Database (Part 4)

Vipra Singh · Follow

21 min read · Jan 20, 2024

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

Posts in this Series

1. *Introduction*
2. *Data Preparation*
3. *Sentence Transformers*
4. *Vector Database (This Post)*
5. *Search & Retrieval*
6. *LLM*
7. *Open-Source RAG*

8. *Evaluation*
9. *Serving LLMs*
10. *Advanced RAG*

. . .

Table Of Contents

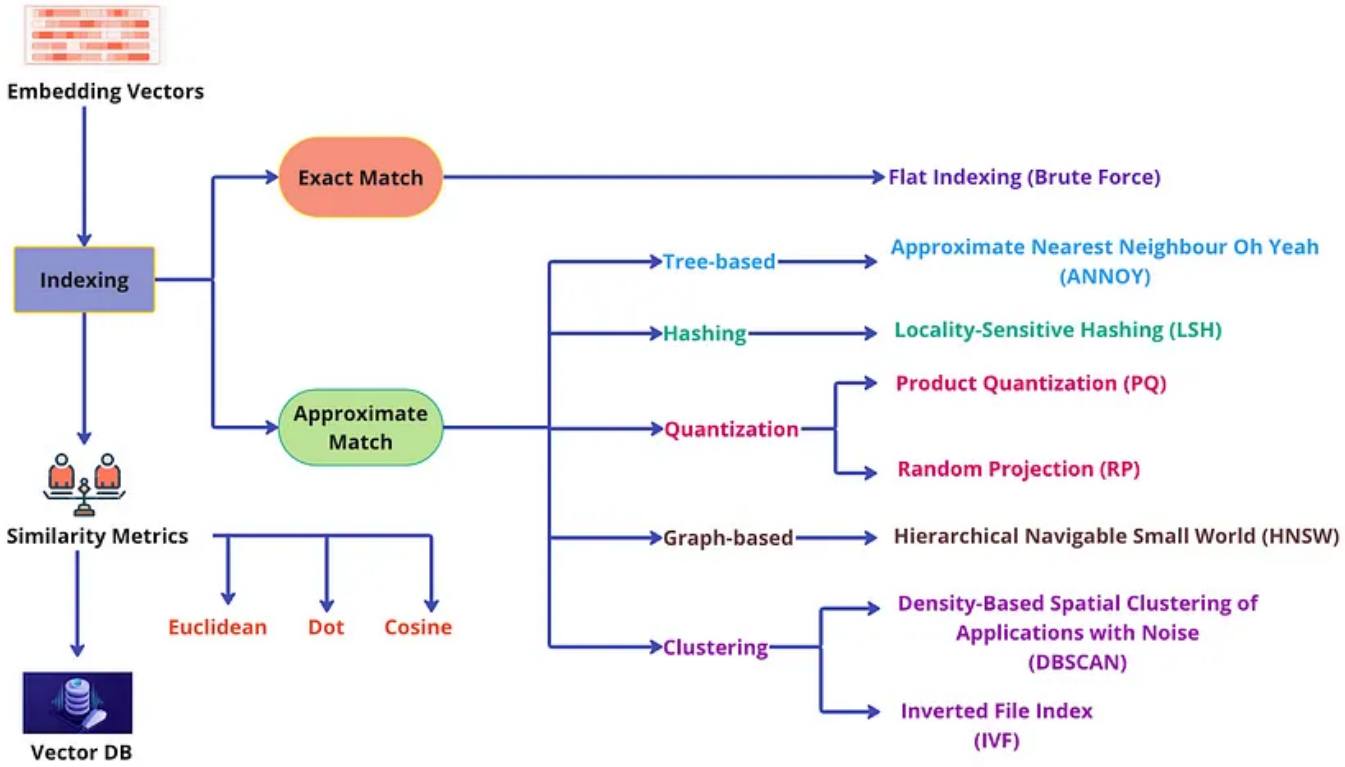


Image by Author

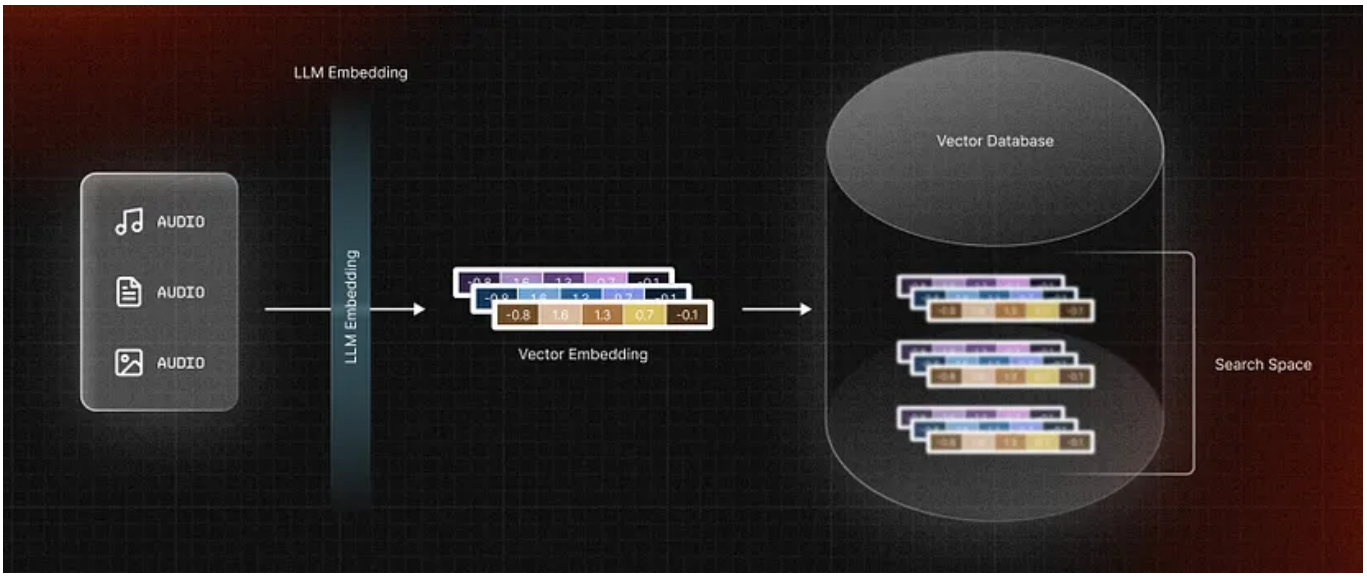
1. Introduction

In the previous blogs, we have covered till embedding of the raw data into vectors. To use the embedded information repeatedly, we need to store the embeddings so that they can be accessed on demand. For this purpose, we use a special kind of database called a **vector database.**

Efficient storage and retrieval of embeddings with capabilities like CRUD operations, metadata filtering, and horizontal scaling are crucial for large-scale applications using Retrieval-Augmented Generation(RAGs). Vector databases like ChromaDB, Pinecone, and Weaviate specialize in this, offering fast retrieval and similarity searches.

Integrating the right vector database is crucial for maximizing RAG performance. A thoughtful selection, considering use case intricacies, ensures seamless storage and retrieval, optimizing the power of Retrieval-Augmented Generative models.

In this blog, we will deep dive into Vector Databases & Indexing methods.



Vector DB

2. Vector Database Vs. Other Databases

	Vector Database	Relational Database	NoSQL Database	Graph Database
Data Structure	*Vectors*	Tables	Key-value pairs, Documents, Wide-column, Graph	Graph
Data Model	*Vector Space Model*	Relational Model	Various models depending on the type (e.g., key-value, document, columnar)	Graph Model
Query Language	*Similarity-based queries*	SQL (Structured Query Language)	Query language specific to the database (e.g., MongoDB Query Language for MongoDB)	Graph query languages (e.g., Cypher for Neo4j)
Schema	*Schema-less*	Schema-based	Schema-less or flexible schema	Schema-less or flexible schema
Horizontal Partitioning	✔	✔	✔	✔
ACID Compliance	*Not usually*	Usually	Not usually	Not usually
Data Relationships	*N/A*	Defined by relations/foreign keys	N/A	Modeled using edges and nodes
Performance	*High*	High	High	Performance varies based on the graph complexity
Use Cases	*Similarity search, recommendation systems, machine learning*	Traditional business applications, transactional systems	Big data, real-time applications, unstructured data	Social networks, fraud detection, recommendation engines, knowledge graphs

