

# Hybrid Approximate Nearest Neighbor Indexing and Search (HANNIS) for Large Descriptor Databases

M M Mahabubur Rahman  
Computer Science  
Texas State University  
San Marcos, TX, USA  
toufik@txstate.edu

Jelena Tešić  
Computer Science  
Texas State University  
San Marcos, TX, USA  
jtesic@txstate.edu

**Abstract**—In this paper, we present a novel method for efficient and effective retrieval of similar deep descriptors. Our new hybrid method for indexing and searching for the approximate nearest neighbors in high-dimensional large deep-descriptor databases retrieves truly similar items in the database, even if the retrieval set is large. The proposed solution — hybrid approximate nearest neighbor indexing and search (HANNIS) — partitions the whole data space using the kmeans++ algorithm and then indexes each cluster using adapted hierarchical navigable graphs. This approach enables us to load items that are truly close to the incoming query at retrieval time. HANNIS outperforms all state-of-the-art methods in terms of recall at depths of up to 100 and offers consistent index loading and retrieval performance.

**Index Terms**—large set retrieval; high dimensional indexing and search; deep descriptors; big data

## I. INTRODUCTION

The increasing amount of data requires efficient and scalable retrieval of similar instances for further analysis in most data science applications, such as data integration, recommender systems, information retrieval, software engineering, cybersecurity, outlier detection, classification, and clustering. A similarity search can be defined as a search for objects from a database that are close to a query based on some sort of sameness, also known as the distance function. According to its definition, the nearest-neighbor problem is to create a data structure that, given any point  $q$  in a metric space  $(X, D)$ , gives the point in  $P$  that is closest to  $q$  (the point *nearest neighbor* in  $P$ ). The data structure maintains more details about the set  $P$ , which is then utilized to locate the closest neighbor without calculating all the distances between  $q$  and  $P$ .

The  $k$ -nearest neighbors ( $k$ -NN) search identifies the top  $k$  nearest neighbors to the query and performs very well for retrieving exact solutions in smaller data sets with a lower dimension. However, the  $k$ -NN search can be sluggish in large datasets and higher dimensions because of the "curse of dimensionality". Instead of generating a model from the training data, the  $k$ -NN search takes into account the entire data set each time

a query is initiated. This quality leads to high memory usage and inefficiency in the retrieval result as the data increases.

To address this problem, several approximate nearest-neighbor (ANN) methods were introduced. ANN methods work very fast but sacrifice some accuracy by loosening the condition of exact nearest-neighbor retrieval. ANN methods handle the "curse of dimensionality" well and thus have superior performance over the  $k$ -NN search in large datasets with higher dimension. ANN methods, which aim to yield any point  $p' \in P$  such that the distance from  $q$  to  $p'$  is at most  $c \cdot D(q, p)$ , for some  $c \geq 1$  (Figure 1), can be used to speed up computations with closest neighbors and reduce memory usage by the data structure. Many effective ANN solutions have been proposed in the past decade and can be grouped as graph-based [1]–[5], hashing-based [6]–[8], and partition-based [9]–[11] methods. In this paper, we focus on indexing and retrieval improvements for real deep feature databases for the unknown class discovery application, and we measure algorithms in terms of high recall at any depth of retrieval, fast index loading, and retrieval times.

A few libraries [1], [9], [12], [13] have used the graph-based Hierarchical Navigable Small World (HNSW) algorithm. Among all ANN methods, the HNSW algorithm [1] shows promising results in terms of retrieval time [14]. However, there is still room for improvement in recall. Moreover, all of the above libraries build a large index that often takes a long time to load the index from memory during nearest neighbor retrieval.

In this paper, we show that the hybrid approximate nearest neighbor indexing and search (HANNIS) approach outperforms existing state-of-the-art ANN method libraries built on the HNSW algorithm. Both the indexing and searching of our proposed method has two phases. For indexing, HANNIS first clusters all data points using the kmeans++ algorithm and then arranges the graph into a hierarchical layer of proximity graphs for each cluster along with their cluster center (centroid) information. During the search, HANNIS first loads the index closest to the query and then performs a layer-by-layer search. In Section II we discuss the indexing and searching procedure in detail. HANNIS outperforms the state-of-the-art libraries built on the HNSW algorithm in terms of accuracy for the publicly available SIFT10M and DEEP10M datasets [15], [16]. In Section III, we discuss our experimental results in detail.

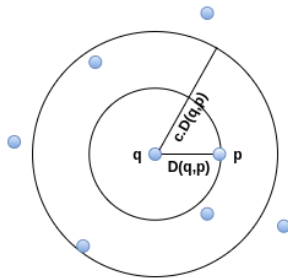


Fig. 1: ANN search

Our HANNIS library is suitable only for vector spaces and does not allow index updating.

