Microsoft DiskANN in Azure Cosmos DB

Authors

Magdalen Manohar
Senior Researcher, Azure Cosmos DB

James Codella
Principal Product Manager, Azure Cosmos DB

Mark Hildebrand
Senior Software Engineer, Azure Cosmos DB

Harsha Vardhan Simhadri
Senior Principal Researcher, Azure Cosmos DB

Ravishankar Krishnaswamy
Principal Researcher, Microsoft Research

Introduction

The use of information retrieval systems to bring data to AI models for personalized inference has exploded in popularity. For years, vector embeddings have been used to extract salient and semantically relevant features from text, image, audio, video, and tabular data. With the rise of Deep Learning approaches, vector embeddings can capture semantic relevancy with high quality. The recent rise in both small and large language models has introduced a wave of model options, services, and APIs to generate vector embeddings across a spectrum of tasks and domains.

Developers have a need to store, manage, index, and search over these vectors to find the most relevant data to a prompt, user question, or other data point. As a result, many specialized vector databases have been developed to solve this problem. Additionally, many operational databases are adding support for vector indexing and search (e.g., pgvector). However, both types of solutions tend to fall short of satisfying the requirements of modern and robust AI applications.

To enable efficient vector search, many databases employ Approximate Nearest Neighbors (ANN) indices based on popular methods like Inverted File Index (IVF), Hierarchical Navigable Small Worlds (HNSW). However, such methods can be slow at scale, requiring inordinate amounts of memory or suffering decreased accuracy when data is incrementally inserted, modified, or frequently deleted[1]; in many cases the only way to restore search accuracy is to fully rebuild the index[2], which is unacceptably costly.

In AI, efficient vector storage and search are crucial. Developers need solutions that manage and index vectors effectively, providing relevant data for prompts or queries. Existing specialized vector databases and operational databases with vector indexing often don't fully meet AI application needs. An ideal solution would offer:

- **Cost-effective vector search at any scale**: Efficient, accurate search capabilities that are affordable, regardless of vector volume.
- **Automatic scaling**: Handles small to planet-sized databases (less than 1M to more than 1B vectors) seamlessly.
- **Robustness to incremental changes**: Ensures data integrity and consistency, as well as high search accuracy, amidst updates.
- **Automated data sharding/partitioning**: Enhances performance and optimizes resource utilization.
- **High availability**: Minimizes downtime, ensuring continuous service.
- **Low-latency transactional operations**: Provides quick, responsive data retrieval and modification.
- **Built-in multitenancy**: Allows multiple users or groups to securely share the same database instance.
- **Flexible data modeling**: Allows developers to structure data to best suit their application needs.

These requirements must be met by **both** the underlying database and the vector indexing and retrieval algorithm: in other words, supporting them requires integrating a state-of-the-art indexing algorithm with a state-of-the-art database. We achieve these high standards by integrating the vector index DiskANN into Azure Cosmos DB.

For more than a decade, Azure Cosmos DB has been at the forefront of enterprise-grade operational databases capable of immense scales with industry-leading 99.999% availability, automated data

[1] FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search [arxiv: 2105.09613]
[2] https://towardsdatascience.com/not-all-hnsw-indices-are-made-equaly-6bc0d7efd8c7

partitioning, guaranteed same-region point reads/writes in under 10ms, and 4-replicas by default. Even more significant is its instant autoscale capability, scaling outward during periods of high traffic, and inward when traffic reduces. Azure Cosmos DB has set the standard for low-latency, highly scalable databases. Positioning vector search in Azure Cosmos DB also allows the users to co-locate source data and vectors, which simplifies management and helps ensure consistency.

In May 2024, Microsoft released vector search built into the Azure Cosmos DB query language for exact search. Vector search in Azure Cosmos DB provides cost-effectiveness and performance by taking advantage of Azure Cosmos DB's scale-out architecture and an index type based on **DiskANN**, a graph-based indexing and search system that can index, store, and search large sets of vector data on relatively small amounts of computational resources. DiskANN stores quantized, or highly compressed, vectors in memory, while storing the full vectors and graph structure in on-cluster, high-speed SSDs that constitute the backbone of Azure Cosmos DB data storage. DiskANN employs highly optimized traversals for fast search, with robust maintenance algorithms to maintain accuracy under replaces and deletions. This combination reduces compute consumption by an order of magnitude, while maintaining low latency, high accuracy, and high throughput. DiskANN also supports efficient query filtering via pushdown to the index to enable fast and cost-effective hybrid queries. DiskANN has been used successfully within Microsoft for years, and today it is part of crucial Microsoft applications such as web search, advertisements, and the Microsoft 365 and Windows copilot runtimes.

In this whitepaper, we dive into how DiskANN achieves high-accuracy, low latency, and cost-effective vector search at scale. We highlight several benefits including **resource efficiency**, **near real-time querying**, **update friendliness**, and **efficient filtering**.

