LANNS: A Web-Scale Approximate Nearest Neighbor Lookup System

Ishita Doshi
LinkedIn
idoshi@linkedin.com

Dhritiman Das
LinkedIn
dhdas@linkedin.com

Ashish Bhutani
Uber
abhutani@uber.com

Rajeev Kumar
LinkedIn
rkumar6@linkedin.com

Rushi Bhatt
Compass
rushi.bhatt@compass.com

Niranjan Balasubramanian
LinkedIn
nbalasubramanian@linkedin.com

ABSTRACT

Nearest neighbor search (NNS) has a wide range of applications in information retrieval, computer vision, machine learning, databases, and other areas. Existing state-of-the-art algorithm for nearest neighbor search, Hierarchical Navigable Small World Networks (HNSW), is unable to scale to large datasets of 100M records in high dimensions. In this paper, we propose LANNS, an end-to-end platform for Approximate Nearest Neighbor Search, which scales for web-scale datasets. Library for Large Scale Approximate Nearest Neighbor Search (LANNS) is deployed in multiple production systems for identifying top-K ($100 \leq k \leq 200$) approximate nearest neighbors with a latency of a few milliseconds per query, high throughput of ~2.5k Queries Per Second (QPS) on a single node, on large (e.g., ~ 180M data points) high dimensional (50-2048 dimensional) datasets.

PVLDB Reference Format:
Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. LANNS: A Web-Scale Approximate Nearest Neighbor Lookup System. PVLDB, 15(4): 850 - 858, 2022.
doi:10.14778/3503585.3503594

1 INTRODUCTION

Nearest-neighbor search (NNS) is an effective technique for information retrieval and several machine learning applications. Despite its simplicity and wide-ranging utility, efficiently building and serving k-nearest neighbor data structures to web-scale has remained a challenge. In this paper, we describe LANNS (Large Scale Approximate Nearest Neighbor Search), a system designed and deployed in a web-scale environment at LinkedIn. LANNS has been deployed in a production environment for identifying top-K (with k ranging from 100-200) approximate nearest neighbors with very low latency (few milliseconds per query), very high throughput (roughly 2.5K Queries Per Second (QPS) on a single node), on large (e.g., 180M data points) high dimensional (e.g., 128, 256, or 2048 dimensional) data sets.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503594

Broadly, nearest neighbor search approaches fall into four categories. They can be tree-based[6, 9, 21, 25], product quantization-based[11, 12, 16, 17], Locality Sensitive Hashing (LSH) based[1, 2, 5, 13, 29], or graph-based[10, 14, 19, 20]. Most of the scalable methods return approximate nearest neighbors (i.e., miss out on some of the k-nearest neighbors in the results) in order to speed up the search. The recall, measured as the fraction of true k-nearest neighbors returned in a result set of size k, is generally traded off for the query latency or throughput. Figure 1[4], shows such a compromise between various state of the art algorithms (Annoy[25], BallTree[7], Faiss-IVF[15, 16], FLANN[24], Hierarchical Navigable Small World (HNSW) graph[20], KGraph[10], PANNG[14], PyNNDescent[22] and SWGraph [19]) on the SIFT1M dataset. It is evident from Figure 1, and other offline benchmarks conducted by us, that HNSW tends to outperform competitors considering QPS vs recall tradeoff.

We have used HNSW as the core approximate nearest neighbor (ANN) algorithm. However, LANNS has been built to be extensible to support other ANN algorithms with a bounded drop in recall.

Despite the favorable performance characteristics and popularity of HNSW[20], building the HNSW data structure does not scale well for production system with large, high dimensional datasets. For example, building the HNSW index on a real dataset of size $2.7M$ with 256 dimensions takes about **2 hours 20 minutes** on a single machine. At LinkedIn, we often have to serve k-NN queries on datasets containing **100M-500M** records with dimensionality of **50-2048**. This renders the default single-machine HNSW index build methods impractical. Furthermore, beyond a certain index size, procurement and maintenance cost of high memory servers also increases compared to commodity hardware. It is therefore necessary to be able to split up the dataset into multiple shards.

In this paper, we present LANNS, our end-to-end platform currently in production at LinkedIn, which enables web-scale nearest neighbor search in a variety of applications. As part of LANNS, we propose a two-level data partitioning strategy that allows us to scale the HNSW algorithm to web-scale datasets at index build time, as well as for online serving. We show that using this parallel building of separate HNSW indices, one for each data partition, and flexible data segmentation, we achieve fast index build and online serving. Our proposed data segmentation techniques also bound the drop in recall as compared to the HNSW algorithm. These segmentation techniques have theoretical guarantees on the recall as a function of the tuneable partitioning parameters that are on similar lines as [9]. We demonstrate the empirical performance of our proposed strategy on two open-source and four real-world datasets.

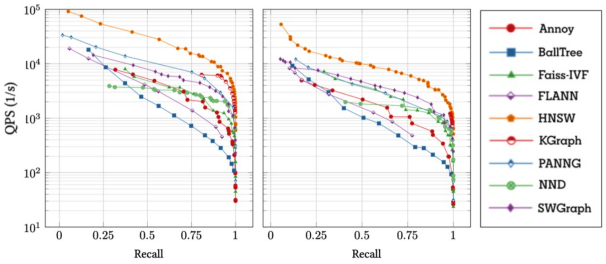


Figure 1: Recall v/s QPS on SIFT1M. Left: 10 nearest neighbors, Right: 100 nearest neighbors

1.1 Our Contributions

Our contributions in this paper are as follows:

(1) We propose a two-level data partitioning strategy that allows us to scale HNSW indexing to web-scale datasets.
(2) We propose a flexible data segmentation framework within each partition which allows further scaling. We propose two segmentation strategies with guarantees for a bounded drop in recall as a function of data size.
(3) We show, through extensive benchmarking, that for a majority of queries, high recall is achieved while querying only one or a few segments, and our partitioning and segmentation framework performs and scales well.
(4) We demonstrate the performance of our end-to-end system on various open-source and real datasets and show its favorable scalability properties.

The paper is organized as follows. We discuss related work in Section 2. In Section 3, we motivate two-level data partitioning, followed by the partitioning strategies. In Section 4, we describe the Spark framework for LANNS. In Section 5, we present our experiments on open source and real datasets, with a brief discussion on design choices. We present our online framework in Section 6, and conclude and discuss future work in Section 7.

