BBC: Improving Large-k Approximate Nearest Neighbor Search with a Bucket-based Result Collector

Ziqi Yin, Gao Cong, Kai Zeng, Jinwei Zhu, Bin Cui
Nanyang Technological University, Singapore Huawei Technologies Co., Ltd Peking University, China
ziqi003@e.ntu.edu.sg,gaocong@ntu.edu.sg,{zhujinwei,kai.zeng}@huawei.com,bin.cui@pku.edu.cn

ABSTRACT

Although Approximate Nearest Neighbor (ANN) search has been extensively studied, large-k ANN queries that aim to retrieve a large number of nearest neighbors remain underexplored, despite their numerous real-world applications. Existing ANN methods face significant performance degradation for such queries. In this work, we first investigate the reasons for the performance degradation of quantization-based ANN indexes: (1) the inefficiency of existing top-k collectors, which incurs significant overhead in candidate maintenance, and (2) the reduced pruning effectiveness of quantization methods, which leads to a costly re-ranking process. To address this, we propose a novel bucket-based result collector (BBC) to enhance the efficiency of existing quantization-based ANN indexes for large-k ANN queries. BBC introduces two key components: (1) a bucket-based result buffer that organizes candidates into buckets by their distances to the query. This design reduces ranking costs and improves cache efficiency, enabling high-performance maintenance of a candidate superset and a lightweight final selection of top-k results. (2) two re-ranking algorithms tailored for different types of quantization methods, which accelerate their re-ranking process by reducing either the number of candidate objects to be re-ranked or cache misses. Extensive experiments on real-world datasets demonstrate that BBC accelerates existing quantization-based ANN methods by up to 3.8× at recall@k = 0.95 for large-k ANN queries.

PVLDB Reference Format:
Ziqi Yin, Gao Cong, Kai Zeng, Jinwei Zhu, Bin Cui. BBC: Improving Large-k Approximate Nearest Neighbor Search with a Bucket-based Result Collector. PVLDB, 19(11): 3468 - 3482, 2026.
doi:10.14778/3836663.3836702

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

1 INTRODUCTION

Driven by the rapid process of large-scale machine learning and generative AI techniques, efficient vector search has become a critical capability in modern data systems [56, 63, 67, 87]. Vector databases [60, 61, 79, 80] now serve as the foundation for querying embeddings generated by deep learning models, where Approximate Nearest Neighbor (ANN) search is the core computational primitive [60, 82], trading off minor accuracy for significantly improved efficiency [31, 84]. In practice, ANN algorithms are typically extended to retrieve approximate k-nearest neighbors to meet the demands of real-world applications.

Although ANN queries have been extensively studied, most existing studies design and evaluate their methods under small k settings, which are typically in the range of a few tens to a few hundreds [13, 46]. This setting is well-suited for some applications such as retrieval-augmented generation (RAG) [43], where an ANN index retrieves the top-10 relevant documents for a large language model to generate the final responses. However, many real-world applications involve large k scenarios (e.g., $k \geq 5{,}000$), where a large number of nearest neighbors need to be retrieved for each query. We refer to such queries as large-k ANN queries and next present several of their applications.

(1) In model training or fine-tuning scenarios, it is often necessary to efficiently retrieve a large set of highly relevant data samples to construct the training dataset. These samples typically number in the tens of thousands in many real-world applications, such as retrieving videos or images that capture specific types of dangerous driving behavior. In such cases, the initial query is often ambiguous, such as an image representing a driving behavior. Data engineers usually need to perform multiple iterations of search, refining the query by selecting better examples from the retrieved results before identifying an effective query and obtaining a satisfactory set of results.
(2) In document retrieval, state-of-the-art methods often adopts a retrieve-and-rerank pipeline [41]. Documents are encoded into embeddings using pre-trained language models [93]. An ANN index built on these embeddings retrieves tens of thousands of candidate documents for each query. Subsequently, a more sophisticated model, such as ColBERT [39], which encodes queries and documents into token embeddings and computes similarity by aggregating token-level similarities, re-ranks the candidates to obtain the final results.
(3) In industrial recommendation systems [22, 38], hundreds of thousands of candidates are first retrieved via an ANN index and then re-ranked using more computationally expensive models to produce the final recommendations.

However, existing ANN methods face significant performance degradation when handling large-k ANN queries, as demonstrated in our evaluation on four representative ANN indexes: the IVF [37], the popular graph-based method HNSW [50], and two quantization-based methods IVF+RaBitQ [21] and IVF+PQ [34]. An example result on the C4 dataset is shown in Figure 1 and similar trends are observed across other datasets. We observe that at recall@k = 0.95, when k increases from 100 to 5,000, the throughput of IVF+RaBitQ

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836702

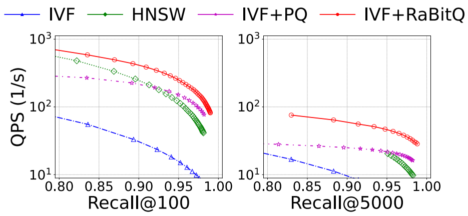


Figure 1: Querying Performance of IVF, HNSW, IVF+PQ, and IVF+RaBitQ on the C4 dataset at k = 100 and k = 5000.

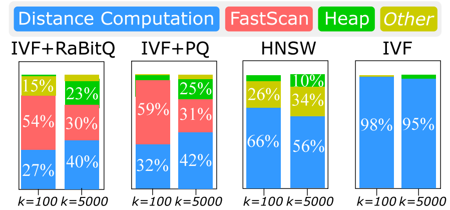


Figure 2: Time Overhead Breakdown of four methods at different k, where "Distance computation" denotes exact distance computation, "FastScan" denotes estimated distance computation, "Heap" denotes heap operations, and "Other" covers the remaining costs.

drops from 227 queries per second (QPS) to 47 QPS, a 4.8 × slowdown; HNSW's throughput falls from 113 QPS to 20 QPS, showing a larger 5.7× slowdown. In this work, we focus on optimizing quantization-based methods for large-k ANN queries for two reasons: 1) quantization-based methods exhibit superior performance for large-k ANN queries; and 2) our empirical results and analysis[1] show that quantization-based methods are more robust to increase of k compared with graph-based methods such as HNSW.

Challenge 1. Quantization-based methods often face significant slowdowns in large-k ANN queries due to two primary challenges. The first stems from the inefficiency of existing top-k collectors when handling large k. These collectors are responsible for maintaining the k nearest candidates by storing each candidate's ID and distance to the query. Whenever a closer candidate is found, it replaces the farthest one, whose distance serves as the comparison threshold. Existing ANN studies commonly employ a binary-heap priority queue as the top-k collector. However, when k is large, the heap becomes inefficient under modern memory hierarchies consisting of L1, L2, L3 caches and main memory. The inefficiency arises because the heap size exceeds the capacity of the fastest L1 cache. For example, when k = 100, the distance–ID pairs occupy only 800 bytes, which is well within the 32 KB L1 cache capacity, allowing the heap reside entirely in the L1 cache and achieve low latency. However, when k = 5,000, the heap grows to 40,000 bytes, exceeding the L1 cache capability and causing frequent L1 cache misses that significantly increase latency. For example, in IVF+RaBitQ, its share of runtime rises from 2% at k = 100 to 23% at k = 5,000 under recall@k = 0.95, as shown in Figure 2. This observation is consistent with prior findings [40] that link priority queue performance to L1 cache misses.

Challenge 2. As k increases, the pruning effectiveness of quantization methods drops. These methods accelerate ANN search by estimating distances to quickly prune distant objects and can be grouped into two categories: (1) *unbounded* methods that prune solely by estimated distances (e.g., PQ), and (2) *bounded* methods that provide probabilistically guaranteed distance bounds (e.g., RaBitQ). Although they differ in querying strategies, both types of approaches require re-ranking a growing number of candidates as k increases. In *bounded* methods, the candidate set is maintained by a top-k collector, and any object whose estimated bounds overlap with the current threshold is re-ranked. As k grows, the number of such objects increases, leading to higher re-ranking overhead. For example, in IVF+RaBitQ, the runtime share of exact distance computation rises from 27% at k = 100 to 40% at k = 5,000, as shown in Figure 2. Similarly, *unbounded* methods retrieve and re-rank a candidate set whose size is typically several times larger than k, to maintain high recall. Consequently, as k increases, the re-ranking cost grows roughly linearly, resulting in a significant slowdown.

Our Method. To address the first challenge, we observe that existing collectors typically maintain an exact top-k set, where each stored candidate may be accessed and replaced. When k is large and the stored distance-id pairs exceed the L1 cache capacity, this results in frequent L1 cache misses and high maintenance overhead. To overcome this, we propose a novel **bucket-based result buffer** (BBC) that reorganizes candidate storage to maximize cache efficiency, preserving exact results without maintaining exact top-k order. Specifically, it partitions the estimated distance range between the query and data objects into non-overlapping sub-ranges through one-dimensional quantization. Each sub-range corresponds to a bucket consisting of two linear buffers that sequentially store the IDs and distances of candidates falling within this sub-range. This design offers two key benefits: (1) it lowers ranking complexity by organizing candidates into buckets based on distance, maintaining ordering across buckets while avoiding ordering within each bucket; (2) it reduces L1 cache misses, as the sequential insertion pattern within each bucket enables the hardware prefetching mechanism to proactively prefetch relevant memory blocks into the L1 cache for subsequent candidate insertions. Leveraging both bucket-level ranking and the number of objects stored in each bucket, we efficiently identify the buckets that contain the exact top-k results and the threshold bucket that holds boundary candidates near the threshold distance. Together, these buckets form a *candidate superset.* The upper bound distance of the threshold bucket serves as a *relaxed threshold.* Both the candidate superset and the relaxed threshold can be maintained at low cost, since only a small number of buckets are involved. This design eliminates per-object access and replacement by operating only on bucket-level structures, where distant buckets are implicitly pruned once they become irrelevant. When the exact top-k set is required, the final selection is restricted to the threshold bucket, as the bucket-level ranking guarantees that all preceding buckets contain only closer candidates. Leveraging the distance concentration phenomenon [31, 84], we further show that under an equal-depth partition of the distance range, the error between the relaxed and exact thresholds are on the order of 10^{-2}, keeping the

[1]The suboptimal performance of HNSW at large k arises from the fact that graph-based ANN indexes are designed for small k. These methods construct a proximity graph during indexing and use it to navigate queries toward nearby objects to reduce search space. However, when k is large, the graph traversal inevitably expands to a larger portion of the graph, incurring significant additional overhead, as shown in Figure 2.

selection cost negligible, as supported by both theoretical guidance (Section 3.2) and empirical evaluation (Section 4.2).

