

Vectorized Database System Architecture Document

Abdallah Ahmed Ali

Ahmed Osama Helmy

Aliaa Abdelaziz

Omar Mahmoud Mohamed

Team: 10

27-10-2024

I. Introduction

This document outlines the initial design ideas for a scalable vector search system capable of handling a large number of high-dimensional vectors (Up to 20 million). The primary objective of this system is to support efficient storage and balance fast retrieval with acceptable accuracy for vectors in a high-dimensional space, with a focus on optimizing speed, memory efficiency, and the ability to scale efficiently.

In this initial phase, we explore various techniques, assess their strengths and limitations, and propose a system architecture that balances accuracy with computational efficiency. We will consider different approaches for storing the vectors & explore different indexing techniques for efficient similarity search to achieve our design objective.

The outcome of this document is to establish a foundation for the vector search system by selecting the most promising indexing approach and defining the storage and retrieval mechanisms to meet the project's performance requirements.

II. System Requirements and Constraints

Functional Requirements:

- **Data Storage:**

The system should be capable of storing a large number of high-dimensional vectors on which the search operation will be performed.

- **Similarity Search:**

The system must support similarity search based on cosine similarity, allowing retrieval of vectors that are closest to a given query vector.

- **Top-K Retrieval:**

For each query, the system should return the top (k) most similar vectors, where (k) is a variable parameter.

Non-Functional Requirements:

- **Memory Efficiency:**

Creating a balance between storage of the vectors on disk and loading vectors into main-memory as limited allowed ram usage is specified in the requirements, this will also prevent us from using certain types of indexes.

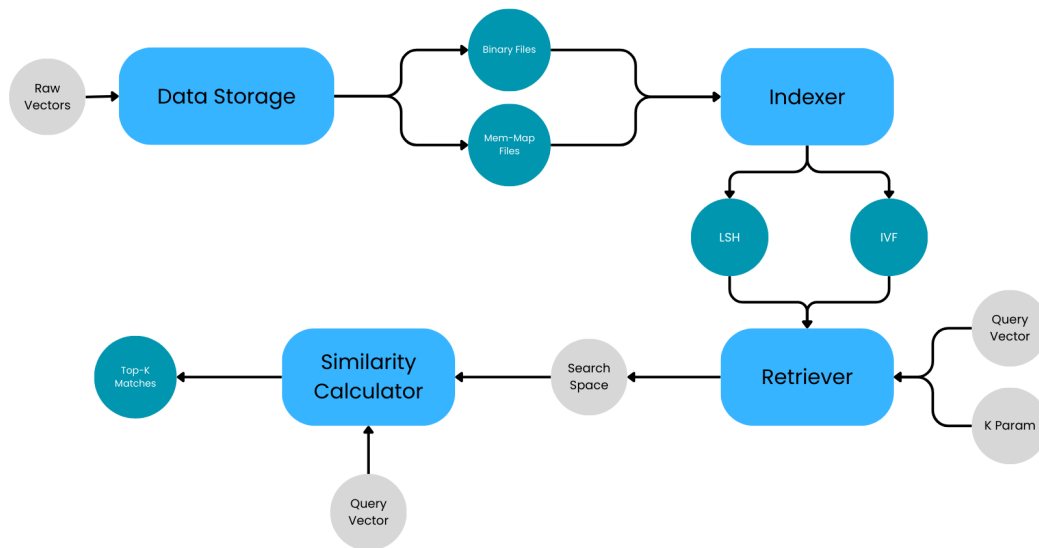
- **Accuracy:**

Being able to retrieve correct & approximately similar results is the main objective of this project.

- **Scalability:**

The system should be able to handle different amounts and sizes of data maintaining acceptable performance with the specified constraints.

III. System Architecture



1. Data Storage:

Data storage is the first critical step in designing a large-scale vector search system. Since the system is expected to handle up to 20 million high-dimensional vectors, choosing an efficient and scalable data storage approach is essential to ensure both performance and scalability.

This component is responsible for storing the raw vectors that will later be indexed and retrieved for similarity searches. Given the large volume of data, it's important to use methods that are both space-efficient and capable of fast data retrieval.

For data storage, the two primary approaches we are considering are **Binary Files** and **Memory-Mapped Files**. Both methods offer efficient, compact storage for handling large volumes of high-dimensional vector data. Initially, we plan to test the system's memory usage to determine if the entire dataset can fit within the available RAM.

If the dataset stays within memory limits, binary files provide a straightforward, high-speed storage solution. However, if testing reveals that we exceed RAM capacity, we will switch to memory-mapped files, which allow us to access data directly from disk without loading it all into memory. This flexibility ensures that we can adapt our storage strategy based on the system's requirements and the number of vectors at hand.

