GRIP: Multi-Store Capacity-Optimized High-Performance Nearest Neighbor Search for Vector Search Engine

Minjia Zhang
Microsoft AI and Research
Bellevue, WA, USA
minjiaz@microsoft.com

Yuxiong He
Microsoft AI and Research
Bellevue, WA, USA
yuxhe@microsoft.com

Abstract

This paper presents GRIP, an approximate nearest neighbor (ANN) search algorithm for building vector search engine which makes heavy use of the algorithm. GRIP is designed to retrieve documents at large-scale based on their semantic meanings in a scalable way. It is both fast and capacity-optimized. GRIP combines new algorithmic and system techniques to collaboratively optimize the use of memory, storage, and computation. The contributions include: (1) The first hybrid memory-storage ANN algorithm that allows ANN to benefit from both DRAM and SSDs simultaneously; (2) The design of a highly optimized indexing scheme that provides both memory-efficiency and high performance; (3) A cost analysis and a cost function for evaluating the capacity improvements of ANN algorithms. GRIP achieves an order of magnitude improvements on overall system efficiency, significantly reducing the cost of vector search, while attaining equal or higher accuracy, compared with the state-of-the-art.

CCS Concepts

• **Information systems → Search engine architectures and scalability**; **Search engine indexing**;

Keywords

Information retrieval; approximate nearest neighbor search; SSD

ACM Reference Format:
Minjia Zhang and Yuxiong He. 2019. GRIP: Multi-Store Capacity-Optimized High-Performance Nearest Neighbor Search for Vector Search Engine. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357384.3357938

1 Introduction

Retrieving relevant documents or images in response to a natural language query is an integral and indispensable task in information retrieval (IR). Due to the importance of this task, both academic research and industrial products have put a significant emphasis on designing effective and efficient IR systems. The recent successful advances of deep neural networks for various tasks have also impacted IR applications. In particular, it is possible to build a *vector search engine* to support *semantic search* by encoding both documents and queries into dense continuous vectors with high-quality neural ranking models and retrieve documents based on vector distances [12, 34]. This approach has demonstrated significant relevance gains in a wide range of IR applications, such as web search [22, 38], ad-hoc retrieval [13, 20, 36], question answering [37], mobile search [6], and product search [33].

The deployment of vector search to large scale is gaining more and more attention, due to the widespread commercial value and the exciting prospect. Fundamentally, vector search can be abstracted as the nearest neighbor search problem in high-dimensional space, and various approximate nearest neighbor (ANN) search algorithms have been proposed by trading the guarantee of exactness against high-efficiency improvement, such as tree structure–based [9, 10, 30], hashing–based [18], product quantization–based [17, 23, 25, 27, 31], and proximity graph–based [16, 29] approaches. However, existing approaches all suffer from one common limitation: they often assume that the entire database would fit in main memory (DRAM); thus they fail if the data exceed the limited memory capacity on a single machine. To overcome the DRAM scaling issue and to meet the high capacity needs of the large-scale vector search, one can either scale up the amount of DRAM in a single machine or scale out the ANN search by partitioning the dataset and utilize the collective DRAM in a distributed setting, both of which are not memory efficient and very costly.

Yet another possibility is to *leverage persistent storage to achieve more capacity*. While storage has always been much slower than DRAM, the arrival of fast storage medium, such as NVMe based SSDs, has delivered unprecedented performance on both latency and peak bandwidth, and their cost is fast approaching that of hard disk drives. Also, SSDs today can scale up to terabytes, whereas DRAM scales only to gigabytes. In this context, it is worth investigating the hybrid memory-storage hierarchy for scalable ANN due to the changes to the hardware landscape. To date, however, few studies have examined how to best leverage these new storage technologies in an ANN algorithm. There are several aspects of SSDs that make existing algorithms challenging for them. For example, the proximity graph based method is at the time being the fastest and most accurate ANN method [16, 29], but they are memory consuming and do not scale well to SSDs because the traversal of the graph generates many non-contiguous memory accesses, which would be much slower on SSDs.

In this paper, we exploit the memory-storage hierarchy simultaneously and rethink the design of ANN search algorithm to address the following question: Can a multi-store ANN algorithm achieve low search latency and high accuracy while being memory-efficient and scalable? Specifically, the algorithm should have a clear advantage over the state-of-the-art ANN implementations.

CIKM '19, November 3–7, 2019, Beijing, China

ACM ISBN 978-1-4503-6976-3/19/11…$15.00
https://doi.org/10.1145/3357384.3357938

To provide a definite answer to this question, we present GRIP tailored to jointly optimize search time, memory usage, and accuracy with both DRAM and SSDs. To avoid excessive memory usage of a dataset, GRIP first applies a product quantization–based technique to compress full-precision vectors into short codes. However, product quantization, as other compression–based techniques, often comes with non-negligible accuracy degradation on large datasets of dense vectors due to quantization errors. To improve the accuracy, instead of finding top-K in one-shot, GRIP identifies a small list of candidates that have a high probability of including the top-K, and then validates those candidates with their full-precision vectors stored on SSDs. However, this causes another two challenges to the search efficiency: 1) product quantization tends to check more vectors than the proximity graph–based approaches to reach the same accuracy; and 2) accessing the storage is still much slower than accessing DRAM. To improve search efficiency, GRIP systematically addresses three key technical challenges: 1) To reduce the number of vectors that need to be checked, GRIP introduces a new routing index, called graph-routing-index, which generalizes the proximity graph idea by replacing each node of the graph with a small group of short codes and navigating the graph to quick select these groups with reachability guaranteed; 2) To accelerate the speed of checking individual vectors, GRIP introduces a technique that reduces memory accesses by almost half with tiny memory cost added to each vector. 3) To support low validation latency, GRIP employs a lightweight mechanism that exploits the asynchrony and internal parallelism of SSDs.

We implement GRIP in C++ on Linux and perform a thorough evaluation to compare its effectiveness and efficiency with state-of-the-art ANN implementations. Our evaluation suggests that compared to compression–based technique such as IVFPQ, GRIP obtains significantly higher accuracy (close to 1), while reducing the latency by 2.7–19.4X. To meet similar accuracy target, GRIP improves the latency by 14.7–23.4X. Compared to the state-of-the-art proximity graph–based approach HNSW, GRIP achieves 12–14X memory cost reduction with similar latency and accuracy.

To evaluate the capacity improvement, we introduce a cost function that expresses the overall system efficiency — *VQ*, i.e., VQ = number of Vectors per machine × Queries processing rate, inspired by DQ metric of web search engine [19]. For a given database with a total number of vectors N and a query processing rate Q, an ANN solution requires $(N \times Q)/VQ$ number of nodes. The higher the VQ, the less the number of machines and cost! To meet the same accuracy target, GRIP improves *VQ* by up to 15.3X compared to the DRAM-only algorithms such as IVFPQ, L&C, and HNSW.

The key contributions of the work include: 1) Introducing GRIP, a capacity-optimized ANN algorithm for large-scale vector search, which is, to the best of our knowledge, the first that goes beyond memory and allow ANN to benefit from both DRAM and SSDs simultaneously. 2) Developing several novel techniques that collaboratively optimize latency, accuracy, and memory usage (§ 4, § 5). 3) A cost analysis of GRIP and identifying an important new metric (i.e., VQ) to evaluate the efficiency/cost of capacity optimization results for high-quality ANN search (§ 6). 4) Implementing and evaluating GRIP, and showing an order of magnitude improvements on VQ compared to the state-of-the-art (§ 7).

