**🔍 Unlocking Intelligent Search at Scale: Introducing DiskANN (Disk Accelerated Nearest Neighbor)**

In a world driven by embeddings, vectors, and intelligent systems, the ability to **search for “semantic similarity” at scale** has become a foundational capability. We’ve now integrated **DiskANN**, a cutting-edge open-source library developed by Microsoft, into our architecture to enable **fast, disk-based approximate nearest neighbor (ANN) search** over massive high-dimensional vector datasets.

**🧠 What is DiskANN?**

**DiskANN (Disk Accelerated Nearest Neighbor)** is a high-performance ANN search system designed to **scale beyond memory limitations**. Unlike traditional vector search engines that require all data to fit in RAM, DiskANN leverages **disk-based hierarchical navigable small-world (HNSW)-like graphs** for fast, memory-efficient retrieval.

**Key Features:**

* **Billions of Vectors**: Supports ultra-large datasets stored on SSD/NVMe disks.
* **High-dimensional Support**: Optimized for 100–10,000+ dimensions.
* **Low Latency**: Delivers sub-second response times using intelligent caching.
* **Microsoft Production-Grade**: Used internally in Bing and Azure Cognitive Services.

**🚀 Why DiskANN is a Game-Changer for Vector Search**

1. **Disk-Scale Indexing**

Traditional solutions like FAISS or HNSW require large amounts of RAM. DiskANN breaks through these constraints using efficient disk access patterns, making it ideal for **cost-effective and scalable AI workloads**.

1. **Real-Time Semantic Intelligence**

DiskANN enables search and retrieval based on **meaning, not just keywords**, empowering use cases like:

* + Semantic search over documents
  + Visual similarity search
  + Matching embeddings from LLMs, images, or graph nodes

1. **Hybrid Integration with PostgreSQL and Apache AGE**

Although DiskANN is standalone, it can be used as a **semantic overlay** on top of relational or graph data:

* + Extract candidate records with DiskANN
  + Enrich or filter them via SQL or Cypher from PostgreSQL/AGE

1. **Azure Native Compatibility**

DiskANN is already embedded into **Azure Cognitive Search with vector search** enabled. For custom workloads, we can:

* + Deploy it as an API-based microservice
  + Use it in Azure Functions or AKS
  + Store indices on Azure Premium Disks or Blob Storage

**🔧 Our Implementation Strategy**

We’ve begun integrating DiskANN for enterprise-grade vector similarity tasks:

**Architecture Highlights:**

* **Embedding Generation**: LLM/ML models generate vector representations (e.g., using Azure OpenAI, Hugging Face, or in-house models).
* **Index Construction**: Vectors are indexed using DiskANN CLI or Python wrappers.
* **Disk-Based Hosting**: Indexes stored on SSDs (local or Azure-managed).
* **API Wrapper**: Exposed DiskANN search via FastAPI endpoints for consumption by internal apps and pipelines.
* **Result Enrichment**: Matched vector IDs are mapped back to PostgreSQL/AGE records for metadata, tags, or relationships.

**🔗 Real-World Application Scenarios**

| **Use Case** | **Description** |
| --- | --- |
| **Semantic Search** | Retrieve documents, articles, or policies based on meaning (not keywords). |
| **Recommendation Engines** | Suggest similar products, customers, or user actions based on vector proximity. |
| **Fraud and Anomaly Detection** | Match patterns in user or transaction embeddings to identify outliers. |
| **Multimodal AI** | Use shared vector spaces to link images, text, and audio. |
| **LLM Memory Index** | Store vectorized LLM conversations, documents, or code snippets for contextual retrieval. |

**🧬 How It Integrates with Our Existing Stack**

| **Component** | **Role** |
| --- | --- |
| **Azure PostgreSQL** | Stores metadata, tags, and relationships for mapped vectors |
| **Apache AGE** | Models entity connections and subgraphs; ideal for re-ranking or structural reasoning |
| **DiskANN API** | Retrieves top-k similar vectors using cosine or L2 distance |
| **Azure OpenAI / Hugging Face** | Generates embeddings for users, products, texts, etc. |
| **Azure Functions / FastAPI** | Wraps DiskANN as a scalable service |
| **Azure Blob / Premium SSD** | Hosts DiskANN index files for fast access |

**🧩 FAQ: Common Technical Questions**

**🔸 Q1. Can DiskANN integrate with Apache AGE or PostgreSQL?**

Yes. DiskANN returns vector IDs; you can use these to query metadata or graph context from PostgreSQL or AGE via standard joins or cypher queries.