To address the second limitation, we design two new re-ranking algorithms tailored to different quantization methods. For *bounded* methods, we aim to skip re-ranking objects that are guaranteed to be either within or outside the top-k based on their estimated distance bounds, and re-rank only uncertain ones. We first formulate an minimal re-ranking scenario that minimizes the number of re-ranked objects without sacrificing accuracy and design a solution to achieve it. However, this approach incurs substantial heap overhead that offsets its benefits. To address this, we design a greedy re-ranking algorithm that integrates seamlessly with the proposed result buffer, significantly reducing the number of re-ranked objects with a small extra cost. For *unbounded* methods, the number of re-ranked objects cannot be reduced since their estimated distances lacking guaranties. Instead, we propose an early re-ranking strategy that tightly couples re-ranking with the result buffer. It computes exact distances for objects predicted to enter the re-ranking pool when their data are accessed, thus effectively reducing cache misses during exact distance computation.

Building on these techniques, we develop a novel bucket-based result collector (BBC) that substantially enhances the efficiency of existing quantization-based methods for large-k ANN queries without compromising accuracy. BBC integrates the proposed result buffer to gather candidates efficiently and incorporates the newly designed re-ranking algorithms to produce the final results.

The main contributions of this work are summarized as follows:

(1) We identify and analyze two major limitations of quantization-based methods for large-k ANN queries: inefficiency of top-k collectors and declined pruning effectiveness, both of which cause substantial performance degradation. To our knowledge, these findings have not been reported in prior work.
(2) We propose a novel bucket-based result collector (BBC), which introduces a bucket-based result buffer serving as a cache-efficient top-k collector and two new re-ranking algorithms tailored to *bounded* and *unbounded* quantization methods. To the best of our knowledge, this is the first framework explicitly designed for large-k ANN queries.
(3) Extensive experiments on real-world datasets show that: (1) BBC accelerates existing quantization-based methods on large-k ANN queries by up to 3.8× speedup at recall@k = 0.95; (2) the proposed result buffer reduces the overhead of the top-k collector by an order of magnitude; and (3) the new re-ranking algorithms speed up the re-ranking efficiency by up to 1.8×.

2 BACKGROUND AND MOTIVATIONS

Problem Statement. Given a dataset D of N data objects, each represented by a d-dimensional vector, the Approximate Nearest Neighbor (ANN) query involves two phases: indexing and querying. In the indexing phase, it constructs a data structure based on the data vectors. In the querying phase, given a query q, it aims to efficiently retrieve the k nearest vectors using the data structure under a similarity metric. Most existing studies [33, 34, 49, 80] focus on the setup where k is small (e.g., $k = 100$). However, as discussed in Section 1, the query with large k (e.g., $k \geq 5,000$) is important in many applications and introduces new challenges in algorithm design. In this study, we aim to develop an efficient solution for large-k ANN queries ($k \geq$ 5,000) under commonly used similarity metrics in vector spaces, such as metric distances like Euclidean distance, as well as cosine similarity and inner product. Unless otherwise specified, we adopt Euclidean distance as the default metric and also discuss extensions to other similarity measures.

Modern Memory Hierarchy. The modern memory hierarchy typically consists of L1, L2, and L3 caches and main memory. The L1 cache, located closest to the CPU core, provides the fastest access speed but has the smallest capacity, typically 32 KB. It stores the most frequently accessed data and instructions to minimize access latency. The L2 and L3 caches are located farther from the CPU cores, offering slower access than L1 but faster than main memory. Although main memory is much larger, typically ranging from several tens to hundreds of gigabytes (GB), its high access latency makes it slower. Therefore, the data required by the CPU are loaded into the L1 cache before processing. The transfer of data from main memory or from the L3/L2 caches to the L1 cache results in L1 cache misses, which introduce high latency, as shown in prior experimental evaluations [40]. To reduce L1 cache misses, modern hardware automatically prefetches memory blocks adjacent to the currently accessed data into the L1 cache, while evicting less frequently used blocks to lower cache levels or main memory.

Top-k Collector. Most ANN studies employ a binary-heap priority queue to maintain the k nearest neighbors. Despite the binary heap achieving low latency when k is small, it incurs frequent L1 cache misses and significantly higher latency at larger values of k, as discussed in Section 1. Although modern hardware supports L1 cache prefetching, it is only effective for data structures with regular memory access patterns, such as sequentially stored linear buffers. For data structures with irregular and unpredictable access patterns, such as binary heaps, its applicability is much more limited [48].

Quantization. Quantization methods accelerate ANN search by efficiently estimating distances to prune distant candidates. During indexing, they construct a quantization codebook, assign each data vector to their nearest codebook vector, and store the codebook ids as compact quantization codes. At query time, they (1) pre-compute distances between the query and the codebook vectors, and (2) use these pre-computed distances to efficiently estimate query-object distances from the stored quantization codes, also known as quantized distances. In practice, these methods are often integrated with an inverted file (IVF) index [37] to improve querying performance, with IVF+RaBitQ [21] and IVF+PQ [34] being representative methods. The IVF index partitions the data vectors into $n_{cluster}$ clusters using the k-means algorithm during indexing and routes each query to the n_{probe} nearest clusters at query time, thereby reducing the search space. Within these clusters, quantization methods execute their querying algorithm to obtain the final results.

Based on their pruning mechanisms and query processing strategies, quantization methods can be categorized into two types. In particular, *bounded* methods such as RaBitQ [20, 21] provide an estimated distance range with a high probabilistic guarantee (e.g., 99%) and leverage these bounds for pruning. Specifically, it employs a top-k collector to maintain the currently found k nearest neighbors and re-ranks any objects whose lower bounds fall below the collector's current threshold, as they may potentially enter the top-k results. Because the objects stored in the collector often

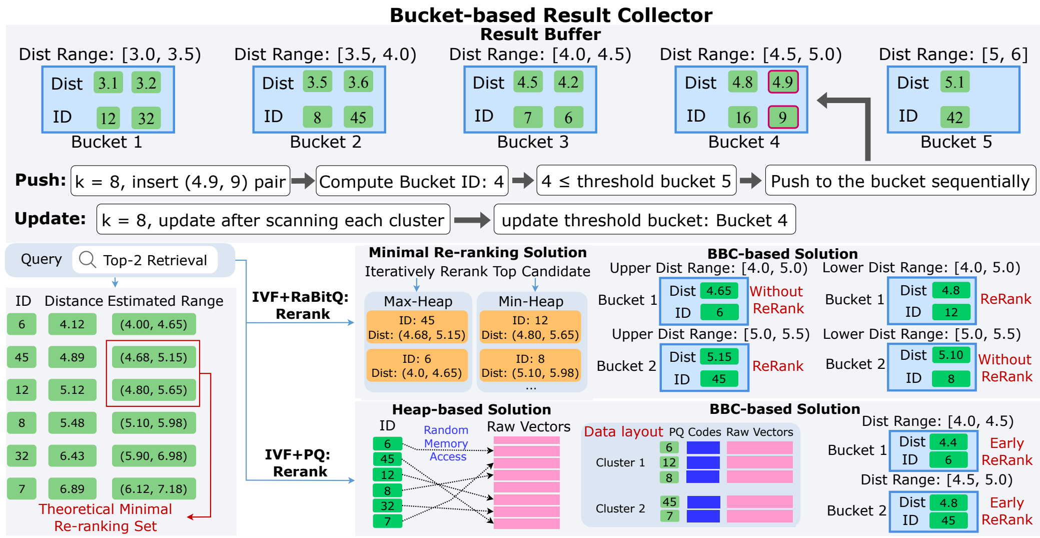


Figure 3: Illustration of the Proposed Bucket-based Result Collector.

change rapidly during the early stage, the number of re-ranked objects is typically several times larger than k, as shown in Section 4.2. When k increases, this results in a significantly higher re-ranking cost, as shown in Figure 2. *Unbounded* methods such as Product Quantization (PQ) [34] generally preset a hyperparameter $n_{cand} \gg k$ (e.g., n_{cand}=3,000 and k=100) to determine how many candidates are retrieved based on their estimated distances. When a query arrives, these methods employ a top-k collector to gather n_{cand} candidates according to their estimated distances, which are then re-ranked to produce the final results. Here, n_{cand} is typically several times larger than k to achieve high recall. As k increases, the number of re-ranked objects increases linearly, leading to a significant increase in the re-ranking cost, as shown in Figure 2.

Motivations. While many ANN methods have been developed, no prior work has specifically investigated large-k ANN queries. Therefore, we evaluate several representative ANN methods on large-k ANN queries, where the top-k collector is implemented using a binary-heap priority queue. We present the evaluation results on the C4 dataset in Figure 1, which contains over 14 million passages. Figure 2 further presents a breakdown of time overheads under different k based on VTune profiling. Details of the experimental setup are given in Section 4.1. The experimental results reveal the performance degradation of these methods on large-k ANN queries, and we have highlighted two major limitations of existing quantization-based methods in the Introduction. This motivate us to design a new algorithm for large-k ANN queries.

3 THE BBC METHOD

3.1 Overview

In this section, we propose a novel bucket-based result collector (BBC), composed of two new components: a bucket-based result buffer serving as the top-k collector and two re-ranking algorithms. We proceed to give an overview of the two components.

First, the result buffer partitions the estimated distance range between the query and data objects into non-overlapping sub-ranges using one-dimensional quantization, as shown in Figure 3. Each sub-range corresponds to a bucket that contains two linear buffers, which sequentially store the IDs and distances of candidates. As presented in the Introduction, this design (1) provides bucket-level ranking of candidates based on their distances to the query, where candidates are ordered across buckets but remain unordered within each bucket (e.g., objects in Bucket 1 always have smaller distances to the query than those in Bucket 2). This design differs from binary heaps, which maintain a strict bucket-level order throughout, thereby reducing the ranking cost; and (2) exploits the hardware prefetching mechanism to reduce L1 cache misses. This is because only the tail of the recently accessed linear buffer typically resides in the L1 cache, while the preceding elements are evicted, substantially alleviating L1 cache pressure.