Azure Cosmos DB's integration with DiskANN provides distinct advantages to meet the needs of modern AI developers. When compared to other state-of-the-art vector indexing algorithms, such as ScaNN, we find:

5x

Up to 5x increased throughput for in-memory DiskANN

10x

Up to 10x lower memory used with SSD DiskANN

Source: Cohere-35M and Arxiv-2M-ada-002-2M datasets

Table of Contents

Introduction to Vector Search and Database Fundamentals

Vector Search Concepts

Machine-learning generated embeddings are a crucial tool in the representation, storage, and retrieval of data such as text, images, and video, which are in turn used in popular applications such as search, recommendation, advertising, generative AI, and retrieval augmented generation (RAG). Embeddings are designed such that semantically similar objects are mapped to nearby vectors, meaning the primary algorithmic tool for retrieval of such objects is **nearest neighbor search** on their corresponding embeddings. Exact nearest neighbor search requires scanning each embedding in the database and is thus unacceptably costly, so most applications instead use **approximate nearest neighbor search (ANNS)**, trading off a small amount of accuracy for a drastically decreased cost. ANNS is evaluated using the **recall@k** metric, which measures what proportion of the true top-k nearest neighbors are reported by the approximate search algorithm. In practice, it is computationally feasible to achieve recall at or above 95% on many workloads.

The rise in popularity of embedding models has led to significant improvements in practical ANNS algorithms aimed at embedding with hundreds to thousands of dimensions over the past decade. Most ANNS algorithms can be placed into one of two categories: partition-based ANNS and graph-based ANNS. Each of them builds a **vector index**, a data structure that assists in the retrieval of embeddings. Vector indices fall roughly into two categories: partition-based ANNS and graph-based ANNS.

At a high level, **partition-based ANNS** algorithms work by partitioning the high-dimensional space into a smaller number of cells, where nearby points are likely to end up in the same cell. During retrieval, a small number of cells are exhaustively searched, and the best results found in those cells are reported as the approximate result. Prominent examples of partition-based algorithms include FAISS-IVF[3], ScaNN[4], and FALCONN[5].

[3] FAISS is a library for efficient similarity search of dense vectors
[4] Accelerating Large-Scale Inference with Anisotropic Vector Quantization [arxiv:1908.10396]
[5] Practical and Optimal LSH for Angular Distance

On the other hand, **graph-based ANNS** indices construct a graph over the embeddings in the database, with one vertex representing each embedding and directed edges between vertices. The search for a query *q* uses the so-called "greedy search," which at a designated start point, examines its neighbors in the graph, walks to the neighbor closest to *q*, and repeats the greedy walk towards the query until it can no longer improve the distance to *q* (see Figure 1). Realistically, in higher dimensions and scenarios requiring more than one similar vector to be retrieved, the greedy search explores many alternative paths towards the query. The graphs are carefully constructed so that greedy search probabilistically leads to the correct answer most of the time. Graph-based search is surprisingly efficient for querying moderate- to large-sized datasets. The fraction of points in the index examined for each query is an order of magnitude (or more) smaller compared to partition-based techniques. Popular examples of graph-based algorithms include DiskANN, HNSW[6], NSG[7] and NGT[8].

An additional algorithmic building block of ANNS is **compression** of the associated vector data, which allows data to be stored more compactly in more expensive storage tiers (e.g., main memory), transmitted more efficiently across memory bus, and distance comparisons to be computed with fewer CPU cycles with little loss in accuracy. Both **scalar quantization** and product **quantization** are widely used on top of a vector index to increase search speeds.

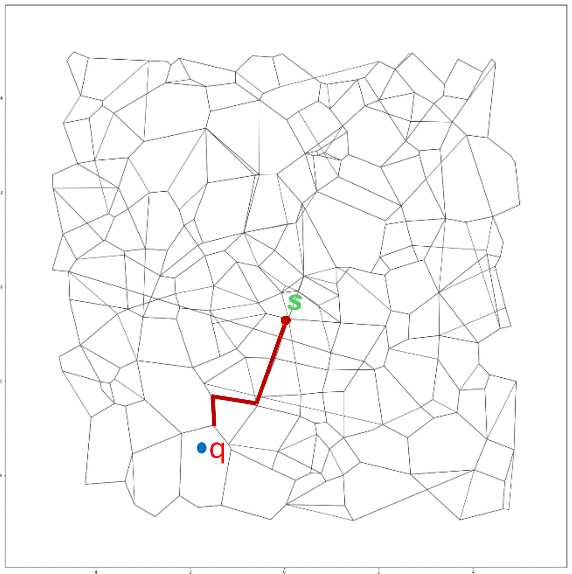

Figure 1: Illustration of greedy search in a graph index over a dataset with two-dimensional vectors. The search starts at starting point s and walks towards the query q.