#### A. Related Work

**Neighbor-ANN methods** for indexing and searching are based on local neighborhood data points. Hierarchical Navigable Small World (HNSW) [1] arranges the graph into a hierarchical layer of proximity graphs, allowing the algorithm to evaluate only a needed portion without being dependent on the entire network. The query process starts at the top layer where the edges are the longest and then performs a greedy search within that layer until it reaches a local minimum. The search then switches to the lower layer, where the edges are shorter, and this time the starting point is the previous local minimum and continues until the query is reached. So far, HNSW has provided the best performance in terms of retrieval time among all ANN methods. However, it takes a significant time to build indexes for a large data set. Also, loading the offline index from the database is slow. NSG [3] selects one node and tries to ensure the existence of monotonic pathways connecting it to all other nodes. "Navigating Node" is the term given to this node. The search always begins with the Navigating Node, which makes the search efficient on an NSG. However, NSG is excessively sparse, which hurts search speed. Furthermore, NSG experiences considerable indexing complexity and offers no theoretical guarantee on the nearest-neighbor search. Navigating the satellite system graph (NSSG) [4] addresses the problem of theoretical guarantee of NSG by introducing the concept of a satellite system graph (SSG). During the SSG indexing process, an angle  $\alpha$  is provided for dataset  $D$ . The first neighbor of each data point  $P \in D$  is the closest data point  $Q$  and all other neighbors  $R$  of  $P$  satisfy the condition  $\cos \angle RPQ < \cos \angle \alpha$ . NSSG also uses the pruning technique during index building to eliminate the farthest and duplicate data points. The search process starts at the fixed point and proceeds towards the neighbor that has the smallest distance from the query. This process repeats until the query is reached. The sparsity of NSSG can be controlled by changing the angle  $\alpha$ , making it faster than non-graph-based methods. However, the indexing and memory cost is higher than that of non-graph based methods.

Neighborhood Graph Tree (NGT) [5] uses a range search during the graph construction mechanism, and, to avoid a high degree of neighboring nodes and reduce memory overhead, applies a three-degree adjustment by connecting each data point to its three nearest neighbors throughout the graph. During the query process, NGT generates a seed using the VP tree [17] and performs a range search to obtain the nearest neighbors. A major drawback of NGT is that if the query and seed are far away from each other in the search space, then it takes many hops in between to reach the query from the seed, and thus increases the retrieval time. One way to address this problem is to transform the  $k$  nearest neighbor graph into a bidirectional one [18], and the other is to construct an undirected graph by continuously inserting elements. All neighboring ANN methods

suffer from a long index building time and low retrieval for large deep-descriptor databases [19].

**Space Partition ANN methods** narrow the entire search space to a smaller group of similar data points. The similarity is defined based on some distance metric, and different partitioning techniques such as clustering, Voronoi partition, random divided partition, etc. are exploited to separate each similar group. The Faiss library enables efficient partitioning of data in Voronoi cells [15], and the index of each cell is a centroid of that cell. Faiss uses different compression techniques, such as product quantization [20] to compress actual vectors, helping to work faster with large data sets. Data points in each cell can be further indexed using different indexing algorithms such as LSH, HNSW, NSG, etc. In the search algorithm, the centroid with the smallest distance from the query point narrows the search space down to the target cell and performs a similarity search on the target cell. However, for large datasets, a large number of Voronoi regions do not contain data points, and a lot of time is spent searching for empty regions [15].

**Data Partition ANN methods** for indexing and searching rely on data-driven indexes. One such approach is the  $kd$ -tree where the whole data space is partitioned in the dimension with the highest variance recursively until a certain number of datapoints remain in the leaf. Improvement of the  $kd$  tree proposes building multiple randomized  $kd$  trees in parallel by splitting the data in one of the top five dimensions with the highest variance, rather than in the dimension with the highest variance. During the search process, a shared priority queue is used to perform a depth-first search with some heuristic scoring function along all the randomized  $kd$ -trees [21]. FLANN combines the randomized  $kd$ -tree approach with hierarchical k-means [22] for indexing. In the hierarchical k-means indexing algorithm, k-means clustering is performed recursively until each leaf contains fewer than  $k$ -children, and the result is the cluster tree. The search method starts at the root node and crosses the inner node that has the lowest distance from the query to the cluster center. FLANN performance degrades in higher-dimensional feature spaces as the splitting based on a single dimension cannot reflect the whole data set in  $kd$ -tree. In addition, the choice of initial cluster centers in the k-means algorithm makes a huge difference in the final clusters.

Annoy [11] uses a random projection and builds a tree. The space is split into two subspaces at each intermediate node in the tree by a randomly selected hyperplane. This process is done  $n$  times to build a forest of trees. One major advantage of Annoy is that it can use static files as indexes and that the indexes can be shared between processes. Additionally, Annoy separates the processes of building an index from loading it, allowing one to share an index as a file and rapidly map it to memory. This helps to reduce the memory footprint. However, Annoy has low performance in terms of speed and accuracy, which they have mentioned in their GitHub library.

**Hashing ANN methods** for indexing attempt to hash similar input items into the same "buckets" with high probability, and the methods are either data independent, such as locality

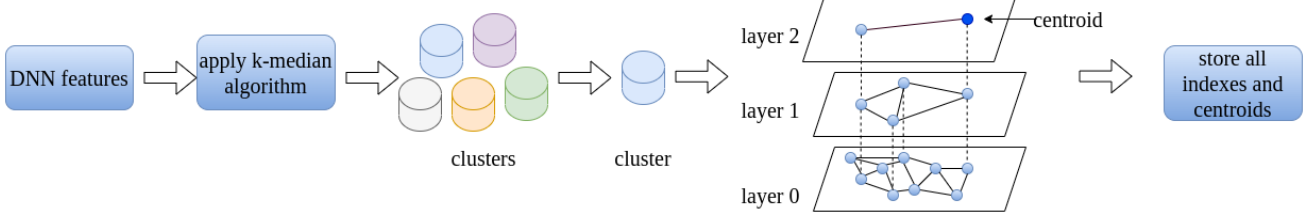


Fig. 2: HANNIS: proposed indexing approach.