Leveraging both bucket-level ranking and the number of objects stored in each bucket, we can efficiently identify the buckets that contain the exact top-k results and the threshold bucket that holds boundary candidates near the threshold distance. For example, in Figure 3, when k = 8, bucket 5 serves as the threshold bucket because the cumulative number of objects in the first five buckets reaches 8 (before inserting the object 9 with distance 4.9). The candidates within these buckets together form a *candidate superset*. The upper bound distance of the threshold bucket serves as the *relaxed threshold*, which is 6 in this case. At query phase, both can be efficiently updated. For example, when inserting a new object (e.g., object 9 with distance 4.9 in Fig. 3), we compute its bucket ID based on its distance to the query, compares it with the threshold bucket ID (e.g., Bucket 5), and appends the distance and ID to its corresponding bucket (e.g., pushing object 9 to Bucket 4). After insertion, the threshold bucket can be updated by traversing buckets in order until the accumulated number of candidates reaches k, as the buckets are organized by distance range. In Figure 3, when k = 8, the threshold bucket shifts from bucket 5 to bucket 4 after inserting object 9. The more distant buckets (e.g., Bucket 5) are no longer visited and are implicitly dropped without incurring additional cost. Finally, we only need to select a subset of objects from the threshold bucket and combine them with the objects in the preceding buckets to obtain the exact top-k results.

Second, two re-ranking algorithms are designed for different types of quantization approaches to accelerate their re-ranking process, as illustrated in Figure 3. For *bounded* methods, we establish a theoretical minimal re-ranking set that minimizes the number of re-ranked objects without accuracy loss and propose a solution to achieve it. However, this solution incurs considerable heap overhead. To mitigate this, we propose a re-ranking algorithm built on our result buffer. It skips re-ranking objects that are definitely either inside or outside the top-k and re-ranks only the uncertain ones near the threshold bucket. For *unbounded* methods, we propose a novel early re-ranking algorithm that re-ranks many objects predicted to enter the re-ranking pool when accessing their data, reducing L1 cache misses from random memory access.

In the rest of this section, we present the two components of BBC: (1) the result buffer (Section 3.2) and (2) the two re-ranking algorithms, along with the improved search algorithm (Section 3.3).

3.2 Result Buffer

We now describe how the result buffer partitions the estimated distance range between the query and objects into m non-overlapping sub-ranges, as defined by the codebook C:

$$C = \{c_1, c_2, \ldots, c_{m+1}\}, \quad c_i < c_{i+1}. \tag{1}$$

Accordingly, the sub-ranges are formally defined, each corresponding to a bucket $B[i]$:

$$B[i] = [c_i, c_{i+1}), \quad i = 1, 2, \ldots, m. \tag{2}$$

For each input distance–ID pair, the result buffer determines the corresponding bucket by locating the interval in which the distance falls. Specifically, an object o_i is assigned to bucket $B[j]$ if its distance from the query, $\mathrm{Dist}(q, o_i)$, satisfies $c_j \le \mathrm{Dist}(q, o_i) < c_{j+1}$. The distance $\mathrm{Dist}(q, o_i)$ can refer to either the exact distance $\mathrm{Dist}_{\mathrm{exact}}$ or the estimated distance $\mathrm{Dist}_{\mathrm{quant}}$. Next we turn to the three core operations of the result buffer: `Push`, `Update`, and the `Collect` function, which are used to collect the top-k results.

The `Push` function in Algorithm 1 details the procedure for inserting an object into the result buffer. Specifically, when a new object is inserted, the result buffer first computes its distance to the query and the corresponding bucket ID (line 2). It then compares this bucket index with the threshold bucket ID (line 3) and appends the distance–ID pair to the assigned bucket if the index does not exceed the threshold bucket ID (line 4). Here, using the threshold bucket ID for comparison essentially treats the upper bound of the threshold bucket's distance range as the relaxed threshold. In addition, bucket ID comparisons can benefit from fast SIMD-based integer comparison instructions, enabling simultaneous comparison of batches of objects. The quantization code computation can also be accelerated using SIMD instructions, as discussed later.

The `update` function in Algorithm 1 (lines 5-11) describes the process of updating the threshold bucket. In particular, the buckets in the result buffer are arranged in ascending order of distance range, as shown in Figure 3. We accumulate the number of candidates in buckets in order until the total number reaches or exceeds k (lines 6-9). Once this condition is met, the visited bucket is identified as the threshold bucket and returned (line 10). If the total number

Algorithm 1: The Workflow of Result Buffer

```
   Input: Result buffer B; a set of clusters cl to be scanned; retrieval
          size k;
   Output: The result buffer B
1  Function Push(q, o_i, τ, B):
2      Compute Dist(q, o_i) and bucket ID j;
3      if j ≤ τ then
4          Append (dist, o_i) to the tail of B[j];
5  Function Update(B, k):
6      s ← 0;
7      foreach B[i] ∈ B do
8          s ← s + |B[i]|;
9          if s > k then
10             return i;
11     return ∞;
12 Function Collect(q, cl, B, k):
13     Initialize threshold bucket τ ← ∞;
14     Construct codebook C for B;
15     foreach cr ∈ cl do
16         foreach o_i ∈ cr do
17             Push(q, o_i, τ, B);
18         τ ← Update(B, k);
19     Res ← ∅;
20     for i ← 0 to τ − 1 do
21         Res ← Res ∪ B[i];
22     s ← k − |Res|;
23     Select the top-s candidates from B_τ and append to Res ;
24     return Res;
```

of objects stored in the result buffer is less than k, the threshold bucket is set to ∞, allowing all objects to be accepted (line 11). Since only dozens of buckets are involved, the update cost is negligible. Once the threshold bucket is updated, the more distant buckets are no longer accessed and are implicitly dropped, thus incurring no additional time cost for candidate maintenance.

The `collect` function in Algorithm 1 describes the workflow of collecting top-k results based on (estimated) distance in IVF-based ANN methods. Specifically, we first initialize the threshold bucket to ∞ and construct the codebook C for the result buffer, whose generation will be discussed later (lines 13–14). Objects are then inserted into the result buffer B within each cluster using the `Push` function (lines 15–17). The threshold bucket is updated after processing each cluster using the `Update` function (line 18). This update is performed once per cluster because the relaxed threshold is very close to the exact threshold (as will be discussed later), and updating once per cluster helps to reduce the update cost. Once all clusters have been processed, the objects in the buckets preceding the threshold bucket are inserted into the result set *Res* (lines 19-21). This is because the bucket-level ranking of candidates ensures that these objects are closer to the query than those in the subsequent buckets, thus falling within the top-k candidates. Finally, we compute the number of objects s that need to be selected from the threshold bucket (line 22), choose the top-s objects from it, and add them to *Res* (line 23). *Res* now contains the exact top-k results and is returned (line 24). This design substantially reduces the cost of maintaining exact top-k results, as the selection operation is performed only once in the final stage within a single bucket.

Deciding the Number of Buckets m. A key consideration is how to determine the number of buckets m. If m is too large, the increased number of vectors to be written raises the L1 cache miss rate. If m is too small, objects are concentrated in just a few vectors, resulting in a costly final selection process (as shown in Section 4.2). To balance these factors, we aim to maximize m based on the L1 cache capacity C_{L1}, while accounting for the space required by quantization codes C_{quant} and lookup tables C_{lut}. Accordingly, the number of buckets m is given by the following equation:

$$m = \frac{C_{L1} - C_{quant} - C_{lut}}{256}, \tag{3}$$

where $C_{quant} = 2 \times 32 \times \frac{d}{M} \times \frac{B}{8}$ is the space for quantization codes of the current and subsequent processed batches. Here, d is the dimensionality, M is the number of sub-vectors, and B denotes the bits per sub-vector. $C_{lut} = \frac{d}{M} \times 2^B$ is the size of the lookup table. The denominator 256 reflects hardware prefetching considerations: the result buffer maintains two separate linear buffers for IDs and distances, and modern hardware typically prefetches two 64-byte cache lines for sequential access, we reserve $m \times 2 \times 2 \times 64 = 256m$ bytes to ensure the active tails of all buckets reside in the L1 cache. Since Equation 3 incorporates hardware parameters, m adapts to changes in hardware settings. It also accounts for the memory overhead of the ANN algorithm and not all buckets are frequently accessed, thus small variations in m do not lead to significant increases in L1 cache misses or latency (see Exp-6).

Deciding the Codebook C. We now describe how to construct the codebook $C = \{c_1, c_2, \ldots, c_{m+1}\}$. The codebook is designed to satisfy two critical properties: (1) Precision, ensuring that the relaxed threshold (e.g., c_i) remains close to the precise value. This is crucial because it affects the efficiency of the final selection, and significant deviations could make the process time-consuming; (2) Efficiency, ensuring low latency in both codebook generation and quantization code computation. Since the result buffer serves as the top-k collector in the ANN search, slow generation and computation would offset its benefits. Note that the codebook requires dynamic generation for each query, rather than pre-computation, as query-object distance distributions vary across queries.

To quantify precision, we formalize it as the total quantization error. Given a query q, for each object $o_i \in D$, let $\mathrm{Dist}(q, o_i)$ denote its distance to the query and a_i denote its assigned quantization code. Each code corresponds to a bucket whose upper boundary is $c_{a_{i+1}}$. The precision objective $Cost(C, D)$ is then defined as:

$$Cost(C, D) = \sum_{i=1}^{N} \left| c_{a_{i+1}} - \mathrm{Dist}(q, o_i) \right|, \tag{4}$$

The corresponding optimal solution $\{C, D\}$ is given by:

$$\{C, D\} = \arg\min_{\{C,D\}} Cost(C, D). \tag{5}$$

The problem is then to find a centroid codebook C that minimizes Equation 4. Because computing the distance for all objects is impractical, an exact solution to this problem becomes infeasible. The

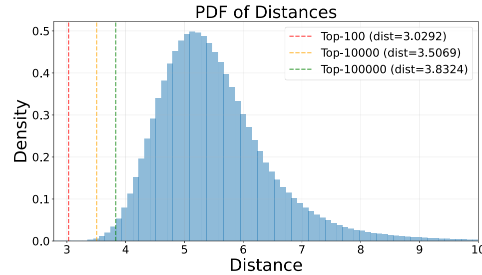


Figure 4: The probability density function (PDF) of distances between the query and data vectors on the C4 dataset.

two common approximate methods for one-dimensional quantization are equal-depth partition and equal-width partition, which are described as follows:

- **Equal-depth partition** [58]: This method divides the data range into intervals with equal data points, resulting in non-uniform intervals based on the data distribution. While it maximizes bucket utilization and has higher precision, it involves a slower generation and computation process.
- **Equal-width partition** [66]: This method divides the data range into intervals of equal width, regardless of the data distribution. While it enables faster codebook generation and code computation, it suffers from lower precision because it does not adapt to the data distribution.