GRIP, with its efficiency and effectiveness, powers our production vector search services. It empowers significantly more vectors served by a single machine with low latency and high accuracy. As online services like search engine host billions of vectors and serves thousands of requests per second through vector search, a highly cost-efficient solution like GRIP could save thousands of machines and millions of infrastructure cost per year for the deployments of large-scale vector search.

2 Background and Related Work

2.1 Preliminaries

Vector search can be abstracted as the nearest neighbor search (NNS) problem, which aims at finding the K vectors from the dataset which minimize the distance to a given query, according to a pair-wise distance function (e.g., Euclidean).

In practice, the size of the database vectors is often large, so the computation cost of an exact solution is extremely high. To reduce the searching cost, approximate nearest neighbor (ANN) search relaxes the guarantee of exactness for efficiency, which returns the true nearest neighbors with high accuracy (high recall), where the recall measures the fraction of the top-K retrieved by the ANN search which are exact nearest neighbors. In particular, assume $TopK^{'}$ denotes the set of results returned to a given query q, the recall is defined as below:

$$Recall = \frac{|TopK \bigcap TopK^{'}|}{|TopK|} = \frac{|TopK \bigcap TopK^{'}|}{K} \quad (1)$$

High recall (i.e., close to 1) is clearly important for high-quality vector search because otherwise users will not be able to find what they are looking for. In this paper, we use recall as an evaluation metric. Next, we summarize previous work on ANN search.

2.2 Related Work

2.2.1 Tree structure–based algorithms. Many tree-based approaches have been proposed to partition the space and index the resulting sub-spaces for fast retrieval, such as KD-Tree [10], R*-Tree [9], and Randomized KD-Tree (FLANN) [30]. These approaches tend to work well in low dimensions. However, it is challenging to partition the subspaces, especially in high dimensional space, so that neighbor areas can be scanned efficiently to identify the nearest neighbors of a given query. The complexity of these approaches is $O(D \times N^{1-1/D})$, which gradually becomes not more efficient than a brute-force search as the dimension D becomes large (e.g., > 32) [26].

2.2.2 Proximity graph–based algorithms. The proximity graph based approaches have recently demonstrated outstanding accuracy and latency trade-offs. In particular, HNSW (Hierarchical Navigable Small World Graph) achieves logarithmic search complexity by navigating with small-world properties [29]. Several literatures have compared HNSW to a wide range of existing ANN algorithms, and the experimental results show that HNSW is by far both the fastest and most accurate, because it checks much fewer vectors to reach the same accuracy than other approaches [15, 29]. Fu. et al. later introduces NSG, which achieves similar performance as HNSW but with concrete theoretical proof of the logarithmic search complexity [16]. Both HNSW and NSG achieve high search efficiency with high accuracy, but they have a high cost of memory, as they still need to store the whole full-precision vectors and graph metadata in memory.

2.2.3 Compression–based algorithms. Another large body of existing ANN work relies on compression. One of the well-known representatives is Locality-Sensitive Hashing (LSH) [18], which approximates the similarity between two vectors with hashed codes. However, LSH and similar approaches have been designed for large bag-of-words sparse vectors with hundreds of thousands of dimensions, not dense continuous vectors with only a few hundreds of dimensions, like those learned by neural networks.

On a separate line of research involves compressing vectors to short codes that contain only tens of bits through *product quantization* (PQ) [23, 24]. Prior work such as OPQ, Cartesian KMeans, and LOPQ extend PQ by making the quantization better fit to the underlying distribution of database vectors [17, 25, 31]. To deal with large datasets, these approaches typically use a two-level approach in combination of the inverted file index (IVF) or inverted multi-index (IMI) [23, 27], which prior work has demonstrated to be more effective on large-scale datasets than hashing-based approaches [25]. Although the PQ–based methods can effectively reduce the database size, one common limitation is that they have a poor recall on large datasets because of quantization errors (discussed in §§ 3.1).

3 Challenges and Opportunities

In this section, we first introduce the challenges prior work face to offer desired accuracy, latency, and memory all together. We then identify opportunities for improvement.

3.1 Challenges

Challenge I. It is less clear how to best utilize storage for ANN design, given the very different performance characteristics between memory and storage. While it takes only around 60ns to read a few bytes of data with DRAM, it takes 10ms to read data from hard disk drives (HDDs) due to its slow mechanical nature. Because of this big latency gap, one of the key assumptions in existing ANN design has been that data fits in memory. More recent non-volatile memory (NVM) based SSDs offer 60 μs read latency, which has improved by more than 1,000X over HDDs. Furthermore, DRAM today scales only to gigabytes per DIMM slot, whereas SSDs can scale up to terabytes per PCIe slot [5]. In terms of cost, SSDs are up to 8X cheaper and consume less than ten times of power per bit than DRAM [28]. These changes could dramatically affect ANN design. However, it is still unclear how to best leverage these new storage technologies. For example, what should be stored on SSDs and what should be kept in memory, and for a response time limit how to deal with the trade-offs between accesses in DRAM vs. in SSDs, which previous design choices do not consider.

Challenge II. Graph-based approaches are efficient, but they are memory consuming and do not scale well to SSDs. While graph-based approaches attain good latency and accuracy, their indices are memory consuming in order to store both full-length vectors and the additional graph structure [29]. As the number of vectors grows, its scalability is limited by the physical memory of the machine, requiring machines with larger memory or more machines. However, it is difficult to make graph-based solutions work effectively on SSDs because the search of the graph structure generates many non-contiguous memory accesses, which are much slower on SSDs than memory and detrimental to the query latency.

Challenge III. Compression provides memory compactness but results in poor recall on large datasets. Compression based approaches, such as PQ, suffer from low recall. As a point of reference, the best results achieved by both PQ and its most advanced variant LOPQ on Deep10M and Deep1B datasets do not exceed 0.76 recall at rank 1 [23, 25]. Douze et al. recently propose *L&C*, which builds an HNSW graph with each node quantized and exploits neighbor nodes to refine the estimation of distance. This approach does not eliminate quantization errors and still creates non-negligible recall loss for large datasets, and the best recall L&C achieves on 1-million vectors (Deep1M) is < 0.65 (Fig.5) when $K = 1$ [15], which is too low. We will provide a comparison with this approach in our experiments.

3.2 Preliminary Analysis and Opportunities

Although product quantization–based approaches have poor recall, they are still promising techniques for capacity optimization because they provide a strong reduction in memory. This section presents several studies that have guided the design of the algorithms and optimizations in § 5. All these evaluations are performed on $X = SIFT1M \subset \mathbb{R}^{128}$, a classical dataset in BIGANN to evaluate nearest neighbor search [23]. We experiment with IVFPQ (16K clusters) by varying the number of clusters being selected and checked in IVF. This lets us identify opportunities for our design.

First, Fig. 1a shows that by checking 512 clusters (3%), the probability of an exact match is only 0.718, but by returning more candidates (e.g., 10), the likelihood that the true top-1 is included in those candidates is 0.997, close to unity (Fig. 1a). Similarly, although the recall is only 0.789 when searching for top-10, the probability of the true top-10 included in 50 candidates is 0.997, again close to unity. We find this issue generally exist towards various datasets (more results in Section 7.3). This observation indicates that — ***O1:*** *although the approximated top-K might not always match the exact top-K for PQ on large datasets, the* ***top-K are more likely to fall within a list of*** *R* ***candidates, where*** *R* ***is larger but not too much larger than*** *K*. Prior work made similar observation when applying product quantization to signal processing [24], but they did not evaluate the efficiency impact of this observation.