2. Indexing Strategy:

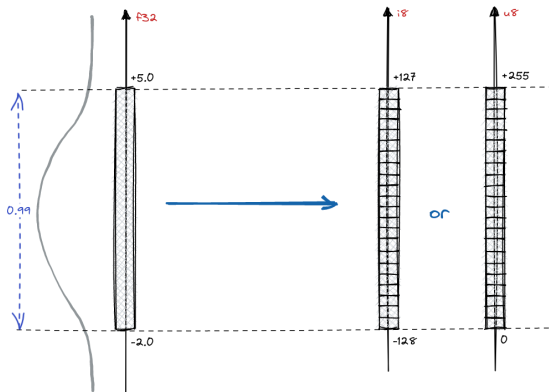
Indexing is the most critical module in this project, so we conducted in-depth research on various approaches. Here, we will outline each method considered, followed by a detailed description of our chosen approach, which combines elements from multiple methods to leverage the strengths of each.

2.1) Compression Algorithms

Used to compress index vectors to get efficient size, Mostly use Quantization (eg., convert from float representation to int etc...). There's 2 main algorithms (scalar quantization & product quantization). Note algorithms with flat in their name have no compression

*Algorithms usually do Convert **Database_Space** (column vectors) of d dimension to **Compressed_Space** of m where m is a hyperparameter that needs to be tuned of the algorithm.*

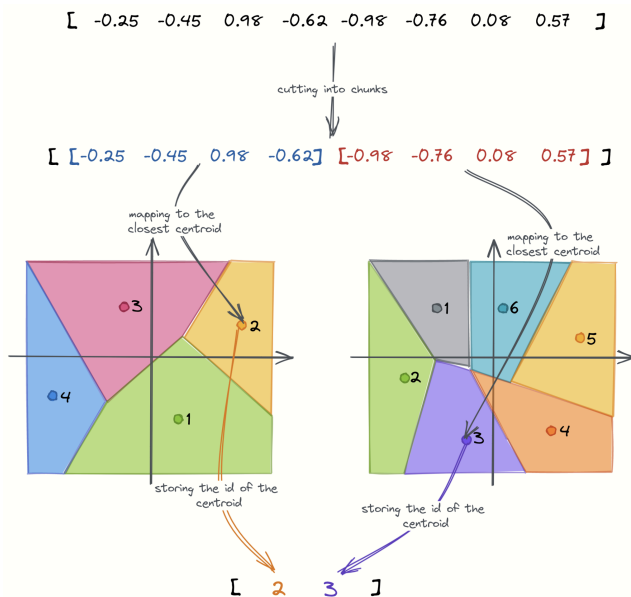
Scalar quantization: mostly used to convert float to int. it uses **min_max_normalization**



by doing 3 steps for each dimension:

1. Finding min and max
2. Finding start and step size
3. Subtracting start and divides by step

here's a reference image from [qdrant db](https://qdrant.github.io/qdrant/docs/concepts/quantization/)



Product quantization: provide more high-compression data, and overcome underutilized bins in Scalar Quantization as it considers distribution of sub-vectors. Start by deciding **m** then divide each vector into **m** sub-vectors.

Then for each sub-vector/chunk:

1. Run **K-means** usually $K=256$ as it's the max value on single byte
2. Assign whole subvector to one integer (index of closest cluster)

To calculate distance between vector and the Quantized index:

To get $d(\mathbf{v}, \mathbf{I})$ where \mathbf{v} is a query vector and \mathbf{I} is the quantized index.

We need first to divide \mathbf{v} into **m** sub-vectors $\mathbf{v} = [v_1, v_2, v_3 \dots v_m]$

Notice $\mathbf{I} = [c_1, c_2, c_3 \dots c_m]$ where \mathbf{c} is centroid index for sub-vector and \mathbf{C} is the centroid vector of size d/m


$$d(\mathbf{v}, \mathbf{I}) = d(v_1, c_1) + d(v_2, c_2) \dots + d(v_m, c_m)$$

2.2) Index Data Structure:

Usually in vector databases we want to get the nearest **K** vectors using a similarity measure, brute-force has become impossible in recent ages as data and embedding goes over 10s and 100s of millions of records. So it now becomes a problem of **ANNS** (approximate nearest neighbors search) where many algorithms become to introduced new tradeoffs between (Latency, Recall, Index Size, Query Throughput)

We can divide algorithms in the data stored by data-structured use we select most popular algorithms for each category:

1. Hash-Based (LSH / LSH-Tree)
2. Tree-Based (LSH-Tree, Annoy)
3. Graph-Based (HNSW, SPANN, VANMA)
4. Inverted Index Based (IVF, IVF-PQ, IMI)

 **Note:** there's also disk-based indexes where part of index stored on disk (eg, SPANN, VANMA); We will skip it as the project has a limit on RAM Size and assume index fit all in memory. We will also filtered some options as it use very large RAM mostly will be the purely Graph & Tree Based (eg, HNSW)

Graph-Based Algorithms:

Usually Graphs are very bad for memory, first it needs larger space and Second very Random Memory Access causes madness. We'll not talk to much here just brief on algorithms

HNSW (hierarchical navigable small world) which creates layers of Graphs where the above layer have vectors with distant similarities and each bottom layer gets more edges with more closer similarity and lesser node-degrees. The layers work as Skip list