sensitive hashing (LSH), or data-dependent methods, such as locality-preserving hashing (LPH). LSH-based methods mostly rely on locality-sensitive hash functions with a view to generate same hash code for similar input data points and different hash codes for dissimilar data points. The most used hash function for creating hash codes is random linear projections, where the 2-stable distribution (normal distribution) is used to select the random projection parameters. LPH-based methods generate specific hash functions from the distribution of the whole dataset. The primary purpose of LPH algorithms is to preserve similarity between the original data space and the hash-coding space. SRS [8] falls under the LSH method, which first generates  $m$  2-stable random projection vectors and calculates the projections for each data point. Then, the  $m$ -dimensional projections associated with their respective IDs are assigned a multidimensional index. SRS uses the  $R$ -tree [23] index structure, which can incrementally return the  $(k + 1)$ th nearest neighbor after calculating the  $k$ th nearest neighbor. During the search, SRS projects the query with the same  $m$  projection vectors to the  $m$ -dimensional space. Then an exact  $k$ -NN query is performed incrementally on the  $R$ -tree index centered on the query projection. The major drawback of SRS is that the points colliding with the query in a part of a hash function in one hash table are ignored, since the hash tables are built before searching, even though they are probably close in the vector space. Unlike constructing a hash table with "static" hash functions, QALSH [7] uses a dynamic collision count approach. By building a  $B+$ -tree on each random projection and performing incremental range queries until the top- $k$  candidates are determined, QALSH pioneered query-aware hash algorithms. It is important to remember that the hash functions are computed intelligently based on the range of possible query executions rather than on a single query point. One drawback of QALSH is that, to meet the query accuracy guarantee, additional hash algorithms are needed as the dataset size increases, which requires more time. Furthermore, when new projections are formed, it can be expensive for QALSH to create  $B+$  trees on top of each projection to maintain accuracy.

## II. METHODOLOGY

In this section, we present our proposed method, which is divided into two sections. First, we introduce the algorithm for efficiently pre-processing the data and build offline indexes to accelerate the search, as illustrated in Figure 2. Next, we

---

### Algorithm 1: BUILD(HANNIS, $X$ , $ncls$ , $niter$ , $M$ , $cand$ )

---

**Input:** multilayer graph  $HANNIS$ , data vector  $X$ , number of clusters  $ncls$ , number of iteration to find centroids  $niter$ , number of established connections  $M$ , size of dynamic candidate list  $cand$

**Output:** Update HANNIS inserting all elements

```

1 clusters, centroids  $\leftarrow$   $KMEANSPP(X, ncls, niter)$ 
2 foreach  $element (cls, cen)$  of  $(clusters, centroids)$  do
3   foreach  $c$  of  $cls$  do
4      $INSERT(HANNIS, c, M, cand)$ 
5   end
6    $SAVE(HANNIS, cen)$ 
7 end
```

---

propose the  $k$ -NN retrieval approach, which makes use of the proposed indexing approach.

---

### Algorithm 2: SEARCHINDEX( $q$ , centroids, $k$ , $n$ )

---

**Input:** query element  $q$ , cluster centers  $centroids$ , number of nearest neighbors  $k$ , number of clusters to load  $n$

**Output:**  $k$  closest neighbors to  $q$

```

1 foreach  $cen$  of  $centroids$  do
2   find the distance from  $cen$  to  $q$ 
3 end
4  $nclus \leftarrow n$  closest centroids to  $q$ 
5 foreach  $iclus$  of  $nclus$  do
6    $HANNIS \leftarrow LOADINDEX(iclus)$ 
7    $nclusK \leftarrow SEARCH(q, k)$ 
8 end
9 neighbors  $\leftarrow k$  nearest neighbors to  $q$  in  $nclusK$ 
10 return neighbors
```

---

a) *Index building:* To reduce the search space and accelerate the index loading time, we first subdivide the entire data space into  $n$  different partitions of  $C_1, C_2, C_3, \dots, C_n$ . Clustering is a technique that can group similar datapoints, and cluster centroids can be used as representative of a particular cluster. For this work, we have used the k-means++ algorithm [24], as it chooses the initial cluster center in a way that results in better clustering over k-means. All centroid information and the distances from the centroid to the data points in each

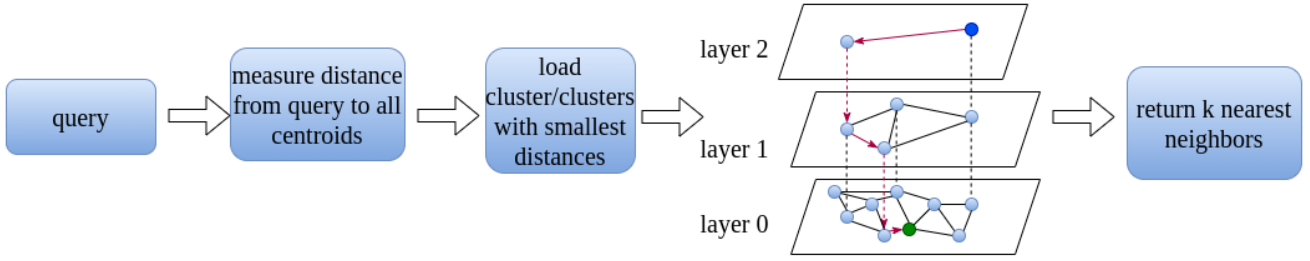


Fig. 3: Proposed HANNIS retrieval pipeline.

