DISTRIBUTEDANN: Efficient Scaling of a Single DISKANN Graph Across Thousands of Computers

Philip Adams [1] **Menghao Li** [2] **Shi Zhang** [1] **Li Tan** [1] **Qi Chen** [3 *] **Mingqin Li** [4 † *] **Zengzhong Li** [1 *] **Knut Risvik** [5 *] **Harsha Vardhan Simhadri** [1 *]

Abstract

We present DISTRIBUTEDANN, a distributed vector search service that makes it possible to search over a single 50 billion vector graph index spread across over a thousand machines that offers 26ms median query latency and processes over 100,000 queries per second. This is 6× more efficient than existing partitioning and routing strategies that route the vector query to a subset of partitions in a scale out vector search system. DISTRIBUTEDANN is built using two well-understood components: a distributed key-value store and an in-memory ANN index. DISTRIBUTEDANN has replaced conventional scale-out architectures for serving the Bing search engine, and we share our experience from making this transition.

1. Introduction

Approximate Nearest Neighbor (ANN) search is a common retrieval technique in web search, multimedia search, and new scenarios like Retrieval-Augmented Generation (Lewis et al., 2020). Given a set of data vectors X, and a query vector q, the goal of an ANN search system is to quickly find as many of the k vectors in X that are nearest to q as possible. ANN search is a classic problem and researchers have developed many techniques, including algorithmic approaches (Malkov & Yashunin, 2018; Andoni & Indyk, 2008; Subramanya et al., 2019; Wang., 2018; Babenko & Lempitsky, 2014), compression techniques (Jegou et al., 2010; Ge et al., 2014; Gao & Long, 2024), and specialized indexes targeting GPUs, flash storage, or external memory (Johnson et al., 2019; Ootomo et al., 2024; Wang, 2021; Jang et al., 2023). The performance of these techniques is closely tracked by benchmarking efforts like ANN-Benchmarks (Aumüller et al., 2018) or Big-ANN-Benchmarks (Big-ANN Benchmarks, 2023).

However, the majority of research effort has been focused on datasets small enough to fit in a single machine's memory or disk. For larger datasets, like searching over hundreds of billions of web documents, the standard approach is to split the corpus into smaller partitions[1]. An independent index is built for each partition, and each partition is hosted on a different machine so that the entire corpus can be searched in parallel. The main downside of this approach is efficiency: inside a single ANN index, query cost scales with $\log|X|$ (as empirically measured), while across many partitions (assuming a fixed partition size) the cost will scale with the number of partitions. In other words, a system with P partitions has a search complexity of $P \log \frac{|X|}{P}$, which is much worse than the complexity of one index over the entire dataset X. Even techniques aimed at improving this tradeoff, like using clustering to assign vectors to partitions (Wang, 2021), face scalability challenges and complex tradeoffs.

In this paper, we argue that state-of-the-art performance on very-large-scale datasets can be achieved through a single large logical index stored in a distributed key-value store. DISTRIBUTEDANN begins with the abstraction of the key-value store as a large shared disk, and then makes modifications to the data and compute placement choices of DISKANN indices in order to make it practical to serve them in this setting. We explore the tradeoffs compared to a traditional approach, and present experimental results demonstrating the scalability of this approach on a fifty-billion vector subset of a web search dataset. DISTRIBUTEDANN has already been deployed in Microsoft Bing to support search over hundreds of billions of vectors. Compared to the previous production system, it delivered over **6× headroom in query throughput** with the same machine footprint, and **7.8 and 4.5 percentage point im-**

[*]Listed alphabetically by surname [†]The work was done at Microsoft. [1]Microsoft, Redmond, United States [2]Microsoft, Bejing, China [3]Microsoft Research Asia, Vancouver, Canada [4]Shopify, Bellevue, United States [5]Microsoft, Trondheim, Norway. Correspondence to: Philip Adams <philipadams@microsoft.com>.

Proceedings of the 1st Workshop on Vector Databases at International Conference on Machine Learning, 2025. Copyright 2025 by the author(s).

[1]For online systems, it is often necessary to choose a partition size much smaller than the total space available on a machine, in order to be able to bring a new replica online quickly enough should a host fail.