2 RELATED WORK

In this section, we will discuss techniques and algorithms used for nearest neighbor search, as well as some works similar to LANNS.

Locality Sensitive Hashing (LSH)[13] – LSH is a hashing based technique where points are assigned to buckets such that, with high probability, similar points are found within the same bucket, while points far from each other are likely to be in different buckets. Variants of LSH can be data dependent[1, 2, 5, 29] or data independent. This method builds the index in linear time and has good theoretical guarantees of sub-linear query time, however, for adversarial data, this algorithm might run as slow as a linear scan.

Tree-based methods[9, 21, 25] – These tree based methods build one or a set of trees by recursively splitting the dataset. In [25], a set of trees are built where each tree is constructed by picking two points at random and splitting the dataset using a hyperplane separating the two points. In [21] the authors propose Approximate Principal Direction Trees, which recursively splits the data points using approximate eigenvectors. They claim that the method reduces average diameter at the same rate as PCA Trees [28] with lower runtime. In [9], a tree is built by randomly partitioning the data using random hyperplanes. They also propose spills, i.e., route data points or queries to multiple partitions based on their distance to the splitting hyperplane. These algorithms give low recall when queries are near the boundaries of splitting hyperplanes.

Product Quantization (PQ)[11, 12, 16, 17] – PQ is a compression based ANN search method. The main motivation behind PQ is to compress the space into a product of lower dimension spaces and to quantize each of these subspaces separately. The dataset is split into multiple smaller, tall datasets based on their dimensions, and each of these sub-datasets are clustered into k clusters. One advantage is the compression of datasets, which results in significant speedup. However, in this approach as well, exact nearest neighbors might lie in other clusters.

Sparsest Cut and Eigenvectors[27] – Sparsest cut aims to partition the vertices of a graph in a way that the weights of the edges cut during this partitioning are as small as possible. This is typically done by using the Laplacian of the adjacency matrix of the graph and using the second smallest eigenvector of the same. [27] shows that using the second smallest eigenvector of the Laplacian has some proven theoretical guarantees.

Hierarchical Navigable Small World(HNSW)[20] – A graph-based technique built on the idea of Small Worlds (SW). Suppose you build a hierarchy of SW graphs that separate links according to their lengths. At earlier stages of the search, you traverse long edges and zoom into a local minima for the query, and at later stages, you search the neighborhood of the local minima to find the nearest neighbors to the query. This method has the benefit of tuning parameters to adjust the accuracy v/s speed trade-off, and the space v/s speed trade-off. It has a polylogarithmic time complexity and is highly competitive on real-world datasets[4]. However, the HNSW indexing is not scalable for large datasets in production. We extend this work to scale to large datasets with an implementation in Apache Spark[30] and employing various techniques motivated by Random Projection Trees[9], Approximate Principal Direction Trees[21] and Sparsest Cuts[27].

Another related work, SONG[31] leverages GPUs to scale NNS. It might not always be feasible to provison GPUs for all practical use cases. With LANNS, we propose leveraging a shared Spark cluster for NNS. Another type of ANNS is where there is a trade-off between RAM and the query time or QPS[3, 26]. This may not be suitable for time sensitive (e.g., search) applications or use cases, where a decrease in QPS could cause a loss in trust of the users.

3 TWO-LEVEL PARTITIONING

In many real use cases, we often require the algorithm to scale to datasets of size 100M-500M. The state-of-the-art, HNSW algorithm, takes about 2 hours 20 minutes to index a dataset of 2.7M records. This becomes infeasible in real-world scenarios. Often these indices are used in production systems that do not have the capacity to support such large datasets.

We propose a horizontal, two-level partitioning of the data such that each partition represents a subset of the dataset. We attain acceptable indexing times by building a separate HNSW index within each partition, and we host one, or a few, partitions on each online server node. The two-levels of partitioning, `sharding` and

segmentation, are two different dimensions that help in solving two aspects of the scaling problem.

3.1 Sharding

Sharding, our first level of partitioning, is necessary for a very large dataset where the memory requirements for keeping the entire dataset is large and cannot be accommodated in a single node. For example, a dataset with 50M records in a 500-dimensional space would require approximately **93G** storage. In addition to this 93G, we also need to consider the memory requirements of building a graph. Assuming that the total storage required would be about 128G, an index of this size would not fit in a production node with a standard memory configuration of 64G. Building customized production nodes with higher memory is not feasible since the cost per GB increases super-linearly with total machine memory due to the higher cost of compatible components, higher failure rates, etc. Thus, we propose our first level of partitioning as sharding. When a point is inserted, it is hashed to **one** particular shard using the key of the data point. This partitioning does not exploit any locality information and each query is routed to **all** shards of the LANNS index. The response is generated by merging all shard level candidates and picking the topK best candidates.

Sharding allows us to scale horizontally by partitioning the dataset. Each shard is hosted on a separate server node, which in turn enables us to keep the memory requirement of a single server node under control, and allows us to use standard configuration server nodes with 64G memory.

Let us consider a use case where the server node has enough memory but the indexing time is unacceptable. In such scenarios, with only one level of partitioning, we would create more shards. There is additional merge cost involved at the master/broker or the system which makes calls to the shards. Higher the number of shards, higher is the merge cost. The master, a system with low memory of 2G-4G, would need to merge the results from these shards and give the final topK responses. Considering the build time, one may use a large number of shards which could mean an increased merge cost, and possibly higher memory for use cases with a large number of shards. Having large number of shards also comes with an additional undesirable operational cost of maintaining a large number of systems in production, and increased hardware footprint in terms of the cluster (collection of server nodes) size.

3.2 Segmentation

Segmentation, our second level of partitioning aims at reducing the disadvantages of sharding. Each shard is further split into smaller partitions called segments. This segmentation can be done using same techniques as sharding, or smarter segmentation techniques that allow queries to be routed to one or only a few segments. Routing to a single segment during query retrieval may have a negative impact on the topK recall, but smarter segmentation strategies can be employed to keep this impact bounded. We propose two "smarter" segmentation strategies learnt using the indexing data in Section 3.3. Employing these same techniques in sharding becomes complicated as the online service employs an external broker in front of the shards, which are not co-located.