First, we demonstrate that the precision level of equal-depth quantization meets our requirements. In particular, in high-dimensional space, the distance concentration phenomenon [31, 84] causes distances between vectors to concentrate around a value with only a small deviation. Based on this, we provide (1) a quantitative analysis with a proof sketch, with the full proof deferred to the technical report [91] and (2) experimental validation on real-world high-dimensional datasets (see Exp-4).

Theorem 3.1 (Expected Mean Absolute Error). *Let $q, o \in \mathbb{R}^d$ be random vectors with norms satisfying $\alpha < \|q\|_2, \|o\|_2 < \beta$, and let $R = \|q - o\|_2 \in [0, 2\beta]$ denote their Euclidean distance. Assume that R has a sub-Gaussian left tail around a concentration value $\mu \in [\alpha, 2\beta]$, i.e., there exist absolute constants $c_1, c_0 > 0$ such that for any $x \leq \mu$,*

$$\mathbb{P}(R \leq x) \leq 2c_1 \exp(-c_0 d(\mu - x)^2).$$

For an integer $m \geq 2$, let $b_1, \ldots, b_{m+1}$ be an equal-depth partition of $[0, \mu]$ with $b_1 = 0$ and $b_{m+1} = \mu$, satisfying

$$\mathbb{P}(R \leq b_i) = \frac{i-1}{m} \mathbb{P}(R \leq \mu), \quad i = 1, \ldots, m+1.$$

Define the quantized distance $\widehat{R}$ by mapping $R \in [b_i, b_{i+1}]$ to b_{i+1}. Then the expected absolute quantization error over $[0, \mu]$ satisfies

$$\mathbb{E}|R - \widehat{R}| \;\leq\; c_1 \sqrt{\frac{\pi}{c_0 d}} \;+\; \sqrt{\frac{\log(2c_1 m)}{c_0 d}}.$$

Proof Sketch. The key idea is to view the quantization error as the area between two cumulative distribution functions, i.e., the integral of the difference between the true CDF and the quantized CDF, which can be represented as $\int_0^{\mu} \Delta(x)\, dx$, where $\Delta(x) = \mathbb{P}(R \leq x) - \mathbb{P}(\widehat{R} \leq x)$. We partition the integration regions $[0, \mu]$ into two regions: $[0, z]$ and $[z, \mu]$, where the transition point is chosen as

$z = \mu - \sqrt{\frac{\log(2c_1 m)}{c_0 d}}$ according to sub-Gaussian concentration. For $[0, z]$, the integral is bounded by the Gaussian integral

$$\int_0^z \Delta(x)dx \leq \int_0^z \mathbb{P}(R \leq x)dx \leq 2c_1 \int_{-\infty}^{\mu} e^{-c_0 d(\mu - x)^2} dx = c_1 \sqrt{\frac{\pi}{c_0 d}}$$

For $[z, \sqrt{2}]$, we use a worst-case bound of 1 on the CDF difference, so the contribution is bounded by the interval width $\sqrt{\frac{\log(2c_1 m)}{c_0 d}}$.

Summing the two contributions yields the stated bound. □

Extension of Theorem 3.1 to real-world datasets. First, when d is in the hundreds and m from Equation 3 is on the order of dozens, the bound is on the order of 10^{-2}, which is sufficiently tight. Second, the assumption that R has a sub-Gaussian left tail around a concentration value follows from the distance concentration phenomenon [1, 21, 31, 84], and is widely observed in real-world datasets. We further illustrate this phenomenon on the C4 dataset (Figure 4), where the query-data distances concentrate around 5.2, with most values lying in $[4, 8]$. Third, we restrict quantization to $[0, \mu]$ because the result buffer targets collecting the top-k results, so equal-depth quantization is applied only to this subset rather than the entire dataset. Figure 4 illustrates this: distances between the query and the top-100,000 candidates (e.g., 3.8324) are significantly smaller than the concentration distance (e.g., 5.2), making the theoretical bound conservative. Skewed data distributions and extreme outliers far from the query therefore do not affect the bound, since such points are excluded from the equal-depth quantization process. We empirically validate the above bound on real-world datasets, showing that the gap between the quantized values and the true threshold values is on the order of 10^{-2} or smaller (see Exp-4).

Codebook Generation Based on Estimated Distance. We now detail how to construct the codebook C for the result buffer. To efficiently estimate the distance range of the result buffer, we sample a subset of the dataset, denoted as D_{sample}, which typically consists of tens of thousands of objects, and quickly compute their estimated distances. In practice, D_{sample} is formed using objects from the 5–10 nearest clusters. We then perform a partial sort on the estimated distances to obtain the top-k results. Since the partial sort is performed only once, its computational cost is negligible. Afterwards, we derive the minimum distance $d_{\min}$ and maximum distance $d_{\max}$ from the local top-k results. It is evident that the $d_{\max}$ are necessarily farther than the true top-k estimated distance. Therefore, the estimated distance range will not overflow at d_{max}, any potential overflow can only occur at d_{min}, which rather rare and can be safely handled by the boundary control, as described below. To maintain computational efficiency, we follow the approach of [57], which first computes n_{ew} equal-width buckets and then reassigns these buckets into m equal-depth buckets, where a lookup table map is used to preserve the correspondence between them.

Quantization Code Computation. For inserted objects, we first compute their equal-width codes and subsequently clamp those outside the range $[0, m]$ to prevent boundary overflow, and then obtain the mapping ID via the lookup table, as formulated below:

$$a_i = \text{map}\left[\text{clamp}\left(\left\lfloor \frac{\text{Dist(q, o}_\text{i}) - d_{\min}}{\delta} \right\rfloor, 0, m\right)\right] \tag{6}$$

where $\text{clamp}(x, y, z) = \max(y, \min(x, z))$. SIMD instructions can be employed to accelerate the computation of $\left\lfloor \frac{\text{Dist(q,o}_\text{i}) - d_{\min}}{\delta} \right\rfloor$ as well as the clamp operation, enabling the batch processing of dozens of objects. We fix n_{ew} to 256, so that the mapping can be stored using uint8, which requires only 256 bytes in total, allowing it to reside in L1 cache with minimal memory overhead. Besides, AVX instructions can process 4× as many uint8 values per instruction as float32 values, which speeds up batch comparison.

Algorithm 2: Minimal Re-ranking solution of IVF+RaBitQ

Input: Query q, retrieval size k, and the object set O to be scanned.
Output: Top-k results

```
Initialize max-heap H_u and min-heap H_l;
foreach o ∈ O do
    Compute the lower and upper bounds o_lb, o_ub from o to q;
    if o_ub < H_u.top_ub then
        Insert (o, o_lb, o_ub) into H_u;
        if |H_u| > k then
            Move top item from H_u to H_l;
    else if o_lb < H_u.top_ub then
        Insert (o, o_lb) into H_l;
Vis ← ∅;
while H_u.top_ub > H_l.top_lb do
    Select o from H_u or H_l with smaller lower bound;
    Compute exact distance of o and insert into H_u;
    Vis ← Vis ∪ {o};
    if |H_u| > k then
        Move top item from H_u to H_l if not visited;
return H_u;
```

Complexity Analysis. The time complexity of the Push operation is $O(1)$, dominated by the bucket ID computation, which is much cheaper than the $O(\log(k))$ cost of binary heap. The `update` operation has a time complexity of $O(m)$. Since m computed by Equation 3 is typically dozens and the operation is performed periodically, its amortized time complexity can be considered O(1), which is more efficient than the $O(\log(k))$ cost of binary heap. The `collect` operation has a time complexity of $O(N)$ but is invoked only once at the end of the query process, its cost is negligible.

3.3 Re-rank Algorithms

We now discuss how to integrate the result buffer with existing quantization-based methods: IVF+RaBitQ and IVF+PQ. We aim to enhance their efficiency for handling large-k ANN queries. To achieve this, we introduce several new techniques.

Integrating with IVF+RabitQ. We introduce a novel re-ranking algorithm based on our result buffer by exploiting the bound property [21], namely, the true distance fall within the estimated bound with high probability (e.g., 99%). This property enables us to efficiently estimate the distance range between each visited data object and the query. As a result, it is unnecessary to re-rank data objects that are guaranteed to be within the top-k or definitively outside the top-k. Re-ranking is required only for ambiguous candidates whose inclusion in the top-k remains uncertain. Based on this insight, we formally define the minimal re-ranking scenario.

Observation 1 (Minimal Re-ranking Scenario). Dist_k *denotes the distance between the query and the boundary top-k object. When*

Algorithm 3: Improved Search Algorithm of IVF+RaBitQ

Input: Query q, retrieved size k, and the clusters to be scanned cl.
Output: Top-k results.

1 Initialize two result buffers B_u and B_l;
2 Generate codebook C for B_u and B_l;
3 **foreach** $cr \in cl$ **do**
4   **foreach** $o_i \in cr$ **do**
5   Compute the lower/upper bounds o_{lb}/o_{ub} for o_i and their respective quantization codes a_{lb}, a_{ub};
6   **if** $a_{ub} < \tau$ **then**
7   Insert o_i into B_u;
8   **else if** $a_{lb} < \tau$ **then**
9   Insert o_i into B_l;
10   $\tau \leftarrow \mathtt{Update}(B_u, k)$;
11 Insert objects from B_u into B_l;
12 $i \leftarrow 0$, $j \leftarrow \tau$;
13 Initialize result buffer B_{exact} with codebook C;
14 **while** $i < j$ **do**
15   **foreach** $o \in B_l[i] \cup B_u[j]$ **do**
16   Compute exact distance $Dist_{\text{exact}}(q, o)$;
17   Insert o into B_{exact} based on $Dist_{\text{exact}}(q, o)$;
18   Clear $B_l[i]$ and $B_u[j]$;
19   $i \leftarrow 0$, $j \leftarrow$ index of the threshold bucket in $B_u \cup B_{\text{exact}}$;
20   **while** $B_l[i]$*.empty()* **do**
21     $i \leftarrow i + 1$;
22 **return** the objects in $B_u \cup B_{\text{exact}}$ as the final top-k results;

the objective is to minimize the number of objects to be re-ranked, the minimal re-ranking scenario for bounded quantization method is to re-rank only those objects $o \in D$ whose lower and upper bounds satisfy $[o_{lb}, o_{ub}] \cap \{\text{Dist}_k\} \neq \emptyset$. This set represents the theoretical minimal set of objects that must be re-ranked without accuracy loss.

Figure 3 illustrates a top-2 case. Objects 6 and 45 are the true top-2 results, with object 45 as the boundary item. The estimated ranges are RaBitQ distance bounds. Candidates whose ranges overlap with the boundary distance must be re-ranked (e.g., objects 45 and 12), forming the theoretical minimal re-ranking set.

