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, vector quantization techniques like product quantization (PQ) are widely used to generate space-efficient representations for large-scale dense vectors. However, the codebooks generated by PQ often reach several gigabytes in size, making them impractical for web-scale, high-dimensional vectors in resource-constrained environments like mobile devices.

In this study, we propose **SegPQ**, a simple yet effective framework for losslessly compressing codebooks generated by **any** PQ variants, enabling efficient **in-memory** vector search on devices with limited memory. SegPQ represents the raw PQ codewords as a trained error-bounded piecewise linear approximation model (ϵ-PLA) and pre-computed low-bit residuals. We theoretically demonstrate that, with high probability, the number of bits per compressed codeword is $1.721 + \lceil \log_2 \epsilon^{\text{OPT}} \rceil$, where ϵ^{OPT} is the optimal error parameter that can be determined by data characteristics. To accelerate query execution, we further design SIMD-aware query processing algorithms on compressed codebooks to fully exploit the hardware parallelism offered by modern architectures. Extensive experimental studies on real datasets showcase that, for **1 billion** vectors, SegPQ reduces PQ codebook memory consumption by up to **4.7×** (approx. **851 MB**) while incurring only **3.3%** additional 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, 18(11): 3730 - 3743, 2025.
doi:10.14778/3749646.3749650

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

*These three authors contributed equally to the paper.
†Correspondence to Dr. Qiyu Liu.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749650

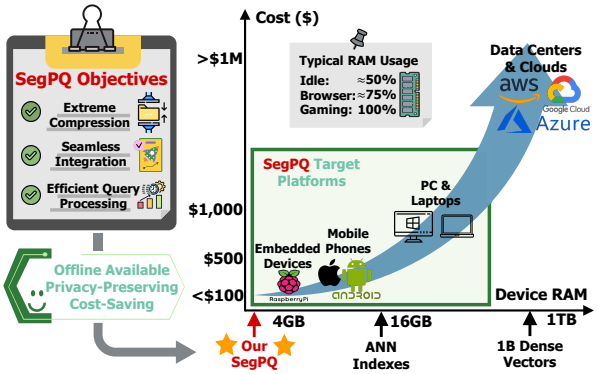


Figure 1: Illustration of SegPQ's design considerations: ❶ compact for resource-constrained devices, ❷ compatible with *any* PQ variants, and ❸ neglectable computation overhead.

1 INTRODUCTION

A defining attribute of large language models (LLMs) is their unprecedented scale, which poses significant challenges for deployment in resource-constrained environments [14, 56, 58, 72]. Recent advancements [3, 15, 27, 69], however, have facilitated LLM services on devices with limited resources, such as laptops, smartphones and embedded devices [46, 54, 55]. Nonetheless, these newly proposed systems often come with heavy optimization and complex workflow, with their actual performance to be verified. In this context, vector databases (VectorDBs) and retrieval-augmented generation (RAG) [5, 30, 59] offer promising solutions for enhancing LLM applications on resource-limited devices. By integrating external knowledge through approximate nearest neighbor (ANN) search, this vector-based infrastructure enables efficient retrieval with minimal memory overhead, allowing LLMs to access private or domain-specific knowledge bases without significantly increasing model size. In contrast to the large-scale nature of LLMs, ANN engines are designed as small and fast as possible [33, 45]. As LLMs are difficult to condense, refining their already-small integration technologies brings new perspectives, to unlock the potential of on-device AI innovations.

Taking the open-domain question answering (OpenQA) task [34, 35] as an example, external knowledge bases such as Wikipedia [76] are often utilized to enhance generation quality. However, by adopting popular embedding models like Cohere [17], vector databases built on Wikipedia data can reach hundreds of gigabytes in size,

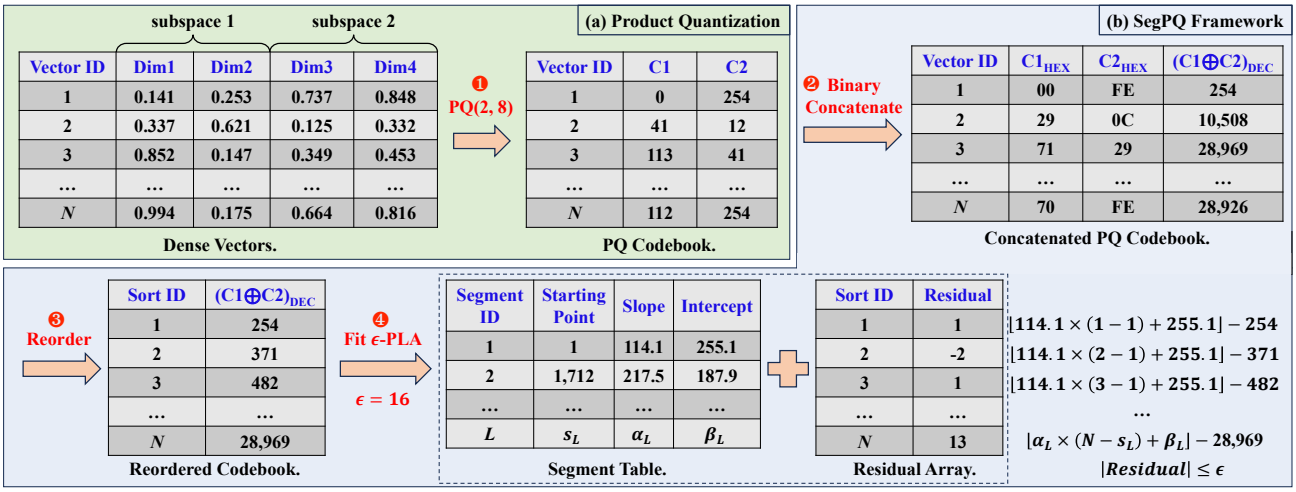


Figure 2: Running examples of (a) the classic product quantization (PQ) workflow and (b) our SegPQ framework. In (a), a PQ(2, 8) codebook is constructed on N 4-D dense vectors. PQ(2, 8) means that the sub-space partition number is 2 and each sub-space will be clustered into $2^8 = 256$ parts using k-means. 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.

making them impractical for deployment and use on personal devices. To reduce their sizes, memory-efficient ANN indexes like Locality-Sensitive Hashing [1, 18, 38], Proximity Graph [49, 50], and Vector Quantization [10, 26, 28, 77] have been explored. Among these, Product Quantization (PQ) stands out as a simple yet effective way, widely supported by mainstream VectorDBs such as Faiss [22], Milvus [52], and Pinecone [61]. PQ reduces memory usage by dividing high-dimensional vectors into subspaces, then independently compressing each subspace through clustering. As illustrated in Figure 2(a), PQ partitions a high-dimensional vector into m equal-sized subspaces, applies k-means clustering [48] to find k centroids for each subspace, and approximates each vector by concatenating the indices of its closest centroids across all subspaces. By PQ, representing each dense vector requires 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($m = 2, b = 8$) codebook on N 4-D dense vectors. Each partitioned sub-space of $4/m = 2$ dimensions will be clustered into $2^b = 256$ clusters, and each sub-vector is approximated by its closest 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. Notably, to ensure that the generated codewords are byte-aligned, b is typically fixed to 8. Therefore, we interchangeably use PQ$m \times b$ and PQ(m, b), such as PQ16, PQ32, PQ64, etc.