Another added advantage of segmentation is that segmentation provides the same scalability as sharding for offline ingestion[1], which is useful for cases where the dataset is small enough to fit into a single server node, but it is large enough to render the HNSW indexing time unacceptable. This helps to avoid setting up a multi-sharded setup till the time the dataset becomes large enough. For a large dataset, this also enables us to keep the number of shards under control. Each partition, i.e., each segment is built separately and in parallel. Thus, segmentation does not hamper the scalability of indexing. Since multiple segments are hosted within the same server node, it also reduces the online hardware footprint.

As mentioned earlier, routing to one or a few segments can cause a drop in the recall. We propose segmentation techniques that bounds this drop in the recall. Another point to note is that in cases where a query is routed to multiple segments, there is an additional merge cost. For our online serving systems, this merge happens within the shard and does not require additional network I/O to send results from each shard to the broker node.

3.3 Segmentation Strategies

In this section, we describe three types of segmenters, the Random Segmenter (RS), Random Hyperplane Segmenter (RH), and Approximate Principal Direction Hyperplane Segmenters (APD). RS is a data-independent segmenter, whereas RH and APD segmenters are data-dependent.

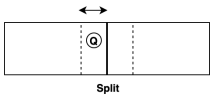
3.3.1 Random Segmenter (RS). In this particular segmenter, no type of learning from data is required. At indexing time, for each document, it randomly selects a segment where it should be routed. Since this type of segmenter has no guarantees about the locality of the data, a query vector would be routed to **all** segments.

3.3.2 Random Hyperplane Segmenters(RH). Random Hyperplane Segmenters, motivated by Randomized Projection Trees[9], builds a short tree of hyperplanes. The motivation behind this work is the following– if two points are similar, they would be close in the space, and it is highly unlikely that a randomly chosen hyperplane would split the two. However, if two points are far, there would be a high probability that the two points would be split. This enforces a sense of locality. With high probability, points similar to each other would lie in the same partition. We exploit this intuition to design our segmenters as– at each internal node of our segmenter, we first generate a random hyperplane from the unit sphere and project all points on this generated hyperplane. We then perform a median split based on these projected values.

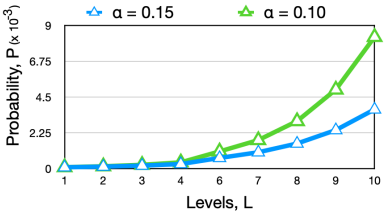
However, with a low probability, this method faces the problem of missing nearest neighbors that lie across the boundary in the other partition. We employ the method of "virtual" spill, where we maintain a left and right boundary around the splitting point. When a query point arrives and it lies within these left and right boundaries, we route the queries to both partitions. [2]

We briefly describe the insertion and querying process and state the theoretical bounds provided in [9] which are directly applicable to RH segmenter. Let the dataset be represented by a matrix $\mathcal{D}$ of

[1] It is worthwhile to note that indexing is done offline for online serving as well.
[2] Note that instead of using a virtual spill, we can also perform data side spill during ingestion, where data points lying within the left and right boundaries are routed to both partitions.



(a) Virtual Spill when query point (Q) is near the splitting hyperplane



(b) Probability of failure with increasing depth

Figure 2: Spills and failure probability of Segmentation

size $n \times d$, where n is the number of points and d is the number of dimensions, and α be the amount of spill. Let $x.h$ refer to the projection of x on h, and U denote the n dimensional vector of projections, $U = \mathcal{D}.h$. For insertion of a point x, if $x.h < median(U)$ route to the left partition, else route to the right. For query of a point q, let $l = 0.5 - \alpha$ fractile point in U, and $r = 0.5 + \alpha$ fractile point in U. If $q.h < l$, route to the left, if $q.h > r$, route to the right, else route to both sides. .

Definition 3.1. For query q, data points $x_1, \ldots x_n$, let $x_{(1)}, \ldots, x_{(n)}$ denote the reordering of points by increasing distance from q. Let us consider the potential function for the k-NN.

$$\Phi_{k,m}(q, x_1, \ldots x_n) = \frac{1}{m} \sum_{i=k+1}^{n} \frac{\sum_{j=1}^{k} ||q - x_{(j)}||/k}{||q - x_{(i)}||} \tag{1}$$

Theorem 3.2. *From [9], Suppose we build a tree on data points $x_1, \ldots x_n$ of depth $\mathcal{L}$, with α spill. If this tree is used to find the nearest neighbors of query q, then the probability that it fails to return $x_{(1)}, \ldots, x_{(k)}$ is*

$$\frac{k}{\alpha} \sum_{i=0}^{\mathcal{L}} \Phi_{(k,(0.5+\alpha)^i n)}(q, x_1, \ldots, x_n) \tag{2}$$

As we increase the depth of the tree, the number of hyperplanes used also increases. As more hyperplanes are used, there is a higher probability of two close points being separated by the segmentation algorithm. In Figure 2b, we approximate the probability that our segmentation algorithm fails to return $x_(1)$.

For the ease of demonstration, let $n = 10000$. As we use more levels, there is a higher probability of missing out on the exact nearest neighbor. Note that, in practice, we only a few levels of segmentation with about 1-8 segments per shard. Inside each of the leaves, we build an HNSW graph which is known to give high recall[4].

3.3.3 Approximate Principal Direction Hyperplane Segmenters(APD). Approximate Principal Hyperplane Segmenters, are motivated by Approximate Principal Directions[21] and Spectral Clustering[27]. Since we would like to minimize the number of queries being routed to multiple segments, we propose using a spectral clustering instead of random hyperplanes. To speed up the process, we also make use a core principle from APD Trees[21]– with a few steps of the power iteration, one can get reasonably close to the eigenvector.

Let the dataset be denoted by $\mathcal{D}$ of dimensions $n \times d$. Let $A_{n \times n}$ denote the adjacency matrix of a similarity graph, G constructed on $\mathcal{D}$. Let D be the degree matrix of A such that $D_{ii} = \sum_j A_i j$, and $C = D^{-1/2} A D^{-1/2}$. It is well-known that the largest eigenvalue of C is 1, and the second-largest eigenvalue and the corresponding eigenvector approximate the sparsest cut [27]. However, for large datasets, it is difficult to compute to the matrices A and C since they are of the order $O(n \times n)$. Along with these restrictions, we also have the added requirement of having a "queryable" hyperplane which not only partitions the data, but allows us to route a new point (query) to the right partition.