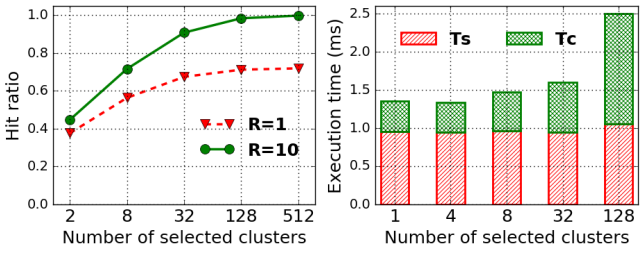


Figure 1: (a) Hit ratio of top-K in the candidate sets. (b) Cluster selection time T_S and vector checking time T_C with two-level quantization-based approach.

Second, we study the search latency of two-level quantization, which is decomposed into two parts: the *cluster selection time* (T_S) and the *vector checking time* (T_C) in selected clusters. We observe that ***O2: quantization-based approaches face a dilemma on optimizing latency.*** Both latency components depend on the number of the first-level clusters NC. During cluster selection, a query computes its distances to all cluster centroids and choose a few clusters whose centroids are closest to the query to scan, and T_S in this case

is linear in NC. When NC is not too large, finding which clusters to scan is computationally inexpensive. Existing approaches rarely choose a large NC, because as NC increases, T_S itself would become too long, prolonging query latency. Fig. 1b shows that sometimes T_S can dominate the computation cost.

On the other end, larger NC leads to a smaller percentage of vectors to be scanned at the second level to reach the same recall. Fig. 2a shows that with NC = 1K, 4K, and 16K, checking the same number of clusters (e.g., 64) all lead to a very similar recall. This observation is kind of intuitive as top-K would belong to at most K clusters regardless of NC value. As larger NC has a less (expected) number of vectors per cluster, this observation indicates that, ***O3: larger*** NC ***reduces the checks of vectors at the second level to reach the same recall***. Fig. 2b shows that by checking 64 clusters, only 0.4% vectors need to be checked when NC is $16K$, whereas it is 6.4% when NC= $1K$, which means the search at the second level takes shorter time with larger NC. The NC dilemma makes it challenging to optimize both T_S and T_C. The problem is beyond selecting a good value for NC and seems to require more fundamental changes in ANN design for latency optimization.

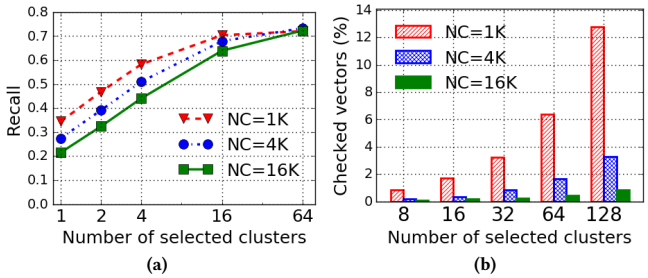


Figure 2: Recall and the percentage of vectors being checked in two-level quantization varying the number of clusters.

4 Design Overview

We introduce GRIP, a capacity-optimized multi-store ANN algorithm to scale up high-quality vector search. GRIP is composed of both DRAM and SSDs, where each vector has a quantized store in DRAM and a full-precision store on SSDs. At a very high level, GRIP indexes and searches vectors in two stages:

- A ***preview*** **stage**, comprising novel in-memory indexing that enables memory compactness and allows a query to quickly identify a short list of candidate vectors;
- A ***validation*** **stage**, employing a lightweight SSD access mechanism to validate every selected candidate with their full-precision representation to select top-K.

Fig. 3 illustrates the overview of a GRIP index, which comprises of several layers. At the very bottom layer is the *validation layer* that stores the entire full-precision dataset on SSDs. Since the capacity of SSDs is much larger than that of DRAM, GRIP can host significantly more vectors, even in full-precision, given enough SSD capacity. Above that is the *preview layer*, which contains short codes encoded by product quantization codebooks. The quantization significantly reduces memory usage. The short codes are clustered into groups, which allow a group of vectors to be selected altogether if its centroid is close to the query. The highest layer uses the graph-routing-index to quickly and accurately dispatch a query to a few closest clusters.

Intuitively, the multi-store design achieves memory savings, through storing quantized data in memory, and high recall through SSD-based validation to enhance accuracy given the high probabilities of true top-K included in the candidate list, as identified in **O1**, but what about latency? Accessing SSDs is still much slower than accessing DRAM. We introduce and develop three techniques to efficiently reduce the search time:

- **Graph-routing-index (GRI)**: The graph-routing-index returns a small number of clusters that are better localized around the query points with low latency, high accuracy, and reachability guarantee. It makes cluster selection more scalable compared to IVF and IMI, handling **O2**, and avoids checking an excessive amount of short codes, as identified in **O3**. (§§ 5.1);
- **PQ***: We introduce PQ^*, a variant of PQ that further cuts the cost of checking vectors by half by storing a small partial-distance-value (PDV) value for each vector (§§ 5.2).
- **Lab-SSD-Val**: It performs lightweight validation on SSDs through a combination of asynchrony and multi-candidate batching techniques(§§ 5.3).

Next, we describe how to construct GRIP index in details and how to search for top-K (§§ 5.4).

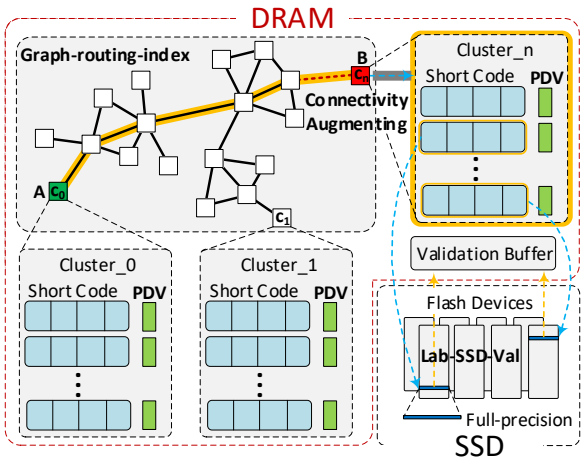


Figure 3: Overview of GRIP.

5 Algorithms and Optimizations

Given a set of vectors, GRIP builds the in-memory index using Algorithm 1, with three major steps:

5.1 GRI Construction

Cluster selection based on exact search incurs linear complexity of $O(NC \cdot D)$. Although prior ANN work such as KD-Tree and PQ (e.g., IMI) may be invoked to speed up this process, they often introduce significantly accuracy loss. Instead, we introduce a coarse-grained routing index based on the state-of-the-art proximity graph–based approaches. In particular, we build GRI based on both HNSW and Kosaraju's algorithm [21]. The goal of GRI is to quickly and accurately select a few closest clusters to check at the next step and to avoid creating unreachable clusters.