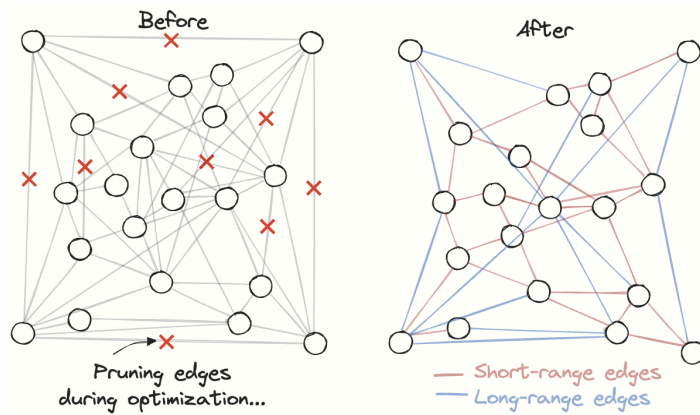
HNSW-PQ (HNSW + PQ) just using compression on nodes of graph

Both algorithms require high Memory Size will *probably exceed 2GB easily for 20 million records*

		Recall100@100	Latency (μ s)	Memory required (MB)
Sift1M Low params	Uncompressed	0.91561	293	1277
	Compressed	0.91361	401 (x1.36)	610 (47.76%)
Sift1M High params	Uncompressed	0.99974	1772	1674
	Compressed	0.99658	1937 (x1.09)	1478 (88.29%)

qHNSW (HNSW with disk storage with memory-mapped access.)
It addresses the memory issue created by HNSW.

Vanma: very new and based on Disk-indexing. It start with building random graph of all vectors then pruning edges using heuristic



Tree-Based Algorithms:

- **Annoy**
 - Annoy uses a forest of random projection trees to perform efficient approximate nearest neighbor search
 - **The algorithm works by:**
 - Projecting points onto random hyperplanes
 - Partitioning the space based on which side of the hyperplane the points fall on
 - Repeating this process recursively to create a binary tree of partitions
 - **Performing Query:**
 - Traverses down each tree in the forest to find the leaf node where the point would belong
 - Collects the points in the leaf nodes found across all tree
 - Returns the top-k points from this list that are closest to the query point

Hash-Based Algorithms:

LSH (Locality Sensitive Hashing) is the most popular ANN search technique which uses a hash-based algorithm and its idea is fairly simple as it only uses a common idea like hashing.

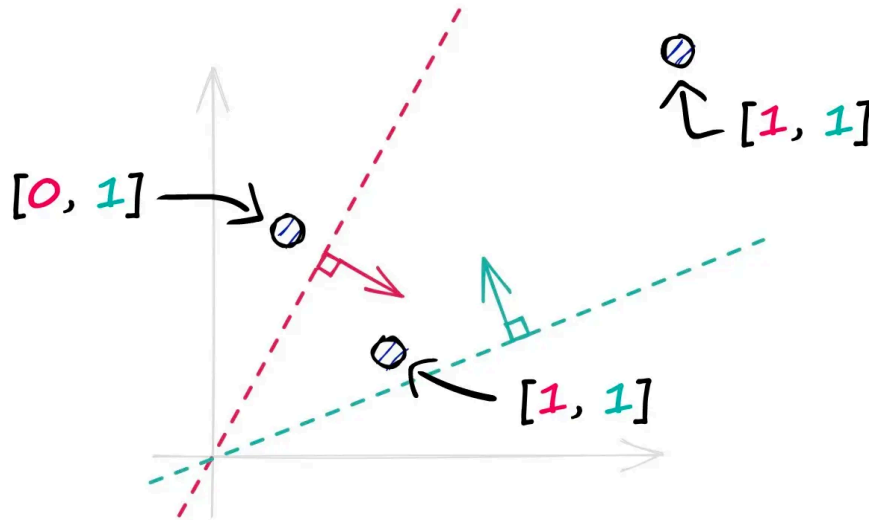
The two most popular approaches are:

- *Shingling, MinHashing, and banded LSH (traditional approach)*
- *Random hyperplanes with dot-product and Hamming distance (The one we will cover)*

The key difference which might seem weird when talking about hashing but makes a lot of sense in this topic is that in normal hashing we typically want to minimize collisions but here we rather want to maximize those collisions such that hopefully all similar vectors are mapped together to the same bucket and those dissimilar ones are mapped to different buckets.

This approach has a variety of different ways for implementation but we are going to focus on the approach which uses random hyperplanes generation to encode the position of data points in the space with respect to these planes which effectively generates regions which gives similar data points the same binary vector encoding and hence the same hash value.

The random hyperplane method for LSH involves:



1. Random Hyperplane Creation:

We generate N -Hyperplanes where N is the length of the binary vector which encodes the position of the data point in the space which means larger N means higher recall but also higher latency so this makes it a hyperparameter for the algorithm which we can tune to match our needs.