To make this method work in practice, we assume $A = \mathcal{D}\mathcal{D}^T$ and $\mathcal{D}$ is almost regular, which allows us to apply the Cheeger inequality described above. The second-largest eigenvector of A can be found using the second largest left singular vector of $\mathcal{D}$. Since $\mathcal{D} = U \Sigma V^T$ where U and V are the left and right singular vectors, we approximate the right singular vectors, as $U = \mathcal{D}.V$. Thus, we use h which is the second-largest right singular vectors of $\mathcal{D}$, and let $u = \mathcal{D}.h$.

This method also has the drawback of near points being across the splitting hyperplane. Again, we employ methods of spill, insertion, and querying as described in Section 3.3.2. Note that the theoretical guarantees from Section 3.3.2 are also applicable to the APD Segmenters. This bound is loose since APD is a data-dependent partitioning technique which boosts the performance in practice.

Consider a LANNS system which leverages these strategies, and Theorem 3.2, we can state the following.

Corollary 3.3. *Suppose we build a tree of depth $\mathcal{L}$, with α spill, and the leaves of this tree are segments of the LANNS index. Let method $\mathcal{A}$ be used to perform an ANN-search within each segment. If for a query q, $\mathcal{A}$ fails to return $x_{(1)}, \ldots, x_{(k)}$ with probability p_A, then LANNS fails to return $x_{(1)}, \ldots, x_{(k)}$ with probability*

$$\frac{k p_A}{\alpha} \sum_{i=0}^{\mathcal{L}} \Phi_{(k,(0.5+\alpha)^i n)}(q, x_1, \ldots, x_n) \tag{3}$$

4 OFFLINE FRAMEWORK

In this section, we describe the various components of LANNS. We propose pre-learning our learnable segmenters and feeding them as input to the indexing algorithm. The indexing algorithm stores the index on HDFS which can be fed into the querying algorithm, or can be exported to an online serving system (see Section 6).

4.1 Learning a Segmenter

Since the data distribution in our shards is uniform, we propose to pre-learn a segmenter and employ the same segmenter across all shards. This has a two-fold advantage– (i) avoiding unnecessary computations to learn a segmenter for each shard on the fly; (ii) storing segmenters for each shard in the offline system. Since the segmenter is shared, only one copy is stored. Given the input dataset, we subsample the dataset uniformly at random. This sampled dataset, say, $\mathcal{D}$, is fed to the segmenter learning algorithm, which is one of RH or APD. These techniques learn a tree of separating hyperplanes. At each internal node of this tree, a hyperplane is generated using RH or APD, that is used to further split the dataset into two partitions. For the APD Segmenter, we use the Spark Machine Learning library[23] implementation for distributed Singular Value Decomposition. Once this tree of hyperplanes is learnt, we

store the tree consisting of the hyperplane, the split points, and the left and right boundaries for each of the internal nodes. This learnt segmenter is fed to the ingestion algorithm.

4.2 Indexing

In Figure 3a, we show the process of scaling indexing for web-scale datasets. Along with the input dataset, we optionally input a pre-learnt segmenter that is loaded within each Spark executor and is used to generate the two-level partitioning. This pre-learnt segmenter is shared across shards. Each document is tagged with a shardID and one or more segmentIDs. The partition tagged dataset is repartitioned based on segmentID and shardIDs. One particular (shard, segment) pair is loaded in an executor and the HNSW Index is built on this subset. The HNSW Index is built inside an executor and hence, all HNSW indexing can happen in parallel. The serialized index inside each executor is stored in the HDFS from the executor itself and the associated metadata and segmenter information is coupled with the index and written from the driver.

4.3 Querying

For our offline use cases, it is of utmost importance to scale not only to big datasets but also to big query sets. To scale our query process, we make use of partitioned query sets. We demonstrate our process of querying with our two-level partitioned index in Figure 3b. We take a large query set and partition them into smaller batches, which are written to the HDFS. We also read the metadata of the index and prepare a '`SearchExecutorContext`' which informs each executor of which segment of which shard, and which query partition to load within it. This SearchExecutorContext is sent to the executors, the respective HNSW Indices and query partitions are loaded inside the executor. Partial search occurs inside each of these executors. The partial results go through a two-level merging as follows– in the first level, partial results are returned to the driver along with the shard and segmentIDs they come from. These partial results are repartitioned on the basis of the queryID and the shardID to perform a segment level merging to obtain shard results. The shard results are repartitioned again based on the queryID for the final merge. This is analogous to how merging would occur in an online system, where segment results would first get merged within the server node containing the shard. These merged shard level responses are further merged within the searcher master/broker node.

4.3.1 Preventing Time-Out Errors. Spark occasionally suffers from time-out errors which could prove to be catastrophic in some large-scale systems. Since a spark cluster is shared among many users and applications, there is heavy load in the cluster, and some nodes which are freed up while waiting for other tasks to finish. These get allocated to other applications, and "die" Consider a scenario where you have 100 *(query, shard, segment)*-partitions, and only 8 executors. After the partitioned search, before segment-level merging is triggered, some executors die and become unreachable. In these scenarios, the results become unavailable and search for those partitions is restarted. While waiting for these recomputed results, some other executors may die, and so on. This leads to cascading failures which may cause catastrophic damages for applications. This is what we refer to as "time-out" errors. In order to prevent such scenarios from happening, we write partial results to a temporary path on the HDFS. After searching and the first phase of merging, the results are written to a temporary path on the HDFS and are loaded from the temporary path for further processing. As soon as our two-level merging finishes, this temporary directory is cleaned. This works well since Spark ensures that for write operations, as soon as an executor finishes processing its task, instead of waiting for other executors to finish execution, it can write to the HDFS. This is in contrast to the repartitioning where the executor keeps the results and waits for all tasks of the stage to finish executing.