cluster are stored and passed to the next phase to build the HNSW index (Algorithm 1 line 2). Once the groups have been calculated, each group is hierarchically arranged. The graph is arranged into a hierarchical layer of proximity graphs where upper layer contains the data points with the longest edges (distance to the neighboring data points), and the lower layer contains the data points with the smallest edges (Figure 2). This layered structure allows the algorithm to evaluate only a needed portion without relying on the entire network and maintains logarithmic complexity. Data points are sequentially added to the graph one at a time. A probability function,  $P(L) = F(L, l_m)$ , is used to determine the insertion layer for an element. The value  $L$  denotes at which layer an element will be inserted. The probability function normalized by the "level multiplier"  $l_m$ , where  $l_m=0$  indicates that vectors are only inserted at layer 0, gives the probability of a vector insertion at a given layer. We attain the highest performance when we reduce the overlap of shared neighbors between layers. We can reduce the overlap by decreasing  $l_m$ . But doing so, as more vectors are moved to layer 0, raises the average number of search traversals. So, we utilize a  $l_m$  value of  $\log_2(4M/5)$  that evenly distributes both, where  $M$  is the maximum number established connection at any data point. The starting point for building an index is chosen randomly, and a heuristic is used to select neighbors. Finally, all indexes are stored with their centroid information so that only a certain index can be loaded based on the query, which in turn saves the index loading time (Algorithm 2 line 5).

*b) Improving effectiveness of retrieval:* Figure 3 shows the  $k$ -NN retrieval approach. During the retrieval, at first distances from the query to all centroids are measured and sorted based on their distances. Then the index with the smallest distance from the query to the centroid is loaded for searching. After loading an index, a greedy search is performed in the graph index. Then finally the top  $k$ -NN to the top  $k$  is returned. In the Algorithm 2 line 3, the distances between all the centroids and query are calculated. Then we sort the distances and load the cluster that has smallest query to centroid distance (Algorithm 2 line 4). A control parameter determines how many indexes to load, and loading multiple indexes improves accuracy but increases the retrieval time (Algorithm 2 line 5). The search within an index starts with the centroid in the upper layer where the edges are the longest, and then a greedy search is used within that layer until it reaches a local minimum (Figure

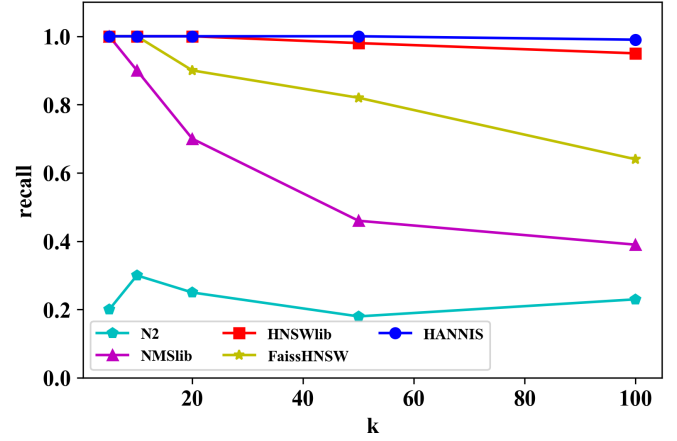


Fig. 4: Recall@k for DOTA 2.0 dataset with 2,697,873 instances of 1024 dimensional floats for five methods N2, Nmslib, Faiss HNSW, HNSWlib, HANNIS.

3). Then the search switches to the lower layer where the edges are shorter. This time, the starting point is the previous local minimum, and this process continues until the query is reached and the top  $k$ -NN to the top  $k$  is returned (Algorithm 2 line 8). For multi-index search, each index returns its top  $k$ -NN to the query. Then, all the retrieval results are sorted on the basis of their distance to the query. Finally,  $k$ -NN are chosen as the final retrieval result (Algorithm 2 line 9). By loading multiple indexes, our multi-index search handles the situation where the query lands on the boundary of a cluster in feature space that some of its actual nearest neighbors are in different cluster.

### III. EXPERIMENTS

For all of our experiments, we have used the benchmark data set DEEP10M with 10 million instances and 96 dimensions [15], SIFT10M with 10 million instances and 128 dimensions [16], and DOTA 2.0 data set with 1,024 dimensions and 2,7 million instances [25]. For the subsequent data set, we extracted deep features using Faster-RCNN [26] with Resnet50 [27] as the backbone and the Detectron2 [28] code baseline.

**Performance analysis** We measured the performance of our HANNIS method with four different state-of-the-art methods built on the HNSW algorithm. Three performance measurement metrics have been used to evaluate the performance of each method: recall@ $k$ , index loading time, and retrieval time.