Though achieving a significant compression ratio compared to raw dense vectors, the resulting PQ codebooks, in practical settings (e.g., $m = 8$, $k = 256$), still necessitate 8 GiB for 1 billion vectors. Given a typical *idle* RAM (≤**4 GiB**) [67], as illustrated in Figure 1, this size is still **not small enough**, thus preventing the deployment of RAG services to resource-constrained devices, such as the latest iPhone or personal laptop. A recent work DeltaPQ [75] first attempts to reduce PQ codebook size by identifying and compressing redundancies in codewords using delta encoding [78]. 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 vector quantization codebook without compromising its effectiveness in resource-constrained environments?"* Specifically, as summarized in Figure 1, our design objectives are threefold: ❶ **Memory Efficiency:** Compress the codebook generated by PQ to fit in resource-constrained environments; ❷ **Generality:** Maintain *lossless* compression and compatibility to *any* PQ variants to ensure retrival quality and seamless integration with existing ANN engines; and ❸ **Query Efficiency:** 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. Figure 2(b) illustrates the workflow of SegPQ. Unlike existing dictionary-based or entropy-based compression algorithms [78], 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 data indexing [11, 23, 24, 32], 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.** This work targets the growing demand for deploying large-scale vector retrieval services on resource-constrained devices, driven by privacy concerns and the need for AI-powered applications in environments with unstable network connectivity. We share a novel perspective on compressing generic PQ codebooks by directly modeling the distribution of codewords.

❷ **Theoretical Insights.** SegPQ is a principled approach where the simple idea delivers non-trivial results. Our analysis reveals that SegPQ achieves an encoding efficacy of $1.721 + \lceil \log_2 \epsilon^{\mathsf{OPT}} \rceil$ bits per compressed PQ codeword, where ϵ^{OPT} is the optimal PLA error parameter determined by data distribution. This extends prior theoretical work [23] on learned indexing to learned compression, achieving near-optimal performance relative to the information-theoretic lower bound [66].

❸ **Empirical Performance.** Extensive benchmarks on six real-world ANN datasets demonstrate that SegPQ reduces PQ codebook memory usage by up to **4.7×** (down to ≈**851 MB** for 1 billion vectors) while incurring only a **3.3%** increase in computational overhead. To further validate its real-world applicability, we integrate SegPQ into an OpenQA pipeline based on Llama3.2-3B and a Wikipedia-based vector database. The results show that SegPQ reduces the total memory footprint of the RAG system from **7.25 GiB** to **4.62 GiB** while improving generation quality by **1.12×**.

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 and formulate the lossless codebook compression problem.

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 [26, 28, 77].

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 unique clustering IDs. 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 $\mathbf{X}$:

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

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

Table 1: Summary of major notations.

Notation	Explanation
m	the partition number foe each dense vector
b	the number of bits required to encode a cluster ID
$\mathbf{x}^1, \cdots, \mathbf{x}^m$	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 line segment

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 efficiently [48]. 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 clustering centroids to reconstruct the original dense vector $\mathbf{x}$. Since PQ constructs codewords from the Cartesian product $\Sigma^1 \times \cdots \times \Sigma^m$, it can efficiently represent an exponentially large set of vectors using only $m \cdot \lceil \log_2 k \rceil$ bits per codeword. We use $\mathrm{PQ}(m, b)$ ($b = \lceil \log_2 k \rceil$) to represent a PQ quantizer with m sub-spaces and k clusters per sub-space. To ensure byte alignment, b is typically set to 8, making each codeword's total bit-width a multiple of 8. Thus, when the context is clear, we interchangeably use $\mathrm{PQ8} \times m$ (e.g., PQ64 = PQ(8, 8)).

For query processing, 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 $\mathrm{SDC}(\mathbf{y}, \mathbf{x}) = \|D(C(\mathbf{y})) - D(C(\mathbf{x}))\|$ or asymmetric distance computation $\mathrm{ADC}(\mathbf{y}, \mathbf{x}) = \|\mathbf{y} - D(C(\mathbf{x}))\|$. Both SDC and ADC can be efficiently processed by pre-computing a distance lookup table of size $O(k^2)$ [28]. In practice, SDC is generally less accurate but more efficient than ADC due to the greater loss of information in the query vector.

Other Vector Quantizers. Beyond PQ, various other vector quantization techniques have been explored to enhance accuracy and reconstruction quality. Optimized PQ (OPQ) refines PQ by applying data-driven orthogonal transformations to subspaces, improving quantization accuracy [26]. Additive Quantization (AQ) represents vectors as linear combinations of full-dimensional codewords, achieving better reconstruction at the cost of increased computation [7]. Residual Quantization (RQ) employs a hierarchical encoding scheme, iteratively quantizing residual errors to minimize reconstruction loss [51]. As discussed in Sections 3.2 and 4.1, our proposed codebook compression framework is not restricted to a specific quantization method and can be applied to a broad range of vector quantizers.

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 $\mathrm{DC}(\mathbf{Q}^c)$, 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* [66, 74], 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

In this section, we first overview the SegPQ framework and then elaborate on the compression and decompression algorithm details.

3.1 SegPQ Framework Overview

The core idea of SegPQ is based on the observation that an arbitrary PQ codebook $\mathbf{Q}$ can be *reorganized* in a way that allows it to be well-fitted by even simple machine learning models. Specifically, we adopt *error-bounded piecewise linear approximation* (ϵ-PLA) to model $\mathbf{Q}$. By precomputing and materializing all *residuals* (i.e., the difference between the prediction and the true codeword), SegPQ ensures lossless recovery of the original codewords.

The SegPQ compression process, illustrated in Figure 2(b), consists of five key steps:

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 *ascending order* and reorder the corresponding codewords in $\mathbf{Q}$ accordingly.

❸ **Model Fitting.** With a *monotonic* sequence $\mathcal{K}$ and a pre-specified error parameter ϵ, we fit an ϵ-PLA model $f(i)$ such that $|\lfloor f(i) \rfloor - \mathcal{K}[i]| \leq \epsilon$. Conceptually, $f(i)$ maps an arbitrary index i to its corresponding value $\mathcal{K}[i]$, which approximates 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 $\delta_i = \lfloor f(i) \rfloor - \mathcal{K}[i]$. Since $\delta_i \in [-\epsilon, \epsilon]$, each residual requires $\lceil 1 + \log_2 \epsilon \rceil$ bits for encoding. Let $\Delta = \{\delta_1, \cdots, \delta_N\}$ denote the residual array. The compressed codebook is a tuple $\mathbf{Q}^c = (f, \Delta)$ of learned PLA model $f(\cdot)$ and the residual array Δ.

