Pyramid: A General Framework for Distributed Similarity Search on Large-scale Datasets

Shiyuan Deng*, Xiao Yan*, Kelvin K.W. Ng, Chenyu Jiang, James Cheng

Department of Computer Science
The Chinese University of Hong Kong
Shatin, Hong Kong
{sydeng7, xyan, kwng6, cyjiang7, jcheng}@cse.cuhk.edu.hk

Abstract—Similarity search is a core component in various applications such as image matching and product recommendation. However, single-machine solutions are usually insufficient due to the large cardinality of modern datasets. We present Pyramid, a general and efficient framework for distributed similarity search. Pyramid supports search with popular similarity functions including Euclidean distance, angular distance and inner product. Different from existing distributed solutions that are based on KD-tree or locality sensitive hashing (LSH), Pyramid is based on the Hierarchical Navigable Small World graph (HNSW), which is the state-of-the-art similarity search algorithm. To achieve high query processing throughput, Pyramid partitions a dataset into sub-datasets containing similar items for index building and assigns a query to only some of the sub-datasets for query processing. Experiments on large-scale datasets show that Pyramid produces quality results for similarity search, achieves high query processing throughput and low latency, and is robust under node failure and straggler.

Index Terms—similarity search, distributed system

I. INTRODUCTION

Given a query \( q \in \mathbb{R}^d \) and a similarity function \( s(q, x) \), similarity search finds the item that is most similar to the query in a dataset \( X \) \[1\]. Popular similarity functions include Euclidean distance, angular distance and inner product. In practice, it is commonly required to return the top-\( k \) most similar items to a query. Similarity search is a key component in a large number of applications including large-scale image search, recommendation, and sequence matching. Exact similarity search is usually too costly and approximate similarity search, which returns a good portion of the exact top-\( k \) neighbors, suffices for most applications. Thus, we focus on approximate similarity search in this paper.

Similarity search algorithms. Existing similarity search algorithms can be roughly classified into four categories, i.e., tree-based methods \[2\], locality sensitive hashing (LSH) based methods \[3\], vector quantization based methods \[4\] and proximity graph based methods \[5\]. Among them, the proximity graph based methods were shown to provide the best performance in a number of empirical studies. Among the proximity graph based methods, the Hierarchical Navigable Small World graph (HNSW) \[5\] represents the state-of-the-art method because of its fast index construction and good search performance. We will give a detailed introduction to HNSW in Section II. There are already some distributed similarity search solutions for large datasets \[6\], \[7\]. However, these solutions use either tree-based methods \[6\] or LSH-based methods \[7\], and scalable solutions for the more recently proposed proximity graph based methods are still lacking.

Single machine solutions. The main challenge of similarity search on large-scale datasets is the memory required to store the raw data and index data structures. For example, the SIFT1B dataset takes up 512GB if each feature is stored as a floating-point number. In addition, the proximity graph of HNSW also takes a large amount of memory as it needs to maintain a neighbor list for every item. To reduce memory consumption, existing single-machine solutions such as FAISS \[8\] use vector quantization techniques (e.g., OPQ \[4\]) to compress the items. However, the quantization error introduced by compression often harms the quality of search results. For example, using OPQ with 8 codebooks, FAISS achieves a precision of only 25.15% when 2\( ^{22} \) items are probed for top-10 Euclidean nearest neighbor search on SIFT1B. To provide high-quality search results and scale to even larger datasets (e.g., with trillions of items) we may encounter in the future, it is necessary to develop distributed solutions that can remove the memory constraint of a single machine.

Requirements. Apart from producing high-quality search results, a similarity search framework needs to fulfill three additional requirements for production use, i.e., high query processing throughput, low query processing latency, and good robustness. Query processing throughput is the number of queries the framework can handle in unit time (per second). Query processing latency measures time taken to process a query (from entering the system to finishing processing) and online applications typically require a query processing latency in the order of several milliseconds. As node failures and stragglers are common for distributed computing, the framework should also be robust under these adversarial situations.