Solution to the Minimal Re-ranking Scenario. We design Algorithm 2 to achieve the minimal re-ranking scenario described in Observation 1, which consists of two phases. In the candidate collection phase, it maintains the top-k candidates with the smallest upper bounds in a max-heap H_u, while storing objects whose lower bounds fall below the k-th upper bound in a min-heap H_l (lines 2-9). Then it iteratively selects the object with the smaller lower bound from the tops of the two heaps for exact distance evaluation, and updates the heaps accordingly (lines 11–16). The process terminates when the largest upper bound in H_u is smaller than the smallest lower bound in H_l (line 13). The correctness proof is provided in the technical report [91]. Figure 3 illustrates this process, where re-ranking stops once objects 6 and 8 reach the top, as the lower bound of object 8 exceeds the upper bound of object 6. However, maintaining a max-heap of size k and an unbounded min-heap incurs prohibitively high overhead when k is large. As a result, this approach can even be slower than IVF+RaBitQ (Section 4.2).

Algorithm 4: Improved Search Algorithm of IVF+PQ

Input: Query q, the number of objects to be retrieved k, the clusters to be scanned cl, the number of objects to be re-ranked n_{cand}.
Output: Top-k results.

Initialize result buffer B;
Sample a subset $D_{\text{sample}} \subset D$;
Produce codebook C for B;
Generate the predicted threshold bucket τ^{pred};
foreach $cr \in cl$ **do**
  foreach $o_i \in cr$ **do**
    Compute $\text{Dist}_{\text{quant}}(q, o_i)$ and bucket id a_i;
    if $a_i < \tau$ **then**
      if $a_i < \tau^{\text{pred}}$ **then**
        Compute exact distance $\text{Dist}_{\text{exact}}(q, o_i)$;
        Insert o and $\text{Dist}_{\text{exact}}(q, o_i)$ into B;
      else
        Insert o and $\text{Dist}_{\text{quant}}(q, o_i)$ into B;
  Update τ^{pred} and τ;
Re-rank remaining candidates in B;
return top-k results;

To address this, we propose a greedy re-ranking algorithm, which significantly reduces the number of items to be re-ranked. Figure 3 provides an example, where we replace the two heaps with two result buffers, and only re-rank items within some boundary buckets, thereby enabling early termination of the re-ranking process. Algorithm 3 summarizes the enhanced search algorithm. Specifically, we maintain two result buffers that share the same codebook (lines 1–2). Next, we collect candidates based on the lower and upper bounds of o_i. In particular, when the object's upper bound lies within the top-k upper bounds, the object is inserted into C_u; otherwise, if its lower bound falls below the threshold, it is inserted into C_l as a possible top-k item (lines 3-9). For the collected candidates, we first re-collect the falsely dropped candidates into C_l from C_u (line 11). Then we greedily re-rank all items in the marginal buckets, that is, the top bucket of C_l and the threshold bucket of C_u, and insert the results into a new result buffer C_{exact} for storing items with exact distance (lines 15-17). After each computation iteration, we clear the candidate buckets and update the marginal buckets until $i \geq j$ (lines 18-21). The loop stops when $i \geq j$ as there is no further potential candidates exist. Finally, we return the results in $C_u \cup C_{\text{exact}}$ as the final results. Experiments show that this method achieves near-minimal re-ranking reduction, leading to a substantial decrease in re-ranking time (as shown in Section 4.2).

Integrating with IVF+PQ. Due to the unbounded nature of the PQ algorithm, we cannot reduce the number of objects to be re-ranked. Therefore, we propose an early re-ranking algorithm built upon our result buffer to reduce the cache misses caused by the random memory access patterns of IVF+PQ, as illustrated in Figure 3. In particular, we optimize the memory data layout to store each object's PQ code and embedding contiguously. When estimating the distance of an object from its PQ code, we predict whether it will enter the re-ranking pool based on the estimated distance. If so, we immediately compute its exact distance to reduce L1 cache misses.

Table 1: Dataset Statistics

Dataset	$\|D\|$	d	$\|Q\|$	Size (GB)
WiKi	10,000,000	1,536	1,000	58
C4	14,252,691	1,024	1,000	54
MSMARCO	18,000,000	768	1,000	51
Deep100M	100,000,000	96	10,000	35

Algorithm 4 details our proposed re-ranking approach and summarizes the improved search algorithm. During the sampling stage of codebook generation (line 2), we use the bucket containing $\left(\frac{|O_{\text{sample}}|}{|O|} \times n_{cand}\right)$-th quantized distance as the predicted threshold bucket τ^{pred} (line 3). Then, during the scanning phase, for each object in a bucket preceding the threshold bucket, we compute the exact distance if its bucket ID $a_i < \tau^{\text{pred}}$ and insert the exact value; otherwise, we insert its quantized distance (lines 5-13). After scanning each cluster, we update the predicted threshold τ^{pred} using the $\left(\frac{|O_{\text{scanned}}|}{|O|} \times n_{cand}\right)$-th quantized distance and update the threshold bucket τ as described in Algorithm 1 (line 14). This approach leads to a substantial reduction in L1 cache misses and re-ranking time, as verified in Exp-5. A concern is that τ^{pred} might be too large, causing unnecessary re-ranking. In practice, this does not occur because clusters are traversed from nearest to farthest based on query-centroid distance (line 5), which produces a distribution skewed toward smaller values, thereby keeping τ^{pred} low.

Extension to different Metrics. Our proposed result buffer applies to various similarity measures, including metric distances such as Euclidean distance, as well as cosine similarity and inner product, as these measures exhibit the distance concentration phenomenon in high-dimensional spaces [1] and our result buffer does not rely on specific properties of metric distances. Similarly, the two re-ranking algorithms are compatible with any distance measure supported by PQ, RaBitQ, or other ANN techniques, as they do not depend on specific properties of metric distances as well.

4 EXPERIMENTS

4.1 Evaluation Setup

Datasets. We conduct experiments on four real-world datasets[2]. Specifically, we evaluate our method and baselines on the Wiki, C4, MSMARCO, and Deep100M datasets. The statistics of these datasets are listed in Table 1. Details of each dataset are stated in the technical report due to page limitations [91].

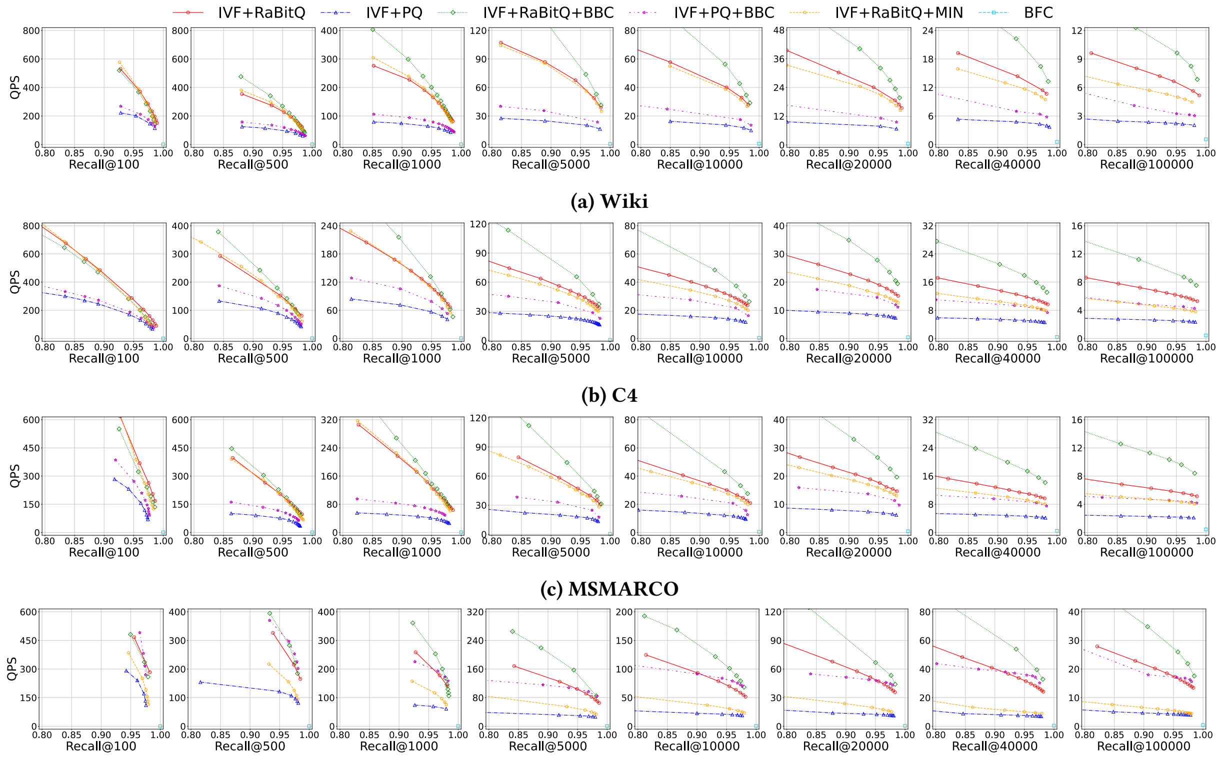
Baselines. First, we evaluate four representative ANN methods for large-k ANN queries, as described in the Introduction and detailed in the technical report [91]. We integrate our proposed BBC with existing quantization-based methods, IVF+PQ and IVF+RabitQ, yielding IVF+PQ+BBC and IVF+RaBitQ+BBC. We compare them with their original counterparts and also include the minimal re-ranking solution for IVF+RaBitQ, described in Section 3.3 and denoted as IVF+RaBitQ+MIN, as a baseline. We also include brute-force search, denoted as BFC, as a baseline for comparison. Second, to compare the efficiency of our proposed result buffer, denoted as RB, for collecting the top-k results, we consider four baselines: the binary heap (denoted as Heap), the cache-optimized d-ary Heap [36] (denoted as d-Heap), the sorted linear buffer used in [2] (denoted as Sorted), and the lazy update method (denoted as Lazy). In particular, Sorted maintains candidates in a sorted linear buffer, shifting the entire buffer upon each insertion. Lazy stores candidates whose distances to the query are below the current threshold in a linear buffer and updates both the buffer and the threshold using a SIMD-optimized partial sorting operation, after each cluster is processed.