Scalar quantization maps each coordinate of the embedding to a smaller representation. For example, 32-bit floating point representations are easily rounded to the nearest 16-bit floating point with little loss of precision. Rounding to the nearest 8-bit integer or even to 1- or 2-bit representations is more lossy at a coordinate level, but still preserves enough information overall to help the query navigate the index.

Product quantization (PQ)[9] is a more sophisticated technique that maps collections of coordinates to a few bytes by clustering data and mapping the data to the identity of the coordinate center. For

[6] Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs [arXiv:1603.09320]

[7] Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph [PVLDB 12(5)]

[8] Neighborhood Graph and Tree for Indexing High-dimensional Data

[9] Product Quantization for Nearest Neighbor Search | IEEE Journals & Magazine | IEEE Xplore

many datasets, product quantization achieves a better compression than scalar quantization normalized for noise introduced in distance calculations. For example, PQ can compress OpenAI's ada-002 embeddings (6KB) by 96x while retaining enough information to navigate the index. This compression is better than the compression that 1-bit quantization achieves, while retaining more signal. While PQ was specifically formulated for preserving Euclidean distances between embeddings, in practice it is also reasonable at preserving inner product distances[10].

When evaluating an ANNS algorithm or supporting vector database, considerations include both how the algorithm performs on various **performance metrics** as well as what additional **features** may be required by the use case. Commonly used performance metrics include **queries per second (QPS)** or **throughput**, query **latency**, **memory consumption, CPU utilization,** and **indexing time.** Beyond the ability to build a vector index and query it, applications may require algorithms capable of spilling to disk efficiently, supporting **streaming** insertions and deletions, and the ability to **filter** based on query metadata such as timestamp or product characteristics.

Database Concepts

There are many types of databases used today. **Non-relational** or **NoSQL databases** (e.g., Azure Cosmos DB, MongoDB, etc.) do not require a predefined structure or schema, allowing for the storage and retrieval of any data type and making them highly flexible, adaptable to changes, and especially useful for operational and transactional data.

It is common for developers and architects to also consider availability as defined by the **service level agreement** in terms of service uptime percentage. Additionally, **scalability** can be done vertically (by increasing resources such as CPU or RAM), or horizontally (by increasing compute nodes). The **elasticity** of a database measures the speed at which a database can scale up/out during periods of high traffic, and down/in during periods of low traffic, including ability to distribute and recombine data efficiently during such changes.

Scaling out is enabled by sharding, a database partitioning technique that splits a large database into smaller, more manageable pieces, called shards, and distributes them across multiple servers or nodes. Automated sharding (offered in Azure Cosmos DB) is crucial as it can help to improve performance, scalability, and availability of the database by reducing the load on each server or node and enables the system to handle large volumes of data and requests efficiently, without the need for manual intervention. **Replicas** are copies of a single data container stored in a different location. Adding replicas to a database has multiple benefits including higher availability, improved performance, reduced downtime, faster data recovery, and increased scalability.

[10] https://github.com/facebookresearch/faiss/wiki/Indexing-1M-vectors#4-bit-pq-comparison-with-scann

DiskANN: A state-of-the-art system for vector search

DiskANN is a graph-based indexing and search system that can index, update, and accurately query large sets of vector data with limited computational resources. It can work effectively with an SSD-based index and limited memory to scale to an order of magnitude more points compared to in-memory indices. It can do so while retaining high queries per second (QPS) and low query latency, thus providing a balance between memory usage and search performance. DiskANN also supports an **in-memory mode** for scenarios requiring extreme throughput, where the entire index is loaded into DRAM (the same index can be used for SSD-based or in-memory mode).

Experiments demonstrate in-memory DiskANN overall matches or outperforms alternate graph-based methods such as HNSW and partition-based methods such as ScaNN in memory consumption and query performance. SSD-based DiskANN consumes much lower memory, and still can achieve performance on par with ScaNN on moderate to large sized datasets.

Key Ideas and Research

The first set of ideas, published in the **eponymous DiskANN research paper**, demonstrates that it is possible to build and serve accurate indices on extremely large billion scale datasets using inexpensive SSDs and an order of magnitude less DRAM than other algorithms. The paper presents a new graph-based algorithm that can effectively use quantized representations of the vectors for search, which are highly compressed and can be cached in a limited memory. For OpenAI's ada-002 embeddings, for example, the algorithm can work with quantized vectors that are 64x smaller (and correspondingly lower memory) than the original. The rest of the index, including the large original embeddings and the graph index terms are stored on SSD. Moreover, the graph itself can be built in a limited-memory environment by sharding the dataset into smaller pieces during the initial graph build and merging them on the SSD.