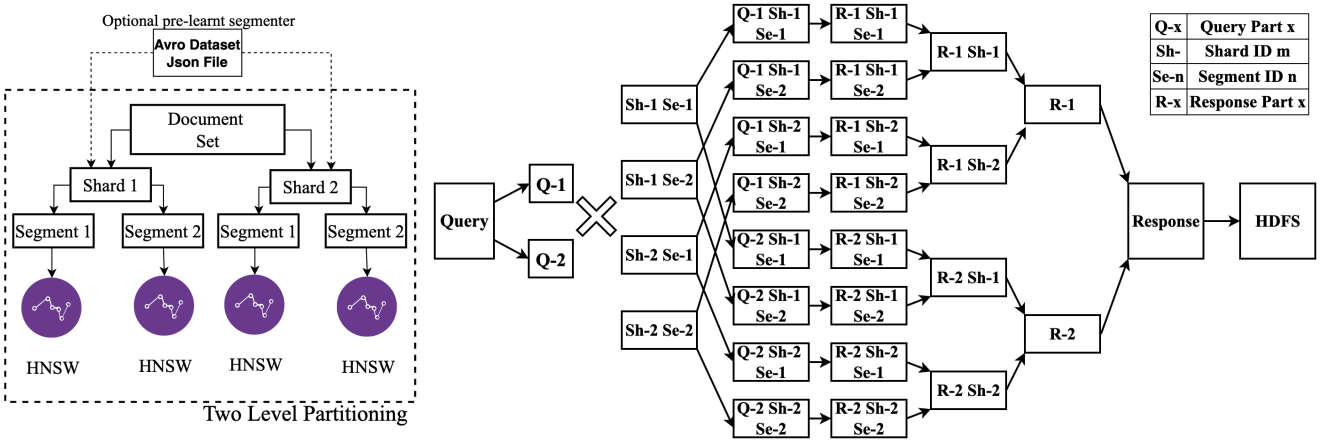
4.3.2 Per shard TopK. Some recommender systems require searching for a very large number of nearest neighbors, of the order of 1000s, with further post-processing to prune candidates. Sending the same "k" or "topK" to each shard can prove to be wasteful since each shard would then return topK responses. These topK responses would use up network I/O bandwidth and also increase the merge cost at the searcher or broker. In order to avoid such cases, we employ a "perShardTopK", which uses the Normal Approximation Interval[8] to reduce the number of nearest neighbors fetched from each randomly partitioned shard. Let S be the number of shards, and p be the confidence (or topK.confidence), and $s' = \frac{1}{S}$, and $cI = s' + f(p) * \sqrt{\frac{s'(1-s')}{topK}}$ then, $perShardTopK = min(topK, \lceil cI * topK \rceil)$, where $f(p)$ is the $(1 - p/2)$ quantile of the standard normal distribution (the probit).

Note that since hyperplane based segmenters may query only a few segments, it is undesirable to apply the concept of a "per segment topK". Employing a per segment topK could lead to fewer than topK results as the final output. Thus, we do not optimize the topK for segments, instead we propagate the shard level perShardTopK to the associated segments. In the online system, querying using the perShardTopK at the segment level does not hamper the network I/O drastically. The merging occurs within a node and only a final perShardTopK are sent over the network. While c-ANNS[18] may also be applied to reduce the number of results, this may not be desirable in all use cases. Some applications may need to generate candidates when the query point is very far from all indexed data (i.e., recommendations for new or inactive users).

5 EXPERIMENTS

5.1 Open Source Evaluation

For our evaluations on open-source data, we use two datasets– (i) `SIFT1M`, the `SIFT1M` dataset with the indexing dataset of $1M$ records and a query set of $10k$ records. Each of these has dimension, $d = 128$, and (ii) `GIST1M`, the `GIST1M` dataset with the indexing dataset of $1M$ records and query set of $1k$ records. Each of these have dimension, $d = 960$. For both these datasets, we consider $topK = 100$ nearest neighbors with the distance function to be Euclidean Distance. For both, `SIFT1M` and `GIST1M`, we compare our performance with the HNSW algorithm. We build (n, m)-partitioned indices, where n is the number of shards and m is the number of segments, using Random Segmenters (RS), Random Hyperplane Segmenters (RH), and APD Segmenters (APD). For the `SIFT1M` dataset, we experiment with $(1, 8)$-partitioned and $(2, 4)$-partitioned indices. For the `GIST1M` dataset, we limit ourselves to $(1, 8)$-partitioned indices. For all experiments, $\alpha = 0.15$, i.e., we route about 30% queries to both partitions at any level. We set the $topK.confidence = 0.95$ to limit



(a) Indexing multi-million datasets

(b) Querying with large query sets on multi-million datasets

Figure 3: Indexing and Querying within LANNS

the number of results obtained from each shard. For all experiments, the building times, query times, and recall are averaged over 5 runs.

5.1.1 Results. In Tables 1,2 and 3, we present the Recall, Build Times and Query Times comparisons with the HNSW algorithm for SIFT1M and GIST1M datasets. For Build and Query Times, we vary the number of executors for our LANNS indices. For both datasets, we observe a $\sim 4.5\times - 5\times$ speed-up in build time using 2 executors, and a $\sim 10\times - 11\times$ speed-up in build time using 8 executors. For the RS segmenter, we see comparable query times with respect to HNSW and 2 executors. However, we see a speedup of $2\times - 2.5\times$ when we increase the number of executors to 8. This comes with comparable recall. With RH segmenter for both datasets, we see a significant drop of about $\sim 15\%$ in recall for $(1, 8)$-partitioning, and this comes with a speed up of $\sim 2\times - 2.5\times$ on the query time using 2 executors, and $\sim 3\times - 4\times$ using 8 executors. For the APD segmenter, for SIFT1M, we observe a loss of 2% in recall with a $(1, 8)$-partitioning and a 1% drop $(2, 4)$-partitioning. For GIST1M, we observe 7% loss in recall with the $(1, 8)$-partitioning. However, for both datasets, the $(1, 8)$-partitioning this comes with a $\sim 2\times - 3\times$ speedup in query time with 2 executors and $\sim 5\times$ speedup in query time with 8 executors.It is worthwhile to note that while both datasets contain 1M points, GIST1M has a lower recall with APD. This can be attributed to the difference in dimensionality, SIFT1M is in 128 dimensions while GIST1M is in 960 dimensions. Theorem 3.2 and Theorem 3.3 also indicates that higher dimensionality and a deeper segmentation tree leads to a higher loss in recall.