**🔸 Q2. Can I combine DiskANN with pgvector?**

Yes. Use pgvector for real-time, in-RAM lookups for small datasets and DiskANN for massive-scale ANN search.

**🔸 Q3. How do we support filtering or metadata-aware queries?**

By chaining DiskANN (for top-K retrieval) with PostgreSQL filters or AGE graph traversal.

**🔸 Q4. Can it be used in Azure cloud-native services?**

Yes. Deploy DiskANN in containers, Azure Functions, or container apps. Store vectors and indices in Azure storage tiers.

**🔸 Q5. How scalable is DiskANN?**

It’s used in Bing-scale workloads. With SSD optimization and proper sharding, it handles billions of vectors with predictable latency.

**🧭 What’s Next**

* **Hybrid Search:** Combine text search (BM25), graph reasoning (AGE), and vector ranking (DiskANN) for intelligent retrieval pipelines.
* **Auto-index Pipelines:** Automate vector generation and DiskANN index updates from batch jobs or event streams.
* **Embedding Store Governance:** Apply role-based access and monitoring around sensitive embeddings, especially for PII or regulated workloads.

**DiskANN in Azure Database for PostgreSQL – Flexible Server**

This document provides a comprehensive overview of DiskANN, its integration into Azure Database for PostgreSQL Flexible Server, and practical examples for leveraging its vector search capabilities within your applications.

**🔍 Understanding DiskANN**

DiskANN is an **Approximate Nearest Neighbor (ANN)** search algorithm developed by Microsoft Research. It's designed to efficiently find similar items (represented as high-dimensional vectors) within extremely large datasets, even when those datasets exceed available memory.

**Key Characteristics:**

* **Approximate Nearest Neighbor (ANN):** It quickly finds "good enough" similar items, making it suitable for real-time applications where speed is critical.
* **Graph-based:** DiskANN builds a graph where data points (vectors) are nodes, and edges connect them to their approximate neighbors. Search involves traversing this graph.
* **Disk-Optimized:** Designed to work efficiently with Solid State Drives (SSDs), allowing it to handle billions of vectors with significantly less RAM compared to purely in-memory solutions. This makes it highly cost-effective and scalable.
* **High Accuracy & Low Latency:** It balances search accuracy (recall) with high query throughput and low response times.
* **Robust to Updates:** Supports real-time data insertions, deletions, and modifications without significant performance degradation.
* **Filtered Vector Search:** A key advantage is its ability to combine vector similarity search with traditional database filters (e.g., WHERE price < 100).

**☁️ DiskANN in Azure**

Microsoft has integrated DiskANN into several Azure services to enable scalable and efficient vector search for AI-driven applications.

**Key Integrations:**

* **Azure Database for PostgreSQL – Flexible Server:** This is highly relevant to your use case. The pg\_diskann extension allows you to build DiskANN indexes directly within your PostgreSQL database, transforming it into a powerful vector database.
* **Azure Cosmos DB:** Also leverages DiskANN for its vector search capabilities in NoSQL environments.

**Benefits of using DiskANN in Azure:**

* **Scalability:** Handles vast datasets of vectors, supporting growth in AI applications.
* **Cost-Effectiveness:** Reduces memory requirements by leveraging SSDs, leading to lower infrastructure costs.
* **High Performance:** Delivers accurate results with low latency, even for complex filtered queries.
* **Simplified Architecture:** Consolidate relational data and vector embeddings in a single PostgreSQL database.

**Typical Use Cases:**

* **Semantic Search:** Finding content based on meaning, not just keywords.
* **Recommendation Systems:** Suggesting relevant items (products, content) to users.
* **Retrieval Augmented Generation (RAG):** Enhancing Large Language Models (LLMs) with external knowledge for more accurate and contextually rich responses.

**🚀 Leveraging DiskANN with PostgreSQL**

For your PostgreSQL database running on Azure Database for PostgreSQL – Flexible Server, DiskANN is enabled through the pg\_diskann extension, which works in conjunction with the pgvector extension. This setup turns your PostgreSQL instance into a full-fledged vector database.

**How it Works:**

1. **Vector Embeddings:** Your application or a connected service generates numerical representations (embeddings) of your data (e.g., text, images, product descriptions) using an LLM or embedding model. These are typically high-dimensional (e.g., 1536 dimensions for text-embedding-ada-002).
2. **Store Vectors:** These embeddings are stored in a VECTOR type column within your PostgreSQL tables.
3. **Create DiskANN Index:** You create a DISKANN index on the VECTOR column.
4. **Perform Similarity Searches:** Your applications execute SQL queries that leverage this index for efficient similarity search.
5. **Hybrid Search:** Combine vector similarity with traditional SQL filters on other columns (e.g., WHERE status = 'available').