We build GRI upon HNSW because it is both the fastest and most accurate ANN index. Furthermore, since we build the graph with only cluster centroids, it does not cause as much memory overhead as indexing the entire database vectors. However, one source of inaccuracy of HNSW is that it has no guarantee of reachability. Analysis shows that HNSW graph is weakly connected

Algorithm 1 **GRIP index construction algorithm**

1: **Input:** Vector set X, vector dimension D.
2: **Output:** GRIP *index*.
3: **Parameter:** Number of database vectors N, number of sub-dimensional spaces M, size of each sub-codebook L, number of clusters NC
4: $centroids \leftarrow clustering(X, D, NC)$ ▷ Partition the vector space with Lloyd's algorithm.
5: $gri\_index \leftarrow CreateGri(centroids)$
6: **for** i **in** $0..(N-1)$ **do**
7: $cluster\_id, centroid \leftarrow assign(X[i], centroids)$
8: $residuals[i] \leftarrow compute\_residual(X[i], centroid)$
9: $cluster\_ids[i] \leftarrow cluster\_id$
10: **for** i **in** $0..(M-1)$ **do**
11: $sub\_codebook \leftarrow train(residuals, D, M, L)$
12: $pq\_codebook.add(sub\_codebook)$
13: $preview\_layer.add(pq\_codebook)$
14: **for** i **in** $0..(N-1)$ **do**
15: $cluster\_id \leftarrow cluster\_ids[i]$
16: $short\_codes[i] \leftarrow product\_quantizer(residuals[i])$
17: $preview\_layer.clusters[cluster\_id].add(short\_codes[i])$
18: **for** i **in** $0..(N-1)$ **do**
19: $pdc\_values[i] \leftarrow comp\_pdc(residuals[i], cluster\_ids[i])$

but not necessarily strongly connected, which causes the issue of having "isolated nodes" in the graph — nodes with zero in-degree that are unreachable. For instance, our experiments show that an HNSW graph with an out-degree of 20 over 200K centroids using the Deep10M dataset has around 900 nodes whose in-degree are zero. These nodes and their corresponding clusters, which are around 45,000 vectors, are unreachable. If these vectors are unreachable, they cannot be retrieved. We do not want to miss results simply because the routing index cannot reach their cluster centroids. To resolve the reachability issue, the *CreatGri* method at line 5 builds an HNSW graph with additional two complete traversals of the HNSW graph to adjust edges to make the graph strongly connected. The benefits of this design is that the number of added edges is theoretically guaranteed to be minimal, and since only minimal edges are added, it preserves the small-world properties, which HNSW relies on for efficiency.

5.2 Enabling High-Efficiency Compression with PQ*

To better suit the design of memory-storage based architecture, GRIP introduces a variant of PQ, called PQ^*, to provide both memory compactness and high-performance. PQ^* reduces the number of bits to store each vector by generating codebooks and encoding short codes the same way as PQ. Different from PQ, PQ^* adds a *partial-distance-value (PDV)* to each vector, which takes a tiny piece of memory but results in significantly search time reduction (line 18–line 19). We first briefly describe the main idea of using PQ to quantize vectors below and please find more details in Jégou et al. [23]. We then talk about how PQ^* helps improve performance.

GRIP first calculates the residual distance of each vector to the cluster it belongs to (line 6–line 9). It then splits the vector space into M sub-dimensional spaces and constructs a separate codebook for each sub-dimension (line 10–line 12). GRIP then generates quantized short codes by assigning the PQ code to each vector (i.e., a concatenation of M indices) and adds each short code to its closest cluster (line 14–line 17).

5.2.1 Partial-Distance-Value (PDV). For each query, the original distance computation of PQ requires $2 \times M$ memory accesses to look-up-tables (LUTs) that store pre-computed distances between codewords. PQ^* reduces the number of memory accesses to $M + 1$, which significantly reduces the data movement cost. Next, we provide the detail of how PDV works.

For each vector x in cluster c_i, PQ^* calculates the distance from query q to x with asymmetric distance computation (ADC) [23]:

$$d(q,x) = d(q-c,r) \sim d_{ADC}(q-c, pq(r)) \quad (2)$$

$$= \sum_{m=1}^{M} d((q-c)^m, vq^m(r^m)) \quad (3)$$

where c is the cluster centroid and r is the residual between x and c, denoted as $[r^1 : \ldots : r^M]$. Eqn.(3) gets expanded into Eqn. (4):

$$\underbrace{\|q-c\|^2}_{Term-A} + \underbrace{\sum_{m=1}^{M} \|c_{x^m}^m\|^2}_{Term-B} + \underbrace{2\sum_{m=1}^{M} \langle c^m, c_{x^m}^m \rangle}_{Term-C} - \underbrace{2\sum_{m=1}^{M} \langle q^m, c_{x^m}^m \rangle}_{Term-D} \quad (4)$$

where $c_{x^m}^m = vq^m(r^m)$, denoting the closest sub-codeword assigned to sub-vector x^m in the m-th sub-dimension.

By looking close into these terms, we notice that each term can be query–dependent (query–dep.), mapped-centroid–dependent (centroid–dep.), and/or PQ-code–dependent (PQ-code–dep.). Table 1 summaries these dependency relationships.

	Query–dep.	Centroid–dep.	PQ-code–dep.
Term-A	✓	✓	✗
Term-B	✗	✗	✓
Term-C	✗	✓	✓
Term-D	✓	✗	✓

Table 1: Dependencies of PQ distance computation.

Since both Term-B and Term-C are query independent, existing algorithms such as PQ compute these two terms and store them in LUTs. At search time, it requires $2 \times M$ memory lookups to retrieve these two terms. This approach does not require to store any additional data for an individual vector and favors memory savings more than latency gains. In contrast, PQ^* computes and stores the total partial sum of Term-B and Term-C (PDV) for every vector at the index construction phase, which adds f-byte memory (e.g., 4-byte if vector type is float) per data point. During query processing, it takes a single lookup to get the PDV. In total, it takes $M + 1$ lookup-add operations to calculate the distance. We show in the evaluation that PQ^* significantly reduces the search time while providing considerable memory overhead reduction.

5.3 Validation Layer Construction

Apart from the in-memory part, GRIP keeps all the full-precision database vectors on SSDs and validate R selected candidates from in-memory search with their full-precision vectors. Loading full-precision vectors from storage is slow because even SSDs typically have 100-1000 times higher round trip latency than DRAM. To make it even worse, the latency of validating an entire set of candidates from SSDs can be detrimental to the response time because there could be as many as 100 candidates that need to be validated.

5.3.1 Lab-SSD-Val. Lab-SSD-Val is a lightweight mechanism for quickly validating a list of candidates with their full-precision vectors on SSDs. Modern SSDs are built on an array of flash memory

packages, which are connected through multiple (e.g., 8) channels to flash memory controllers, and data accesses can be conducted independently in parallel [11]. This hardware architecture has two benefits: (1) Accessing data from multiple flash memory packages in parallel can provide high-aggregate bandwidth; (2) High latency operations can be effectively hidden by other concurrent operations. Fortunately, given that the validation process does not have to be sequential, the *asynchronous batching* query submission based technique [32] well addresses the problem by dividing the queries into multiple batches and loading each batch asynchronously. Specifically, we implement Lab-SSD-Val using the Linux NVMe Driver. The implementation uses the Linux kernel asynchronous IO syscalls. We combine B candidate vectors in the candidate list R into one big batch and submits $S = \lceil R/B \rceil$ batched requests to SSDs to cover the entire list. We then validate a candidate as soon as its full-precision vector has been loaded from SSDs into DRAM. Batched requests allow GRIP to exploit the internal parallelism of SSDs and make better performance I/O request scheduling decisions. Asynchronous IO unblocks GRIP search process and allows the distance computation to overlap with I/O and hide the SSD access latency.

5.4 Query Processing

In this section, we describe how GRIP searches top-K. Algorithm 2 shows the online search process. First, the search starts from GRI in memory, which efficiently selects a few closest clusters with graph navigation (line 5). The system parameter that controls the trade-off between search time and accuracy is $efSearch$, which bounds the length of the search candidate queue. This step is very fast because of the limited number of nodes and the logarithmic search complexity. Second, for those selected clusters, GRIP checks quantized vectors in them with PQ^* [1] (6–17) and keeps track of R candidates ($R >^{\epsilon} K$, where R is larger but not significantly larger than K) with a priority queue (line 4). Third, eventually R candidates are validated on SSDs to select the final top-K (line 18).