❺ **Query Processing.** Given a compressed codebook $\mathbf{Q}^c = (f, \Delta)$, each codeword $\mathbf{q}_i$ is reconstructed as $\mathcal{K}[i] = \lfloor f(i) \rfloor + \Delta[i]$, considering that the projection from $\mathbf{Q}$ to $\mathcal{K}$ is a *bijection* according to 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 (Step ❶ and Step ❷), we define a mapping function $\mathcal{M} : \mathbb{N}^m \mapsto \mathbb{N}$ that projects a codeword $\mathbf{q}$ into a integer sorting key

Algorithm 1: SegPQ Codebook Compression

Input: the original PQ codebook $\mathbf{Q}$
Output: the compressed codebook $\mathbf{Q}^c$

$\mathcal{K} \leftarrow \{\mathsf{BinCat}(\mathbf{q}) | \mathbf{q} \in \mathbf{Q}\}$ // Project to 1-D key
$sort(\mathcal{K})$ // Reorder by sorting key
$\epsilon^{\mathrm{OPT}} \leftarrow \sqrt{\frac{2C \cdot \ln 2 \cdot (m \log_2 k + 2F)}{N}}$ // Configure by Theorem 4.5
$f \leftarrow \mathrm{OptPLA}(X = \{1, \cdots, N\}, Y = \mathcal{K}, \epsilon^{\mathrm{OPT}})$ // Fit PLA
$\Delta \leftarrow \{\lfloor f(i) \rfloor - \mathbf{Q}[i] | i \in \{1, \cdots, N\}\}$ // Compute residuals
return $(f.segments, \Delta)$ // Return segments and residuals

k. To ensure lossless recovery, $\mathcal{M}$ is required to be a *bijection*. In SegPQ, we choose the following binary concatenation operation $\mathsf{BinCat}(\cdot)$ as the mapping function.

Definition 3.1 (Binary Concatenation). Given a PQ codeword $\mathbf{q} = \{\mathbf{q}^1, \mathbf{q}^2, \cdots, \mathbf{q}^m\}$, the binary concatenation operation is defined 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 $\mathbf{q}_2 = [29_{\mathsf{hex}}, 0\mathsf{C}_{\mathsf{hex}}]$, $\mathsf{BinCat}(\mathbf{q}_2) = \mathsf{dec}(290\mathsf{C}_{\mathsf{hex}}) = 10508$.

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 equally simple: the binary representation of a projected codeword $\mathsf{bin}(\mathsf{BinCat}(\mathbf{q}))$ can be evenly partitioned into m parts to reconstruct the original m sub-codes. Since the projected key set $\mathcal{K} = \{\mathsf{BinCat}(\mathbf{q}) | \mathbf{q} \in \mathbf{Q}\}$ is logically equivalent to the original codebook $\mathbf{Q}$, the problem of compressing $\mathbf{Q}$ reduces 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) [26], Additive Quantization (AQ) [7], and Residual Quantization (RQ) [51]. Since the aforementioned vector quantizers all employ k-means to learn the codebook, the resulting codewords in real-world applications often exhibit a *near-uniform* distribution (see Section 4.1 for details). Leveraging this property, the theoretical results derived in this paper are applicable to a broad range of vector quantizers beyond PQ, making our SegPQ a generalizable framework that supports various quantization methods as plugable components. As demonstrated in our experiment study (Section5), SegPQ+**X** consistently compresses codebooks generated by various quantizers, where **X** can be PQ, OPQ, AQ, and RQ.

3.2.2 Reordering and Model Fitting. To prepare the training data, the original codewords in $\mathbf{Q}$ are sorted based on their corresponding binary concatenation values $\mathcal{K}$ in increasing order. With the sorted set $\mathcal{K}$, SegPQ learns 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 and lossless representation of $\mathbf{Q}$, enabling query processing directly on the compressed data.

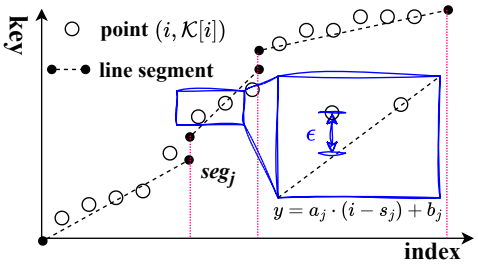


Figure 3: Illustration of an ϵ-PLA model.

Intuitively, learning f is equivalent to learning the inverse cumulative distribution function (ICDF, a.k.a. the quantile function) of $\mathcal{K}$. Inspired by recent learned index studies like RMI [32] and PGM-Index [24], to balance the model expressivity and inference efficiency, simple machine learning models like piecewise linear functions [12, 24] are trained to approximate the mapping f with controllable error. 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,

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

such that $|\mathcal{K}[i] - \lfloor f(i) \rfloor| \le \epsilon$ 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 that require heavy runtimes such as Pytorch [63] and Tensorflow [70], ϵ-PLA offers both space and time efficiency in training and inference. ❷ By making reasonable statistical assumptions, we can derive a quantitative relationship between the model complexity (i.e., the required number of segments) and the error bound, which is essential for subsequent optimal parameter configuration. In contrast, for deep learning models, establishing such a relationship is often intractable due to the inherent complexity and black-box nature.

For implementation, we adopt the online algorithm [57] to construct an *optimal* ϵ-PLA such that the required number of line segments is minimized to satisfy the given 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. In Section 4, we provide the first theoretical analysis to estimate L, enabling a precise characterization of SegPQ's space overhead and compression ratio.

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, the 1-st segment, with a slope 114.1 and an 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 precomputed as $\delta_3 = 483 - 482 = 1$. Finally, the trained ϵ-PLA model and precomputed residuals are materialized as the lossless compressed representation of the original codebook.

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 [57]. Therefore, the total time complexity of Algorithm 1 is $O(N \log N + N) = O(N \log N)$.

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

Limitation of BinCat Projection. In Step ❶, we employ BinCat to map PQ codewords to integer sorting keys. For high-bit-width codebooks (e.g., PQ128/PQ256), BinCat generates excessively large value spaces, impairing learned PLA efficacy. This aligns with information theory: expanding the value range (for fixed dataset sizes) increases entropy, thereby making compression much more challenging. In this work, we prioritize practical resource-constrained scenarios (e.g., PQ32/PQ64) while supporting higher bit-widths via partial compression by only compressing the upper 64 bits while leaving the remaining bits unchanged. Experiments demonstrate that, this partial compression solution achieves **1.43×** (for PQ128) and **1.18×** (for PQ256) compression ratios, outperforming other baselines.

Codeword 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. This sorting operation inevitably modifies the original vector ID assignments. In our work, we assume a generic scenario where the original vector IDs can be safely updated to their new sorting indexes.

Example 3.5 (RAG with SegPQ). In a RAG-powered open-domain question answering (OpenQA) 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 from an external knowledge base (e.g., Wikipedia) to be retrieved, and e_d is the learned embedding of 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 value 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 SegPQ's compressed codewords are kept in memory, enabling efficient in-memory kNN query processing on 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 $\mathcal{K}$.