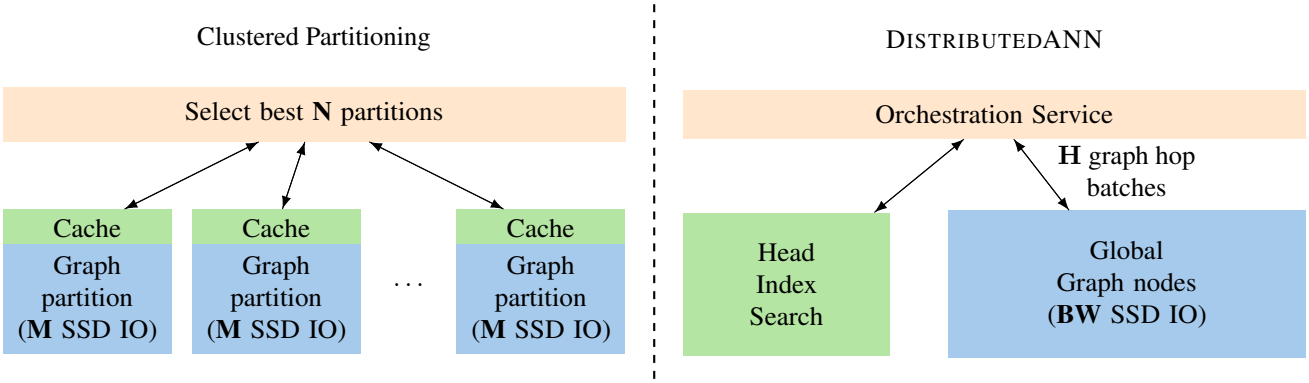

Figure 1. High-level architecture comparison between a conventional system using clustered partitioning and DISTRIBUTEDANN.

provements in recall@5 and recall@200 respectively.

2. DISTRIBUTEDANN

Existing ANN indices like DISKANN are optimized for tiered storage by reducing the number of round-trips to SSD. Therefore, DISKANN is well suited for adaptation to a distributed serving environment, since the main drawback of accessing a networked disk is added access latency. However, directly serving a multi-hundred-terabyte DISKANN graph from network-attached storage encounters new bottlenecks:

- The memory-resident portions of the index are too large to fit in a single machine, and accessing them over the network would incur significant overhead.
- As the diameter of the graph grows, more hops are needed to traverse it, increasing latency.
- Transmitting the full graph node data over the network requires too much serial compute per query and consumes excessive network bandwidth.

In the rest of this section, we first review the data layout of DISKANN, and then discuss how we adapt the index to overcome these scaling bottlenecks.

2.1. DISKANN Index Layout

A DISKANN index has 3 components:

1. An array of compressed (PQ or OPQ[2]) representations of all vectors in the index, stored in memory.
2. An array of graph nodes, one per vector, stored on SSD. Each node stores the full precision vector and a list of IDs representing out-neighbors.
3. An in-memory cache of frequently-accessed graph nodes.

Searching this index begins at a fixed point, and scores neighbor candidates of each visited node using their in-memory compressed representations to determine which nodes to visit in the next iteration of beam search. Depending on the index size and the recall required, tens to hundreds of nodes may be read (depending on I, the limit on the total amount of IO), and for each node many candidate neighbors (depending on R, the graph degree) will be considered.

2.2. Index Layout Modifications

Compressed Vectors Duplicated into Graph Nodes. Our first modification is based on the observation that, for a sufficiently large index, the array of compressed vectors will not be able to fit in a single machine. One option would be to store these compressed vectors in a memory-based key-value store. However, due to the large number of candidates that must be considered ($I \times R$ may be tens of thousands of lookups per search for typical parameters) and the latency implications of doing an extra network hop per beam search iteration, we instead decide to duplicate the compressed representation of each vector into all the graph nodes it is a neighbor of. This introduces a significant space amplification[3] of

$$\frac{(1+R)\,\mathrm{sizeof}(\mathrm{id}) + d + Rd_{\mathrm{OPQ}}}{R\,\mathrm{sizeof}(\mathrm{id}) + d}, \tag{1}$$

but it reduces the number of read operations per search to I. This approach mirrors that of (Tatsuno et al., 2024; Pan et al., 2023).

[2]Product Quantization and Optimized Product Quantization, described in detail in (Jegou et al., 2010; Ge et al., 2014).