6 Cost Analysis

Compared to in-memory only ANN approaches, the GRIP multi-store design has more parameters. Having a much larger design space, it is preferable to have a systematic way for parameter selection. In this section, we build a performance model, with an eye towards being able to set the parameters properly to achieve the design targets. Table 2 presents the notation used in the analysis.

6.1 Search Cost

The search cost consists of three parts: the cost of cluster routing (CR), the cost for checking vectors (CV), which includes sub-distance calculation and actual distance calculation using sub-distance results (codebook look-ups plus summing up sub-distances), and the cost of validation (VA). Assuming a complete overlap between computation and data movement (best case scenario), the execution time can be estimated using the roofline model [35].

$$\begin{aligned} Time &\geqslant Max(CompTime, DataMoveTime) \\ &= Max(\frac{TotalComp}{ComputePeak}, \frac{DataMoved}{DataBandwidth}) \end{aligned} \tag{5}$$

[1] Similar as IVFPQ, GRIP skips (line 7–8) and directly calculates term-D on-the-fly at line 15 when it is cheaper to compute term-D directly.

Algorithm 2 **GRIP online search algorithm**

1: **Input:** Query vector q, K
2: **Output:** Top-K
3: **Parameter:** number of selected clusters to scan NS, size of search queue $efSearch$, size of candidate list R.
4: $R \leftarrow min\_priority\_queue()$
5: $I^{NS}, D^{NS} \leftarrow gri\_index.search(q, NS)$
6: **for** m **in** $0..(M-1)$ **do**
7: **for** l **in** $0..(L-1)$ **do**
8: $termD\_LUT[m][l] \leftarrow -2 \times \langle q^m, c_l^m \rangle$ ▷ Init LUTs for computing all possible Term-D
9: **for** n **in** $0..(NS-1)$ **do**
10: $cluster \leftarrow preview\_layer.clusters[I^{NS}[n]]$
11: $t1 \leftarrow \|q - D^{NS}[n]\|^2$ ▷ Compute Term-A
12: **for all** v **in** cluster **do**
13: $dist \leftarrow 0$
14: **for** m **in** 0..(M-1) **do**
15: $dist \leftarrow termD\_LUT[m][v_m] + dist$
16: $dist \leftarrow pdc\_values[v_{id}] + dist$ ▷ Lookup-add the PDV value
17: $R.push(v, dist)$
18: $TopK \leftarrow lab\_ssd\_val(q, R)$ ▷ Validation on SSDs.

Based on that, the search cost (SO) is:

$$SO = CR + CV + VA \tag{6}$$

$$CR \geqslant Max(\frac{2 \cdot logNC \cdot D}{ComputePeak}, \frac{logNC \cdot D \cdot f}{DRAMBandwidth(logNC \cdot D \cdot f)}) \tag{7}$$

$$\begin{aligned} CV \geqslant Max(&\frac{N \cdot NS/NC \cdot (M+1)}{ComputePeak}, \\ &\frac{N \cdot NS/NC \cdot (M+1) \cdot logL/8}{DRAMBandwidth(N \cdot NS/NC \cdot (M+1) \cdot logL/8)}) \\ &+ N \cdot NS/NC \cdot logR \end{aligned} \tag{8}$$

$$VA \geqslant Max(\frac{2 \cdot R \cdot D}{ComputePeak}, \frac{R \cdot D \cdot f}{SSDBandwidth(R \cdot D \cdot f)}) + R \cdot logK \tag{9}$$

On modern architectures, the computational throughput is significantly higher than the DRAM data movement throughput, which is significantly higher than the SSD data movement throughput. Let us look at a machine with Xeon Gold 6152 2.10GH, DDR4 (2666 MHz) DRAM, and Samsung 960 Pro SSD. It has peak computational performance of 1.48TFlops while the DRAM peak bandwidth is 10.75 GigaFloats/s (43GB/s) and the SSD peak bandwidth is 0.525 GigaFloats/s (2.1GB/s). Because of this big difference of hardware throughput, the total execution time can easily be bottlenecked by the data movement. Our GRI reduces both the computation and data movement overhead of CR from $O(NC)$ to $O(logNC)$. Our PQ^* technique reduces the DRAM accesses of each vector from $2 \cdot M$ to $M + 1$, effectively reducing CV. While the SSD accessing cost is high from an absolute standpoint, our in-memory index is designed with the goal of reducing excessive accesses to SSDs, which results in only a small set of candidates (e.g., 10–100) , which together with Lab-SSD-Val only cause marginal cost associated with SSDs (VA).

Symbol	Meaning	Example
	GRIP design parameters	
NC	number of clusters	$200K$
L	codebook length per sub-dimension	256
M	number of sub-dimension	24
OD	out-degree of proximity graph	20
NS	number of selected clusters	256
R	number of candidates	50
	Workload characteristics	
N	number of database vectors	80M
D	number of dimensions	128
f	number of bytes of data type	4
K	number of nearest neighbors to retrieve	5
	Constraints	
ξ	lower limit of allowable accuracy	0.97
ϕ	response time limit	10ms
	Hardware parameters	
MS	memory capacity size	$32GB$
SS	SSD storage size	$1TB$

Table 2: Notation.

6.2 Construction Cost

The complexity of the GRIP index construction consists of three parts: (1) the clustering cost, which takes $O(NC \cdot |X|^2 \cdot D)$ to run each iteration; (2) the GRI construction cost, which has $O(NC \cdot logNC + 2 \cdot (NC + NC \cdot OD))$ complexity; (3) the PQ^* cost, which takes M times the complexity of performing K-Means clustering with L centroids of dimension $\frac{D}{M}$ to learn the codebooks and then takes $O(N \cdot (D \cdot L + 2M))$ to encode and compute the PDV values for all vectors. Among the three parts, clustering usually dominates the construction time because it has a quadratic complexity to the number of vectors due to the pairwise comparisons. When X is really large, it is possible to reduce clustering cost by applying the algorithm to a smaller subset $X^* \subset X$.

6.3 Space Consumption

We express memory consumption as the sum of bytes required to store the GRIP index, which includes the GRI, the codebooks, the quantized short codes, and the PDV values. The GRI uses $NC \cdot (D + OD) \cdot f$ bytes. The codebooks take $L \cdot D \cdot f$ bytes. Each vector consumes $M \cdot \frac{log(L)}{8(bit)}$ plus f bytes, which includes both codebook indices and a PDV value. The memory overhead is given by Eqa. 10:

$$MO = NC \cdot (D + OD) \cdot f + L \cdot D \cdot f + N \cdot (M \cdot \frac{log(L)}{8\text{-}bit} + f) \quad (10)$$

First, this equation indicates that even a large NC (e.g., two orders of magnitude smaller than N) does not significantly increase the memory consumption because it is still relatively small compared to the quantized representation of the entire set of vectors. However, when NC gets closer to N and M gets closer to D, GRIP will consume increasingly more memory. In the extreme case when $NC = N$ (no clustering) and $M = D$, then GRIP turns out to be more similar to an HNSW graph with scalar quantizer applied to every node. Second, the selection of parameters needs to satisfy the following relationship between the memory capacity, SSD size, and memory overhead:

$$MO \leqslant MS \ \ \& \ \ N \times D \times f \leqslant SS \quad (11)$$

Under this condition, GRIP can fit more vectors on a single machine even if the dataset size exceeds the memory capacity.