Algorithm 2: SegPQ Codebook Decompression

1 **Function** AT($\mathbf{Q}^c = (Segs, \Delta)$, i):
2 $\ell^* \leftarrow$ minimum $\ell \in \{1, \cdots, |Segs|\}$ s.t. $Segs[\ell].s \geq i$
3 $\mathbf{q}^* \leftarrow \lfloor Segs[\ell^*].\alpha \times (i - Segs[\ell^*].s) + Segs[\ell^*].\beta \rfloor$
4 **return** Split($\mathbf{q}^* + \Delta[i]$)
5 // Traverse the whole codebook $\mathbf{Q}^c$
6 **Function** DC($\mathbf{Q}^c = (Segs, \Delta)$):
7 $\mathbf{Q} \leftarrow [\,]$
8 // The loop can be optimized using SIMD
9 **for** $i \in \{1, 2, \cdots, |Segs|\}$ **do**
10 **for** $j \in \{1, 2, \cdots, Segs[i].coverage\}$ **do**
11 $\mathbf{q}^* \leftarrow \lfloor Segs[i].\alpha \times j + Segs[i].\beta \rfloor$
12 $\mathbf{Q}.add(\text{Split}(\mathbf{q}^* + \Delta[Segs[i].s + j]))$
13 **return** $\mathbf{Q}$

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*, implying the existence of a global optima. An analytical optimal solution for ϵ, as used in Line 3 of Algorithm 1, is derived to maximize SegPQ's compression efficacy.

Dynamic Updates. Similar to learned indexes, SegPQ's reliance on machine learning models makes efficient updates non-trivial, particularly given that SegPQ leverages sorted codeword indices as model inputs. To handle this, one can employ the gap array design commonly used by updatable learned indexes such as ALEX [21]. Specifically, we introduce "gaps" in the data, allowing new codewords to fill these gaps without immediate reconstruction. If inserting new codewords violates the original error bound ϵ, the affected segment is split into two smaller segments to maintain accuracy. However, this approach requires reserving space for future insertions, leading to extra storage overhead. Additionally, a large number of insertions within a localized range could quickly exhaust available gaps, necessitating a complete model re-training. In this work, we focus on edge-side, infrequently updated vector databases (Example 3.5). In such applications, updates can be handled server-side, where compressed codebooks are periodically rebuilt and distributed to edge devices.

3.3 Querying Compressed Codebooks

Due to the bijective nature of BinCat-based projection and error-bounded property of ϵ-PLA, the i-th PQ 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 $C_1 = \text{Split}(\text{bin}(255-1)) = \text{Split}(00\text{FE}_{\text{hex}}) = [0, 254]$ and $C_2 = \text{Split}(\text{bin}(369+2)) = \text{Split}(0173_{\text{hex}}) = [1, 115]$.

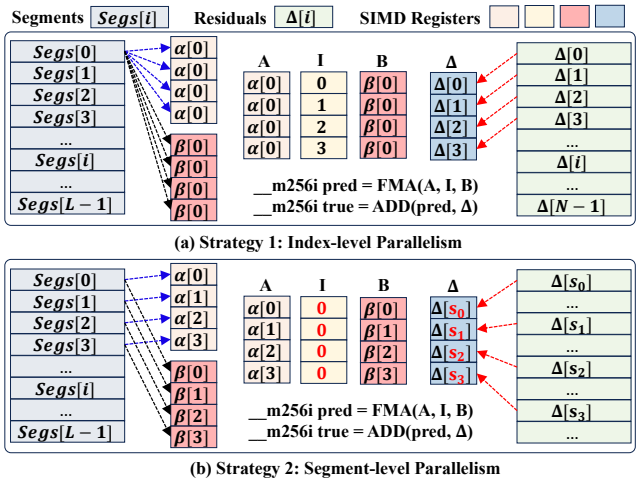


Figure 4: Illustration of SIMD-aware traversal: (a) Index-level parallelism, and (b) Segment-level parallelism. FMA(A, I, B)=A×I+B and (s_i, α_i, β_i) refers to the i-th segment $Segs[i]$.

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

3.3.1 Random Access. The pseudo-code is given in Lines 1–4 of Algorithm 2. Specifically, Line 2 searches for the corresponding line segment given an index i. Line 3 computes the approximate key 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 $O(\log_2 L)$ time by adopting 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 could build an in-memory index like B+-tree on the segment breakpoints. However, we do not incorporate auxiliary index structures, as they introduce extra storage and computation overhead. Moreover, random access is not frequently used in PQ-based ANN processing.

3.3.2 SIMD-Aware Traversal. In practice, every codeword must be accessed to find the nearest neighbors to a query vector, using either ADC or SDC as discussed in Section 2. Therefore, traversing the entire codebook is more prevalent than random access. To accelerate full traversal, we leverage the hardware parallelism offered by *single instruction multiple data* (SIMD), a feature available on most modern CPUs[1]. SIMD enables vectorized computation, allowing *multiple* arithmetic operations to be executed within a single instruction cycle. For example, a 256-bit SIMD register can process 8 single-precision floating-point values in parallel, while a 512-bit register can handle up to 16.

Depending on how SIMD parallelism is applied, we consider two strategies, as illustrated in Figure 4: ❶ **index-level parallelism**, where multiple indexes are processed simultaneously for

[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.

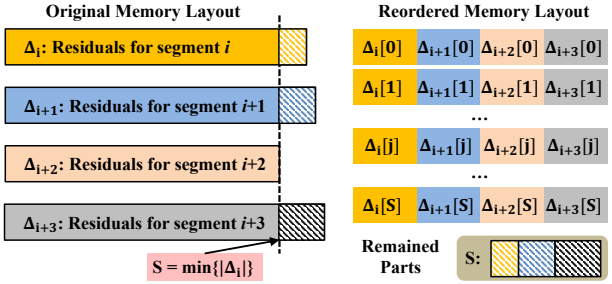


Figure 5: Illustration of the memory re-layout for segment-level parallelism (i.e., Strategy ❷ in Section 3.3). W.l.o.g., we assume that four segments are processed simultaneously for each index starting from 0.

each segment, and ❷ **segment-level parallelism**, where multiple segments are processed simultaneously for each index. Strategy ❷ is generally more efficient because, within the same segment, once $f(i) = \alpha \cdot i + \beta$ is computed, the next prediction $f(i+1)$ can be derived incrementally as $f(i+1) = f(i) + \alpha$, eliminating redundant multiplications. Due to page limits, we put the code snippets for SIMD-aware codebook traversal in our technical report [68]. As no segment search operation is required, traversing the entire codebook requires an amortized time of $O(1)$ per codeword.

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. 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 Assumption 1 introduces our core assumptions on the distribution of PQ codewords, based on which Lemma 4.2 derives the variance of the *codeword gap* distribution;

❷ Theorem 4.3 presents our central theoretical results regarding the *expected* line segment coverage;

❸ Based on Theorem 4.3, we further derive the space overhead and compression ratio of SegPQ in Corollary 4.4;

❹ Theorem 4.5 derives the optimal error parameter ϵ^{OPT} in a closed form, and Corollary 4.6 gives the corresponding bits per compressed codeword as $1.721 + \lceil \log_2 \epsilon^{\text{OPT}} \rceil$;

❺ Finally, Theorem 4.7 generalizes the results of Theorem 4.3 by relaxing the uniformity assumption, extending our methodology to arbitrary distributions.