2. Binary Vector Encoding:

By using multiple hyperplanes, binary vectors are created that capture position information in lower dimensions. Using dot product to know the position of the data point relative to each hyperplane.

3. Hashing and Retrieval:

Query vectors are hashed similarly, and the Hamming distance between binary vectors identifies close matches, offering fast, approximate similarity search.

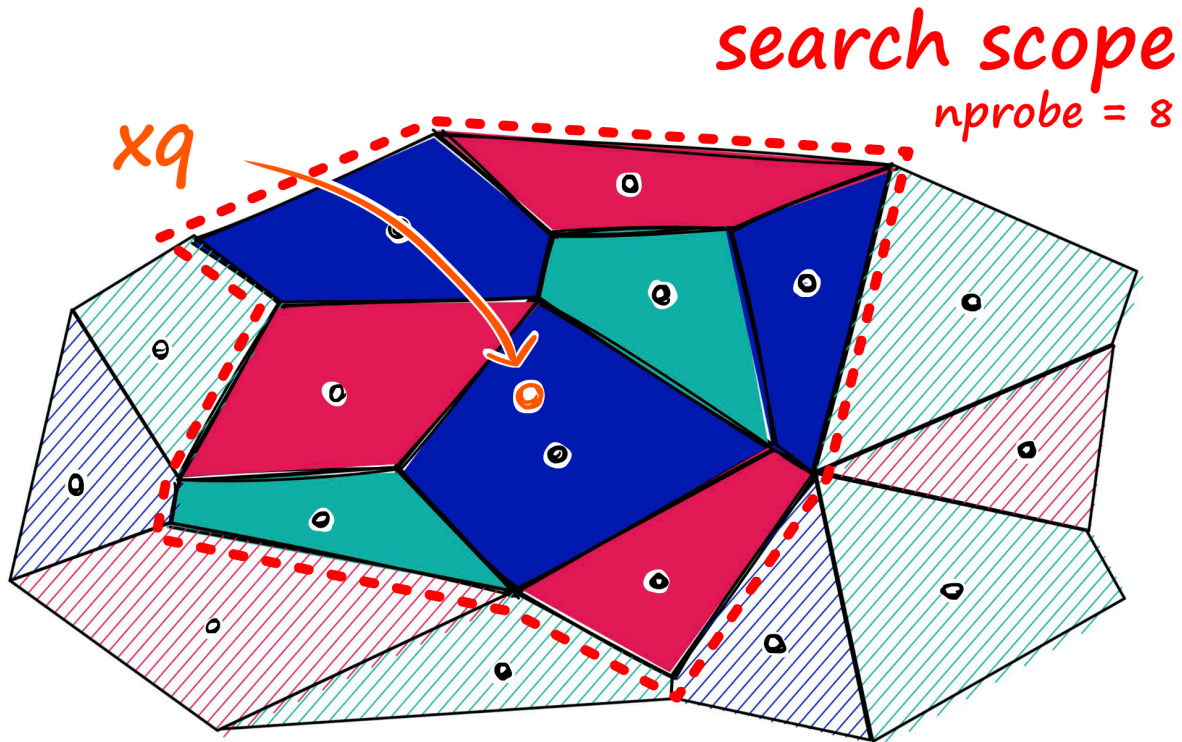
Locality Sensitive Hashing (LSH) is effective for approximate nearest neighbor searches, especially in high-dimensional data. However, it often requires a large N value, increasing memory and potentially slowing performance. At very high N values, the speed gains over a flat index can be marginal.

So LSH can be an effective approach for low dimensional data, or small datasets.

Inverted Index-Based Algorithms:

Inverted File Index (IVF) is one the simplest yet most effective methods for building efficient and scalable indexes.

It's built on the concept of voronoi diagrams (dirichlet tessellation) which is just a fancy way of saying that it's based on the k-means clustering algorithm.



So as the above figure shows we use the k-means algorithm where the K is our first hyperparameter here which will depend on the size of the dataset which encodes the number of clusters created where each centroid will act as a key for the vectors in its cluster.

Another hyperparameter to tune is n -probe which means the number of centroids to match the search vector which is directly proportional to the search quality but comes at the price of time & speed.

So we just retrieve these centroids and return the most similar vectors.

Finally let's take a look at how these two parameters (K,N) affect the performance of our system.