Evaluation metrics. First, for ANN query, we use recall rate recall@k = $\frac{\mathcal{R}\cap\tilde{\mathcal{R}}}{k}$ [46, 82] to evaluate the accuracy of search results and queries per second QPS = $\frac{|Q|}{t}$ [18] to evaluate the search's efficiency. Here, $\mathcal{R}$ represents the result retrieved by the ANN index, $\tilde{\mathcal{R}}$ denotes the ground-truth result computed by the brute-force search. QPS= $\frac{|Q|}{t}$ [18] is the ratio of the number of queries ($|Q|$) to the total search time (t), representing the number of queries processed per second. We also report the relative error between the distances of approximate nearest neighbors in $\mathcal{R}$ and true nearest neighbors in $\tilde{\mathcal{R}}$ to assess the quality of the retrieved results. The relative error [46, 73] is computed as $\frac{1}{k}\sum_{i=0}^{k}(\frac{\text{Dist}(q,\mathcal{R}[i])}{\text{Dist}(q,\tilde{\mathcal{R}}[i])}-1)$, where q denotes the query, $\tilde{\mathcal{R}}[i]$ is the i-th true nearest neighbor of q in $\tilde{\mathcal{R}}$, and $\mathcal{R}[i]$ is the i-th approximate nearest neighbor retrieved by the ANN algorithm. Second, to compare our proposed result buffer with alternative approaches, we evaluate the time overhead (milliseconds) of collecting top-k results under varying dataset sizes and different values of k using VTune profiling.

Parameter Settings. Following the suggestion from Faiss [37], the number of clusters for IVF, IVF+RaBitQ, and IVF+PQ is set to approximately $\sqrt{|D|}$, which is 4,096 in our experiments. For IVF+RaBitQ, we use the default quantization parameters from the original paper [21], $\epsilon_0 = 1.9$ and $B_q = 4$. For IVF+PQ, the number of sub-vectors M is set to $\frac{d}{4}$, and the number of bits per sub-vector is set to 4, resulting in $B = d$, following the original settings [4, 27, 37]. For HNSW, during indexing, we set the candidate list size $ef_{\text{construction}}$ and the maximum number of edges per node M to 200 and 32, respectively, following previous studies [50]. To evaluate the capability of methods in handling large-k ANN queries under different k, we vary k from 5,000 to 100,000, which are widely used in practical scenarios [22, 41]. Additionally, we also present results for k values ranging from 100 to 1,000, showing that BBC does not slow down existing methods for small k. At query time, IVF and IVF+RaBitQ vary the number of clusters for routing n_{probe} from 10 to 1,200 and IVF+PQ use the same n_{probe} as IVF+RaBitQ. For IVF+PQ, for each dataset and given k, the n_{cand} parameter is then fine-tuned to maximize QPS at a target recall of 0.95, under the constraint that the configuration must also be capable of achieving 0.98 recall, which is listed in the full version due to page limitations [91]. For HNSW, we increase ef_{search} from k in increments of $\frac{k}{2}$. For BBC, since our CPU, like most modern CPUs, has an L1 cache of $C_{L1} = 32$ KB, we set the number of buckets m according to Equation 3: 56 for Wiki, 80 for C4, 92 for MSMARCO, and 120 for Deep100M.

Implementations. The baselines and our method are all implemented in C++. First, we use the hnswlib implementation [51], a widely adopted industry-standard library, for HNSW. For IVF+RaBitQ, we adopt its open-source implementation [21]; and for IVF and

[2]Note that our techniques introduce negligible additional space overhead. Therefore, we defer the memory cost analysis to the technical report [91].



(a) Wiki

(b) C4

(c) MSMARCO

(d) Deep100M

Figure 5: The accuracy-efficiency trade-off results under different k (upper and right is better).

IVF+PQ, we implement these methods based on IVF+RaBitQ because they share a common index structure. Second, for Heap, we use the STL implementation. For d-Heap, we use the Boost Library. For Sorted, we use its official implementation. For Lazy, we use the x86simdsort library. All experiments are run on a machine equipped with an AMD Ryzen Threadripper PRO 5965WX 7.0GHz processor (supporting AVX2) and 128 GB of RAM.

4.2 Experimental Results

Exp-1: ANN Query Performance. The querying results over the four datasets are shown in Figure 5. We have the following observations: **(1) BBC achieves 1.4×–3.8× speedup over existing quantization-based methods for large-k ANN queries.** Specifically, on the Deep100M dataset, when k = 100,000 and recall@k = 0.95, IVF+PQ+BBC requires 61 ms per query, compared to 233 ms per query for IVF+PQ, achieving a 3.8× speedup. When k = 5,000, on the Wiki dataset at recall@k = 0.98, IVF+PQ+BBC achieves a 1.4× acceleration over IVF+PQ (42 ms vs. 61 ms). The gain stems from the high efficiency of our proposed result buffer and the effectiveness of the newly designed re-ranking algorithm. **(2) The acceleration provided by BBC becomes more significant as k increases.** For example, on Deep100M at recall@k = 0.95, as k increases from 5,000 to 100,000, the acceleration ratio of IVF+PQ+BBC over IVF+PQ increases from 2.9× at k = 5,000 (2.8 ms vs. 8.2 ms) to 3.8× at k = 100,000 (61 ms vs. 233 ms). This is because, as k increases, existing collectors and re-ranking algorithms incur significantly higher costs, whereas our proposed result buffer remains efficient (Exp-3), and the newly designed ranking algorithm can further reduce the re-ranking cost substantially (Exp-5). **(3) When k is large, IVF+RaBitQ+MIN is significantly slower than IVF+RaBitQ+BBC, and even slower than IVF+RaBitQ.** Across the four datasets, IVF+RaBitQ+MIN performs worse than IVF+RaBitQ and IVF+RaBitQ+BBC at k = 5,000 and the performance gap widens considerably as k increases. This can be attributed to the high cost of heap operations, which outweighs the efficiency gained from re-ranking fewer objects and leads to higher overhead when k is large, consistent with our previous analysis. **(4) IVF+RaBitQ+BBC and IVF+PQ+BBC are comparable to IVF+RaBitQ and IVF+PQ at k = 100, while outperforming them when $k \geq 500$, with the performance gap widening as k increases.** This suggests BBC should be used when $k \geq 500$ to improve query performance. **(5) Brute-force search is significantly slower than existing ANN indexes.** For example, on the Wiki dataset, at recall@100,000 = 0.98, BFC takes 1.8s per query, whereas the slowest ANN method, IVF+PQ, requires only 0.5s.

Exp-2: Relative error for large-k ANN queries. Figure 6 reports the relative error of large-k ANN queries on the C4 dataset, and similar trends are observed across the other datasets. The results show that **(1) As recall increases, the relative error drops sharply from 0.3% (recall@5,000 = 0.89) to 0.01% (recall@5,000 = 0.98)**, and **(2) the relative error remains nearly constant across different k.** These indicate that retrieved results remain highly accurate for large-k ANN queries, regardless of the value of k.

Exp-3: Latency of Top-k Collectors. We evaluate the time overhead of our proposed result buffer, denoted as RB, and compare it with four baselines. Using quantized distances from IVF+RaBitQ as

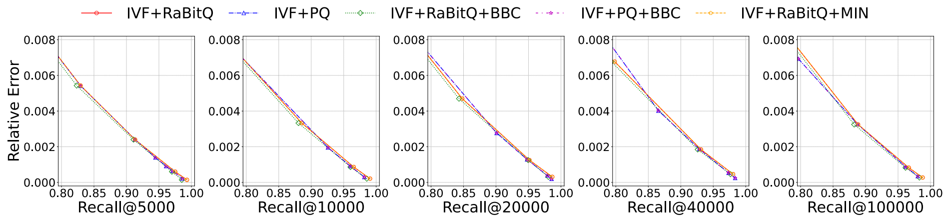


Figure 6: Relative error of large-k ANN queries on the C4 dataset (lower is better).

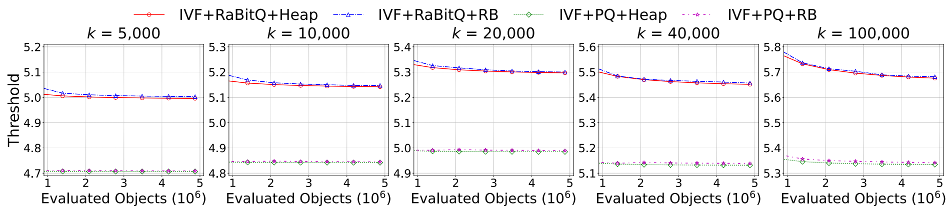


Figure 7: Comparison of threshold values generated by the result buffer and binary heap on the C4 Dataset.

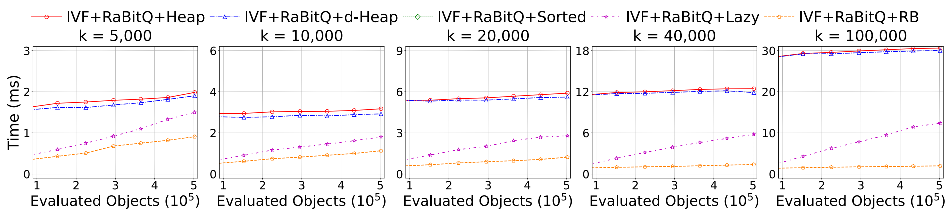
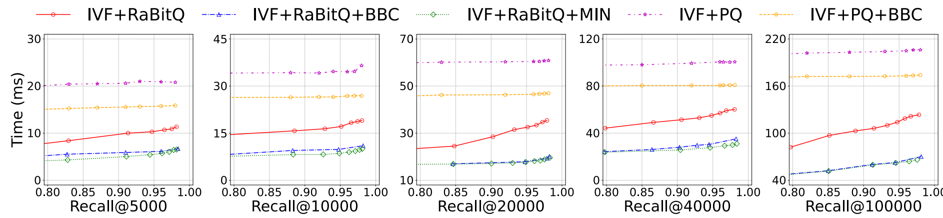


Figure 8: Overhead of top-k collectors under varying numbers of evaluated objects on the Deep100M dataset.