To build a graph that supports queries using fewer round trips to the SSD, DiskANN constructs a graph on which greedy search converges to right answer using very few hops (since each hop in the graph corresponds to one random IO to SSD). The original DiskANN paper empirically demonstrates that it is possible to design graphs where over 90% of queries reach their answer in 6 hops or less; on large indices spanning hundreds of gigabytes, the index provides high recall with *as few as fifty* 4KB reads from SSD, where partition-based indices might use several thousand or more reads. Later analysis of the algorithms has shown that graphs constructed with this logic provably route queries to the correct answer[11] even when navigation uses a cheaper metric[12].

[11] Worst-case Performance of Popular Approximate Nearest Neighbor Search Implementations: Guarantees and Limitations [arxiv:2310.19126]
[12] A Bi-metric Framework for Fast Similarity Search [arxiv:2406.02891]

Fresh-DiskANN introduces new ideas to support real-time updates to the index without needing to rebuild it, while maintaining the accuracy and efficiency of the index. Although many other indices support updates, they do not necessarily preserve recall. Therefore, most other indices are rebuilt periodically from scratch after a certain number of updates[13][14], which can be computationally expensive. The first idea here is that DiskANN index construction can be fundamentally incremental so that point inserts are as effective as batch index builds and can be done with an extreme amount of parallelism. The second idea is a carefully designed logic for locally fixing the graph index when a node is deleted, since naïve deletion policies can degrade recall[15]. The third idea is that the logic for locally editing the graph can be effectively batched with low memory and amortized CPU (as compared to rebuilding).

Filtered-DiskANN is designed to efficiently answer filtered ANNS queries, which ask for the nearest neighbors of a query's embedding among points in the index that match additional predicates, such as filters on date/time, numerical ranges, or strings representing information such as product category or user id. It presents algorithms with native support for faster and more accurate filtered ANNS queries. Central to these algorithms is the construction of a graph-structured index which forms connections not only based on the geometry of the vector data, but also the associated filter set. As a result, for some categories of filtered queries, this index can respond two to three orders of magnitude faster than alternates such as inline or post filtering.

This research has inspired considerable follow-on work in academia and industry. Award-winning adaptations to new storage technologies[16] and high-end GPUs[17] have pushed the envelope of cost- and power-normalized query throughput. Alternate parallel and deterministic data structures have pushed the parallelism and scalability of the algorithms to over a hundred cores[18]. New data layouts that improve IO per query[19][20], methods for real time updates[21] and new classes of filter pushdown are being developed[22].

---

[13] Speed up HNSW merge by writing combined vector data [LUCENE-10375] · Issue #11411 · apache/lucene · GitHub
[14] Update and rebuild an active index | Vertex AI | Google Cloud
[15] big-ann-benchmarks/neurips23/notes/streaming/hnsw_result/hnsw_result.md at main · harsha-simhadri/big-ann-benchmarks · GitHub
[16] Winning the NeurIPS BillionScale Approximate Nearest Neighbor Search Challenge
[17] BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU [arxiv:2401.11324]
[18] ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms [arxiv: 2305.04359]
[19] DiskANN++: Efficient Page-based Search over Isomorphic Mapped Graph Index using Query-sensitivity Entry Vertex [arxiv: 2310.00402]
[20] LM-DiskANN: Low Memory Footprint in Disk-Native Dynamic Graph-Based ANN Indexing | IEEE Conference Publication | IEEE Xplore
[21] Enhancing HNSW Index for Real-Time Updates: Addressing Unreachable Points and Performance Degradation [arxiv: 2407.07871]
[22] ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data [arxiv: 2403.04871]

Empirical Comparison with other Vector Indexing Methods

In this section, we benchmark two variants of DiskANN that differ in how the index terms are stored – entirely in DRAM (we call this configuration in-mem), or with the quantized vectors stored in-memory and the remaining terms are stored on a fast SSD (we call this configuration SSD). We compare the two configurations to two state-of-the-art vector indices, ScaNN[23] and HNSW[24]. ScaNN is a partition-based search tree augmented with anisotropic vector quantization designed for inner product distances, while HNSW is a graph-based index which employs a multi-level graph-based index. Each experiment constructs a static vector index and runs a batch of queries. We find that DiskANN generally produces the most favorable QPS/Recall tradeoff and consumes the least memory.

Datasets. Performance was evaluated for two scenarios: a smaller dataset consisting of two million embeddings, where in-memory algorithms are most relevant, and a larger dataset of 35 million embeddings, where good disk-based performance becomes crucial. **Recall/QPS** tradeoffs as well as **build time** and peak **memory consumption** during query of each index, were reported. Note that while DiskANN indices of over a Terabyte size for much larger datasets (per partition) are routinely deployed on commodity machines with <256GB RAM, the publicly available implementation of ScaNN, and the original public implementation of HNSW (hnswlib-v0.8.0) and many other implementations, would not work in such a setting. Therefore, we limit the datasets to sizes where all three libraries can work reasonably well.