6.4 VQ Efficiency

For capacity optimization, we are facing a threefold trade-off between latency, memory efficiency, and accuracy. Among them, accuracy is an effectiveness metric, and both latency and memory consumption are important system efficiency metrics which have a crucial impact to the scalability and capacity. In this section, we introduce a cost function that expresses the system efficiency as ***VQ* (Vector–Query)**, the product of the number of vectors V per node and the query processing rate Q, inspired by the DQ (Document–Query) metric from a web search engine [19]. Our constraints are that the accuracy must exceed the fixed threshold ξ and the latency must be lower than a response time limit ϕ . V is inversely proportional to the amount of memory consumption per vector. Q is inversely proportional to the query latency. Therefore

$$VQ \propto \frac{1}{SO \cdot MO} \quad (12)$$

For a dataset with a total number of vectors N, an ANN solution requires $(N \times Q)/VQ$ number of machines, given constraints are met. The higher the VQ, the less number of machines needed!

7 Evaluation

We evaluate GRIP and show how its design and algorithms contribute to its goals.

7.1 Methodology

7.1.1 Datasets. We conduct experiments on three datasets.

- *SIFT1M* is a classical dataset in BIGANN to evaluate nearest neighbor search [23].
- *Deep10M* is a more recent dataset that consists of dense continuous vectors generated by a deep neural network [7].
- *SpaceV80M* is a dataset that consists of 80 millions of 128-dimensional feature vectors extracted by a neural ranking model.

We choose these three million-scale datasets because not all the state-of-the-art approaches can scale to billion-scale vectors. We choose SIFT1M and Deep10M because they are widely used in related literature [8, 14, 15, 23, 29]. We choose SpaceV80M, which is one partition of a multi-billion scale dataset used for semantic understanding of queries and web data for better search quality. There is an attempt to build a benchmark suite for ANN [1], but no dataset contains more than 1.2M vectors, which is insufficient.

7.1.2 Comparison Algorithms. We compare with three approaches.

- *IVFPQ:* **Faiss** [2] is a recently released library from Facebook and has the most state-of-the-art implementation of product quantization–based methods.
- *HNSW:* **NMSLib** [4] is a well known ANN search library, which contains well-implemented codes for HNSW.
- *Link-and-code (L&C):* **L&C** [3] is an ANN algorithm that combines quantization with graph–based retrieval, incurring low latency and low memory usage.

All three of them are memory-only ANN solutions. We choose IVFPQ as a baseline because we leverage product quantization for in-memory vector compression. We choose HNSW because it is both the fastest and most accurate method at the time being, as discussed in § 2. We choose L&C because it is a state-of-the-art attempt to optimize latency, accuracy, and memory usage altogether.

7.1.3 Evaluation metrics. Latency, memory, and recall are important metrics for ANNs. We measure query latency as the average time of per-query execution (one query at a time) time in millisecond. The memory cost is calculated as the total allocated DRAM for

the ANN index. The recall is calculated as Eqa. 1. Since it is essential to be both fast and with high accuracy in real scenarios, we focus on the performance of all algorithms in the high recall range. We report VQ improvement over baseline ANNs as a product of the latency speedup and memory cost reduction rate.

7.1.4 Testbed. We conduct the experiments on Intel Xeon 6152 CPU (2.10GHz) with 64GB DRAM and 1TB Samsung 960 Pro SSD.

7.2 ANN Search Performance Comparison

7.2.1 Comparison to IVFPQ. Table 3 reports the *recall* (for $K = 1$), latency, memory cost, and VQ improvement of GRIP, in comparison to IVFPQ. Both IVFPQ and GRIP partition the input vectors into NC clusters (20K for SIFT1M, 200K for Deep10M, and 400K for SpaceV80M) and generate the same codebooks to encode all vectors with a compression ratio of 16X. We vary the selected clusters NS (N_{scan}) from 64 to 1024. This is the range we start to see that further increasing NS leads to a marginal return of recall from IVFPQ. Two main observations are in order.

	NS	IVFPQ			GRIP			VQ
		Recall	Lat.	Mem.	Recall	Lat.	Mem.	Impr.
SIFT1M	1024	0.679	8.8	40	**0.904** ($NS:32$)	**0.6**	49	**12.0X**
	512	0.678	5.4	40	**0.989**	**1.8**	49	**2.4X**
	256	0.676	3.2	40	0.986	1.2	49	2.2X
	64	0.662	2.0	40	0.948	0.7	49	2.3X
Deep10M	1024	0.602	18.5	302	**0.906** ($NS:64$)	**1.2**	377	**12.3X**
	512	0.601	15.6	302	**0.975**	**5.4**	377	**2.3X**
	256	0.599	14.1	302	0.965	3.4	377	3.3X
	64	0.580	12.8	302	**0.906**	**2.0**	377	**5.1X**
SpaceV80M	1024	0.767	28.1	2552	**0.925** ($NS:32$)	**1.2**	4036	**14.8X**
	512	0.765	27.8	2552	**0.977**	**4.7**	4036	**3.7X**
	256	0.763	27.5	2552	0.971	2.7	4036	6.4X
	64	0.752	27.2	2552	**0.946**	**1.4**	4036	**12.3X**

Table 3: Recall, latency (ms), memory (MB), VQ improvement of GRIP in comparison with IVFPQ.

First, the results show that *GRIP provides significant improvement to the recall from about 0.58–0.77 to 0.90–0.99, while at the same time improving VQ consistently by 2.3–12.3 times among tested configurations.* As expected, by varying NS from 64 to 512, the recall and latency increase for both implementations as scanning more clusters increases both the likelihood and time of finding top-K. The recall of IVFPQ is constantly lower than GRIP, and it starts to saturate at $NS = 128$. In contrast, GRIP significantly improves the system's effectiveness by bringing the recall to the 0.97+ range because of GRIP's multi-store design. In terms of efficiency, GRIP overall speedups query latency by 2.7–19.4X even with accessing SSDs, because GRI, PQ^*, Lab-SSD-Val all reduce search cost considerably. Compared to IVFPQ, GRIP takes 1.2–1.6X more memory due to storing the GRI index metadata and PDV values. But the absolute memory savings (10–13.3X) are still quite significant compared to the original dataset.

Second, *GRIP improves VQ by 12–14.8 times to meet similar or higher recall target.* The highest recall IVFPQ gets is still far below 1. As discussed earlier, this is because product quantization has difficulties in distinguishing top-K with only short codes. In contrast, GRIP achieves similar or much higher recall (0.90+) by checking a much smaller number of clusters (e.g., 32 and 64 clusters), improving the latency by 14.7–23.4X. Compared to IVFPQ, GRIP is more suitable for high recall and fast response scenarios.

7.2.2 Comparison to HNSW. HNSW is one of the state-of-the-art proximity graph–based approaches. We compare them by choosing configurations from both that achieve the same recall targets [1].

As reported in Table 4, *GRIP significantly and consistently outperforms HNSW, with an average VQ improvement of 2.5–15.3X among tested configurations.* GRIP reduces the memory cost by 12–14X compared to HNSW because it uses PQ^* to compress vectors into short codes. HNSW runs faster than GRIP in a few cases. However, the latency gap between GRIP and HNSW decreases as we increase the recall target. This is presumably because for isolated nodes in HNSW, further increasing $efSearch$ does not help discover them but leads to significantly more node exploration. We will discuss the benefits of HNSW and GRIP and how to determine which one to use at the end of this section.