Build times do not change across $(1, 8)$-partitioning and the $(2, 4)$-partitioning and across segmenters. This is because we pre-learn the segmenters and feed them to the ingestion setup. While RS doesn't require any pre-learning, for the SIFT1M dataset, RH segmenter takes 2.1 minutes and 1.8 minutes for $(1, 8)$-partitioning and $(2, 4)$-partitioning respectively on a subsample of 250k data points, and APD segmenter takes 3 minutes and 2.6 minutes for $(1, 8)$-partitioning and $(2, 4)$-partitioning respectively on a subsample of 250k data points. For the GIST1M dataset, RH segmenter takes 6.3 minutes on a subsample of 250k data points, and APD segmenter takes 18 minutes on a subsample of 250k data points. For all segmenter learning on GIST1M, we use 30 executors.

5.2 Real-World Datasets

We use four large-scale datasets for real-world use cases:

(1) Groups Search: Groups is a dataset of $\sim$ 2.7M groups on LinkedIn. Each group is embedded in a 256-dimensional space. We evaluate the offline performance on 10k queries.
(2) People You May Know: (PYMK) is a database of 100M users of the platform. Each record is an embedding in 50 dimensions. We evaluate the offline recall performance on a subset of 1M queries, and offline query latency on 372M queries.
(3) People Search: (People) is a database of 180M users of the LinkedIn platform, represented as an embedding in 50 dimensions. This use case leverages LANNS for people search. We evaluate the offline performance on 20k queries.
(4) Near-Duplicates: (NearDupe) consists of embeddings of images posted on the LinkedIn feed. The training set has 148k records in 2048 dimensions with a query set of 500k records.

Groups, People and NearDupe use-cases were first tested using our offline platform and then onboarded to the online platform (See Section 6). PYMK use case, one of our biggest use-cases employs our offline platform for an in-production system.

We first provide benchmarking results on the Groups dataset. We evaluate the following alternatives– (i) Physical Spill: A data point close to the splitting plane is routed to both children or segments, (ii) Virtual Spill: A query close to the splitting plane is routed to both children or segments. The physical spill uses data side duplication and uses a higher memory footprint as compared to the virtual spill. However, the Queries per Second (QPS) in case of a physical spill is slightly higher than the virtual spill since the query is routed to only one segment in case of a physical spill. The results with both types of spill are presented in Table 4. For both of these, we see that the recall values are comparable with only a slight difference in the QPS values. For virtual spills, since queries are being routed to multiple segments, the number of unique queries that can be

Table 1: Recall for SIFT1M and GIST1M datasets. R@k refers to Recall at topK = k. Method suffix (n, m) refers to (n, m)-partitioning.

	SIFT1M						GIST1M					
Method	**R@1**	**R@5**	**R@10**	**R@15**	**R@50**	**R@100**	**R@1**	**R@5**	**R@10**	**R@15**	**R@50**	**R@100**
HNSW	**0.991**	**0.996**	**0.997**	**0.998**	**0.998**	**0.998**	0.994	0.995	0.995	0.995	0.993	0.989
RS(1,8)	0.979	0.986	0.986	0.986	0.98	0.98	**0.995**	**0.998**	**0.999**	**0.999**	**0.999**	**0.999**
RH(1,8)	0.841	0.818	0.804	0.798	0.776	0.762	0.872	0.858	0.851	0.843	0.827	0.812
APD(1,8)	0.977	0.977	0.975	0.973	0.966	0.961	0.931	0.919	0.912	0.91	0.908	0.905
RS(2,4)	**0.989**	**0.994**	**0.995**	**0.995**	**0.996**	**0.996**	-	-	-	-	-	-
RH(2,4)	0.916	0.913	0.906	0.903	0.892	0.885	-	-	-	-	-	-
APD(2,4)	**0.989**	**0.995**	**0.994**	**0.994**	**0.992**	**0.991**	-	-	-	-	-	-

Table 2: Build times for SIFT1M and GIST1M datasets with varying number of executors ($\mathcal{E}$). Time is in minutes for total $1M$ data points.

	SIFT1M				GIST1M			
$\mathcal{E}$	**HNSW**	**RS**	**RH**	**APD**	**HNSW**	**RS**	**RH**	**APD**
2	40	8.2	8.1	8.4	577	132	128	140
4	-	6.6	6.8	6.3	-	96	108	106
8	-	**4.3**	**4.4**	**4.1**	-	**48**	**54**	**52**

Table 3: Query times for SIFT1M and GIST1M dataset with varying number of executors ($\mathcal{E}$). Time is in milliseconds per query for total $10k$ query points for SIFT1M and $1k$ query points for GIST1M.

	SIFT1M							GIST1M			
		(1,8)-partitioning			(2,4)-partitioning				(1,8)-partitioning		
$\mathcal{E}$	**HNSW**	**RS**	**RH**	**APD**	**RS**	**RH**	**APD**	**HNSW**	**RS**	**RH**	**APD**
2	50	58	21	16	49	46	44	336	330	156	144
4	-	46	16	12	38	25	25	-	222	132	108
8	-	25	13	**10**	33	**17**	**17**	-	132	96	**66**

served is lower, leading to slight degradation in the QPS. Since LANNS indices are used in production systems, it is undesirable to have a high memory footprint. Employing physical spills for large datasets such as PYMK would increase the memory footprint by 30% (30M records), which increases the number of server nodes required. Thus, we use virtual spill with $\alpha = 0.15$.

In Table 5, we present our building and querying times[3][4] for our real-world datasets. We include the improvements in the build time for the dataset mentioned in the Introduction. For the Groups dataset, a (2,2)-partitioned index reduces build time to 38 minutes. For People and the PYMK use cases, owing to the large size of the data, it is infeasible to compare with HNSW. For the NearDupe use cases, we essentially use the HNSW index with distributed querying. We present results on real datasets with the parameters reflecting the optimal trade-off for our in-production services. We also present our recall evaluations in Table 6. For each dataset, we obtain a high recall of over 95%, evaluated on query sets of

[3]Note that these times are inclusive of the times required for requesting a cluster and assigning executors.

[4]The building and querying time presented here are averaged over 5 runs.

Table 4: Recall on Groups dataset with (1,m)-partitioning using APD Segmentation. R@k refers to the recall for k-nearest neighbors, *m* refers to the number of segments

		Physical Spill		Virtual Spill	
m	**Spill**	**R@15**	**QPS**	**R@15**	**QPS**
1	**0%**	**0.9458**	**863.29**	**0.9458**	**863.29**
4	10%	0.8400	2619.02	0.8526	2186.93
4	20%	0.8861	2432.23	0.8853	2010.44
4	**30%**	**0.9268**	**2392.42**	**0.9272**	**1984.21**
8	**30%**	**0.9105**	**2710.24**	**0.9112**	**2573**
16	**30%**	**0.8836**	**2797.42**	**0.892**	**2985.34**

Table 5: Build and Query Times for Real-World datsets. dim refers to dimensions, n to number of Shards, and m to number of Segments.