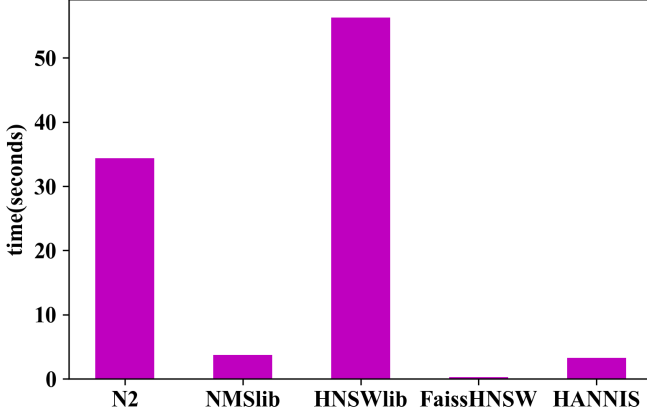


Fig. 5: Index loading times into memory for DOTA 2.0 dataset with 2,697,873 instances of 1024 dimensional floats for  $k = 100$  for five methods N2, Nmslib, Faiss HNSW, HNSWlib, HANNIS.

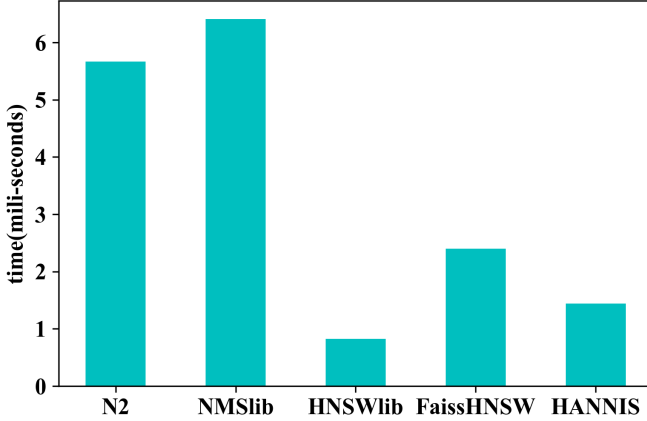


Fig. 6: Retrieval times for DOTA 2.0 dataset with 2,697,873 instances of 1024 dimensional floats for  $k = 100$  for five methods N2, Nmslib, Faiss HNSW, HNSWlib, HANNIS.

Fairness to all, we used the same parameters for all search methods, including HANNIS. The number of established connections and the size of the dynamic candidate list were set at 16 and 200, respectively, throughout all experiments. All experiments were carried out on Ubuntu 20.04.3 server with 11th generation Intel® Core™ i9-11900K @ 3.5GHzX16 CPU with 128GB RAM and NVIDIA GeForce RTX 3070 8GB mem GPU.

**Recall @  $k$**   $R@k$  is measured as the percentage of the closest neighbors in the top  $k$  closest in the  $k$ ,  $k$  in  $[5, 10, 20, 50, 100]$  retrieval set of the 5 methods compared to the truth of the ground in  $k$ ,  $GT_k$ , a brute force search for nearest neighbors in Euclidean space:  $P_k = |M_k \cap GT_k| / |GT_k|$ .  $M_k$  is the set of descriptors recovered by the method  $M$ ,  $M \in [N2, NMSlib, HNSWlib, FaissHNSW, HANNIS]$ , and  $|M_k \cap GT_k|$  is the number of true positives, the number of nearest neighbors recovered by the method  $M_k$  that match  $GT_k$ . **The load time**

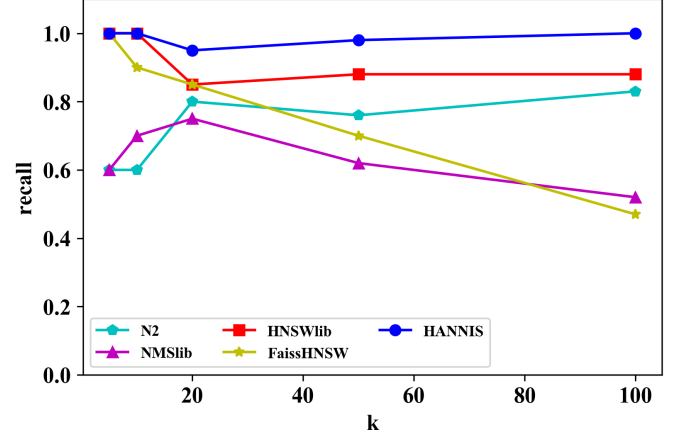


Fig. 7: recall@ $k$  for SIFT dataset with 10 million instances of 128 dimensional integers for five methods N2, NMSlib, FaissHNSW, HNSWlib, HANNIS.

**of the index** measures the time it takes to load the index from memory for retrieval. **Retrieval time** defines the time to retrieve the nearest neighbor  $k$  for a single query.

**DOTA2.0** dataset performance measures for five methods are compared in Figure 4. The set contains 2,697,873 instances of 1024 dimensional object deep feature descriptors extracted from DOTA2.0 using ResNet, and we are evaluating the method most suitable for unknown class discovery in the DOTA2.0 dataset. Our criteria is that recall@ $k$  needs to stay consistent as  $k$  increases while keeping the index loading time and retrieval time comparable to state-of-art. We tried different numbers of clusters during the  $k$ -means++ clustering phase and got the best results on 20 clusters. Fig. 4 recall@ $k$ -retrievals demonstrate HANNIS consistent retrieval recall@ $k$  slightly better than HNSWlib for  $k > 20$ , while HANNIS index loading time is much faster in Fig. 5, and retrieval times are comparable in Fig. 6. NMSlib performance rapidly degrades for  $k > 5$  (Fig. 4), and while its retrieval times are the shortest (Fig. 6), its index loading times are highest Fig. 5, and the method is not suitable. N2 and FaissHNSW have comparably fast index loading times (Fig. 5) and retrieval times (Fig. 6), but their performance quickly degrades for the larger retrieval set (Fig. 4). HANNIS shows to be the most suitable algorithm for unknown class discovery for DOTA 2.0 dataset.