[3]For parameters of $R = 100$, $d = 384$, $d_{\mathrm{OPQ}} = 64$ and using 8-byte IDs instead of 4-byte IDs to allow an index of more than 4 billion vectors, this is approximately a 10x amplification.

In-Memory Head index. We also observe that while a node cache in the key-value store still has the benefit of reducing IO operations, it does not provide the same latency benefit as in single-machine DISKANN because even cached nodes will still incur network hop latency when read. This is a major issue, since the shortest path to the furthest node from the starting point of the graph has at least $\log_R|X|$ edges. In conventional DISKANN, the first few beam search iterations almost always hit the node cache, reducing the number of round trips to disk and thus the latency. To achieve a similar effect in DISTRIBUTEDANN, we introduce a dense cache of the top layers of the graph. We first conduct a breadth-first traversal to collect C vectors from the top layers of the DISKANN graph. We then build a conventional sharded in-memory ANN index over these vectors. We call this smaller index the *head index*. At search time, we first search in the in-memory head index, and then use the results as the starting points for beam search of the DISKANN graph.

2.3. Near-data Computation

Our next modification is also motivated by latency. While log scaling within a single index is favorable compared to linear scaling across indices, it will still require $\log 100 \approx 6$ times as much computation to achieve similar quality searching a 50-billion vector index as a 500-million vector one. If we treat the key-value store purely as a virtual disk and do this work sequentially in a single machine like in DISKANN, there will be a corresponding increase in latency. We instead introduce a near-data node scoring service running on each key-value host, described in Algorithm 1.

Algorithm 1 Node Scoring Service

Input: Node keys $\{k_i\}$, threshold score t, candidate limit l, full-dimension query q, SDC encoded query q_{SDC}
Static Data: OPQ distance table
Output: Sorted result IDs and distances R, sorted candidate IDs and distances C
Initialize $R \leftarrow \varnothing, C \leftarrow \varnothing$
Batch read the node entries n_i for all k_i
for all n_i **do**
  Compute $d(q, v)$ for full-dimension vector $v \in n_i$ and insert v into R
  for all OPQ candidate $p \in e_i$ **do**
    if $d_{\text{OPQ}}(q_{\text{SDC}}, p) < t$ **then**
      Insert p into C
    end if
  end for
end for
Sort R and partial-sort C up to l
Truncate C to l

This change has the benefit of parallelizing the computation, while also allowing resources to be consumed in small uniform chunks (each key read will incur a similar amount of scoring work[4]) that work well with existing resource management/load balancing systems[5]. Additionally, since we only transmit scores over the network instead of full nodes, we achieve a bandwidth savings[6] of

$$\frac{(1+R)\,(\text{sizeof}(id) + \text{sizeof}(score)) + d + d_{\text{OPQ}}}{(1+R)\,\text{sizeof}(\text{id}) + d + Rd_{\text{OPQ}}} \quad (2)$$

compared to a naive virtual disk approach.

2.4. Orchestration Service

The final component of DISTRIBUTEDANN is an orchestration service that maintains lists of the best seen results and candidate vectors. This service will first issue a search in the head index, and then issue H rounds of calls to the node scoring service before returning a final set of results to the caller, described formally in Algorithm 2. Because this service has a small amount of persistent state, it can be hosted on many machines with low overhead, ensuring that the load

Algorithm 2 Orchestration Service

Input: Full dimension query vector q. Beam width BW. Beam iterations (hops) H. Result count k. Head index result count k_{head}. Candidate size $L \geq \max(\text{BW}, k)$.
Static Data: OPQ distance table, OPQ codebooks
Output: Sorted result IDs and distances R
Initialize result heap H_R of size k, candidate heap H_C of size L.
Encode OPQ query q_{SDC} using the codebooks.
Search for k_{head} results in the head index, and insert into H_C
for $i = 1$ **to** H **do**
  Let $t = \text{peekworst}(H_C)$
  Take best BW candidates from H_C as keys K.
  Let $\{R_i\}, \{C_i\} = \text{NodeScoring}(K, t, L, q, q_{\text{SDC}})$.
  Partially merge-sorted-lists of $\{R_i\}$ upto k and $\{C_i\}$ upto L, then insert into respective heaps.
end for
Sort H_R into R

[4]We observe that due to the high ratio of shards to BW, the typical batch size of this service is 1, resulting in very predictable resource usage.