Our solution. Pyramid is a distributed solution based on HNSW and supports popular similarity functions including

\[1\text{To calculate precision, the items are ranked according to their approximate similarity scores, i.e., } s(\tilde{x}, q). \text{ If } k' \text{ of the ground truth top } k \text{ Euclidean nearest neighbors are identified in the items ranking top } k, \text{ the precision is } k'/k. \]

978-1-7281-0858-2/19/$31.00 ©2019 IEEE 1066
Euclidean distance, angular distance and inner product. To achieve high query processing throughput, Pyramid adopts novel dataset partitioning and query assignment strategies. The key idea is to build a much smaller meta-HNSW that captures the structure of the entire dataset, which allows us to build two levels of indexes and pinpoint the neighbors of a query efficiently. Specifically, by partitioning the bottom layer of the meta-HNSW, Pyramid assigns dataset items to sub-datasets of roughly equal size and ensures that items in the same sub-dataset are similar to each other. By searching the meta-HNSW, Pyramid quickly identifies the sub-datasets that are likely to contain the neighbors of a query and involves only these sub-datasets in query processing without hurting the quality of the search results. For failure recovery and straggler mitigation, Pyramid replicates the sub-datasets and their corresponding HNSW graphs across the machines.

We tested Pyramid on three large-scale datasets, Deep500M, SIFT500M and Tiny10M. The results show that Pyramid provides high-quality search results and the precision can easily reach 90%. The throughput of Pyramid is over 2x compared with a naive solution that randomly partitions a dataset among the machines and builds an HNSW for each machine. Comparing with the famous FLANN library [6] that uses tree-based method for distributed similarity search, Pyramid provides a throughput that is over 100x higher and achieves better precision. Pyramid is able to keep the query processing latency within 2-3ms. Pyramid is also robust to stragglers and node failures with its replication strategy.

II. HIERARCHICAL NAVIGABLE SMALL WORLD GRAPH

In this section, we introduce the index building and query processing procedures of Pyramid. We also discuss the special considerations to make Pyramid work for MIPS.

For distributed similarity search with Pyramid, a straightforward solution (denoted as Pyramid-naive) is to randomly partition a dataset among the workers (i.e., machines) in a cluster and build an independent HNSW graph on each worker. A

Algorithm 1 HNSW: Query Processing via Graph Walk

1: Input: HNSW graph G, similarity function s, query q, search factor l for the bottom layer
2: Output: k approximate neighbors of q
3: Set queue $C.add(G.entryVertex, s(q, G.entryVertex))$
4: for $t = G.maxLayer$ downto 1 do
5: $C ← Search-Level(G, t, q, s, C, l)$
6: $C ← \{\max(C)\}$
7: $C ← Search-Level(G, 0, q, s, C, l)$
8: return Top k vertices with the maximum $s(q, x)$ in C

HNSW improves the KNN graph, which contains a single layer and each item is connected to its exact top-k neighbors. In a KNN graph, graph walk can only take small steps as the connections are local. If the starting vertex is far from the neighborhood of the query, graph walk needs a large number of steps to reach the true neighbors. In contrast, the upper layers of HNSW (containing uniformly sampled items from the dataset) allow graph walk to take large steps and approach the neighborhood of the query. Graph construction is also efficient for HNSW as it does not require each item to connect to its exact top-k neighbors. Instead, HNSW inserts items in the dataset sequentially into the graph, and for an item $x$, the highest layer it can appear (denoted by $u$) is first randomly generated from an exponential distribution. Then graph search is conducted using $x$ as the query to find the items most similar to it (in the current graph) in all layers below $u$ and $x$ is connected to them as neighbors.

In a number of empirical studies, HNSW is found to significantly outperform other similarity search algorithms, including tree-based methods, LSH-based methods and vector quantization based methods. These search algorithms are usually based on a near-optimal similarity function. However, HNSW is originally designed for metrics such as Euclidean distance and edit distance, it has been shown recently that HNSW also achieves state-of-the-art performance for maximum inner product search (MIPS) [9].

III. PYRAMID

In this section, we introduce the index building and query processing procedures of Pyramid. We also discuss the special considerations to make Pyramid work for MIPs.
query \( q \) is distributed to all workers and each worker processes the query with its own HNSW graph using Algorithm 1. The final search results are obtained by merging and re-ranking the partial results reported by the workers. However, a query invokes computation on all workers in HNSW-naive, which results in low query processing throughput.

If a query is handled by only some rather than all of the workers, query processing throughput can be improved as each query invokes less workload. This is the main design goal of Pyramid, which is achieved by novel dataset partitioning and query assignment strategies. In the indexing build phase, Pyramid partitions the dataset into sub-datasets containing items similar to each other and assigns each sub-dataset to a worker. Due to the partitioning, some sub-datasets are likely to contain the neighbors for a query while others are not. Then for query processing, it is sufficient to handle the query by the workers holding these potential sub-datasets and the other workers do not need to be involved, which results in high query processing throughput. In Pyramid, both dataset partitioning and query assignment are conducted using a small meta-HNSW built on samples from the dataset.