4.1 Expected Segment Coverage

As discussed in Section 3.2, 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.

Assumption 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 $\text{BinCat}(\mathbf{q}_i)$ follows a uniform distribution $U(0, k^m)$.*

Assumption 1 is reasonably made for most real-world applications. The reason is that, embedding vectors used in vector databases are typically produced by *well-trained* deep learning models (e.g., BERT, Transformer-based models), which tend to exhibit an approximately isotropic distribution in high-dimensional space [20]. On these vectors, mainstream vector quantization techniques (e.g., PQ, OPQ, AQ) adopt k-means clustering, which generates clusters of approximately equal sizes, finally leading to a near-uniform codeword distribution after binary concatenation. Section 5.2 reports the distributions of $\text{BinCat}(\mathbf{q})$ on real-world datasets to validate this assumption.

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 4.2. Notably, in the subsequent analysis, we interchangeably use x_i and $\text{BinCat}(\mathbf{q}_i)$ if the context is clear.

Definition 4.1 (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 4.2 (Gap Distribution Characteristics). *Under Assumption 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 [19], $U_{(i)} - U_{(i-1)} \sim \textbf{Beta}(1, N)$. Under Assumption 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 $g_i \sim k^m \cdot \textbf{Beta}(1, N)$. 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 4.2, we then analyze the expected line segment coverage.

Theorem 4.3 (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,*

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

Proof. We first obtain the result that the expected segment coverage is $\frac{\epsilon^2}{\mathbf{Var}[g]}$ by extending the theoretical results in [23]. Then, by combining Lemma 4.2, we have the result stated in Theorem 4.3. Please refer to our technical report [68] for detailed proof. □

4.2 Memory Footprint Analysis

Intuitively, Theorem 4.3 supports the rationale behind SegPQ. Considering a vector set of 1 million, we construct a PQ(4, 4) codebook, where the gap variance is measured as 0.477. By allocating 6 bits per residual (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 4.3). Therefore, representing 1 million vectors requires approximately 1,000 segments along with 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 4.3.

Corollary 4.4 (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*

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

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 4.3, in expectation[2], by setting the slope of a line segment to $\mathbf{E}[g]$, the number of line segments $L(\epsilon)$ can be derived as follows,

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

[2]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 [23].

In practice, the *optimal* piecewise linear fitting technique [24, 57] is adopted in SegPQ to learn segments where the number of segments is guaranteed to be minimized under the error constraint ϵ. Thus, $L(\epsilon) = O(k^{2m}/N\epsilon^2)$.

We then discuss how to configure ϵ to minimize the space overhead described in Eq. (6). In practice, we can collect a small set of 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 4.5 (Optimal Parameter Setting). *The SegPQ space overhead $M(\epsilon)$ given in Corollary 4.4 is minimized by setting*

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

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

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

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

By setting the derivative of Eq. (10) to zero, we obtain the optimal solution given in Eq. (9). For detailed proofs, please refer to our technical report [68] due to space limitations. □

Corollary 4.6 (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,*

$$\begin{aligned} \frac{M(\epsilon^{\text{OPT}})}{N} &= \lceil 1 + \log_2 \epsilon^{\text{OPT}} \rceil + \frac{1}{2 \ln 2} \\ &\approx 1.721 + \lceil \log_2 \epsilon^{\text{OPT}} \rceil. \end{aligned} \tag{11}$$

By further expanding Eq. (11), 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 [66].

4.3 Discussion on Non-Uniform Codebooks

While Section 4.1 discusses the practical validity of the approximately uniform distribution assumption for PQ codewords, the theoretical behavior of SegPQ under non-uniform distributions remains an interesting question. Fortunately, we show that even without the uniformity assumption, SegPQ maintains comparable asymptotic performance, as demonstrated by a generalized bound analogous to Theorem 4.3.

Theorem 4.7 (Segment Coverage for Non-Uniform Cases). *Given a set of sorted values X^* and an error threshold ϵ, assume that:*

A1: *X^* is the realization of the order statistics of n **i.i.d. samples** $K_1, \cdots, K_i$ drawing from an **arbitrary** distribution $\mathcal{D}$ with maximum range as ρ.*

A2: *The inverse cumulative function $F^{-1}(\cdot)$ exists for $\mathcal{D}$, and for arbitrary $i \in \{1, \cdots, |\mathbb{I}|\}$ where $|\mathbb{I}| = o(n)$, there exists a constant γ such that $|\frac{n}{\rho} \cdot F^{-1}(\frac{i}{n}) - i| \leq \gamma$.*

A3: *For arbitrary $i \in \{1, \cdots, |\mathbb{I}|\}$, the density function $f(\cdot)$ of $\mathcal{D}$ is continuous and non-zero at $F^{-1}(\frac{i}{n})$. Moreover, there exists a constant ξ such that $f(F^{-1}(\frac{i}{n})) \geq 1/\xi$.*

Under assumptions A1, A2, and A3, when n is sufficiently large, the expected coverage of a line segment $\ell(x) = \frac{\rho}{n+1} \cdot x$ is given by:

$$\frac{N^2 \cdot (\epsilon - \rho \cdot \gamma/n)^2}{\xi \cdot \rho^{3/2}} \propto \Theta(\epsilon^2 \cdot N^2). \tag{12}$$

Proof. We develop a new theoretical framework to analyze the statistical behavior of ϵ-PLA that does not rely on the gap model used by PGM-Index [23, 24]. Due to space constraints, please refer to our technical report [68] for detailed proofs. □

According to Theorem 4.7, by ignoring the constant terms, the asymptotic behavior of Eq. (12) remains unchanged compared to the uniform cases as discussed in Theorem 4.3. This suggests that although our method was originally designed under a uniformity assumption, it naturally generalizes to non-uniform distributions.

5 EXPERIMENTAL STUDY

In this section, we report the experimental results to demonstrate the effectiveness and efficiency of SegPQ. All compared methods are implemented in C++ and compiled by g++ 11. All experiments are conducted on a Linux server with an AMD EPYC 7413 CPU and 512 GB of main memory.

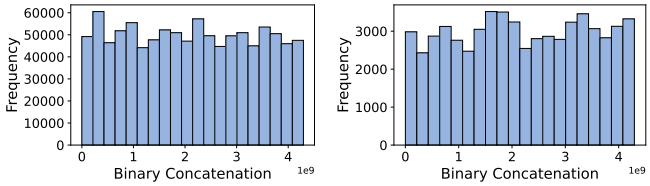
5.1 Datasets and Setups

5.1.1 Benchmark Datasets. We adopt **six** large-scale datasets that are commonly evaluated in ANN and VectorDB studies. **Audio** is a set of 128-D audio features extracted from YouTube video frames [80]. **SIFT** and **GIST** are sets of 128-D/960-D SIFT/GIST image descriptors [28, 29]. **Deep1B** is a billion-scale dataset of pre-trained embeddings from a deep neural network [8], where each embedding vector is reduced to 96-D by PCA. **MSMARCO** is a widely used dataset in retrieval tasks [9], while **Wiki** is derived from Wikipedia passages. Both MSMARCO and Wiki are embedded to 384-D dense vectors using Cohere-embed-light-v3.0 [17]. As discussed in Section 5.3.1, PQ32 is adopted by default. Table 2 summarizes the statistics of the benchmark datasets.