**SIFT** dataset performance measures for five methods are compared in Figure 7. The set contains 10,000,000 instances of 128-dimensional integer SIFT [16], [29] image descriptors extracted from Caltech-256 41  $\times$  41 whole image patches. As the features were extracted from image patches and not object regions, our criteria here is to evaluate the most suitable method to match small image regions for large values of  $k$  for “needle in the haystack” search scenarios while keeping the index loading time and retrieval time in check for  $k = 100$  [30] (Fig. 7). We tried different numbers of clusters during the  $k$ -means++ clustering phase and got the best results on 15 clusters. Recall@ $k$ -retrievals demonstrates interesting behavior



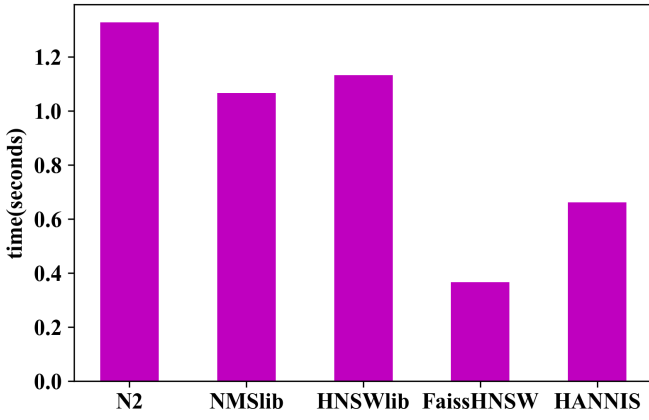


Fig. 8: Index loading times into memory for SIFT dataset with 10,000,000 instances of 128 dimensional integers for  $k = 100$  for five methods N2, NMSlib, FaissHNSW, HNSWlib, HANNIS.

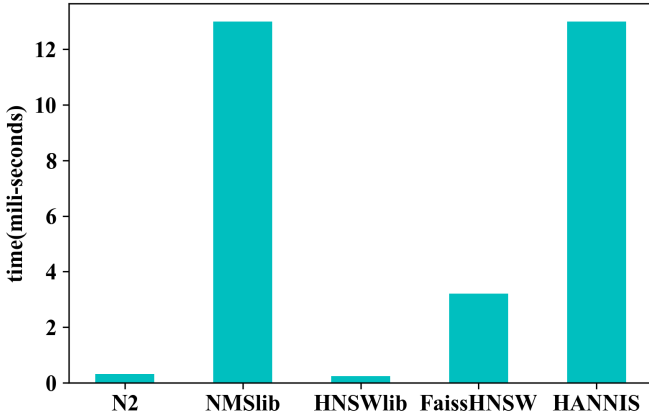


Fig. 9: Retrieval time for SIFT dataset with 10,000,000 instances of 128 dimensional integers for  $k = 100$  for five methods N2, NMSlib, FaissHNSW, HNSWlib, HANNIS.

for all methods that differ from DOTA2.0. HANNIS is now consistently dominating in effectiveness of the retrieval at **all**  $k \in [5, 10, 20, 50, 100]$ , and consistently better than HNSWlib for  $k > 10$  with smaller index loading times (Fig. 8) at the price of much higher retrieval time (Fig. 9). FaissHNSW has faster index loading times Fig. 8 and comparable retrieval times Fig. 6, but FaissHNSW performance quickly degrades for  $k > 5$  Fig. 4 and we do not consider the method appropriate for this scenario. We also observe interesting behavior of N2 in Fig. 7: the effectiveness of the indexing method is *improving* with larger  $k$  and approaching the effectiveness of *HNSWlib* for  $k = 100$  at low index loading time (Fig. 8) and low retrieval time (Fig. 9). NMSlib performance improves for  $k = 10, 20$  and then quickly degrades, even if the retrieval times are the fastest for  $k = 100$  Fig. 9. HANNIS shows to be the most robust algorithm for the “needle in the haystack” similar patch search in SIFT Caltech-256 dataset.

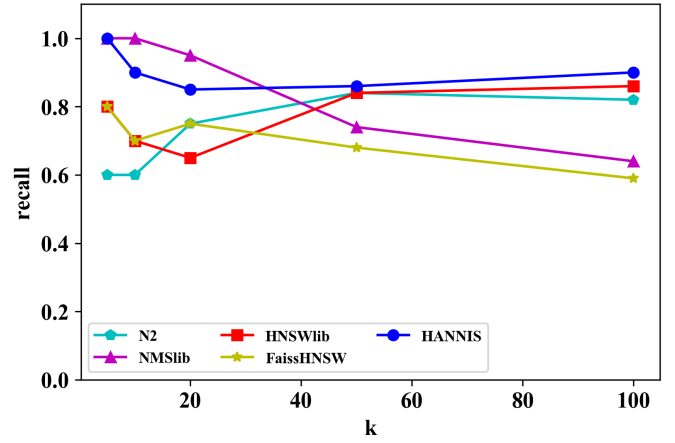


Fig. 10: Recall@k for Deep10M deep descriptors w.r.t. brute force search in Deep10M PCA 96-dimensional space for five methods N2, NMSlib, FaissHNSW, HNSWlib, HANNIS.