The larger dataset, Wikipedia-Cohere-35M, consists of 35 million Wikipedia articles embedded using the state-of-the-art cohere.ai multilingual 22-12 embedding model. The embeddings have 768 floating-point coordinates. The smaller dataset, ArXiv-OpenAIv2-2M, consists of two million ArXiV article abstracts embedded using the 2nd generation OpenAI embeddings with 1536 floating-point coordinates. Experiments were conducted on an Azure Lsv3-series virtual machine with a local NVMe disk capable of 400,000 read IOps.

Code and Parameters. Code and parameters were chosen according to publicly available guidelines for both ScaNN[25] and HNSW[26], and each algorithm was run using its Python wrapper[27][28]. Given that there are many possible implementations of HNSW to choose from, we use the original authors' hnswlib-v0.8.0, as it is the most widely recognized and benchmarked. We note that hnswlib-v0.8.0 lacks some features such as quantization that alternate re-interpretations of HNSW have adopted.

[23] Accelerating Large-Scale Inference with Anisotropic Vector Quantization [arxiv: 1908.10396]
[24] Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs [arxiv:1603.09320]
[25] google-research/scann/docs/algorithms.md at master · google-research/google-research (github.com)
[26] hnswlib/ALGO_PARAMS.md at master · nmslib/hnswlib (github.com)
[27] hnswlib/python_bindings at master · nmslib/hnswlib (github.com)
[28] google-research/scann/docs/example.ipynb at master · google-research/google-research (github.com)

Search parameters were chosen by exhaustively searching possible combinations and reporting the ones that gave the best QPS/recall tradeoff. See Figure 2 and Figure 3 for the exact build parameter choices. For DiskANN, index build parameters were chosen by exhaustive search over possible ranges, selecting for those which gave the best QPS/Recall tradeoff.

Memory Consumption. The experiments measure **peak memory** consumption of each algorithm during the query phase; in practice, this includes the size of the indexing structure, as well as the aspects of vector data used during search (quantized vectors, full-precision vectors, etc.). Two innovations significantly reduced the peak memory consumption of DiskANN compared to HNSW and ScaNN. For In-Mem DiskANN, **scalar quantization** is used to cut the size of the dataset in half without needing to store full-precision vectors, unlike ScaNN and other systems using product quantization followed by re-ranking using full-precision vectors. In the interest of a fair comparison with HNSW, since hnswlib does not include quantization, we also report the full-precision version of DiskANN in our results. On the other hand, SSD DiskANN stores only **quantized** vectors in-memory and pulls the necessary vector data and graph data directly from the SSD at the point of search, keeping its peak memory usage to a small fraction of the total index size. SSD DiskANN also provides the ability to cleanly trade off throughput for memory consumption by caching a small number of frequently accessed graph nodes in the main memory.

Experimental Results

Table 1 and Figure 2 show the performance of In-Memory DiskANN, SSD DiskANN, ScaNN, and HNSW on Wikipedia-Cohere-35M. Table 1 shows the build time and peak memory consumption of each algorithm, while Figure 2 shows the QPS/Recall tradeoff.

Table 2 and Figure 3 show analogous results for the ArXiv-OpenAIv2-2M dataset.

Table 1: Index build times and peak memory usage for DiskANN, HNSW, and ScaNN for the Wikipedia-Cohere-35M dataset.

Method	Build Time (s)	Peak Memory During Query (GB)	Parameters
ScaNN	2059	135	num_leaves=8000, dimensions per block = 2, anisotropic quantization threshold = 0.2
In-Mem DiskANN	2948 (SC) 4556 (FP)	56 (SC) 115 (FP)	R=64, L=128, 16-bit scalar quantizer (SC) where indicated, full precision (FP) where indicated
HNSW (hnswlib, v0.8.0)	13778	116	M=32, ef_search=400
SSD DiskANN	14609	18 (5% caching), 13 (0% caching)	R=64, L=400, caching both 0% and 5% of graph nodes in-memory

Table 2: Build times and peak memory usage for DiskANN, HNSW, and ScaNN for the ArXiV-OpenAIv2-2M dataset

Method	Build Time (s)	Peak Memory During Query (GB)	Parameters
ScaNN	206	17.3	num_leaves=1200, dimensions per block = 2, anisotropic quantization threshold = 0.2
DiskANN	851 (SC) 1210 (FP)	9.4 (SC) 16.3 (FP)	R=40, L=400, 16-bit scalar quantizer (SC) where indicated, full precision where indicated
HNSW (hnswlib, v0.8.0)	1137	16.5	M=32, ef_search=400