				Indexing		Querying	
Dataset	**n**	**m**	**dim**	**Size**	**Time**	**Size**	**Time**
PYMK	20	1	50	100M	480m	370M	10h
People	32	1	50	180M	520m	20k	10m
NearDupe	1	1	2048	148k	80m	500k	5m
Groups	1	1	256	2.7M	133m	20k	7m
Groups	2	2	256	2.7M	38m	20k	3m

Table 6: Recall for Real-World datasets. dim refers to dimensions, and R@K refers to the Recall at topK = K.

Dataset	**n**	**dim**	**Index Size**	**Query Size**	**K**	**R@K**
People	32	50	180M	20k	50	97%
PYMK	20	50	100M	1M	100	95%
NearDupe	1	2048	148k	0.5M	100	97%
Groups	1	256	2.7M	20k	100	96.9%

reasonable sizes. For large datasets, we employ an in-house Spark implementation of distributed brute-force search.

5.3 Choice of Parameters

As demonstrated in Table 1, for both open source datasets, we observe that the RH segmenter has a significant loss in the recall as compared to the APD segmenter, with comparable query time. This comes with a trade-off on the time to learn a segmenter. RH might be a good fit for certain time sensitive applications with highly dynamic datasets where a trade-off between indexing time

and recall is acceptable. For use cases where the dataset is not very dynamic, APD allows a better trade-off with respect to indexing time and recall. Other factors to consider in these scenarios is the (n, m) configuration. While the spark indexing time is the same for $(1, 8)$ and $(2, 4)$, the choice can depend on a large number of factors. The $n = 1$ case requires only one node in the online setup, whereas, the $n = 2$ case would require two nodes and a broker node to merge the results. The query times for both cases would vary as well. For the $n = 2$ case, each query would perform ANN on two shards, and for $n = 1$, it would be on one shard, and recall loss for $m = 4$ v/s $m = 8$ varies with the dimensionality. Another interesting trade-off could be the physical v/s virtual spill. For cases where there are enough resources to allow for a physical spill, but the application is highly time sensitive and requires high parallelization of the queries, one may decide to use physical spill described in [9], and in Table 4. In this case, each query would be routed to only one segment, and queries which are routed to different segments can be served in parallel. In cases where data duplication across segments is not tolerable, virtual spill gives similar results with a small drop in QPS. For any use case, the choice would be dependent not only on the nature of the use case, the indexing time, query time, and recall quality but also on the dimensionality, the cost budget and available resources.

6 ONLINE SERVING

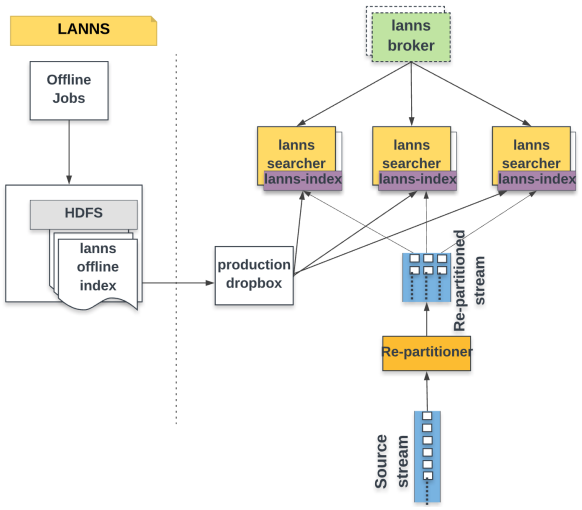


Figure 4: Online Service Architecture

Throughout the paper, we discuss the offline implementations evaluations of our proposed method for scaling HNSW builds. In this section, we will discuss our online service architecture, as shown in Figure 4. To enable nearest neighbor search capability in an online environment, we build the index offline on our Spark cluster and ship the serialized HNSW index (as Avro datasets) to online searcher nodes. The serialized index consists of the graph index, the embeddings (vectors) and additional metadata (like the segmenter, distance function used during index build, etc). The searcher nodes, when starting up, deserialize the index to native Java data-structures optimized for online serving using persisted metadata with minimal additional configuration. This ensures that the platform doesn't allow accidental differences in algorithm configuration between offline build and online serving. The majority of storage needed in the online node comes from the vector representations of entities in the index, the index itself is quite small. Fast lookup access to the embeddings for a document is critical for low latency online serving as most of the search time is spent on doing <query, document> distance comparisons. The difference in the online architecture is that each shard is hosted on a different node. The first stage of the two-step merging, i.e., the shard level merging, happens on the node where the shard is hosted (called a "searcher"), and the final merge happens at the broker or client. The broker is also responsible for calculating and passing the *perShardTopK* to each shard. We also built additional constructs to support use cases hosting different indices in the same searcher for online A/B tests between different embedding representations of the documents. For one of our large use cases with **180M documents** and embeddings of **dimension 128**, we benchmarked the online searcher to achieve a **2.5K QPS** at a **p99 latency of 20ms**.

Our production systems make use of both, online and offline approaches. Some use cases perform nearest neighbor search offline, using Spark, at a fixed cadence and send the results to online services for further processing. Three in-production use cases are hosted online with each shard hosted on a separate searcher node.

Offline v/s Online Serving – For applications with very large datasets and fixed query sets (for example, connections of members of a social network), we suggest offline search and sending results to online services. For very large datasets, the number of shards could be high and require several dedicated host machines, whereas, in offline mode, the processing is done once and a cluster could be shared with various applications. Thus, if the use case is not latency sensitive, or if precomputation of nearest neighbors or recommedations is feasible, it is preferred to use the offline service. Much of the cost of offline processing can be delegated to the offline grid infrastructure. However, this cannot be done for applications where the query set is dynamic and the results are required instantly. For cases such as a search application or nearline spam detection, the offline service would have delays and an unacceptable latency and the online service is the only option.