### A. Index Building

The indexing build procedure of Pyramid is shown in Algorithm 2. We first randomly sample \( m \) items from the original dataset and built a meta-HNSW \( G_m \) on the sampled items. Then, we partition the bottom layer (which is a proximity graph) of \( G_m \) into \( w \) balanced graph partitions (in the sense that each graph partition has similar total vertex weights) and try to minimize the number of edges across the graph partitions. Note that by minimizing the number of cross edges, we ensure that the items in each graph partition are similar to each other. Currently, we use the Karlsruhe Fast Flow Partitioning Algorithm [10] for partitioning, which adopts an efficient multi-level local improvement strategy to search for the best partitioning. Finally, for each item \( x \) in the original dataset, we assign it to sub-dataset \( \lambda^i \) according to its most similar item in \( G_m \). For each sub-dataset \( \lambda^i \), we build a sub-HNSW \( G_i \), independently. Note that there is a one to one mapping between the sub-datasets and the graph partitions (of the bottom layer of \( G_m \)) and the \( i \)-th graph partition corresponds to sub-dataset \( \lambda^i \).

We also consider load balancing in dataset partitioning. Assume that each item in \( \lambda \) is equally likely to be accessed by the queries, the sub-datasets \( \lambda^1, \lambda^2, \ldots, \lambda^w \) should have roughly equal sizes to balance the workloads of the workers. Therefore, we set the weight of a vertex in \( G_m \) as the number of items in \( \lambda \) (a larger sample from \( \lambda \)) for which the vertex is their nearest neighbor in \( G_m \) and ensure that the partitions of \( G_m \) have similar total vertex weights. Empirically, we observed that a small meta-HNSW (e.g., \( m = 10,000 \)) is sufficient for good performance and a large meta-HNSW is not favorable as it prolongs the query processing time.

**Distributed workflow.** The index construction procedure starts with each worker reading a part of the dataset from the distributed file system, and sampling some items from its local dataset. The sampled items are shuffled to one worker for meta-HNSW construction and graph partitioning. When graph partitioning finishes, the worker also builds a one-to-one mapping between the graph partitions (and the sub-dataset indexes) and the workers. After that, the meta-HNSW is broadcast to all workers along with related data structures. The workers decide for each item in its local dataset which sub-dataset it belongs to using the meta-HNSW and shuffle the items to their destination workers in parallel. When shuffle finishes, each worker builds an HNSW on its own sub-dataset.

### B. Query Processing

The query processing procedure of Pyramid is shown in Algorithm 3. Query processing starts by finding the top \( K \) neighbors of the query in the meta-HNSW \( G_m \), which can be done very efficiently as the meta-HNSW is usually very small. For each graph partition of the bottom layer of \( G_m \), if it contains one or more of these \( K \) neighbors, the query will be dispatched to its corresponding sub-dataset. We call \( K \) the branching factor as it controls how many sub-datasets a query will be forwarded to and a larger \( K \) means more sub-datasets will be involved. For each sub-dataset that receives the query, graph walk is conducted to find the top \( k \) neighbors of the query on its corresponding sub-HNSW. The final search results are obtained by selecting the top \( k \) neighbors from the partial results returned by the sub-datasets.

**Algorithm 2** Pyramid: Index Construction

1. **Input:** Dataset \( \lambda \), meta-HNSW size \( m \), number of sub-HNSWs \( w \)
2. **Output:** A meta-HNSW \( G_m \) and \( w \) sub-HNSWs
3. Sample \( m \) items from dataset \( \lambda \) uniformly at random
4. Build the meta-HNSW \( G_m \) on the sampled items
5. Partition the bottom layer graph of \( G_m \) into \( w \) partitions
6. For each item \( x \) in the original dataset \( \lambda \) do
   7. Find its top neighbor \( n(x) \) in \( G_m \) with Algorithm 1
   8. If \( n(x) \) is in the \( i \)-th graph partition of \( G_m \) then
      9. Assign \( x \) to sub-dataset \( \lambda^i \)
6. For sub-dataset from 1 to \( w \) do
   11. Build sub-HNSW \( G_i \) on sub-dataset \( \lambda^i \)