5.1.2 Vector Quantization Methods. We adopt **four** different vector quantizers to construct the codebook: ❶ **PQ**, the classic product quantization [28]; ❷ **OPQ**, the optimized product quantization that minimizes the vector quantization distortions w.r.t. space partitioning and quantization codebooks [26]; ❸ **AQ**, the additive quantization that encodes dense vectors by the *sum* of multiple codebooks [7]; ❹ **RQ**, the residual quantization that iteratively quantizing residual errors [51]. By default, PQ is used as the quantizer to generate codebooks with varying configurations. For OPQ, AQ, and RQ, the hyper-parameters are adjusted to ensure the resulting codebook sizes match those of PQ. The vector quantization implementations are chosen from Facebook's faiss library [22].

Table 2: Summary of evaluation datasets.

Dataset	Domain	#Vectors	#Dims	Raw Size
GIST	Image	1 M	960	3.7 GB
SIFT	Image	1 B	128	123 GB
Deep1B	Image	1 B	96	361 GB
Audio	Audio	438 M	128	211 GB
MSMARCO	Text (en)	113 M	384	170 GB
Wiki	Text (multi)	94 M	384	141 GB



(a) Key distribution on SIFT **(b) Key distribution on Deep1B**

Figure 6: Validation of Assumption 1 that the binary concatenations of PQ codewords are near-uniformly distributed.

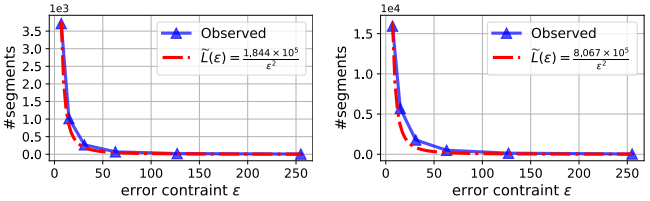
5.1.3 Compared Baselines. We implement and evaluate **seven** compression methods: ❶ **SegPQ**, the full-fledged SegPQ framework with SIMD-based traversal acceleration (by default, the segment-level parallelism is enabled); ❷ **DeltaPQ** [75], the state-of-the-art PQ codebook compression method based on indexing similar codewords' difference; ❸ **LZ4** [47], a widely used lossless compressor for byte stream, and we configure LZ4 to achieve the highest compression ratio (i.e., `LZ4 -9`); ❹ **LZMA** [60], the Lempel-Ziv-Markov chain algorithm, which is another lossless compression scheme optimized for compression ratio; ❺ **BUFF** [37], a generic lossless compressor originally designed for bounded floats. Additionally, we also compare ❻ **Bolt** [10], another vector quantizer that tradeoffs space overhead and accuracy, and ❼ **PQFS** [2], an efficient PQ codebook traversal algorithm by leveraging cache locality.

Remarks. 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 [1] or HNSW [49] variants as they are orthogonal to our work and have been intensively evaluated in previous ANN benchmarks such as [6, 36].

5.2 Validation of Theoretical Results

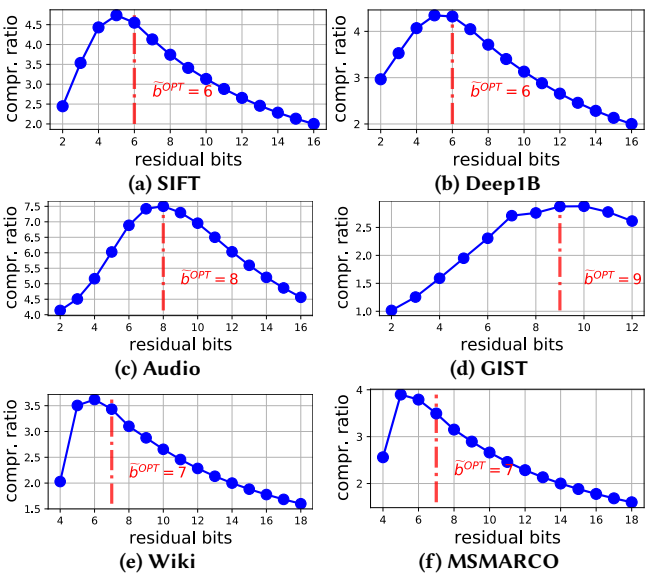
5.2.1 Binary Codeword Distribution. We first validate Assumption 1 on codeword distribution. Figure 6 shows the histograms of PQ codewords after binary concatenation (i.e., the projection step of SegPQ) on various datasets. The results reveal that the projected codewords exhibit *approximate* uniform 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, aligning with the results given in Theorem 4.7. To further evaluate the effectiveness of SegPQ on non-uniform codebooks, we manually construct two datasets following normal and power-law distributions. The corresponding results are analyzed in Section 5.3.6.

5.2.2 Average Segment Coverage. We then empirically verify our core theoretical results on the expected segment coverage (i.e., Theorem 4.3). To do this, we construct four datasets, each consisting of 10^5 vectors sampled from SIFT, with varying gap variances $\mathbf{Var}[g]$. 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 $\widetilde{L}(\epsilon) = C/\epsilon^2$ using the observed points. The results indicate that $C \propto N\mathbf{Var}[g]$, aligning with Theorem 4.3 and Corollary 4.4. As the optimal error configuration (i.e., Eq. (9) in Theorem 4.5) requires C as input, by extensive testing, we adopt $\widetilde{C} \approx 1.734 \cdot k^{2m}/N$ 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.



(a) $\mathrm{Var}[g] = 8.3$. (b) $\mathrm{Var}[g] = 75.0$.

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 $\widetilde{L}(\epsilon) = C/\epsilon^2$.



(a) SIFT (b) Deep1B (c) Audio (d) GIST (e) Wiki (f) MSMARCO

Figure 8: Compression ratios w.r.t. varying bits for storing each residual on six datasets.

5.2.3 Effectiveness of Optimal Parameter Setting. Figure 8 shows the results of compression ratio w.r.t. different bits (b) allocated for each residual. Note that, maximizing the compression ratio is equivalent to minimizing $M(\epsilon)$ as in Theorem 4.5. The red dashed lines mark the optimal error bits $\widetilde{b}^{\mathrm{OPT}}$ estimated by Theorem 4.5. 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 4.5 and ❷ the effectiveness of constant C configured previously.