**Prerequisites & Setup:**

1. **Azure Database for PostgreSQL – Flexible Server:** Ensure you have an instance created (refer to previous setup instructions). For demo purposes, the **Burstable B1MS tier** can be used within the Azure Free Tier limits for 12 months, but for larger datasets or production, a higher tier is recommended.
2. **Enable Extensions in Azure Portal:**
   * Navigate to your PostgreSQL Flexible Server in the Azure Portal.
   * Go to **"Server parameters"** under "Settings".
   * Search for azure.extensions and add pg\_diskann,vector,azure\_ai (comma-separated) to the VALUE field. Save changes.
3. **Azure OpenAI Service (for Embedding Generation):**
   * You need an Azure OpenAI Service resource with an embedding model deployed (e.g., text-embedding-ada-002 or text-embedding-3-small). Note its **Endpoint URL** and an **API Key**.
   * **Crucially for Private VNet Integration:** If your PostgreSQL server is in a private VNet, you **MUST create a Private Endpoint** for your Azure OpenAI Service within the *same VNet and subnet*. This resolves network connectivity issues.
     + **Steps to Create Private Endpoint:** Go to your Azure OpenAI Service resource -> Networking -> Private endpoint connections -> + Private endpoint. Follow the wizard, selecting the correct VNet/Subnet and ensuring "Integrate with private DNS zone" is set to "Yes" to link privatelink.openai.azure.com to your VNet.

**SQL Setup & Examples:**

Connect to your PostgreSQL database using psql or Azure Data Studio.

**1. Create Extensions & Configure Azure AI:**

SQL

-- Create the necessary extensions in your database

CREATE EXTENSION IF NOT EXISTS pg\_diskann CASCADE;

CREATE EXTENSION IF NOT EXISTS azure\_ai;

-- Configure the Azure OpenAI endpoint and API key

-- Replace with your actual values from Azure OpenAI Service's "Keys and Endpoint"

SELECT azure\_ai.set\_setting('azure\_openai.endpoint', 'https://<your-openai-resource-name>.openai.azure.com/');

SELECT azure\_ai.set\_setting('azure\_openai.subscription\_key', '<YOUR\_AZURE\_OPENAI\_API\_KEY>');

-- (Optional) Verify the settings are correctly stored

SELECT azure\_ai.get\_setting('azure\_openai.endpoint');

SELECT azure\_ai.get\_setting('azure\_openai.subscription\_key');

**2. Create Table with Auto-Generated Embeddings:**

The GENERATED ALWAYS AS column simplifies data insertion by automatically creating embeddings.

SQL

CREATE TABLE products (

product\_id SERIAL PRIMARY KEY,

name TEXT,

description TEXT,

price NUMERIC(10, 2),

in\_stock BOOLEAN,

category TEXT, -- Example for filtering

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP, -- Example for filtering

description\_embedding VECTOR(1536) GENERATED ALWAYS AS (

azure\_openai.create\_embeddings(

'YOUR\_OPENAI\_EMBEDDING\_DEPLOYMENT\_NAME', -- Use the exact deployment name from Azure OpenAI Studio

description

)

) STORED

);

**Example Data Insertion:**

SQL

INSERT INTO products (name, description, price, in\_stock, category) VALUES

('Premium Smartwatch', 'A sleek smartwatch with advanced health tracking and long battery life.', 299.99, TRUE, 'Electronics'),

('Wireless Noise-Cancelling Headphones', 'Immersive audio experience with superior noise cancellation for travel and work.', 199.99, TRUE, 'Audio'),

('Ergonomic Office Chair', 'Supportive and comfortable chair designed for long hours of work, adjustable.', 350.00, TRUE, 'Furniture'),

('High-Speed External SSD', 'Fast and portable storage solution for backing up large files and gaming.', 150.00, TRUE, 'Electronics'),

('Smart LED Light Bulbs (4-pack)', 'Control your home lighting with your voice or smartphone app, energy efficient.', 45.00, TRUE, 'Smart Home');

The description\_embedding will be populated automatically for each row.