**Deep10M** dataset performance measures for five methods are compared in Figure 7. The set contains 10 million instances 96 dimensional float vectors. The original 10 million  $10^9$ -dimensional image embedding outputs of the Googlenet’s last fully-connected layer [15] were compressed and normalized into 96-dimensional vectors using principal component analysis. We tried different numbers of clusters during the  $k$ -means++ clustering phase and got the best results on 15 clusters. HANNIS shows better performance than HNSWlib for  $k \in [5, 10, 20]$  (Fig. 10) and similar performance with HNSWlib and N2  $k = 100$  where HANNIS has lower index loading time (and N2 even lower) (Fig. 11), and higher retrieval than HNSW and comparable to N2 time for  $k = 100$  (Fig. 12). Since the dimension of the original data ( $10^9$ ) was larger than the sample size  $10M$ , principal component analysis compression has shown to exhibit upward bias in sample eigenvalues, inconsistency of sample eigenvectors [31], and eigenvalue pollution [32]. It also has been shown that the PCA dimensionality reduction is not meaningful for indexing search and retrieval tasks [33]. Thus, it is hard for us to judge effectiveness of the indexing methods for specific application on this dataset as it has no practical meaning or application in its current form.

Next, we analyze the five methods index load timing and retrieval timing on a log scale for 2.7 million DOTA2.0 1024-dimensional float vectors (2764 million floats), 10 million SIFT10M 128-dimensional integer vectors (1,280 million integers), and 10 million Deep10M 96-dimensional float vectors (960 million floats) *per method* to uncover any common trends. DOTA2.0 is approximately 3 times the size of Deep10M, and nine times size of SIFT10M. Figure 13 compares the index loading time, and N2, NMSlib, HNSWlib, and HANNIS index load time is proportional to the size of the dataset, and HANNIS has the best overall index load timing. FaissHNSW has the lowest index loading time, and it does not correlate to data set size in instances and in feature dimension, and type. Figure 14

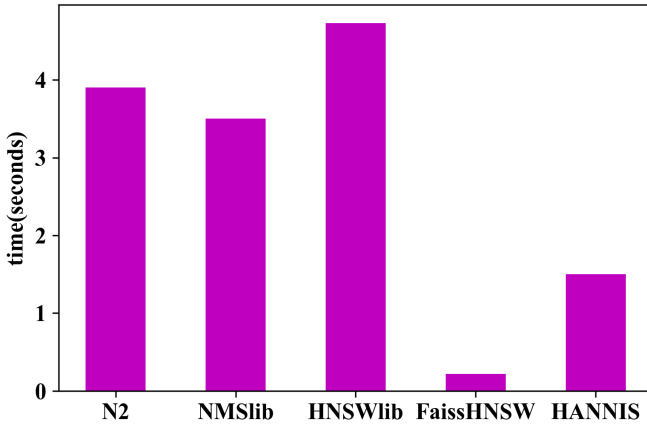


Fig. 11: Index loading times into memory for Deep10M deep descriptors w.r.t. brute force search in Deep10M PCA 96-dimensional space for  $k = 100$  for five methods N2, NMSlib, FaissHNSW, HNSWlib, and HANNIS.

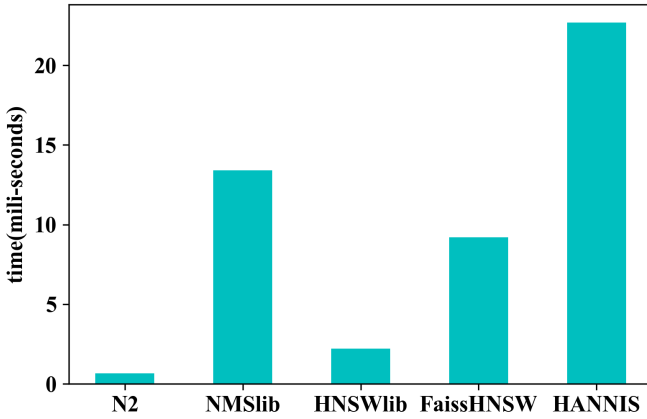


Fig. 12: Retrieval times into memory for Deep10M deep descriptors w.r.t. brute force search in Deep10M PCA 96-dimensional space for  $k = 100$  for five methods N2, NMSlib, FaissHNSW, HNSWlib, and HANNIS.

compares retrieval time per method for  $k = 100$ . For NMSlib, FaissHNSW and HANNIS methods, retrieval time corresponds to number of instances in the dataset, and HANNIS seems to do better on high-dimensional descriptors than the comparable methods. For N2, the retrieval time is directly proportional to the dataset size, and it is hard to interpret the rule for HNSWlib methods.