[5]Multiple scenarios search the web index, and each has a different ideal resource vs. quality tradeoff. By consuming resources in roughly equal chunks, the existing key-value store load balancer can transparently accommodate different search parameters. In a partitioned index, different search parameters required separate benchmarking to set appropriate load factors for each scenario.

[6]Using the same parameters as in Footnote 3, this is approximately a 6x saving. We increase the savings further by pruning any neighbors that are worse than the current worst member of the candidate heap before returning to the orchestration service.

is evenly distributed. This service is also able to use hedged requests (Dean & Barroso, 2013), track replica health across requests, and allow partial failures of batches of node reads in order to reduce the tail latency normally associated with a high-fanout system like DISTRIBUTEDANN.

3. Constructing a Large Graph Index

DISKANN graphs are typically built incrementally by searching with the vector to be inserted as a query, tracking all the visited nodes during the search for use as potential graph neighbors, and then pruning to at most R actual neighbors. While it is possible to insert vectors into an DISTRIBUTEDANN graph by this procedure, it would require significantly more computation than building an equivalently sized partitioned graph, because the partitioned approach only needs to search in one smaller partition for each insertion. To reduce the graph construction cost, we employ a graph stitching approach similar to the one described in (Subramanya et al., 2019). We first build an index with clustered partitioning, with vectors in the closure of multiple clusters inserted into all of them as described in (Wang, 2021). Because these vectors occur in multiple partition's graphs, we are able to stitch together a unified graph by taking the union of their neighbors from all the partitions they are present in, as shown in Figure 2. In order to ensure that the entire graph is reachable, we build the head index from the union of the top layers of each partition's graph, rather than the top layers of the stitched-together graph. The quality of a graph built by this stitching process is lower than one built entirely incrementally, but is sufficient to get good results and is much faster to build.

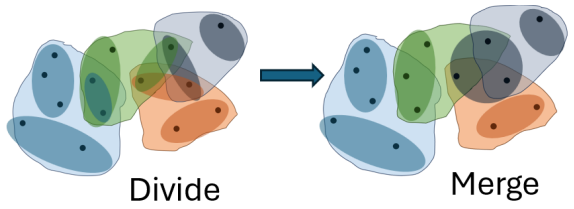

Figure 2. A visual depiction of the unified graph construction process. Partitions are represented by colored regions, and neighborhoods by dark shading. When points exist in multiple partitions, their neighborhoods will be merged by taking the union of neighbor lists, yielding a unified graph.

4. Evaluation

The Bing web index is composed of multiple independent slices to allow portions of the index to be updated atomically using less capacity than would be required to update the entire index atomically. Each slice consists of roughly 50 billion 384-dimensional `int8` vectors. We compare the performance characteristics of DISTRIBUTEDANN and a traditional clustered partitioning approach (with roughly 200 million vectors per partition) on one slice of the index. Because of the graph stitching approach described in Section 3, we are able to ingest identical indexes for both approaches, though for DISTRIBUTEDANN we only ingest the first 72 neighbors in each node to reduce storage consumption. All experiments are conducted in a production environment which has a mix of host SKUs and multiple workloads sharing the resources of each host. A typical host in this environment has 256 to 768 `GiB` of memory, 5 to 10 `TiB` of SSD storage, roughly 200 IOPS per `GiB` of SSD storage, 32 to 64 physical cores, and 40 `Gbps` of network bandwidth. We ensure that the SKU mix hosting each system is roughly equivalent. The index parameters are chosen so that each system has a similar footprint (by bounding resource) at 15k QPS. The conventional index is bound by IO while DISTRIBUTEDANN is bound by SSD space and can continue scaling to over 100k QPS in the same footprint. Performance data was collected on a set of sampled web search queries, presented in Table 1. The parameters for DISTRIBUTEDANN are $H = 5, \mathrm{BW} = 128, R = 72, k = L = 200, k_{\mathrm{head}} = 200$, with a head index size of 2.5 billion vectors. The index built with clustered partitioning selects the top 40 out of 203 partitions, and searches in each with parameters $I = 120, \mathrm{BW} = 6, R = 106, k = L = 120$.

