Apple Intelligence Foundation Language Models. https://machinelearning.apple.com/papers/apple_intelligence_foundation_language_models.pdf. Accessed: 2024-07-31.

[5] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. 1998. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *J. ACM* 45, 6 (1998), 891–923.

[6] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Retrieval-based Language Models and Applications. In *ACL (tutorial)*. Association for Computational Linguistics, 41–46.

[7] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020).

[8] Artem Babenko and Victor Lempitsky. 2014. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 931–938.

[9] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *CVPR*. IEEE Computer Society, 2055–2063.

[10] Davis W. Blalock and John V. Guttag. 2017. Bolt: Accelerated Data Mining with Fast Vector Compression. In *KDD*. ACM, 727–735.

[11] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2022. A Learned Approach to Design Compressed Rank/Select Data Structures. *ACM Trans. Algorithms* 18, 3 (2022), 24:1–24:28.

[12] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2022. A learned approach to design compressed rank/select data structures. *ACM Transactions on Algorithms (TALG)* 18, 3 (2022), 1–28.

[13] Tolga Bozkaya and Z. Meral Özsoyoglu. 1997. Distance-Based Indexing for High-Dimensional Metric Spaces. In *SIGMOD Conference*. ACM Press, 357–368.

[14] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[15] Jin Chen, Zheng Liu, Xu Huang, et al. 2024. When large language models meet personalization: perspectives of challenges and opportunities. *World Wide Web (WWW)* 27, 4 (2024), 42.

[16] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al. 2023. PaLM: Scaling Language Modeling with Pathways. *J. Mach. Learn. Res.* 24 (2023), 240:1–240:113.

[17] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*. ACM, 253–262.

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

[19] faiss [n.d.]. Faiss. https://faiss.ai/. Accessed: 2024-06-12.

[20] 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.

[21] 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.

[22] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. *CoRR* abs/2312.10997 (2023).

[23] Crispin W Gardiner. 1985. Handbook of stochastic methods for physics, chemistry and the natural sciences. *Springer series in synergetics* (1985).

[24] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755.

[25] gemma2b [n.d.]. Smaller, Safer, More Transparent: Advancing Responsible AI with Gemma. https://developers.googleblog.com/en/smaller-safer-more-transparent-advancing-responsible-ai-with-gemma/. Accessed: 2024-08-02.

[26] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

[27] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*. IEEE, 861–864.

[28] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12 (2023), 248:1–248:38.

[29] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *CoRR* abs/2001.08361 (2020).

[30] 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.

[31] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. 1998. Efficient Search for Approximate Nearest Neighbor in High Dimensional Spaces. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, Jeffrey Scott Vitter (Ed.). ACM, 614–623.

[32] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.

[33] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. 2021. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2586–2598.

[34] Haomiao Liu, Ruiping Wang, Shiguang Shan, and Xilin Chen. 2016. Deep Supervised Hashing for Fast Image Retrieval. In *CVPR*. IEEE Computer Society, 2064–2072.

[35] Ting Liu, Andrew W. Moore, Alexander G. Gray, and Ke Yang. 2004. An Investigation of Practical Approximate Nearest Neighbor Algorithms. In *NIPS*. 825–832.

[36] llamacpp [n.d.]. LLM Inference in C/C++. https://github.com/ggerganov/llama.cpp. Accessed: 2024-06-12.

[37] LZ4 [n.d.]. Etremely fast compression. https://lz4.org/. Accessed: 2024-06-12.

[38] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.

[39] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.

[40] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[41] Julieta Martinez, Holger H Hoos, and James J Little. 2014. Stacked quantizers for compositional vector compression. *arXiv preprint arXiv:1411.2173* (2014).

[42] Jaume Masoliver, Miquel Montero, and Josep Perelló. 2005. Extreme times in financial markets. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 71, 5 (2005), 056130.

[43] milvus [n.d.]. The High-Performance Vector Database Built for Scale. https://milvus.io/. Accessed: 2024-06-12.

[44] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *NeurIPS*. 462–471.

[45] mlc [n.d.]. MLC LLM. https://llm.mlc.ai/. Accessed: 2024-06-12.

[46] mlx [n.d.]. MLX: An array framework for Apple silicon. https://github.com/ml-explore/mlx. Accessed: 2024-06-12.

[47] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023).

[48] Joseph O'Rourke. 1981. An On-Line Algorithm for Fitting Straight Lines Between Data Ranges. *Commun. ACM* 24, 9 (1981), 574–578.

[49] Long Ouyang, Jeffrey Wu, Xu Jiang, et al. 2022. Training language models to follow instructions with human feedback. In *NeurIPS*.

[50] Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. 2024. Unifying Large Language Models and Knowledge Graphs: A Roadmap. *IEEE Trans. Knowl. Data Eng.* 36, 7 (2024), 3580–3599.

[51] Igor Pavlov. [n.d.]. LZMA SDK. https://7-zip.org/sdk.html. Accessed: 2024-06-12.

[52] pinecone [n.d.]. Build knowledgeable AI. https://www.pinecone.io/. Accessed: 2024-06-12.

[53] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *SSTD*. Springer, 46–65.

[54] Parikshit Ram and Kaushik Sinha. 2019. Revisiting kd-tree for Nearest Neighbor Search. In *KDD*. ACM, 1378–1388.

[55] Sidney Redner. 2001. *A guide to first-passage processes*. Cambridge university press.

[56] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. 2022. Can Learned Models Replace Hash Functions? *Proc. VLDB Endow.* 16, 3 (2022), 532–545.

[57] Kunihiko Sadakane and Roberto Grossi. 2006. Squeezing succinct data structures into entropy bounds. In *SODA*. ACM Press, 1230–1239.

[58] samsung [n.d.]. How much phone memory and storage do I need? https://www.samsung.com/us/explore/mobile/how-much-phone-memory-and-storage-do-I-need/. Accessed: 2024-08-07.

[59] segpq [n.d.]. SegPQ (technical report). https://github.com/qyliu-hkust/segpq. Accessed: 2024-11-12.

[60] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2020. Energy and Policy Considerations for Modern Deep Learning Research. In *AAAI*. AAAI Press, 13693–13696.

[61] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).

[62] Hugo Touvron, Louis Martin, Kevin Stone, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.

[64] Sebastiano Vigna. 2013. Quasi-succinct indices. In *WSDM*. ACM, 83–92.

[65] Runhui Wang and Dong Deng. 2020. DeltaPQ: Lossless Product Quantization Code Compression for High Dimensional Similarity Search. *Proc. VLDB Endow.* 13, 13 (2020), 3603–3616.

[66] Donna Xu, Ivor W. Tsang, and Ying Zhang. 2018. Online Product Quantization. *IEEE Trans. Knowl. Data Eng.* 30, 11 (2018), 2185–2198.

[67] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *WWW*. ACM, 401–410.

[68] Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA*. ACM/SIAM, 311–321.

[69] youtube [n.d.]. YouTube-8M. https://research.google.com/youtube8m/download.html. Accessed: 2024-06-12.