7.2.3 Comparison to L&C. The comparison to L&C first focus on the trade-off between latency and accuracy. Fig. 4 presents a comparison of GRIP to L&C in terms of recall vs query time for different parameters of $efSearch$ [2]. The results show that L&C can offer much lower latency at the low recall range (i.e., 0.4–0.7). However, at the high recall area (i.e., 0.90–1), L&C is significantly less accurate and has difficulties in reaching recall 0.8 with 250ms. In contrast, GRIP reaches 0.977 recall in less than 5ms. This is because fundamentally L&C still uses quantized short codes for distance computation. In contrast, GRIP's multi-store design allows it to leverage SSDs to boost accuracy with full-precision vectors. Furthermore, the memory usage of L&C is 8.96GB, whereas GRIP takes only 4.04GB of memory, which is 2.2X lower. This is because GRIP can afford a larger compression ratio from quantization by recovering the precision loss from full-precision vectors on SSDs. Because of this large gap of latency and memory at the high accuracy range, GRIP is more competitive when high precision, low memory usage, and fast response are all required.

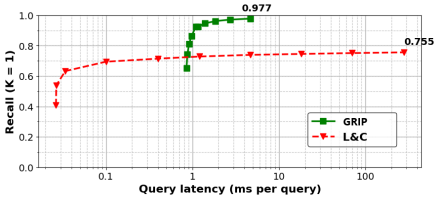


Figure 4: Comparison of GRIP with L&C on SpaceV80M.

7.3 Effect of Different Components

We also conduct an in-depth evaluation across different design points of GRIP.

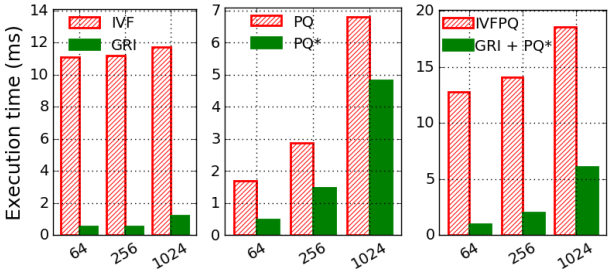
7.3.1 Effect of in-memory search latency. Fig. 5 shows the breakdown of query latency on searching the in-memory index.

Latency of cluster selection. Fig. 5a shows that GRI yields significant improvements on cluster selection time compared to the exact search in IVF. Overall, GRI boosts the cluster selection time by 10–22 times for Deep10M ($NC = 200K$). GRI scales better as

[1] We build HNSW graph with standard settings: $efConstruction$=200 and OD=10. We trade-off accuracy and latency by varying $efSearch$ from 160 to 1280.
[2] We build L&C with 6 links per vector and 32 bytes per vector as suggested by [3].

	HNSW				GRIP				VQ
	Recall	efSearch	Latency	Memory	NS	Recall	Latency	Memory	Improvement
SIFT1M	0.993	1280	2.3	588	512	**0.989**	1.8	**49**	**15.3X**
	0.973	320	0.6	588	128	0.976	0.8	49	9.0X
	0.947	160	0.3	588	64	0.948	0.7	49	5.1X
Deep10M	0.998	1280	3.1	4662	512	**0.994**	3.9	**377**	**9.8X**
	0.985	320	0.9	4662	256	0.983	2.6	377	4.3X
	0.969	160	0.4	4662	128	0.961	2.0	377	2.5X
SpaceV80M	0.972	2560	4.1	57554	512	**0.977**	4.7	**4036**	**12.4X**
	0.943	640	1.4	57554	128	0.961	1.8	4036	11.1X
	0.918	320	0.9	57554	64	0.946	1.4	4036	9.2X

Table 4: Latency (ms), memory (MB), and VQ improvement of GRIP in comparison with HNSW.



(a) Cluster selection time (b) Vector checking time (c) In-memory time

Figure 5: Breakdown of different components on the in-memory search query latency. The x-axis represents the number of selected and scanned clusters *NS*.

NC increases and the improvement becomes more significant with larger *NC*, because of its logarithmic search complexity.

Latency of checking vectors. Fig. 5c illustrates the improvements of query latency of the preview layer using PQ^* compared to PQ. As *NS* increases, the execution time increases almost linearly for both PQ and PQ^*. However, PQ^* consistently outperforms PQ by 1.4–3.5X. This is because PQ^* reduces the distance computation per vector from $2 \times M$ to $M + 1$, cutting both the number of adds and memory bandwidth consumption by almost half.

Latency of the in-memory search. Fig. 5c reports the total in-memory search time. GRI and PQ^* together offer 3–12.7X latency reduction over IVFPQ.

7.3.2 GRI Accuracy. We also evaluate how accurately GRI identifies *NS* clusters compared to doing an exact search. Fig. 6 reports the accuracy and latency of routing $200K$ clusters of Deep10M when *NS* varies from 1 to 256. Overall, GRI can achieve fairly high accuracy (e.g.. when $efSearch$ is 320) for various *NS*, thanks to the outstanding performance of recent proximity graph–based ANN search (e.g., HNSW). Gradually increasing $efSearch$ (efS) leads to higher accuracy at the expense of increased routing latency. We also observe that under the same $efSearch$, larger *NS* sometimes leads to slightly worse accuracy if $efSearch$ is not big enough (e.g., $efSearch$ is less than 160), as the closest centroids not visited during the routing phase are definitely lost. In practice, we find [320, 640] as a good range that provide close to unity accuracy for GRI.

7.3.3 Sensitivity of *K*. Different scenarios might have different *K*. Table 5 shows the recall of GRIP at different *K* compared to IVFPQ varying *NS* from 1 to 1024 on Deep10M. We make two observations. First, GRIP offers significant recall improvement for K=1 as well as $K > 1$. In both cases, the recall of IVFPQ has reached

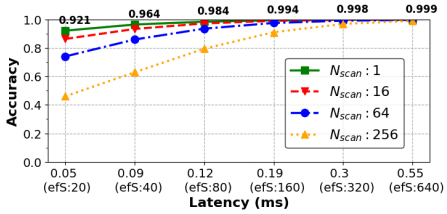


Figure 6: Trade-offs between GRI accuracy and latency on 200K centroids of Deep10M. Higher accuracy is better.

a plateau around 0.60–0.70, whereas GRIP consistently brings the recall to 0.98+. Second, increasing *K* does not significantly increase *R* and a small *R* sharply improves the recall. Although larger *R* is better for getting a higher recall, further increasing *R* from 10 to 100 when $K = 1$ or from 50 to 100 when $K = 10$ does not bring much improvement on recall, which indicates that a small *R* (10–100) is often sufficient to get a high recall.

NS	K=1			K=10		
	IVFPQ	GRIP		IVFPQ	GRIP	
		R=10	R=100		R=50	R=100
64	0.580	0.906	0.920	0.657	0.864	0.868
256	0.599	0.965	0.983	0.691	0.958	0.966
1024	0.602	**0.978**	**0.998**	0.697	**0.983**	**0.994**

Table 5: Effect of *K* on recall for IVFPQ and GRIP.

7.3.4 Performance of validation on SSDs. Fig. 7 shows the validation latency on SSDs without Lab-SSD-Val (Seq.) and with Lab-SSD-Val on SpaceV80M, varying the size of candidate list *R* from 10 to 100. As expected, the execution time all increases almost linearly with the increase of the number of candidates. However, compared to the baseline, Lab-SSD-Val provides a much shorter execution time consistently. Without Lab-SSD-Val, it takes 6.8ms to validate 100 candidates, which already takes more time than in-memory search latency. With Lab-SSD-Val, GRIP validates 100 candidates in less than 0.6ms. Overall, Lab-SSD-Val achieves 4.3–11.5X speedup over the baseline. This is because Lab-SSD-Val exploits the internal parallelism of SSDs with asynchronous batching. As GRIP requires only a small set of candidates to be validated, Lab-SSD-Val allows GRIP to scale much better on SSDs.