A higher **K** means that we must compare our vector to more centroid vectors — but after selecting the nearest centroid's cells to search, there will be fewer vectors within each cell. So, *increase K to prioritize search-speed.*

As for **N**, we find the opposite. *Increasing N increases the search scope — thus prioritizing search-quality.*

Comparing between different versions of IVF

Flat Index

- *The Flat index does not compress vectors*
- *Flat index can guarantee exact search results*
- *Flat index results can serve as a benchmark for evaluating other indexes that have less than 100% recall*

IVF_Flat Index

- *According to Milvus benchmark tests, the recall rate of the IVF_Flat index drops to 0.99 when compared to the Flat index*
- *IVF_Flat does not perform any compression on the vectors*
- *The index files produced by IVF_Flat are roughly the same size as the original, raw non-indexed vector data*
- *1B SIFT dataset that is **476 GB** in size will have an IVF_Flat index of around **470 GB** which means adding all the IVF_Flat index files into memory will consume approximately **470 GB** of storage, which is about **98.7%** of the original*

IVF_SQ8

- *Comparing it with the IVF Flat Index, the same dataset for the 1B SIFT dataset, the IVF_SQ8 index files require just **140 GB** of storage, which is about **30%** of the original.*

IVF_PQ

- *Comparing it with the IVF Flat Index, the same dataset for the 1B SIFT dataset, the IVF_SQ8 index files require just **140 GB** of storage, which is about **30%** of the original.*

Memory Comparison that was conducted between different versions of IVF

*A database with 200K vectors of 1280 dimensions was tested on different versions of IVF. Assuming the float is stored as 32-bit , the size of this database will be **0.95 GB**. As shown in the below table, the best option memory-wise is IVF-PQ. However, this is not taking into consideration the recall or accuracy percentage.*

	<i>Memory used in MB</i>	<i>Memory percentage of data</i>
<i>IVF_FLAT</i>	<i>983.04MB</i>	<i>100.6 %</i>
<i>IVF_SQ8</i>	<i>334.23MB</i>	<i>34.22%</i>
<i>IVF_PQ</i>	<i>13.10MB</i>	<i>1.34%</i>

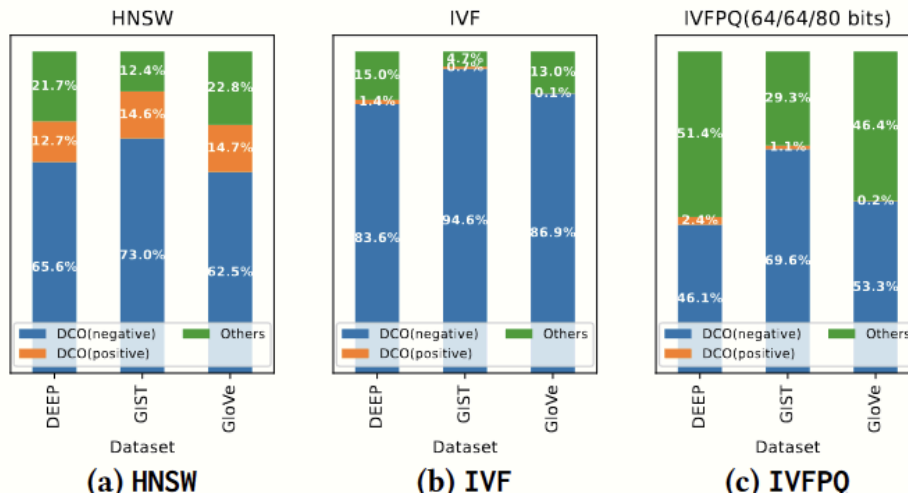
General comparison between the different approaches

<i>Approach</i>	<i>Speed</i>	<i>Accuracy</i>	<i>Memory Usage</i>
<i>Tree-Based</i>	<i>Efficient for low-mod dims Degrades in high dims</i>	<i>High in lower dims Diminishes in higher dims</i>	<i>Generally fast</i>
<i>Hash-Based</i>	<i>Fast indexing & querying</i>	<i>Lower due to collisions</i>	<i>Memory-efficient (codes only)</i>
<i>Graph-Based</i>	<i>Fast search (graph traversal)</i>	<i>High (greedy search)</i>	<i>Memory-intensive (graph structure)</i>
<i>Quantization-Based</i>	<i>Fast (compressed vectors)</i>	<i>High depends on codebook</i>	<i>Highly memory-efficient (compact codes)</i>

2.3) Distance Comparison Operations:

Which used to compute distance/similarity between vectors (eg., Cosine Similarity). There's another used Algorithm Called **ADSampling** used in **ChromaDB**

It simply say since most ANNS spend their time rejecting vectors so let's optimize rejection and make it in logarithmic time (it actually really really speed things up according to ChromaDB guy)



Algorithm 1: ADSampling

Input : A transformed data vector \mathbf{o}' , a transformed query vector \mathbf{q}' and a distance threshold r

Output: The result of DCO (i.e., whether $dis \leq r$): 1 means yes and 0 means no; In case of the result of 1, an exact distance is also returned

```

1 Initialize the number of sampled dimensions  $d$  to be 0
2 while  $d < D$  do
3   Sample some more dimensions  $y_i$  incrementally with
4      $y_i = \mathbf{o}'_i - \mathbf{q}'_i$ 
5   Update  $d$  and the approximate distance  $dis'$  accordingly
6   Conduct a hypothesis testing with the null hypothesis
7      $H_0$  as  $dis \leq r$  based on the approximate distance  $dis'$ 
8   if  $H_0$  is rejected and  $d < D$  then // Case 1
9     return 0
10  else if  $H_0$  is not rejected and  $d < D$  then // Case 2
11    continue
12  else // Case 3
13    return 1 (and  $dis'$ ) if  $dis' \leq r$  and 0 otherwise

```

Failure Probability Analysis. Note that ADSampling terminates in either Case 1 (with the hypothesis being rejected and $d < D$) or

We will added if we have time or we needed it (as the paper very mathy I am not going to document much)

Our Approach:

Now after highlighting the different available approaches and where each approach shines and what each of them lacks we decided that our approach is going to be a mixture of these,

combining them into what's known as a composite index utilizing the strengths of each of these algorithms which is similar to the approach a lot of the current popular vector database systems use.