**Algorithm 3** Pyramid: Query Processing with meta-HNSW

1. **Input:** Query \( q \), branching factor \( K \), number of required neighbors \( k \), meta-HNSW \( G_m \), \( w \) sub-HNSWs
2. **Output:** \( k \) approximate neighbors of \( q \)
3. Initialize \( resSet = \emptyset \)
4. Find the top \( K \) neighbors of \( q \) in \( G_m \) with Algorithm 1
5. For the \( w \) graph partitions of the bottom layer of \( G_m \) do
   6. If partition \( i \) contains some of the \( K \) neighbors then
      7. Search sub-HNSW \( G_i \) for \( k \) neighbors of \( q \)
      8. Add the results to \( resSet \)
9. Extract the top \( k \) neighbors of \( q \) from \( resSet \)

**Distributed workflow.** When a query comes, it is assigned to a random worker as the coordinator. The coordinator searches
the meta-HNSW with the query and decides the sub-datasets that will be involved in processing the query according to the search results. Then, the query is dispatched to the corresponding workers and the workers search their own sub-HNSW with the query. Each involved worker returns \( k \) tuples of \((\text{item id}, \text{similarity score})\) to the coordinator. When all responses for a similarity score) query are gathered, the coordinator selects items with the top \( k \) similarity scores as the final results.

We give an illustration of Pyramid in Figure 2. If we treat the sub-HNSWs on the workers as the bottom layer, the meta-HNSW is equivalent to some common upper layers of the sub-HNSWs. By searching the meta-HNSW, we can quickly identify the sub-HNSWs that are likely to contain the neighbors of a query. This is analogous to the upper layers of an HNSW built using the entire dataset, which help quickly approach the neighborhood of the query.

### C. Pyramid for MIPS

One interesting property of MIPS is that items with large norm are very likely to be the search results, which we illustrate with Figure 1b. For the ImageNet dataset, which contains about 2 million, 150-dimensional descriptors of images, we found the exact top-10 MIPS results of 1,000 randomly selected queries. This gives us a result set containing 10,000 items (without de-duplication) and we calculated the percentage that items with different norm percentiles take in this result set. For example, the first bar (from left to right) in Figure 1b means that items ranking top 5\% in norm takes up 93.1\% of the result set.

The bias towards items with large norm in MIPS causes problems for both dataset partitioning and query processing. Items with large norm tend to be strongly connected with each other in the inner product HNSW as they are likely to be the MIPS results of each other. In Algorithm 2, we partition the meta-HNSW by minimizing the number of cross edges, which means the large norm items will be put into the same graph partition (denote this partition as the large norm partition). For dataset partitioning, most items will find their MIPS in the large norm partition and this means that one of the sub-datasets will be much larger than the others. This can cause the worker holding this sub-dataset to run out of memory. For query processing, the large norm partition is very likely to contain the top-\( K \) MIPS of most queries for meta-HNSW search, which makes the corresponding worker much more heavily loaded than the other workers.

We introduce two changes to Algorithm 2 to solve the aforementioned problems. Firstly, the sampled items are normalized to unit norm before building the meta-HNSW. As all meta-HNSW vectors have unit norm, each sub-dataset \( X' \) covers items pointing to similar directions, which avoids the problem that the large norm partition attracts much more items than the others. The problem that the large norm partition is hot for query processing is also solved as the query is assigned to the sub-datasets that are similar to it in direction and no sub-dataset is more likely to attract queries. Secondly, we introduce an additional item assignment stage after Line 9 in Algorithm 2, which adds the top \( r \) MIPS neighbors (in the entire dataset \( X \)) of each meta-HNSW vector to its corresponding sub-dataset. The motivation is that items with large norm can be the results of MIPS even if they do not point to similar direction as the query and these items may scatter across many sub-datasets. The additional item assignment stage allows items with large norm to be assigned to multiple sub-datasets, and thus reduces the number of sub-datasets to be accessed for processing a query.

### D. API and Implementation

As a distributed system, Pyramid consists of three kinds of major components, i.e., coordinators, executors and brokers. The coordinators receive queries from some upstream applications (e.g., image search), search the meta-HNSW with the queries and send query processing requests to the executors with the relevant sub-HNSWs. The executors conduct search on their own sub-HNSWs with the received query processing requests and return the partial results to the coordinators. The coordinators merge these partial results to obtain the final results. A coordinator handles a query asynchronously and it immediately exits after sending the query processing requests to executors. A callback will be invoked to conduct merge when all partial results are available. The brokers handle the delivery of the query processing requests from the coordinators to the executors and use Kafka for reliable message passing on unreliable network. A Zookeeper cluster is used to monitor the workers in the system to detect failures.