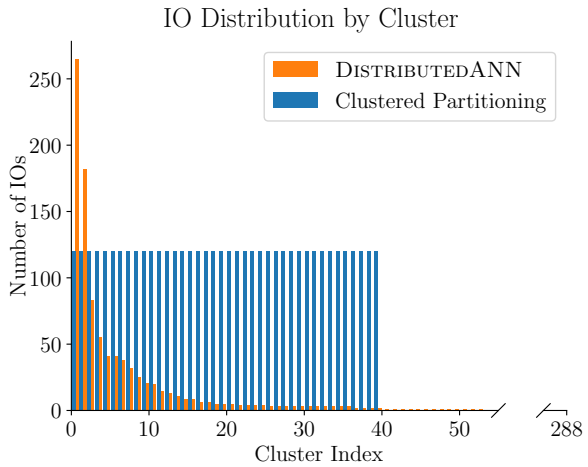

Figure 3. IO in each cluster for a single query, as served by DISTRIBUTEDANN and a conventional clustered partitioning index. DISTRIBUTEDANN is able to implement a much more flexible traversal strategy, improving efficiency.

4.1. Scaling

We observe that while DISTRIBUTEDANN does consume significantly more storage space and has higher latency than a conventional system, it uses significantly less IO and is able to achieve higher throughput and recall on the same

Table 1. Performance and accuracy comparison between DISTRIBUTEDANN and conventional approach with 3 replicas

Metric	DISTRIBUTEDANN	Clustered Partitioning
Recall@5 (%)	**90.8**	83.0
Recall@200 (%)	**71.9**	67.4
Latency@50-ile (ms)	26	**16**
Latency@99-ile (ms)	35	**22**
SSD Space (`TiB`)	780	**270**
Memory (`TiB`)	42	**18**
IO per query	**640**	4800
Network Bandwidth per query (`MiB`)	1.4	**0.3**
Throughput (QPS)	>**100k**	∼15k

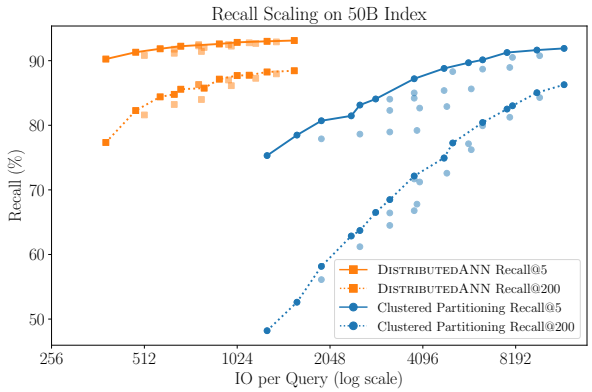

Figure 4. Optimal Recall/IO frontier for indexing 50 billion vectors. Grid search of parameters, DISTRIBUTEDANN: H from 4 to 8, BW $= 32i$ for i from 3 to 6. Clustered Partitioning: selected clusters $N = \{20, 25, 30, 40, 50, 60\}$, IO per cluster $M = 32i$ for i from 2 to 6.

graph. Because DISTRIBUTEDANN has unified graph, it can more efficiently allocate its IO budget, both by searching more deeply in relevant clusters and by touching more total clusters[7]. In comparison, a traditional clustered partitioning approach allocates a fixed amount of IO to each chosen partition (as shown in Figure 3). This allows DISTRIBUTEDANN to achieve consistently higher recall across a range of IO budgets, shown in Figure 4.

DISTRIBUTEDANN also enjoys more flexible and efficient throughput scaling than a conventional system, because the underlying key-value store can be sharded across more hosts to increase the available IOPS and CPU. In a conventional architecture, increasing the number of partitions would require more partitions to be searched for each query, increasing the IO per query. So, the main way to improve throughput is by increasing the number of replicas, which requires resources to be allocated in much coarser-grained chunks and consumes additional SSD and memory in addition to the extra IOPS and CPU.

One significant bottleneck we identified in the current implementation of DISTRIBUTEDANN is the head index, which at 3 replicas becomes CPU-bound before the node scoring service. Our solution in the current system is to increase the number of head index replicas. Due to the relatively small size of the head index and the bounding resource of the overall system (SSD space), this does not increase the number of machines required to serve the index.