Our indexing pipeline is going to include these main components:

1. **Vector transform** — a pre-processing step applied to vectors before indexing (PCA, OPQ).
2. **Coarse quantizer** — *rough* organization of vectors to sub-domains (for restricting search scope, includes IVF, IVF-PQ and HNSW).
 - We are likely to use IVF here but this is prone to changes
3. **Fine quantizer** — a *finer* compression of vectors into smaller domains (for compressing index size, such as PQ).
 - This is likely to be used if we used HNSW or another approach where the size of the index itself is large
4. **Refinement** — *a final step at search-time which re-orders results using distance calculations on the original flat vectors.*
 - *Re-order results based on their original flat vectors (RFlat)*

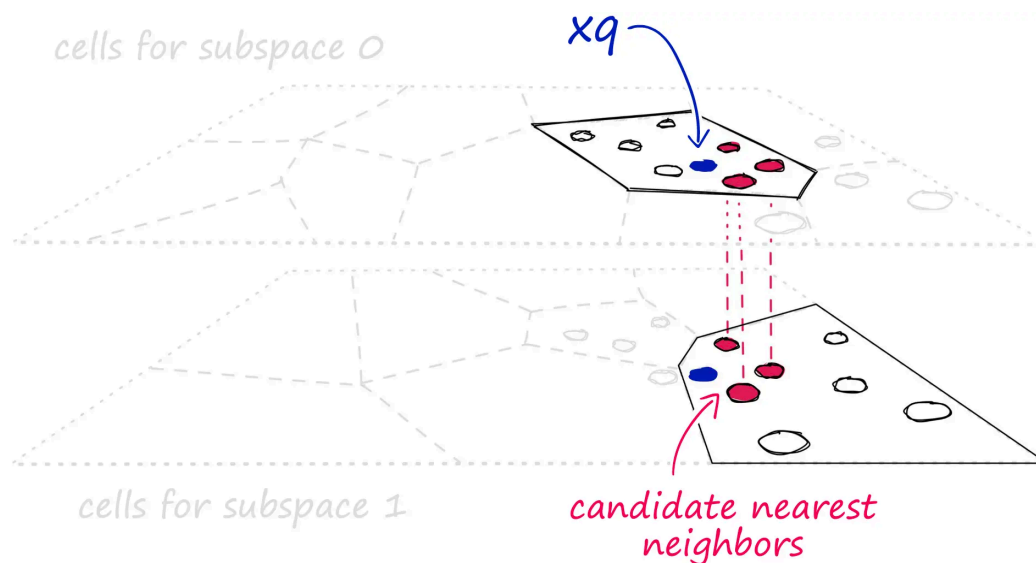
We will be considering the upcoming two indexing approaches (shown in the next pages), though our final choice will depend on upcoming resource allocations and performance testing. While both methods offer unique advantages in terms of speed and recall, we will wait for further information on available resources to determine which approach aligns best with our limitations and project goals. Once resources are confirmed, we'll conduct tests to assess how each approach performs under our specific constraints, allowing us to make an informed decision.

We are going to consider these candidate implementations in more detail:

1. Multi-D-ADC:

The main indexing strategy in this approach is IMI (Inverted Multi-Dimensional Index) hence the Multi-D in the name of the approach, this is quite similar to the IVF indexing explained above but with a little twist where we have two subspaces in which we cluster our vectors rather than a single subspace.

The figure below shows the info mentioned in the previous paragraph:



Now for the ADC part which stands for Asymmetric Distance Comparison where the main distinction between Asymmetric Distance Computation (ADC) and Symmetric Distance Computation (SDC) lies in the handling of the query and database vectors during distance calculations:

- **Symmetric Distance Computation (SDC):** Both the query vector and the dataset vectors are compressed or quantized. The distance is then calculated between the compressed representations of both, which saves memory but can reduce precision because both vectors are approximated.
- **Asymmetric Distance Computation (ADC):** The query vector remains in its original, uncompressed form, while only the dataset vectors are quantized. This approach allows for more accurate distance measurements between the query and the approximate vectors in the index.

This spoils the final component of this approach which is quantization for which we decided to use Product Quantization (PQ) explained above to compress the vectors in our index.

A rather minor yet effective modification is our choice to use Optimized Product Quantization (OPQ) in the vector transformation as our initial step in indexing, which has proven to speed up this approach by nearly 100%.