7 CONCLUSION

In this work, we propose LANNS, an end-to-end platform for Approximate Nearest Neighbor Search. We enable scaling HNSW to web-scale datasets through a two-level partitioning scheme using the Spark framework. We demonstrate the excellent empirical performance on LinkedIn's production use-cases, and through extensive offline evaluations on various datasets. We demonstrate our scalable and highly competitive performance. We also briefly discuss the design choices and the trade-off between using an offline pipeline or an online, deployable service.

As future work, our approach of using segments can be explored for other purposes as well. For example, for context-based searches, we can build a segment per context and perform search in one or a few segments based on the contexts selected at query time.

REFERENCES

[1] Alexandr Andoni, Assaf Naor, Aleksandar Nikolov, Ilya Razenshteyn, and Erik Waingarten. 2018. Data-Dependent Hashing via Nonlinear Spectral Gaps. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing* (Los Angeles, CA, USA) *(STOC 2018)*. Association for Computing Machinery, New York, NY, USA, 787–800. https://doi.org/10.1145/3188745.3188846

[2] Alexandr Andoni and Ilya Razenshteyn. 2015. Optimal Data-Dependent Hashing for Approximate Near Neighbors. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing* (Portland, Oregon, USA) *(STOC '15)*. Association for Computing Machinery, New York, NY, USA, 793–801. https://doi.org/10.1145/2746539.2746553

[3] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-Index: Pushing the Scalability-Accuracy Boundary for Approximate KNN Search in High-Dimensional Spaces. *Proc. VLDB Endow.* 11, 8 (April 2018), 906–919. https://doi.org/10.14778/3204028.3204034

[4] Martin Aumuller, Erik Bernhardsson, and Alexander John Faithfull. 2018. ANN benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *CoRR* abs / 1807.05614 (2018), 1–20. arXiv:1807.05614 http://arxiv.org/abs/1807.05614

[5] X. Bai, H. Yang, J. Zhou, P. Ren, and J. Cheng. 2014. Data-Dependent Hashing Based on p-Stable Distribution. *IEEE Transactions on Image Processing* 23, 12 (2014), 5033–5046.

[6] Jon Louis Bentley. 1990. K-d Trees for Semidynamic Point Sets. In *Proceedings of the Sixth Annual Symposium on Computational Geometry* (Berkley, California, USA) *(SCG '90)*. Association for Computing Machinery, New York, NY, USA, 187–197. https://doi.org/10.1145/98524.98564

[7] Leonid Boytsov and Bilegsaikhan Naidan. 2013. Engineering Efficient and Effective Non-metric Space Library. In *Similarity Search and Applications*, Nieves Brisaboa, Oscar Pedreira, and Pavel Zezula (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 280–293.

[8] Lawrence D. Brown, T. Tony Cai, and Anirban DasGupta. 2001. Interval Estimation for a Binomial Proportion. *Statist. Sci.* 16, 2 (2001), 101–117. http://www.jstor.org/stable/2676784

[9] Sanjoy Dasgupta and Kaushik Sinha. 2015. Randomized Partition Trees for Nearest Neighbor Search. *Algorithmica* 72, 1 (2015), 237–263. https://doi.org/10.1007/s00453-014-9885-5

[10] W. Dong. 2014. *KGraph*. Retrieved 24 June, 2021 from https://github.com/aaalgo/kgraph

[11] T. Ge, K. He, Q. Ke, and J. Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, USA, 2946–2953.

[12] J. Heo, Z. Lin, and S. Yoon. 2014. Distance Encoded Product Quantization. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, USA, 2139–2146.

[13] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[14] Masajiro Iwasaki. 2016. Pruned Bi-directed K-nearest Neighbor Graph for Proximity Search. In *Similarity Search and Applications*, Laurent Amsaleg, Michael E. Houle, and Erich Schubert (Eds.). Springer International Publishing, Cham, 20–33.

[15] J. Johnson, M. Douze, and H. Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7 (2019), 535 – 547. Issue 3.

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

[17] Yannis Kalantidis and Yannis Avrithis. 2014. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, USA, 2321–2328.

[18] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. 2020. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. *Proc. VLDB Endow.* 13, 9 (May 2020), 1443–1455. https://doi.org/10.14778/3397230.3397240

[19] Yu Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2013. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (01 2013), 61–68. https://doi.org/10.1016/j.is.2013.10.006

[20] Y. A. Malkov and D. A. Yashunin. 2016. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2016), 824–836.

[21] Mark McCartin-Lim, Andrew McGregor, and Rui Wang. 2012. Approximate Principal Direction Trees. In *Proceedings of the 29th International Coference on International Conference on Machine Learning* (Edinburgh, Scotland) *(ICML'12)*. Omnipress, Madison, WI, USA, 1611–1618.

[22] L. McInnes. 2018. *PyNNDescent*. Retrieved 20 June, 2021 from https://github.com/lmcinnes/pynndescent

[23] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016), 1235–1241.

[24] Marius Muja and David G. Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *VISAPP 2009 - Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, Lisboa, Portugal, February 5-8, 2009 - Volume 1*, Alpesh Ranchordas and Helder Araújo (Eds.). INSTICC Press, Portugal, 331–340.

[25] Spotify. 2013. ANNOY library. https://github.com/spotify/annoy. Accessed: 2019-08-01.

[26] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. DiskANN: Fast Accurate Billion-Point Nearest Neighbor Search on a Single Node. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 1233, 11 pages.

[27] Luca Trevisan. 2013. Lecture notes on expansion, sparsest cut, and spectral graph theory.

[28] Nakul Verma, Samory Kpotufe, and Sanjoy Dasgupta. 2009. Which Spatial Partition Trees Are Adaptive to Intrinsic Dimension?. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence* (Montreal, Quebec, Canada) *(UAI '09)*. AUAI Press, Arlington, Virginia, USA, 565–574.

[29] Yair Weiss, Antonio Torralba, and Robert Fergus. 2008. Spectral Hashing. In *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*, Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou (Eds.). Curran Associates, Inc., Canada, 1753–1760. https://proceedings.neurips.cc/paper/2008/hash/d58072be2820e8682c0a27c0518e805e-Abstract.html

[30] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

[31] Weijie Zhao, Shulong Tan, and Ping Li. 2020. SONG: Approximate Nearest Neighbor Search on GPU. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, USA, 1033–1044. https://doi.org/10.1109/ICDE48307.2020.00094