4.2. Reliability

One potential concern for a distributed architecture is its resilience to issues in system components, like network partition or host failure. Although the mechanics of operating a high-availability distributed key-value store are well-understood in industry (DeCandia et al., 2007), DISTRIBUTEDANN should still be resilient to partial failures in the node-storage layer. To test this resilience, we modified the node scoring service to accept a configurable failure rate parameter. We then examined the impact on recall@5 and recall@200 of different failure rates, shown in Table 2.

The service experiences a graceful degradation in recall roughly proportional to the failure rate. This not only gives confidence in the reliability of the system, but also the performance stability, as we can safely timeout node scoring requests experiencing tail latencies without a significant adverse effect on recall.

We contrast this with our experience operating a conventional partitioned ANN service. In such a service, availability is more difficult to maintain because of the higher partition sizes requiring more time to bring a new replica online. When a partition becomes unavailable, a large chunk of the dataset is missed, causing a dramatic drop in search quality. Additionally, when vectors are partitioned by clustering and only a subset of partitions are used for each search, load becomes imbalanced, as described in Subsection 4.4. The

[7]Note that due to the graph stitching approach described in Section 3, a single read in DISTRIBUTEDANN may touch multiple clusters.

most popular clusters receive the most traffic and so are most likely to experience performance degradation, meaning that common queries will have the largest drop in search quality. In practice, independent partition scaling and heavy over-provisioning are required to achieve good availability, leading to lower resource utilization.

Table 2. Recall with degraded node scoring service availability

Availability (%)	Recall@5 (%)	Recall@200 (%)
100	90.8	71.9
99	89.7	70.1
98	88.8	69.4
97	87.5	68.8
96	87.0	67.8

4.3. Comparison with GPU-based systems

Recent work (Johnson et al., 2019; Zhao et al., 2020; Ootomo et al., 2024; Khan et al., 2024) has sought to exploit the massive parallelism of GPUs to greatly increase the search throughput of a single machine. These systems have meaningly shifted the tradeoff between vertical and horizontal scaling of ANN search, and offer extremely competitive price-to-performance for high-throughput, billion-scale indices. However, these systems still have limitations on the size of index that can fit in a single machine, and eventually encounter the same linear scaling properties and operational complexities as other partitioned indices on very large datasets. For our scenario, the scalability benefits of DISTRIBUTEDANN outweigh the advantages of GPU-based indices.

4.4. Comparison with Advanced Partitioning Schemes

Recent work such as (Dong et al., 2019; Gottesbüren et al., 2024) improves the performance of partitioned graph indices through schemes that reduce the number of cross-partition edges in an k-NN graph over the dataset. Further work is needed to compare the empirical performance of these approaches and DISTRIBUTEDANN. In particular, these approaches may be preferable in very latency-constrained scenarios where the multi-hop network overhead of DISTRIBUTEDANN is unacceptable. However, we feel that DISTRIBUTEDANN has some notable operational advantages:

- DISTRIBUTEDANN can use one algorithm to build the entire index, rather than separate approaches for coarse-grained partitioning and fine-grained index building. Because of the incremental insertion approach of DISKANN, build and search use the same code and can be optimized simultaneously.
- Semantic partitioning schemes are difficult to load-balance. Since queries will only access a subset of partitions, efficient serving requires independently scaling each partition with traffic, which is often difficult due to the different timescales between query pattern shifts and replica migration. By contrast, the underlying key-value store of DISTRIBUTEDANN is randomly sharded and so receives a predictable traffic distribution.

5. Conclusion

DISTRIBUTEDANN achieves sublinear scaling of ANN search on very large datasets, in exchange for reasonable increases in latency and disk space overhead. It also reduces operational complexity by reusing existing distributed key-value store infrastructure. We feel these tradeoffs are favorable and now use DISTRIBUTEDANN in Microsoft Bing to enable search volume growth and quality improvements on the web index.