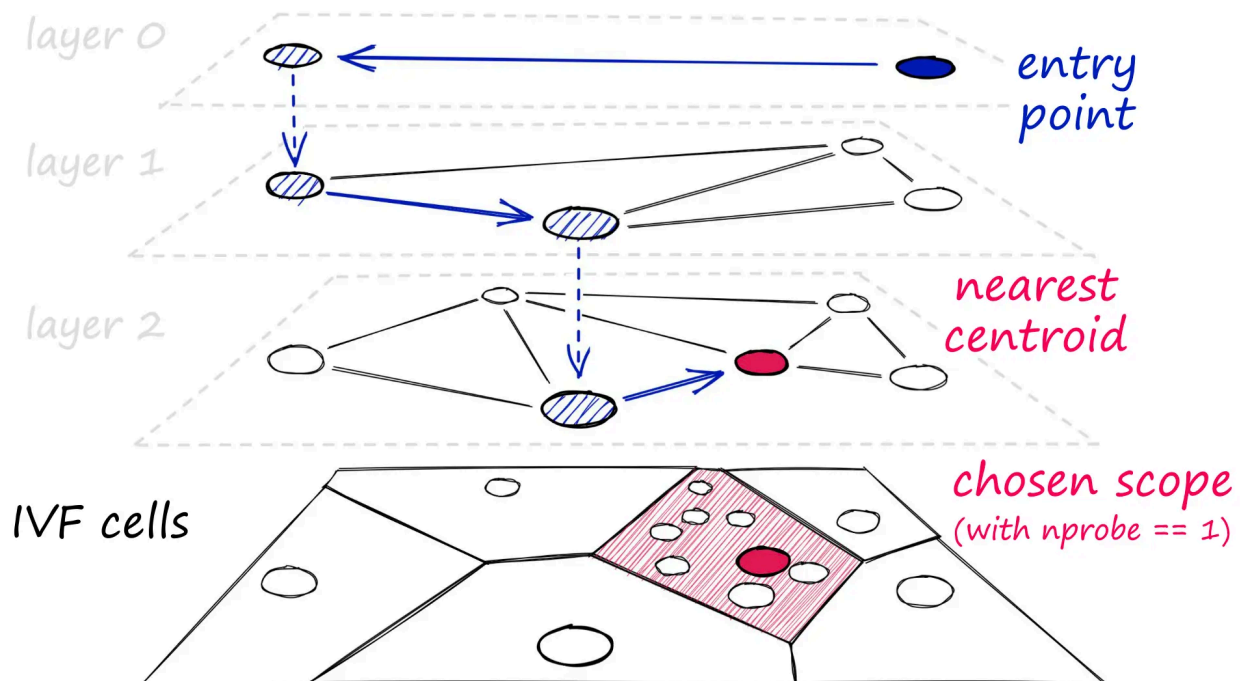
A paper detailing Optimized Product Quantization (OPQ) is attached [here](#), as we haven't yet explored how its rotation-based transformation reduces quantization error and enhances retrieval accuracy.

2. IVF-HNSW:

This is the second design for our index which we are considering as an alternative for the previous approach but this will depend on the limitations on the resource-usage yet to be announced.

The IVF with Hierarchical Navigable Small-World (HNSW) graph is our final composite index, combining IVF's cell-based partitioning with the efficiency of HNSW for faster and more accurate searches. In this setup, vectors are organized into cells using IVF, and HNSW is applied to optimize the traversal across these cells. The HNSW component leverages small-world graph theory, where nodes are traversable in a few steps, creating faster search paths by utilizing long-range links at higher levels of the graph and short-range links at deeper levels for finer search resolution.

This is shown in this figure:



HNSW's layered structure improves speed and recall compared to standard IVF by allowing efficient approximation without exhaustive centroid comparisons. Instead of directly comparing a query vector to each cell centroid in an exhaustive search, HNSW approximates this by creating a graph over centroids and narrowing the search space efficiently. This method is particularly useful for larger indexes, where standard IVF with only 256 centroids (common number used with the SIFT-1M Dataset seen in many implementations) might limit performance due to large vector counts per cell, making it challenging to achieve high recall.

3. IVF-PQ

In IVF-PQ, an Inverted File index (IVF) is integrated with Product Quantization (PQ) to facilitate a rapid and effective approximate nearest neighbor search by initial broad-stroke that narrows down the scope of vectors in our search.