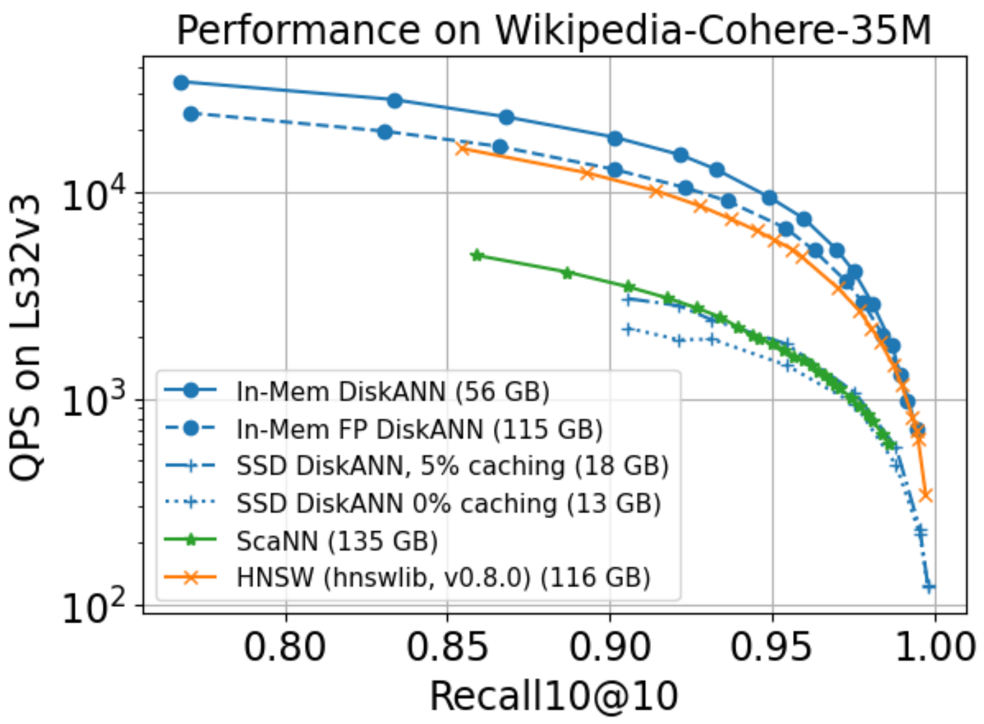

Figure 2: QPS/Recall Curves for disk-based and in-memory DiskANN, HNSW, and ScaNN for the Wikipedia-Cohere-35M dataset using 8 threads. The legend indicates peak memory usage of each algorithm during the query phase. Table 1 lists parameters for each algorithm.

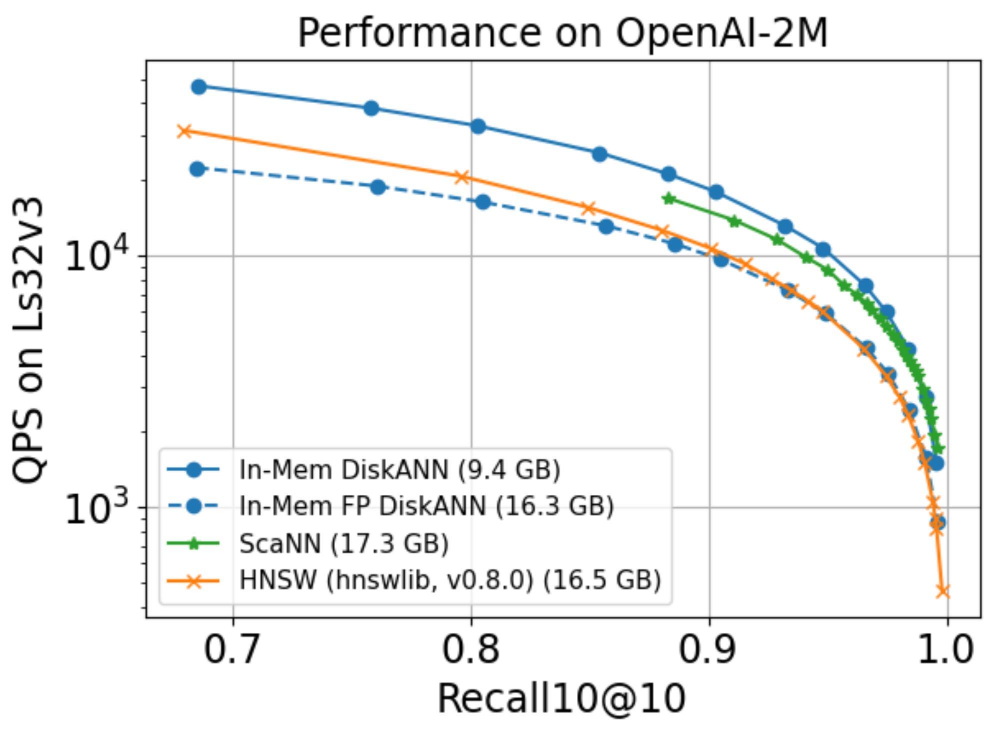

Figure 3: QPS/Recall Curves for in-memory DiskANN, HNSW, and ScaNN for the ArXiV-OpenAIv2-2M embedding dataset using 32 threads. The legend indicates peak memory usage of each algorithm during the query phase. Table 2 lists parameters choices for each algorithm.