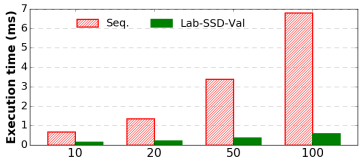


Figure 7: Validation latency w/o and with Lab-SSD-Val.

7.4 Why GRIP? A Combined View

GRIP provides a memory–efficient, high–performance ANN algorithm for building large-scale vector search engines. To make vector search more scalable, GRIP adopts a multi-store design with the goal of achieving high-quality capacity-optimized ANN search on a single node. Take the SpaceV workload as an example, which is a hundred-billion scale dataset under a distributed search setting. SpaceV80M is only one partition of this dataset. Without counting other index metadata, the dataset itself already takes 40GB of memory and is the maximal size non-compression–based approaches such as HNSW can run on a single machine without failing. The latency SLA (ϕ) is 10ms for each query and the accuracy needs to be high ($\xi = 0.97$) to guarantee user satisfaction.

For compression–based approaches such as IVFPQ, although they can reduce memory consumption considerably, their recall cannot meet the accuracy target even on multi-million scale datasets given the latency constraint. HNSW takes 4.1ms to reach 0.972 recall, meeting latency SLA and recall target, but it consumes 56GB of memory (on average 720-byte per vector) to host a single partition of the dataset, which means it would require thousands of distributed machines to handle a peak load of thousands of queries per second. In contrast, although GRIP takes 0.6ms longer to do the search, which is marginal given it is already within the response time limit, it not only reaches a higher recall 0.977, but more importantly, it requires approximately only 51 bytes of DRAM per vector and in total 4G of memory, which is a 14X memory cost reduction. For a larger dataset such as SpaceV120M, HNSW runs out-of-memory and fails. In contrast, GRIP uses 6GB of memory and 60GB of SSD space, both are still far below their hardware capacity on a single machine, while meeting the accuracy target and latency SLA. GRIP is therefore a scalable and more suitable approach to building large-scale vector search engines.

8 Conclusion

The scalability requirement of large scale vector search differentiates our problem from traditional ANN problems, where the data is assumed to fit in memory. We present GRIP, a multi-store capacity-optimized ANN algorithm for building next-generation large-scale vector search engines which collaboratively optimizes accuracy, latency, and memory usage using both DRAM and SSDs. Our evaluation results show that GRIP achieves an order of magnitude improvements on system capacity, making ANN search fast and scalable on large-scale data while attaining high–quality search results, compared to the state-of-the-art.

Acknowledgments

We thank Junhua Wang, Jason Li, Shi Zhang, and Han Zhang from Search & AI and DLVS for their support of GRIP. We thank David Andersen, Conglong Li, Olatunji Ruwase, Samyam Rajbhandari, Wenhan Wang, Qi Chen, and Mingqin Li for their helpful discussions. We thank our anonymous reviewers for their valuable feedback that helps improve the quality of the work.

References

[1] Accessed: 05-20-2019. Benchmarking nearest neighbors. https://github.com/erikbern/ann-benchmarks.
[2] Accessed: 05-20-2019. Faiss: A library for efficient similarity search and clustering of dense vectors. https://github.com/facebookresearch/faiss.
[3] Accessed: 05-20-2019. Link & code source code. https://github.com/facebookresearch/faiss/blob/master/benchs/link_and_code/README.md.
[4] Accessed: 05-20-2019. NMSLib. https://github.com/nmslib/nmslib.
[5] Accessed: 05-20-2019. Samsung drops 128TB SSD and kinetic-type flash drive bombshe00lls. https://www.theregister.co.uk/2017/08/09/samsungs_128tb_ssd_bombshell/.
[6] Mohammad Aliannejadi, Hamed Zamani, Fabio Crestani, and W. Bruce Croft. 2018. Target Apps Selection: Towards a Unified Search Framework for Mobile Devices. In *SIGIR 2018*. 215–224.
[7] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *CVPR 2016*. 2055–2063.
[8] Artem Babenko and Victor S. Lempitsky. 2017. AnnArbor: Approximate Nearest Neighbors Using Arborescence Coding. In *ICCV 2017*. 4895–4903.
[9] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD 1990*. 322–331.
[10] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517.
[11] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *TOS* 12, 3 (2016), 13:1–13:39.
[12] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A library for fast approximate nearest neighbor search.* https://github.com/Microsoft/SPTAG
[13] Mostafa Dehghani, Hamed Zamani, Aliaksei Severyn, Jaap Kamps, and W. Bruce Croft. 2017. Neural Ranking Models with Weak Supervision. In *SIGIR 2017*. 65–74.
[14] Matthijs Douze, Hervé Jégou, and Florent Perronnin. 2016. Polysemous Codes. In *ECCV 2016*. 785–801.
[15] Matthijs Douze, Alexandre Sablayrolles, and Hervé Jégou. [n. d.]. Link and Code: Fast Indexing With Graphs and Compact Regression Codes. In *CVPR 2018*.
[16] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. In *VLDB'19*.
[17] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR 2013*.
[18] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB'99*. 518–529.
[19] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. BitFunnel: Revisiting Signatures for Search. In *SIGIR '17*.
[20] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W. Bruce Croft. 2016. A Deep Relevance Matching Model for Ad-hoc Retrieval. In *CIKM 2016*. 55–64.
[21] D. Frank Hsu, Xiaojie Lan, Gabriel Miller, and David Baird. 2017. A Comparative Study of Algorithm for Computing Strongly Connected Components. In *DASC'17*.
[22] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *CIKM '13*. 2333–2338.
[23] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. In Product Quantization for Nearest Neighbor Search. *TPAMI 2011*.
[24] Herve Jegou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *ICASSP 2011*.
[25] Yannis Kalantidis and Yannis S. Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR 2014*. 2329–2336.
[26] D. T. Lee and C. K. Wong. 1977. Worst-case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees. *Acta Informatica* 9, 1 (March 1977), 23–29.
[27] Victor Lempitsky. 2012. The Inverted Multi-index. In *CVPR '12*. 3069–3076.
[28] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: a memory-efficient, high-performance key-value store. In *SOSP 2011*. 1–13.
[29] Yury A. Malkov and D. A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* arXiv preprint abs/1603.09320 (2016).
[30] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *TPAMI 2014* 36, 11 (2014), 2227–2240.
[31] Mohammad Norouzi and David J. Fleet. 2013. Cartesian K-Means. In *CVPR 2013*.
[32] Karthik Ramachandra, Mahendra Chavan, Ravindra Guravannavar, and S. Sudarshan. 2015. Program Transformations for Asynchronous and Batched Query Submission. *IEEE Trans. Knowl. Data Eng.* 27, 2 (2015), 531–544.
[33] Christophe Van Gysel, Maarten de Rijke, and Evangelos Kanoulas. 2016. Learning Latent Vector Spaces for Product Search. In *CIKM '16*. 165–174.
[34] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *KDD 2018*. 839–848.
[35] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. In *CACM'09*.
[36] Chenyan Xiong, Zhuyun Dai, Jamie Callan, Zhiyuan Liu, and Russell Power. 2017. End-to-End Neural Ad-hoc Ranking with Kernel Pooling. In *SIGIR 2017*. 55–64.
[37] Lei Yu, Karl Moritz Hermann, Phil Blunsom, and Stephen Pulman. 2014. Deep Learning for Answer Sentence Selection. *CoRR* abs/1412.1632 (2014).
[38] Hamed Zamani, Bhaskar Mitra, Xia Song, Nick Craswell, and Saurabh Tiwary. 2018. Neural Ranking Models with Multiple Document Fields. In *WSDM '18*.