Through the `execute(query, para)` method of the `Coordinator` class, a user can provide parameters for query processing, including the branching factor \( K \) and the number of required neighbors \( k \). For graph construction, a user can provide the dataset path and the similarity metric to the `GraphConstructor` class, which will build the meta-HNSW and sub-HNSWs. Users can also specify the parameters for graph construction, such as the size of the meta-HNSW \( m \) and the number of sub-HNSWs \( w \). For straggler mitigation, Pyramid relies on the message distribution mechanism of Kafka, which periodically re-balances the message queues of the executors serving the same sub-HNSW. Pyramid also replicates the one sub-HNSW over multiple machines such that query processing can still continue even if one machine fails.
TABLE I: Datasets

<table>
<thead>
<tr>
<th>Name</th>
<th># item</th>
<th># dimension</th>
<th>size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deep500M</td>
<td>500,000,000</td>
<td>96</td>
<td>192</td>
</tr>
<tr>
<td>SIFT500M</td>
<td>500,000,000</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>Tiny10M</td>
<td>10,000,000</td>
<td>384</td>
<td>15.4</td>
</tr>
</tbody>
</table>

IV. EXPERIMENTAL RESULTS

In this section, we evaluate the performance Pyramid with extensive experiments. First, we explore the influence of the parameters on the performance of Pyramid. Second, we compare Pyramid with two distributed similarity search solutions, HNSW-naive and FLANN [6]. Finally, we evaluate the performance of Pyramid under straggler and failure.

We use three performance metrics, i.e., precision, throughput and latency. For top-k similarity search, an algorithm is allowed to return k items. If k' of these k returned items belong to the ground-truth top-k neighbors of a query, the precision is said to be k'/k. For query processing latency, we report its 90th percentile instead of the average, which is more critical to online applications. For the meta-HNSW and the sub-HNSWs in Pyramid, we set their parameters according to the recommendation in the HNSW paper [5]. We used the three datasets listed in Table I for the experiments. We conducted Euclidean NNS on Deep500M and SIFT500M, and MIPS on Tiny10M. All experiments were conducted on a cluster of 10 machines connected with 10 Gbps Ethernet. Each machine is equipped with two 8-core Intel Xeon E5-2620v4 2.1GHz processors and 48GB RAM, running on CentOS 7.2. As the cluster contains 10 machines, we use 10 sub-HNSWs. The reported results are for top-10 neighbor search.

A. Influence of The Parameters

In Pyramid, the meta-HNSW has two main parameters, i.e., size m (the number of vectors in the bottom layer graph) and branching factor K (the number of top neighbors in the meta-HNSW that are used to choose the sub-HNSWs for a query). On the Deep500M dataset, we tested their influence on different aspects of Pyramid’s performance and report the results in Figure 3 to Figure 4. Figure 3a reports the average access rate, which is defined as the fraction of sub-HNSWs that are accessed for processing a query. Given a fixed meta-HNSW size, the access rate increases with the branching factor K while under the same K, a larger meta-HNSW size results in a lower access rate. Figure 3b shows that the precision first increases rapidly with the branching factor and then stabilizes. Moreover, the precision is slightly higher for a smaller meta-HNSW size under the same branching factor.

Figure 4a shows that the throughput drops when K increases. This is because a larger K results in a higher access rate, which means that more sub-HNSWs are searched to answer a query. Under the same K, although the access rate is lower for a larger meta-HNSW size, meta-HNSW 100,000 does not always achieve higher throughput than meta-HNSW 10,000. This is because it is more complex to search a larger meta-HNSW. Figure 4b shows that the latency increases with the branching factor K. This is because a coordinator needs to wait for the results from more executors under the larger access rate that comes with larger K. The influence of the meta-HNSW size on latency is more complicated as a large meta-HNSW reduces the access rate but requires long time for searching the meta-HNSW. As Pyramid performs well with a meta-HNSW size of 10,000, we use this configuration in all subsequent experiments.

B. Comparison with Other Methods

We compared Pyramid with two other distributed similarity search solutions, i.e., HNSW-naive (introduced in Section III) and FLANN [6]. FLANN is a widely used library for distributed similarity search that is based on KD-tree. Similar to HNSW-naive, FLANN randomly partitions the data among the workers and builds an index on each worker. For a fair throughput comparison between Pyramid and HNSW-naive, we adjusted their query processing parameters to achieve a precision of approximately 90%. As it is difficult for FLANN to achieve a 90% precision, we report both its precision and throughput under the setting recommended in [6].
Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on September 09, 2020 at 07:44:11 UTC from IEEE Xplore. Restrictions apply.