#### IV. CONCLUSION

The approximate nearest neighbor search approach mitigates the high cost of brute force  $k$ -Nearest Neighbor Search in large and high-dimensional data. In this paper, we have proposed a nearest-neighbor indexing and search method (HANNIS) to index and retrieve similar content from high-dimensional deep-descriptor data set, and demonstrated method's effectiveness for two real scenarios: unknown class discovery and whole image

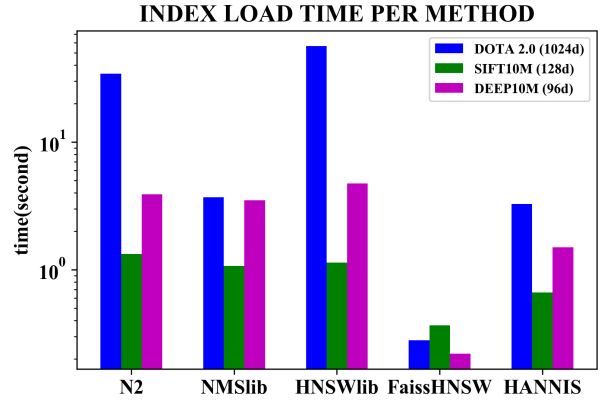


Fig. 13: Index loading time per method for 5 approaches over 3 datasets.

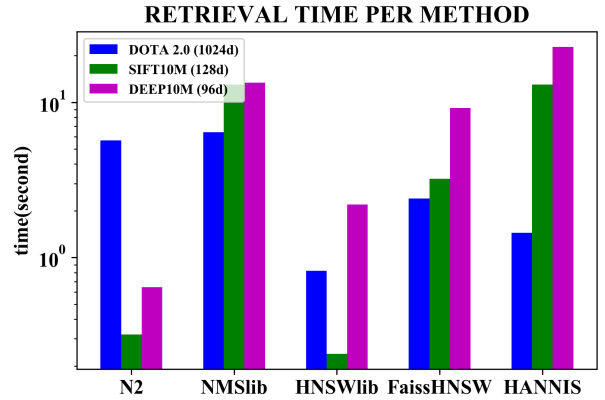


Fig. 14: Retrieval time for 5 approaches for 3 datasets.

feature search. HANNIS efficiently clusters all the data points using kmeans++ and then builds a Hierarchical Navigable Small World (HNSW) graph index for each cluster. During retrieval, HANNIS only loads the indexes that are most similar to the query. Recall exceeds that of all existing state-of-the-art libraries built on the HNSW algorithm up to 100. HANNIS is up to 18 times faster than state-of-the-art libraries in terms of index loading time. The retrieval times are similar to the state-of-the-art libraries for searching in vector space.

#### REFERENCES

- [1] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.
- [2] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586, 2011.
- [3] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

- [4] Cong Fu, Changxu Wang, and Deng Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.
- [5] Masajiro Iwasaki. NgT: Neighborhood graph and tree for indexing, 2015.
- [6] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. *Advances in neural information processing systems*, 28, 2015.
- [7] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.
- [8] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment*, 2014.
- [9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.
- [10] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence*, 36(11):2227–2240, 2014.
- [11] Erik Bernhardsson. *Annoy: Approximate Nearest Neighbors in C++/Python*, 2018. Python package version 1.17.1.
- [12] GeonHee Lee. *TOROS N2 - lightweight approximate Nearest Neighbor library which runs fast even with large datasets*, 2017. Python package version 0.1.7.
- [13] Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In *International Conference on Similarity Search and Applications*, pages 280–293. Springer, 2013.
- [14] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.
- [15] Dmitry Baranchuk, Artem Babenko, and Yuri Malkov. Revisiting the inverted indices for billion-scale approximate nearest neighbors. *CoRR*, abs/1802.02422, 2018.
- [16] Xiping Fu, Brendan McCane, Steven Mills, Michael Albert, and Lech Szymanski. Auto-jacobian: Auto-encoder jacobian binary hashing. *CoRR*, abs/1602.08127, 2016.
- [17] Masajiro Iwasaki and Daisuke Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355*, 2018.
- [18] Masajiro Iwasaki. Pruned bi-directed k-nearest neighbor graph for proximity search. In *International Conference on Similarity Search and Applications*, pages 20–33. Springer, 2016.
- [19] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.
- [20] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.
- [21] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.
- [22] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE transactions on computers*, 100(7):750–753, 1975.
- [23] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.
- [24] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.
- [25] Gui-Song Xia, Xiang Bai, Jian Ding, Zhen Zhu, Serge Belongie, Jiebo Luo, Mihai Datcu, Marcello Pelillo, and Liangpei Zhang. Dota: A large-scale dataset for object detection in aerial images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3974–3983, 2018.
- [26] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.
- [27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [28] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. <https://github.com/facebookresearch/detectron2>, 2019.
- [29] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.
- [30] Jelena Tesic and B. S. Manjunath. Nearest neighbor search for relevance feedback. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2003)*, 16-22 June 2003, Madison, WI, USA, pages 643–648. IEEE Computer Society, 2003.
- [31] Iain M. Johnstone and Debashis Paul. Pca in high dimensions: An orientation. *Proceedings of the IEEE*, 106(8):1277–1292, 2018.
- [32] Edward B Davies and Michael Plum. Spectral pollution. *IMA journal of numerical analysis*, 24(3):417–438, 2004.
- [33] Jelena Tešić. *Managing large-scale multimedia repositories*. University of California, Santa Barbara, 2004.