Experimental Conclusions

On moderately large datasets represented with modern embeddings, in-memory DiskANN matches or exceeds alternate graph-based methods such as HNSW and partition-based methods such as ScaNN on memory consumption and query throughput, sometimes at the cost of higher build times. SSD-based DiskANN consumes much lower memory and still achieves performance on par with ScaNN on Wikipedia-Cohere-35M.

On Wikipedia-Cohere-35M, In-Mem DiskANN achieves up to **5x** speedup over ScaNN and up to **2x** speedup over the hnswlib implementation of HNSW, while consuming less than **half** the memory. SSD DiskANN achieves performance on par with ScaNN while using **less than 1/5-th** the memory and achieves only marginally reduced performance when restricted to **1/10-th** the amount of memory used by ScaNN.

On ArXiv-OpenAIv2-2M, In-Mem DiskANN achieves up to **1.2x** speedup over ScaNN and up to **1.5x** speedup over the hnswlib implementation of HNSW, in both cases using less than **2/3** of the memory.

The DiskANN and Azure Cosmos DB Advantage

A high-performance implementation of DiskANN within Microsoft

A stateless library for multiple storage systems. At Microsoft, the ideas behind DiskANN power many large-scale mission-critical systems such as web search, advertisement, and the Microsoft 365 copilot and Windows copilot runtimes[29]. To take the best of the algorithmic ideas to multiple storage systems and data structures (e.g., Files, B-trees, Bw-trees, K-V stores), we have built a high-performance library in Rust that abstracts the logic of the DiskANN indexing and search algorithms from the details of how index terms are stored and managed. This allows separate and continuous innovation of the algorithms, and management of the index terms. The algorithms can be adapted to a target system by defining how various indexing terms are persisted, cached, and replicated. Extremely high throughput systems may choose to pin everything to memory, while cost-conscious applications may store some terms in slower storage media.

Coupling with Azure Cosmos DB. We coupled the DiskANN library with a highly optimized Azure Cosmos DB inter-op to leverage the performance and scalability features of Azure Cosmos DB. Specifically, we store a PQ compressed vector for each embedding in a Bw-Tree[30] (see Concepts for a discussion of vector compression). A Bw-Tree is a fast concurrent key lookup data structure optimized for storage media like SSD and allows the most frequently accessed terms to be paged into the main memory cache. Terms corresponding to the graph are also stored and updated in a Bw-Tree. The document with the full precision vector and its associated data (e.g., text and metadata) is stored in a lower-cost durable disk-based store. Larger indices can use the native partition feature in Azure Cosmos DB to scale out. We largely reuse the existing query syntax and infrastructure with the addition of a paginated vector search API provided by DiskANN. This search uses quantized vectors to search more broadly in the graph to satisfy additional predicates associated with the query. We re-

[29] Unlock a new era of innovation with Windows Copilot Runtime and Copilot+ PCs - Windows Developer Blog
[30] The Bw-Tree: A B-tree for New Hardware

rank the results discovered by the query layer using quantized vectors using a limited set of full precision vectors to determine a high-recall answer. See Figure 4 for an illustration of re-ranking.

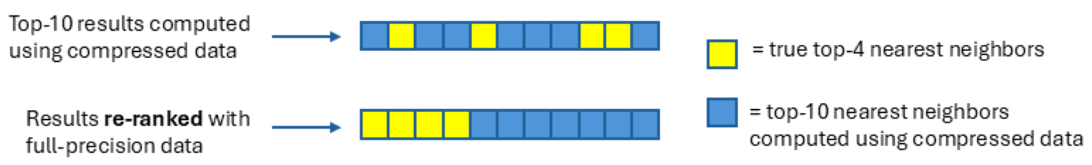
Top-10 results computed using compressed data
Results **re-ranked** with full-precision data
= true top-4 nearest neighbors
= top-10 nearest neighbors computed using compressed data

*Figure 4 illustrates the concept of **re-ranking**, where a search engine retrieves more than the desired k results using compressed data, and then **re-ranks** using full-precision data to retrieve the true top-k.*

This design allows the user two indexing types – Quantized Flat and DiskANN. In Quantized Flat, the query linearly scans through all the quantized vectors corresponding to documents that satisfy the query predicate. This is useful in scenarios where either the index or the set of documents that satisfy the predicate is very small (a few hundred to a few thousand documents). For moderate to large indices with less selective predicates, DiskANN is recommended due to its efficiency at query time.