5.1. Future Directions

We believe significant improvements in latency and space overhead are possible. Some opportunities include:

- DISTRIBUTEDANN uses traditional kernel-based TCP networking for remote service calls. Kernel-bypass networking would likely reduce the tail latency of these network calls significantly. More ambitiously, the relatively simple node scoring service logic could be implemented in a computational storage device attached to the orchestration host.
- As we note in Subsection 4.1, the head index quickly becomes compute bounded in our tests. It may be cost-effective to serve this index from GPU rather than increasing the number of replicas to serve more traffic.
- Because traditional DISKANN does not incur the space amplification that DISTRIBUTEDANN does from replicating compressed vectors into each node, techniques for reducing the average number of graph edges per vector have not been explored in depth. A significant reduction in space overhead is likely possible by placing multiple nearby full-dimension vectors into a single graph node, but further experimentation is needed to understand the tradeoffs of this approach.
- DISTRIBUTEDANN was designed to efficiently serve an index with relatively high traffic. However, further storage tiering to HDDs may allow efficient search on datasets with many mostly-cold tenants, such as per-user indices.
- DISTRIBUTEDANN is deployed across many machines in a cloud region. Inter-zone latency within a region can be up to 2ms, and cross-rack bandwidth is

oversubscribed. If DISTRIBUTEDANN were deployed on a dense cluster of machines with a fully connected network, performance would improve and it might become feasible to store the compressed vectors in a shared memory pool to reduce storage amplification.

6. Acknowledgments

We would like to thank Dafan Liu, Gena Tertychnyi, and Adelin Miloslavov who gave valuable performance advice about the key-value store, node scoring service, and orchestration service.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Andoni, A. and Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122, 2008.

Aumüller, M., Bernhardsson, E., and Faithfull, A. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, 2018. URL https://arxiv.org/abs/1807.05614.

Babenko, A. and Lempitsky, V. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence*, 37(6):1247–1260, 2014.

Big-ANN Benchmarks. Big-ann benchmarks: Neurips 2023. https://big-ann-benchmarks.com/neurips23.html, 2023. Accessed: 2025-01-04.

Dean, J. and Barroso, L. A. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

Dong, Y., Indyk, P., Razenshteyn, I. P., and Wagner, T. Learning space partitions for nearest neighbor search. In *International Conference on Learning Representations*, 2019. URL https://api.semanticscholar.org/CorpusID:189999681.

Gao, J. and Long, C. Rabitq: quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

Ge, T., He, K., Ke, Q., and Sun, J. Optimized product quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):744–755, 2014. doi: 10.1109/TPAMI.2013.240.

Gottesbüren, L., Dhulipala, L., Jayaram, R., and Lacki, J. Unleashing graph partitioning for large-scale nearest neighbor search, 2024. URL https://arxiv.org/abs/2403.01797.

Jang, J., Choi, H., Bae, H., Lee, S., Kwon, M., and Jung, M. CXL-ANNS: Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 585–600, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/jang.

Jegou, H., Douze, M., and Schmid, C. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

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

Khan, S., Singh, S., Simhadri, H. V., Vedurada, J., et al. Bang: Billion-scale approximate nearest neighbor search using a single gpu. *arXiv preprint arXiv:2401.11324*, 2024.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

Malkov, Y. A. and Yashunin, D. A. 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.

Ootomo, H., Naruse, A., Nolet, C., Wang, R., Feher, T., and Wang, Y. Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 4236–4247. IEEE, 2024.

Pan, Y., Sun, J., and Yu, H. Lm-diskann: Low memory footprint in disk-native dynamic graph-based ann indexing. In *2023 IEEE International Conference on Big Data (BigData)*, pp. 5987–5996. IEEE, 2023.

Subramanya, S. J., Devvrit, Kadekodi, R., Krishaswamy, R., and Simhadri, H. V. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems*, 32, 2019.

Tatsuno, K., Miyashita, D., Ikeda, T., Ishiyama, K., Sumiyoshi, K., and Deguchi, J. Aisaq: All-in-storage anns with product quantization for dram-free information retrieval. *arXiv preprint arXiv:2404.06004*, 2024.

Wang, Q. C. B. Z. H. W. M. L. C. L. Z. L. M. Y. J. Spann: Highly-efficient billion-scale approximate nearest neighbor search. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

Wang., Q. C. H. W. M. L. G. R. S. L. J. Z. J. L. C. L. L. Z. J. SPTAG: A library for fast approximate nearest neighbor search. https://github.com/Microsoft/SPTAG, 2018.

Zhao, W., Tan, S., and Li, P. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1033–1044. IEEE, 2020.