5.3 SegPQ Evaluation Results

5.3.1 Retrieval Quality and PQ configuration. Although ANN studies primarily focus on retrieval quality, typically measured by Recall@x, SegPQ, as a lossless compression method, its retrieval performance is entirely determined by the underlying vector quantizer (e.g., PQ, OPQ, AQ). To better guide parameter configuration, Figure 9 illustrates the relationship between Recall@x and x for PQ, OPQ, and AQ on SIFT dataset. The results show that AQ and OPQ achieve relatively high recall even at a 32-bit quantization width, whereas PQ performs slightly worse. Additionally, increasing the bit width from 64 to 128 yields only marginal improvements compared to the more significant gain from 32 to 64. Given that

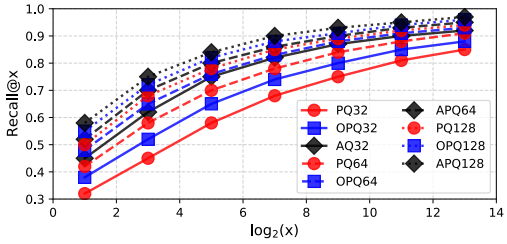
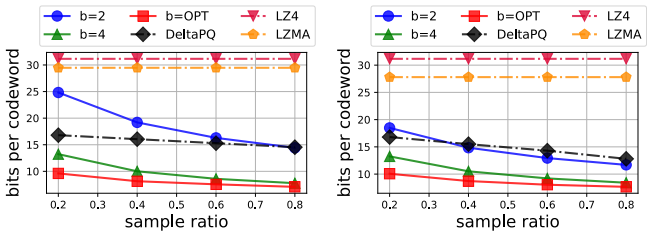


Figure 9: Retrieval quality (measured by Recall@x) of PQ, OPQ, and AQ on SIFT. Note the logarithmic scale for x-axis.



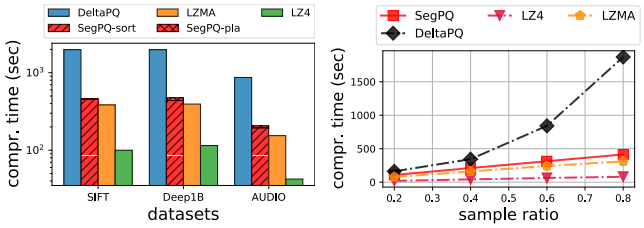
(a) SIFT: memory v.s. s (b) Deep1B: memory v.s. s

Figure 10: Bits per codeword (BPC) w.r.t. different sample ratios s. 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).

SegPQ is designed for resource-constrained scenarios, we set 32 bits as the default quantization width for all quantizers. Results on other datasets follow a similar trend and are therefore omitted for brevity.

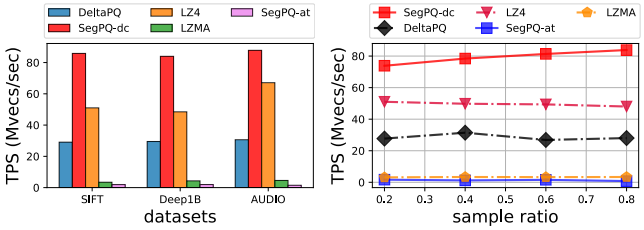
5.3.2 Memory Footprint. To demonstrate the power of memory reduction for SegPQ, we report the average bits per codeword (BPC) by varying different data sizes (controlled by a sample ratio $s \in \{0.2, 0.4, 0.6, 0.8\}$ for each dataset). By adopting PQ32, without compression (i.e., the raw codewords), BPC for SIFT and Deep1B is 32. The results presented in Figure 10 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 two billion-scale vector sets SIFT and Deep1B are reduced to **838 MB** and **882 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.

5.3.3 Compression Efficiency. Figure 11 reports the scalability of codebook compression across various datasets. From the results, LZ4 is the most efficient, processing all billion-scale data within 100 seconds, at the cost of the worst compression ratio (as discussed in Section 5.3.2). 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 PQ32 codebook using faiss takes 12,390 seconds, which is about **30**× larger than SegPQ compression. Overall, SegPQ, LZ4, and LZMA



(a) Overall comparison (b) SIFT: compression time

Figure 11: Evaluation results on codebook compression time. Note the logarithm scale in Figure 11a.



(a) Overall comparison (b) SIFT: throughput v.s. s

Figure 12: Evaluation results on codebook decompression throughput (unit: million vectors per second).

scale well to billion-scale datasets; in contrast, DeltaPQ, the SOTA codebook compression technique, takes over 30 min to compress 1 billion vectors and exhibits super-linear growth as data size increases. This inefficiency is because DeltaPQ relies on indexing differences between codewords, which grows much faster as data size scales. Notably, the compression time of SegPQ primarily contains two parts: codebook reordering (>95%) and PLA training. For the overhead of reordering raw vectors stored on disk (or associated documents in an RAG application), we adopt an external sorting algorithm from `stxxl::sort`, which can sort SIFT (123 GB) in 419 seconds on our experiment platform (4×2.5 TB NVMe and 512 GB main memory). While unneglectable, this is usually a "one-time" preprocessing cost in practical deployments.

5.3.4 Decompression Efficiency. We then investigate the decompression efficiency of compressed codebooks. Figure 12 reports the processing throughput (measured in million vectors per second) w.r.t. different data sizes (controlled by varying sampling ratio). Among all datasets, SegPQ's SIMD-aware traversal (i.e., SegPQ-dc in Figure 12) 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 could be further widened with multi-threading, which we do not explore here as none of the baselines are optimized for multi-threaded execution. Besides, while SegPQ-at is as slow as LZMA, it is the only method that supports efficient random access 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 4.3, ultimately requiring less time to access the segments.

5.3.5 ANN Query Efficiency. We further evaluate the overall ANN query processing efficiency with SegPQ. As shown in Table 3, SegPQ achieves a significant space reduction of up to **352%** compared to PQ and OPQ, while incurring only minor increases in index and query costs, limited to **8.3%** and **6.2%**, respectively. More results on integrating SegPQ with other vector quantization codebooks are provided in Section 5.3.7. Among all methods, PQFS demonstrates the highest query efficiency, as its codebook structure is specifically designed for fast ANN query processing. However, like standard PQ, PQFS comes with a high memory overhead. We further compare SegPQ with other vector compression techniques, including Bolt, BUFF, and DeltaPQ. The results show that SegPQ achieves the *best* trade-off between memory overhead and ANN query efficiency.

5.3.6 Evaluation on Non-Uniform Distributions. We construct two non-uniform datasets, SIFT-Normal and SIFT-Power, by re-sampling

Table 3: Evaluation of ANN query processing efficiency on the Deep1B dataset. 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

Table 4: Evaluation of SegPQ+X on Deep1B, where X represents different quantization methods (PQ, OPQ, AQ, and RQ). We evaluate each quantizer under varying bit-width settings (16, 32, and 64). "OOM" indicates that the quantizer failed to train the codebook due to out-of-memory (OOM) errors.