Vector Search in Azure Cosmos DB

Azure Cosmos DB stands out as a world-class enterprise-ready database compared to dedicated vector databases due to its robust set of features and capabilities. The integration of DiskANN search library within the Azure Cosmos DB combines the power of low latency and accurate vector search with the reliability of a planet-scale distributed database engine. Azure Cosmos DB's architecture, built on high-speed SSDs, enables efficient I/O operations during vector search while ensuring cost-efficiency in large-scale scenarios. The features and capabilities highlighted in this section are also summarized in Table 3.

Capability	Azure Cosmos DB	Most Vector Databases
Reliability	99.999%	99.9%
Data replicas	4 (configurable to more)	1 (configurable to more)
Instant autoscale	Yes	No
Automatic sharding/partitioning	Yes	No
SLA on point-reads/writes	<10ms[31]	No
Consistency options	5 levels: Strong, Bounded staleness, Session, Consistent prefix, Eventual	1-level: Eventual
Multitenancy options	4-levels of isolation per tenant: partition key, container, database, account	2 levels of isolation per tenant: index and compute node
Serverless to provisioned throughput	Yes	No
Rich SQL-like query syntax	Yes	Some (e.g. PostgreSQL)

Table 3: summary of Azure Cosmos DB features and capabilities that support efficient vector search.

Reliability

One of the key benefits of Azure Cosmos DB its high availability, which it guarantees through multi-region replication and failover capabilities. It offers a comprehensive Service Level Agreement (SLA) with up to 99.999% availability, which is unmatched today in any other database offering built-in vector search capabilities.

Data replicas

Within each region, Azure Cosmos DB automatically writes data to 3 out of 4 replicas (local majority). This results in an RTO (Recovery Time Objective) of 0 and an RPO (Recovery Point Objective) of 0 for individual node outages, without requiring any application changes or configurations, minimizing chance of service disruption to the developer's applications in the rare event of outages. Moreover, the developer can add or remove the regions associated with their Azure Cosmos DB account at any time, configuring more secondary regional replicas as needed. Again, such capabilities are unavailable in specialized vector databases that offer one replica by default and applications that incur large costs for additional replicas.

SLA on point-reads/writes

With a single-digit latency SLA for point read and write operations in the same region, Azure Cosmos DB can efficiently manage both transactional data workloads and vector workloads. This provides a

[31] SLA is for 1 KB of data; some vector embeddings may be larger than 1 KB per vector.

unified solution for diverse data operations, simplifying architecture and reducing operational overhead.

Consistency options

Azure Cosmos DB offers five well-defined consistency models to cater to different application requirements: Eventual, Consistent Prefix, Session, Bounded Staleness, and Strong. These options allow developers to make precise trade-offs between consistency, availability, and the performance of their transactional and operational workloads.

Automatic sharding/partitioning

Azure Cosmos DB's automatic sharding or partitioning feature dovetails perfectly with DiskANN's ability to operate at scale. Cosmos DB can distribute indices across multiple partitions in a collection, allowing it to manage larger datasets and achieve higher ingest throughput.

Instant Autoscale

With instant per partition autoscaling, Azure Cosmos DB can automatically adjust throughput based on traffic, scaling out during high-demand periods and scaling in during low-demand periods, helping to reduce costs while maintaining consistent performance.

Serverless to Provisioned Throughput

With both serverless and provisioned throughput configurations, and the ability to seamlessly move from one model to the other, Azure Cosmos DB provides the flexibility to adapt as the developer's applications evolve to new scenarios, usage patterns, and scales.

Multitenancy options

With multiple options for multitenancy, from partition-isolation, collection, database, and resource isolation, Azure Cosmos DB is ready for vector search scenarios where different applications or services might need to perform searches on data that is logically or physically isolated from each other. The variety of multitenancy options enables efficient resource utilization, cost savings, and simplified management in vector search scenarios for any level of isolation an AI application would require.

Rich SQL-like query syntax

Azure Cosmos DB provides a NoSQL query language allowing complex filters to be applied. These filters support a variety of operators such as equality, range, and array string, spatial, and more. When combined with the low-latency vector search provided by DiskANN, such query filters can significantly enhance the freshness and relevancy of search results.

Conclusion

Modern AI applications demand both efficient, secure, and reliable storage of vectors and core data, as well as sophisticated and fast indexing and retrieval algorithms; in other words, a best-in-class database system must be combined with a best-in-class indexing algorithm. The low latency, high reliability, and seamless scaling capabilities of Azure Cosmos DB combined with the high throughput, update friendliness, and efficient filtering capabilities of DiskANN make vector search in Azure Cosmos DB an unparalleled offering in the universe of vector databases.