**3. Create DiskANN Index:**

SQL

CREATE INDEX idx\_products\_description\_embedding ON products USING diskann (description\_embedding VECTOR\_COSINE\_OPS);

-- Use VECTOR\_COSINE\_OPS for cosine distance (common for semantic search),

-- VECTOR\_L2\_OPS for Euclidean distance, or VECTOR\_IP\_OPS for Inner Product.

-- (Optional) With tuning parameters for larger datasets:

-- CREATE INDEX idx\_products\_description\_embedding\_tuned ON products USING diskann (description\_embedding VECTOR\_COSINE\_OPS)

-- WITH (max\_neighbors = 64, l\_value\_ib = 200, pq\_param\_num\_chunks = 384);

**4. Perform DiskANN Searches:**

We'll use a Common Table Expression (CTE) to generate the query embedding for cleaner SQL.

**a. Basic Similarity Search (Top N):**

SQL

WITH search\_query AS (

SELECT azure\_openai.create\_embeddings('YOUR\_OPENAI\_EMBEDDING\_DEPLOYMENT\_NAME', 'gadgets for daily convenience') AS q\_vec

)

SELECT

p.product\_id,

p.name,

p.description,

p.price,

p.category,

p.description\_embedding <=> sq.q\_vec AS cosine\_distance -- Smaller value = more similar

FROM

products p,

search\_query sq

ORDER BY

cosine\_distance

LIMIT 5;

**b. Filtered Similarity Search (Vector + Traditional SQL Filters):**

SQL

-- Search for 'office seating' that is 'in stock' and 'under $400' in the 'Furniture' category

WITH search\_query AS (

SELECT azure\_openai.create\_embeddings('YOUR\_OPENAI\_EMBEDDING\_DEPLOYMENT\_NAME', 'comfortable chair for work') AS q\_vec

)

SELECT

p.product\_id,

p.name,

p.description,

p.price,

p.in\_stock,

p.category,

p.description\_embedding <=> sq.q\_vec AS cosine\_distance

FROM

products p,

search\_query sq

WHERE

p.in\_stock = TRUE

AND p.price < 400.00

AND p.category = 'Furniture'

ORDER BY

cosine\_distance

LIMIT 3;

**c. Time-Based Filtered Search:**

SQL

-- Find 'recent tech releases' from the last 90 days

WITH search\_query AS (

SELECT azure\_openai.create\_embeddings('YOUR\_OPENAI\_EMBEDDING\_DEPLOYMENT\_NAME', 'innovative new technology products') AS q\_vec

)

SELECT

p.product\_id,

p.name,

p.description,

p.created\_at,

p.description\_embedding <=> sq.q\_vec AS cosine\_distance

FROM

products p,

search\_query sq

WHERE

p.created\_at >= NOW() - INTERVAL '90 days'

ORDER BY

cosine\_distance

LIMIT 5;

**d. Demonstrate Index Usage with EXPLAIN ANALYZE:**

SQL

EXPLAIN (ANALYZE, VERBOSE, BUFFERS)

WITH search\_query AS (

SELECT azure\_openai.create\_embeddings('YOUR\_OPENAI\_EMBEDDING\_DEPLOYMENT\_NAME', 'portable sound devices') AS q\_vec

)

SELECT

p.product\_id,

p.name,

p.description,

p.description\_embedding <=> sq.q\_vec AS cosine\_distance

FROM

products p,

search\_query sq

ORDER BY

cosine\_distance

LIMIT 5;

Look for Index Scan using idx\_products\_description\_embedding in the EXPLAIN output, which confirms DiskANN is being utilized.

**Key Takeaways for Application Team:**

* **Familiar SQL Interface:** Leverage existing SQL knowledge for vector search.
* **Simplified Architecture:** Store relational data and embeddings in one database, reducing complexity.
* **Scalability & Performance:** Efficiently handle large datasets and high query loads for AI features.
* **Cost-Optimized:** DiskANN's design reduces the high memory costs associated with purely in-memory vector databases.
* **Hybrid Search:** Combine semantic search with precise filters for powerful and relevant results.
* **Considerations:**
  + **Embedding Model Selection:** Choose an embedding model that aligns with your data and use case (e.g., OpenAI, Cohere, open-source models).
  + **Vector Dimension Consistency:** Ensure all embeddings (stored and queried) match the VECTOR column's dimension.
  + **Index Tuning:** For production, experiment with pg\_diskann parameters (like max\_neighbors, l\_value\_ib, pq\_param\_num\_chunks) to balance accuracy, speed, and resource usage.
  + **Monitoring:** Keep an eye on query latency, recall, and database resource usage to ensure optimal performance.