Method	Memory Unit: GiB	Index Time Unit: sec	Query Time Unit: sec
PQ16	1.9	436	53.92
OPQ16	1.9	691	56.26
RQ16	1.9	65447	94.88
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+RQ16	**0.23 (↓715%)**	65905 (↑0.7%)	100.20 (↑5.6%)
SegPQ+AQ16	**0.24 (↓692%)**	78351 (↑0.1%)	126.48 (↑4.1%)
PQ32	3.8	543	60.73
OPQ32	3.8	579	85.95
RQ32	3.8	206228	113.41
AQ32	OOM	OOM	OOM
SegPQ+PQ32	**0.91 (↓322%)**	672 (↑23.8%)	64.52 (↑6.2%)
SegPQ+OPQ32	**0.84 (↓352%)**	716 (↑23.6)	90.04 (↑4.8%)
SegPQ+RQ32	**0.99 (↓284%)**	206401 (↑0.1%)	119.31 (↑5.2%)
SegPQ+AQ32	OOM	OOM	OOM
PQ64	7.5	1199	115.31
OPQ64	7.5	2220	208.14
RQ64	OOM	OOM	OOM
AQ64	OOM	OOM	OOM
SegPQ+PQ64	**2.95 (↓154%)**	1299 (↑8.3%)	119.81 (↑3.9%)
SegPQ+OPQ64	**3.39 (↓121%)**	2322 (↑4.6%)	213.14 (↑2.4%)
SegPQ+RQ64	OOM	OOM	OOM
SegPQ+AQ64	OOM	OOM	OOM

from the PQ32 codewords of SIFT. To better adapt to non-uniform distributions, we introduce a partition-based parameter-tuning approach. Specifically, we employ PELT [31], an efficient change-point detection algorithm, to partition the codewords into disjoint chunks $\mathcal{P} = P_1, \ldots, P_M$. Within each chunk, codewords can be approximately treated as uniformly distributed, allowing us to allocate an optimal error threshold ϵ_i independently for each partition P_i. Using this strategy, SegPQ achieves a compression ratio of **6.85** on SIFT-Normal and **5.73** on SIFT-Power. For comparison, DeltaPQ achieves compression ratios of **4.67** and **3.98**, respectively.

5.3.7 Evaluation on SegPQ+X. We perform supplementary experiments to showcase the query efficiency of SegPQ when integrated with PQ, OPQ, AQ, and RQ 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 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.

5.4 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 [75], 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: ❶ LLMs, RAG and vector databases, ❷ ANN indexing, and ❸ learned data indexing.

LLMs, RAG and VectorDB. Recent advancements in LLMs [16, 56, 71] have revolutionized NLP tasks, demonstrating transformative power of Transformer architectures [73] with billion-scale parameters. In parallel, integral techniques like VectorDB and RAG with the ANN search engine [5, 25, 30, 59] are investigated to enhance LLM's performance. VectorDB enables efficient storage and retrieval of high-dimensional data, while ANN facilitates similarity searches. RAG [25] 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 [4, 36, 45]. Exact NN solutions typically include kd-tree variants [62, 64] and metric tree variants [13, 79]. 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 [1, 18, 38], proximity graph based [49, 50], and vector quantization based [26, 28, 77] approaches. Among the various ANN techniques, PQ variants are generally adopted by mainstream VectorDB products [22, 52, 61] 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 [24, 32, 39], R-tree [40], histogram [42], hash tables [43, 65], and Bloom filters [44, 53], leading to the concept named *learned index*. Recent theoretical studies [23, 39, 41] demonstrate the effectiveness of piecewise linear models in fitting distributions with controllable error. In our work, we extend the results in [23] to the case of learned PQ codebook compression by fully utilizing the distribution information behind PQ codewords. In our technical report [68], 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, a novel approach for *further lossless* compression of PQ codebooks. We establish simple yet non-trivial theoretical results to demonstrate SegPQ's efficacy. Extensive evaluations on web-scale vector search scenarios demonstrate that SegPQ can reduce the raw codebook size to **less than 1 GiB** while incurring **negligible** computational overhead for processing 1 billion 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.

ACKNOWLEDGMENTS

Dr. Qiyu Liu is supported by the Fundamental Research Funds for the Central Universities (No. 5330501376). Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2023YFF0725100, National Science Foundation of China (NSFC) under Grant No. U22B2060, Guangdong-Hong Kong Technology Innovation Joint Funding Scheme Project No. 2024A0505040012, the Hong Kong RGC GRF Project 16213620, RIF Project R6020-19, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, CRF Project C2004-21G, Key Areas Special Project of Guangdong Provincial Universities 2024ZDZX1006, Guangdong Province Science and Technology Plan Project 2023A0505030011, Guangzhou municipality big data intelligence key lab, 2023A03J0012, Hong Kong ITC ITF grants MHX/078/21 and PRP/004/22FX, Zhujiang scholar program 2021JC02X170, Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab and 2023 HKUST Shenzhen-Hong Kong Collaborative Innovation Institute Green Sustainability Special Fund, from Shui On Xintiandi and the InnoSpace GBA.

REFERENCES

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, et al. 2016. Ms marco: A human generated machine reading comprehension dataset. *arXiv preprint arXiv:1611.09268* (2016).

[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] cohere [n.d.]. Cohere. https://huggingface.co/Cohere. Accessed: 2024-11-12.

[18] 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.

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

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 4171–4186.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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).

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Rebecca Killick, Paul Fearnhead, and Idris A Eckley. 2012. Optimal detection of changepoints with a linear computational cost. *J. Amer. Statist. Assoc.* 107, 500 (2012), 1590–1598.

[32] 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.

[33] 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.

[34] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. 2019. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics* 7 (2019), 453–466.

[35] Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent Retrieval for Weakly Supervised Open Domain Question Answering. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 6086–6096.

[36] 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.

[37] 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.

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

[39] Qiyu Liu, Siyuan Han, Yanlin Qi, Jingshu Peng, Jin Li, Longlong Lin, and Lei Chen. 2024. Why Are Learned Indexes So Effective but Sometimes Ineffective? *arXiv preprint arXiv:2410.00846* (2024).

[40] 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), 1–29.

[41] Qiyu Liu, Yuxin Luo, Mengke Cui, Siyuan Han, Jingshu Peng, Jin Li, and Lei Chen. 2025. BitTuner: A Toolbox for Automatically Configuring Learned Data Compressors. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 4548–4551.

[42] Qiyu Liu, Yanyan Shen, and Lei Chen. 2021. LHist: Towards Learning Multi-dimensional Histogram for Massive Spatial Data. In *ICDE*. IEEE, 1188–1199.

[43] Qiyu Liu, Yanyan Shen, and Lei Chen. 2022. HAP: An Efficient Hamming Space Index Based on Augmented Pigeonhole Principle. In *SIGMOD Conference*. ACM, 917–930.

[44] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable Learned Bloom Filters for Data Streams. *Proc. VLDB Endow.* 13, 11 (2020), 2355–2367.

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

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

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

[48] 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.

[49] 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.

[50] 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.

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

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

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

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

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

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

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

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

[59] 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.

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

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

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

[63] pytorch [n.d.]. PyTorch. https://pytorch.org/. Accessed: 2024-06-12.

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

[65] 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.

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

[67] 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.

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

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

[70] tensorflow [n.d.]. TensorFlow. https://www.tensorflow.org/. Accessed: 2024-06-12.

[71] 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).

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

[73] 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.

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

[75] 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.

[76] wikidata [n.d.]. Wikidata. https://www.wikidata.org/wiki/Wikidata:Main_Page. Accessed: 2024-11-12.

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

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

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

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