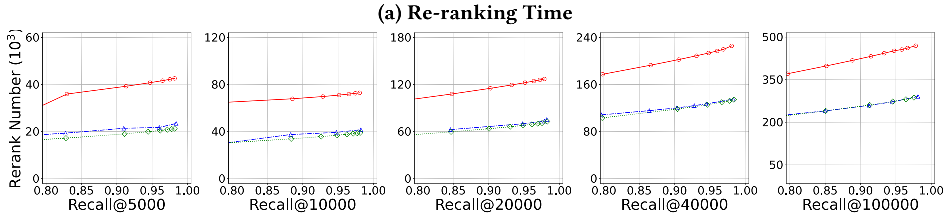
keys, we use these methods to gather top-k results, with k ranging from 5,000 to 100,000, under varying numbers of evaluated objects. The number of evaluated objects is controlled by n_{probe}, which is tuned as in the previous experiment. The isolated time overhead of these methods is measured by VTune. Due to page limitations, we present the results on Deep100M in Figure 8 for IVF+RaBitQ, while similar trends are observed on other datasets and IVF+PQ. We find that: **(1) Our result buffer RB is significantly faster than existing collectors, achieving up to an order of magnitude improvement.** For example, on the Deep100M dataset, when k = 100,000 and n_{probe} = 210, for IVF+RaBitQ, RB takes only 2.0 ms, compared to 30.6 ms for Heap and 12.3 ms for Lazy, achieving an order-of-magnitude speedup. **(2) RB remains efficient at higher values of k, while Lazy and Heap face significant degradation.** For example, on Deep100M, when k = 5,000 and n_{probe} = 210, for IVF+RaBitQ, RB takes 0.9 ms, compared to 1.5 ms for Lazy and 1.9 ms for Heap. When k = 100,000 and n_{probe} = 210, RB takes 2.0 ms, while Lazy takes 12.3 ms and Heap takes 30.6 ms. This efficiency is due to RB avoiding maintaining an exact top-k set throughout, which is consistent with previous experimental results. We further report the overall time for top-k collection, including collectors, quantized distance computation, and other operations, and the total L1 cache misses during this process in the technical report [91] to validate that our speedup stems from reduced cache misses.

Exp-4: Gap between the relaxed and exact thresholds. We compare the threshold values generated by our proposed result buffer with those produced by the binary heap, thereby empirically validating the effectiveness of the equal-depth method. The experimental results on the C4 dataset are shown in Figure 7. **It indicates that the gap between the relaxed and exact threshold values remains consistently small, with differences on the order of 10^{-2}, thereby validating the effectiveness of Theorem 3.1.**

Exp-5: Comparison of Re-ranking Algorithms. We compare our proposed re-ranking algorithm with the naive algorithm. In particular, Figure 9 presents the recall@k and the corresponding re-ranking times/re-ranked objects on the C4 dataset, based on VTune profiling results. The key observations are as follows: **(1) Our re-rank algorithms significantly accelerate the re-ranking process.** Specifically, when k = 20,000 and recall@k = 0.95, IVF+RaBitQ+BBC requires about 32 ms per query, whereas IVF+PQ takes around 18 ms, achieving a 1.8× speedup. Similarly, when k = 20,000 and recall@k = 0.95, IVF+PQ+BBC requires approximately 45 ms per query for re-ranking, compared to about 60 ms for IVF+PQ, resulting in a 1.3× speedup. The speedup of IVF+RaBitQ+BBC mainly results from a reduction in the number of re-ranked items, while the speedup of IVF+PQ+BBC can be attributed to reduced cache misses, as detailed below. **(2) The number of re-ranked objects in IVF+RaBitQ+BBC is significantly reduced compared to IVF+RaBitQ.** Specifically, when k = 100,000 and recall@k = 0.95, IVF+RaBitQ re-ranks 450,067 objects, whereas IVF+RaBitQ+BBC re-ranks 223,142 objects, representing a reduction of nearly 50%. This result is consistent with the 1.8× speedup observed above. Notably, the number of re-ranked objects in IVF+RaBitQ+BBC is only slightly higher than that in the minimal re-ranking scenario of IVF+RaBitQ+MIN, demonstrating the effectiveness of our method. Due to page limitations, we report the L1 cache miss counts of different re-ranking methods in the technical report [91] to further demonstrate the effectiveness of our proposed re-ranking method.



(a) Re-ranking Time



(b) Number of Re-ranked Objects

Figure 9: Re-ranking Time and Number of Re-ranked Objects on the C4 Dataset

Table 2: Top-k collection time (ms) under varying values of m on the C4 dataset (n_{probe} = 90)

$m \backslash k$	5,000	10,000	20,000	40,000	100,000
8	4.74	5.22	5.65	5.91	6.28
32	3.59	3.74	3.99	4.28	4.91
80	3.50	3.68	3.82	4.14	4.81
128	3.58	3.72	3.89	4.16	4.87
256	3.70	3.74	3.91	4.28	5.00

Exp-6: Parameter Sensitivity Study. We evaluate the impact of the number of buckets m following the procedure in Exp-3. Specifically, we measure the top-k collection latency of IVF+RaBitQ+BBC on the C4 dataset with n_{probe} = 90 and vary m from 8 to 256. The experimental results are shown in Table 2, where m = 80 is the optimal number of buckets as determined by Equation 3. The results indicate that (1) the value of m computed by Equation 3 achieves the lowest latency; (2) small deviations from this value result in only marginal latency increases; (3) very small values of m cause a substantial latency increase because objects become concentrated in a few buckets, making the final selection costly; and (4) very large values of m also increase latency due to more frequent L1 cache misses induced by the large m.

5 RELATED WORK

Approximate Nearest Neighbor Search. Various Approximate Nearest Neighbor (ANN) search methods have been proposed [46, 54, 61, 64], which are typically classified into four categories: graph-based methods [8, 17, 18, 26, 28, 49, 50, 82, 89, 90], quantization methods [9, 15, 20, 21, 23, 24, 34, 35, 52, 62, 70, 77, 85, 92], hashing-based methods [12, 19, 29, 30, 42, 45, 47, 59, 65, 75–77, 81], and tree-based methods [5, 10, 11, 69, 88]. Among these methods, IVF and graph-based indexes are widely used in industry [44, 61] and quantization methods have proven highly effective in saving memory and accelerating query processing [4, 21, 27, 32]. For a comprehensive overview, we refer readers to recent tutorials [14], benchmark/experimental evaluations [6, 7, 13, 71, 83], and surveys [54, 60, 61] for details. Although ANN queries have been extensively studied, to the best of our knowledge, the large-k ANN query studied in this work has not yet been specifically investigated.

Quantization. We focus on improving quantization-based methods. The quantization of high-dimensional vectors has been extensively explored in the literature [9, 20, 21, 23, 27, 34, 35, 53, 55, 62, 85, 92]. Early research on quantization focuses on reducing quantization error, with Product Quantization (PQ) [23, 34] as the representative method. Recently, RaBitQ [20, 21] is proposed, which provides theoretical bounds for estimated distances. With the help of SIMD-based implementations (a.k.a. FastScan), these methods have achieved great success in accelerating other ANN approaches [3, 20, 21, 25, 27, 32]. In this study, we integrate BBC into IVF+PQ and IVF+RaBitQ to demonstrate its plug-and-play ability to improve the efficiency of quantization methods for large-k ANN queries.

Priority Queue. Priority queues have been extensively studied, with the binary heap and its variants being the most common implementations due to their $O(\log(n))$ insertion and deletion time complexity [86]. However, in large-k ANN queries, this theoretical efficiency is no longer effective, as L1 cache misses dominate the runtime overhead. This observation is consistent with previous experimental evaluations [40], which report a strong correlation between the priority queue's processing time and L1 cache miss rates. Consequently, studies that focus on optimizing the theoretical time complexity of priority queues [16, 68, 78] offer limited benefits in our context. It is noted that [2] replaces the heap with a sorted linear buffer, which is also used as a baseline for comparison. Several top-k collectors are designed for GPUs [37, 72, 74], such as FAISS's WarpSelect. However, as reported in [37], these methods also face performance degradation when k is large.

6 CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we propose a novel bucket-based result collector (BBC) to accelerate quantization-based methods for large-k ANN queries, which consists of two key components: a bucket-based result buffer and two re-ranking algorithms. One potential future direction is to modify the BBC to support graph-quantization methods for large-k ANN queries. Another promising direction is adapting BBC for GPU settings to accelerate batch large-k ANN queries.

REFERENCES

[1] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. 2001. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*. 420–434.

[2] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore L. Willke. 2023. Similarity search in the blink of an eye with compressed indices. *Proc. VLDB Endow.* 16, 11 (2023), 3433–3446.

[3] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *Proc. VLDB Endow.* 9, 4 (2015), 288–299.

[4] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2017. Accelerated Nearest Neighbor Search with Quick ADC. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval, ICMR 2017, Bucharest, Romania, June 6-9, 2017*. 159–166.

[5] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-Index: Pushing the Scalability-Accuracy Boundary for Approximate kNN Search in High-Dimensional Spaces. *Proceedings of the VLDB Endowment* 11, 8 (2018), 906–919.

[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] Martin Aumüller and Matteo Ceccarello. 2023. Recent Approaches and Trends in Approximate Nearest Neighbor Search, with Remarks on Benchmarking. *IEEE Data Eng. Bull.* 47, 3 (2023), 89–105.

[8] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-based vector search: An experimental evaluation of the state-of-the-art. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–31.

[9] Artem Babenko and Victor S. Lempitsky. 2014. Additive Quantization for Extreme Vector Compression. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. 931–938.

[10] Alina Beygelzimer, Sham M. Kakade, and John Langford. 2006. Cover Trees for Nearest Neighbor. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006 (ACM International Conference Proceeding Series)*, Vol. 148. 97–104.

[11] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB conference, Athens, Greece*. 426–435.

[12] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*. 253–262.

[13] Magdalen Dobson, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2023. Scaling Graph-Based ANNS Algorithms to Billion-Size Datasets: A Comparative Analysis. *CoRR* abs/2305.04359 (2023).

[14] Karima Echihabi, Themis Palpanas, and Kostas Zoumpatianos. 2021. New Trends in High-d Vector Similarity Search: AI-Driven, Progressive, and Distributed. *Proc. VLDB Endow.* 14, 12 (2021), 3198–3201.

[15] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. 2000. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proceedings of the ninth international conference on Information and knowledge management*. 202–209.

[16] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)* 34, 3 (1987), 596–615.

[17] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search with Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 8 (2022), 4139–4150.

[18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. *Proceedings of the VLDB Endowment* 12, 5 (2019), 461–474.

[19] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-Sensitive Hashing Scheme Based on Dynamic Collision Counting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 541–552.

[20] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2025. Practical and asymptotically optimal quantization of high-dimensional vectors in euclidean space for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[21] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[22] Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. 2021. Learning an end-to-end structure for retrieval in large-scale recommendations. In *Proceedings of the 30th ACM international conference on information & knowledge management*. 524–533.