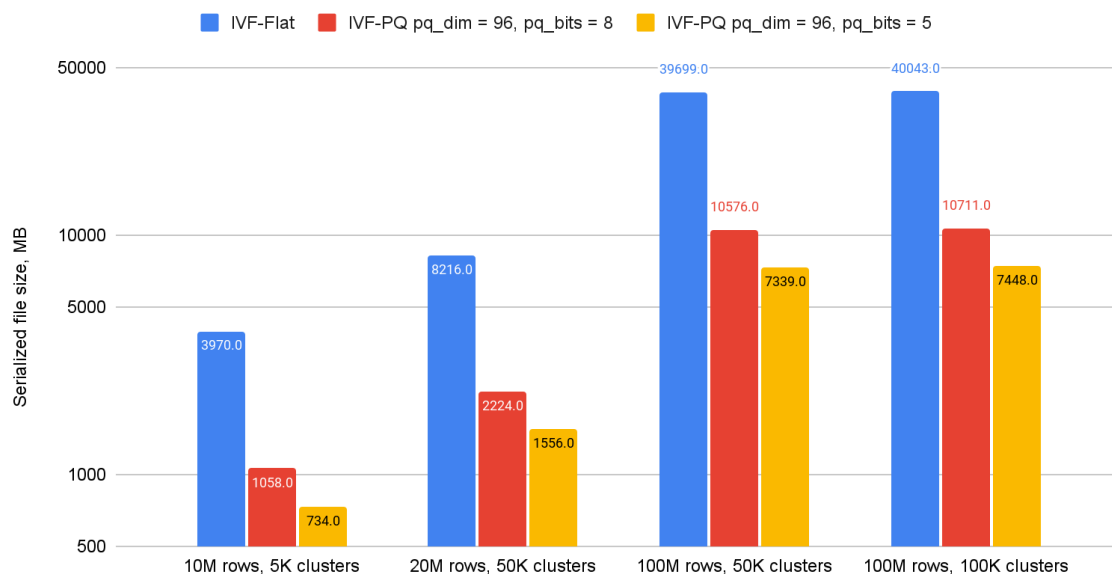
When we introduce our query vector, it restricts our search to the nearest cells only because of which searching becomes way faster compared to PQ.

*In the IVF-PQ index, a database vector y is approximated with **two-level** quantization. The first-level quantizer maps the vector y to the nearest cluster center. In this case, IVF-PQ employs the same balanced hierarchical k -means algorithm as IVF-Flat. The second-level quantizer is called the product quantizer and it encodes the residual, or distance to the closest cluster center.*

The chart below shows the results of tests performed on this

IVF-Flat vs IVF-PQ: index size

DEEP dataset, H100 PCIe GPU

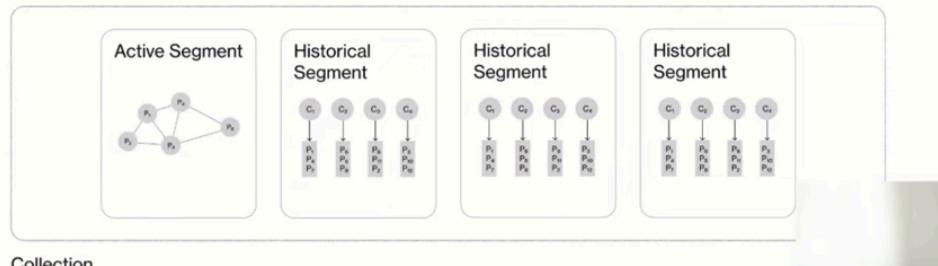


Case-Study (ChromaDB)













Chroma Index



- Divide data into realtime and historical segments
- Index realtime data into HNSW
- Compact historical data into Inverted indices SPANN style and query with ADSampling
 - Store centroids in HNSW
 - Posting lists can be stored separately and allow for separation of storage and compute



Market Study

 Pinecone	Proprietary composite index
 milvus /  zilliz	Flat, Annoy, IVF, HNSW/RHNSW (Flat/PQ), DiskANN
 Weaviate	Customized HNSW, HNSW (PQ), DiskANN (in progress...)
 drant	Customized HNSW
 chroma	HNSW
 LanceDB	IVF (PQ), DiskANN (in progress...)
 vespa	HNSW + BM25 hybrid
 Vald	NGT
 elasticsearch	Flat (brute force), HNSW
 redis	Flat (brute force), HNSW
 pgvector	IVF (Flat), IVF (PQ) in progress...

References

- [Chroma Vector Database: Retrieval for LLMs \(Hammad Bashir + Liquan Pei\)](#)
- [Pinecone Tutorial on FAISS: Hierarchical Navigable Small Worlds \(HNSW\)](#)
- [Random Projection For Locality Sensitive Hashing](#)
- [High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations](#)
- [Data Quarry Blog: Vector databases \(3\): Not all indexes are created equal](#)
- [Weaviate Blog: HNSW+PQ - Exploring ANN algorithms Part 2.1](#)
- [Product Quantization in Vector Search | Qdrant](#)
- [Optimized Product Quantization \(OPQ\) | Microsoft](#)
- [Scalar Quantization: Background, Practices & More | Qdrant](#)
- [Pinecone IVF Tutorial Using FAISS](#)
- [Accelerating Similarity Search on Really Big Data with Vector Indexing](#)
- [Milvus IVF_SQ8 Memory Usage Check](#)
- [A Comprehensive Guide on Indexing Algorithms in Vector Databases](#)
- [Accelerating Vector Search: NVIDIA cuVS IVF-PQ Part 1, Deep Dive](#)