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

[24] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2013. Iterative Quantization: A Procrustean Approach to Learning Binary Codes for Large-Scale Image Retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 12 (2013), 2916–2929.

[25] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2025. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 1 (2025), 80:1–80:26.

[26] Yuntao Gui, Peiqi Yin, Xiao Yan, Chaorui Zhang, Weixi Zhang, and James Cheng. 2026. Pilotann: Memory-bounded gpu acceleration for vector search. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*. 348–358.

[27] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, Vol. 119. 3887–3896.

[28] Ben Harwood and Tom Drummond. 2016. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5713–5722.

[29] Jae-Pil Heo, Youngwoon Lee, Junfeng He, Shih-Fu Chang, and Sung-Eui Yoon. 2015. Spherical Hashing: Binary Code Embedding with Hyperspheres. *IEEE Trans. Pattern Anal. Mach. Intell.* 37, 11 (2015), 2304–2316.

[30] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 9, 1 (2015), 1–12.

[31] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*. 604–613.

[32] Yahoo Japan. 2018. Neighborhood Graph and Tree for Indexing High-dimensional Data. https://github.com/yahoojapan/NGT. Accessed: 2024-04-17.

[33] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems* 32 (2019).

[34] H Jégou, M Douze, and C Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.

[35] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cédric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, and Gustavo Alonso. 2023. Co-design Hardware and Algorithm for Vector Search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*. 87:1–87:15.

[36] Donald B Johnson. 1975. Priority queues with update and finding minimum spanning trees. *Inform. Process. Lett.* 4, 3 (1975), 53–57.

[37] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[38] Sujay Khandagale, Bhawna Juneja, Prabhat Agarwal, Aditya Subramanian, Jaewon Yang, and Yuting Wang. 2025. InteractRank: Personalized Web-Scale Search Pre-Ranking with Cross Interaction Features. In *Companion Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025 - 2 May 2025*. 287–295.

[39] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.

[40] Daniel H Larkin, Siddhartha Sen, and Robert E Tarjan. 2014. A back-to-basics empirical study of priority queues. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 61–72.

[41] Jinhyuk Lee, Zhuyun Dai, Sai Meher Karthik Duddu, Tao Lei, Iftekhar Naim, Ming-Wei Chang, and Vincent Zhao. 2023. Rethinking the role of token retrieval in multi-vector retrieval. *Advances in Neural Information Processing Systems* 36 (2023), 15384–15405.

[42] Yifan Lei, Qiang Huang, Mohan S. Kankanhalli, and Anthony K. H. Tung. 2020. Locality-Sensitive Hashing Scheme Based on Longest Circular Co-Substring. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, Online Conference [Portland, or, USA], June 14-19, 2020*. 2589–2599.

[43] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[44] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June*

14-19, 2020. 2539–2554.

[45] Jinfeng Li, Xiao Yan, Jie Zhang, An Xu, James Cheng, Jie Liu, Kelvin Kai Wing Ng, and Ti-Chung Cheng. 2018. A General and Efficient Querying Method for Learning to Hash. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 1333–1347.

[46] 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 Transactions on Knowledge and Data Engineering* 32, 8 (2020), 1475–1488.

[47] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. *Proc. VLDB Endow.* 13, 9 (2020), 1443–1455.

[48] Varun Malhotra and Christos Kozyrakis. 2006. Library-based prefetching for pointer-intensive applications. In *Technical report*. Computer Systems Laboratory, Stanford University.

[49] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate Nearest Neighbor Algorithm Based on Navigable Small World Graphs. 45 (2014), 61–68.

[50] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.

[51] Yu A Malkov and Dmitry A Yashunin. 2020. hnswlib: Hierarchical Navigable Small World graphs. https://github.com/nmslib/hnswlib. Accessed: 2025-09-09.

[52] Julieta Martinez, Shobhit Zakhmi, Holger H. Hoos, and James J. Little. 2018. LSQ++: Lower Running Time and Higher Recall in Multi-codebook Quantization. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XVI (Lecture Notes in Computer Science)*, Vol. 11220. 508–523.

[53] Yusuke Matsui, Ryota Hinami, and Shin'ichi Satoh. 2018. Reconfigurable inverted index. In *Proceedings of the 26th ACM international conference on Multimedia*. 1715–1723.

[54] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, and Shin'ichi Satoh. 2018. A Survey of Product Quantization. *ITE Transactions on Media Technology and Applications* 6, 1 (2018), 2–10.

[55] Yusuke Matsui, Toshihiko Yamasaki, and Kiyoharu Aizawa. 2015. Pqtable: Fast exact asymmetric distance neighbor search for product quantization using hash tables. In *Proceedings of the IEEE International Conference on Computer Vision*. 1940–1948.

[56] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-throughput vector similarity search in knowledge graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[57] Hamid Mousavi and Carlo Zaniolo. 2011. Fast and accurate computation of equi-depth histograms over data streams. In *Proceedings of the 14th international conference on extending database technology*. 69–80.

[58] M Muralikrishna and David J DeWitt. 1988. Equi-depth multidimensional histograms. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*. 28–36.

[59] Parth Nagarkar and K Selçuk Candan. 2018. Pslsh: An index structure for efficient execution of set queries in high-dimensional spaces. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 477–486.

[60] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *VLDB J.* 33, 5 (2024), 1591–1615.

[61] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the 2024 International Conference on Management of Data*. Santiago AA Chile, 597–604.

[62] John Paparrizos, Ikraduya Edian, Chunwei Liu, Aaron J. Elmore, and Michael J. Franklin. 2022. Fast Adaptive Similarity Search through Variance-Aware Quantization. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. 2969–2983.

[63] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4171–4184.

[64] Marco Patella and Paolo Ciaccia. 2009. Approximate similarity search: A multi-faceted problem. *Journal of Discrete Algorithms* 7, 1 (2009), 36–48.

[65] Ninh Pham and Tao Liu. 2022. Falconn++: A locality-sensitive filtering approach for approximate nearest neighbor search. *Advances in Neural Information Processing Systems* 35 (2022), 31186–31198.

[66] Gregory Piatetsky-Shapiro and Charles Connell. 1984. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record* 14, 2 (1984), 256–276.

[67] Jeffrey Pound, Floris Chabert, Arjun Bhushan, Ankur Goswami, Anil Pacaci, and Shihabur Rahman Chowdhury. 2025. MicroNN: An On-device Disk-resident Updatable Vector Database. In *Companion of the 2025 International Conference on Management of Data*. 608–621.

[68] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[69] Parikshit Ram and Kaushik Sinha. 2019. Revisiting Kd-Tree for Nearest Neighbor Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. 1378–1388.

[70] Patrick Schäfer, Jakob Brand, Ulf Leser, Botao Peng, and Themis Palpanas. 2024. Fast and Exact Similarity Search in less than a Blink of an Eye. *CoRR* abs/2411.17483 (2024).

[71] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. 2022. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*. 177–189.

[72] Nikos Sismanis, Nikos Pitsianis, and Xiaobai Sun. 2012. Parallel search of k-nearest neighbors with synchronous operations. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–6.

[73] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment* (2014).

[74] Xiaoxin Tang, Zhiyi Huang, David Eyers, Steven Mills, and Minyi Guo. 2015. Efficient selection algorithm for fast k-nn search on gpus. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 397–406.

[75] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2010. Efficient and Accurate Nearest Neighbor and Closest Pair Search in High-Dimensional Space. *ACM Transactions on Database Systems* 35, 3 (2010), 20:1–20:46.

[76] Yao Tian, Xi Zhao, and Xiaofang Zhou. 2024. DB-LSH 2.0: Locality-Sensitive Hashing with Query-Based Dynamic Bucketing. *IEEE Transactions on Knowledge and Data Engineering* 36, 3 (2024), 1000–1015.

[77] Ertem Tuncel, Hakan Ferhatosmanoglu, and Kenneth Rose. 2002. VQ-Index: An Index Structure for Similarity Searching in Multimedia Databases. In *Proceedings of the 10th ACM International Conference on Multimedia 2002, Juan Les Pins, France, December 1-6, 2002*. 543–552.

[78] Jean Vuillemin. 1978. A data structure for manipulating priority queues. *Commun. ACM* 21, 4 (1978), 309–315.

[79] Jianguo Wang, Eric Hanson, Guoliang Li, Yannis Papakonstantinou, Harsha Simhadri, and Charles Xie. 2024. Vector Databases: What's Really New and What's Next?(VLDB 2024 Panel). *Proceedings of the VLDB Endowment* 17, 12 (2024), 4505–4506.

[80] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2614–2627.

[81] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2018. A Survey on Learning to Hash. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 4 (2018), 769–790.

[82] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1964–1978.

[83] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph- and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *IEEE Data Eng. Bull.* 47, 3 (2023), 3–21.

[84] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. 194–205.

[85] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. 194–205.

[86] John William Joseph Williams. 1964. Algorithm 232: heapsort. *Commun. ACM* 7, 6 (1964), 347–348.

[87] Wen Yang, Tao Li, Gai Fang, and Hong Wei. 2020. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2241–2253.

[88] Peter Yianilos. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. (1993).

[89] Peiqi Yin, Xiao Yan, Shiyuan Deng, Hui Li, Yifan Zhu, Xiangyu Zhi, Jingqi Mao, Ran Xu, Wenliang Zhang, and James Cheng. 2026. {DistVS}: Large-scale Vector Search with {Compute-Memory} Disaggregation. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 449–467.

[90] Peiqi Yin, Xiao Yan, Qihui Zhou, Hui Li, Xiaolu Li, Meiling Wang, Lin Zhang, Xin Yao, and James Cheng. 2026. GPS: Revisiting the Data Layout for Disk-based High-Dimensional Vector Search. *Proceedings of the ACM on Management of*

Data 4, 3 (SIGMOD (2026), 1–29.
[91] Ziqi Yin, Gao Cong, Kai Zeng, Jinwei Zhu, and Bin Cui. 2026. BBC: Improving Large-k Approximate Nearest Neighbor Search with a Bucket-based Result Collector. *arXiv preprint arXiv:2604.01960* (2026).
[92] Ting Zhang, Chao Du, and Jingdong Wang. 2014. Composite Quantization for Approximate Nearest Neighbor Search. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014 (JMLR Workshop and Conference Proceedings)*, Vol. 32. 838–846.
[93] Wayne Xin Zhao, Jing Liu, Ruiyang Ren, and Ji-Rong Wen. 2024. Dense text retrieval based on pretrained language models: A survey. *ACM Transactions on Information Systems* 42, 4 (2024), 1–60.