Comparison of Databases

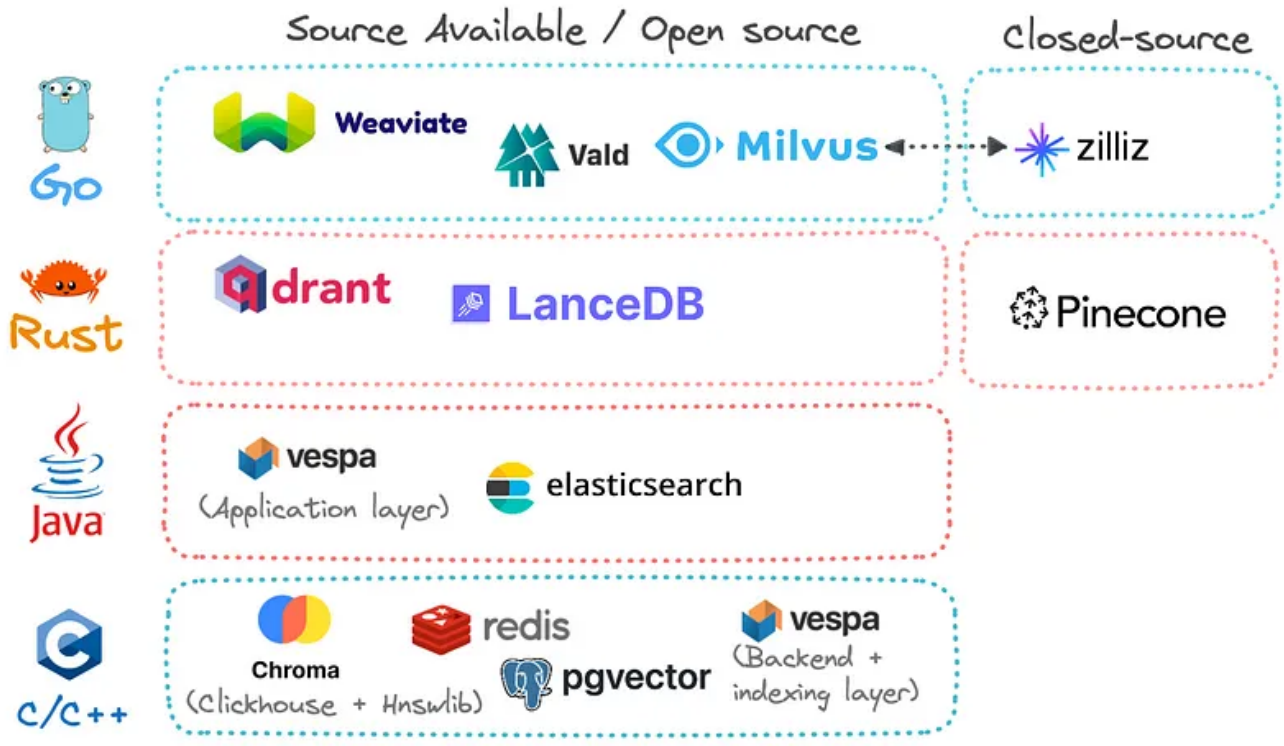
3. Vector Database Vs. Vector Index

In the tech industry, a prevailing misconception suggests that vector databases are simply wrappers for Approximate Nearest Neighbor (ANN) search algorithms.

At its core, a **vector database** is a comprehensive solution for unstructured data. Contrary to this misconception, it encompasses user-friendly features found in today's structured/semi-structured database management systems, including **cloud-nativity, multi-tenancy, and scalability**. It tackles the limitations of standalone vector indices, addressing scalability challenges, integration complexities, and the lack of real-time updates and built-in security measures. This becomes evident as we delve deeper into this tutorial.

On the other side, lightweight ANN libraries like **FAISS** and **ScaNN** serve as tools to construct vector indices. These libraries aim to accelerate nearest-neighbor searches for multi-dimensional vectors. While suitable for small datasets in production systems, scalability becomes a challenge as datasets grow.

4. Popular Vector Databases



Popular Vector Databases

4.1. How does a vector database work?

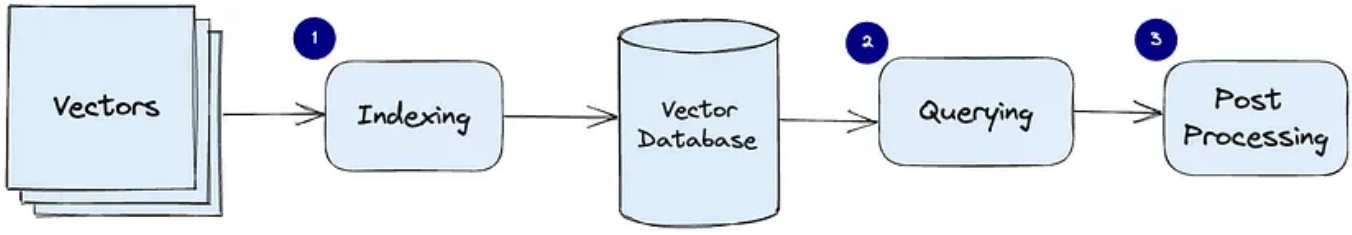
We all know traditional databases store strings, numbers, and other types of **scalar** data in rows and columns. On the other hand, a vector database operates on **vectors**, so the way it's optimized and queried is quite different.

In traditional databases, we are usually querying for rows in the database where the value usually exactly matches our query. In vector databases, we apply a similarity metric to find a vector that is the **most similar** to our query.

A vector database uses a combination of different algorithms that all participate in the Approximate Nearest Neighbor (ANN) search. These algorithms optimize the search through various indexing techniques.

These algorithms are assembled into a pipeline that provides fast and accurate retrieval of the neighbors of a queried vector. Since the vector database provides **approximate** results, the main trade-offs we consider are between accuracy and speed. The more accurate the result, the slower the query will be. However, a good system can provide ultra-fast search with near-perfect accuracy.

Here's a common pipeline for a vector database:



By Pinecone

1. **Indexing:** The vector database indexes vectors using an algorithm such as PQ, LSH, or HNSW (find more on these below). This step maps the vectors to a data structure that will enable faster searching.
2. **Querying:** The vector database compares the indexed query vector to the indexed vectors in the dataset to find the nearest neighbors (applying a similarity metric used by that index)
3. **Post Processing:** In some cases, the vector database retrieves the final nearest neighbors from the dataset and post-processes them to return the final results. This step can include re-ranking the nearest neighbors using a different similarity measure.

In the following sections, we will discuss each of these algorithms in more detail and explain how they contribute to the overall performance of a vector database.

5. Indexing Techniques

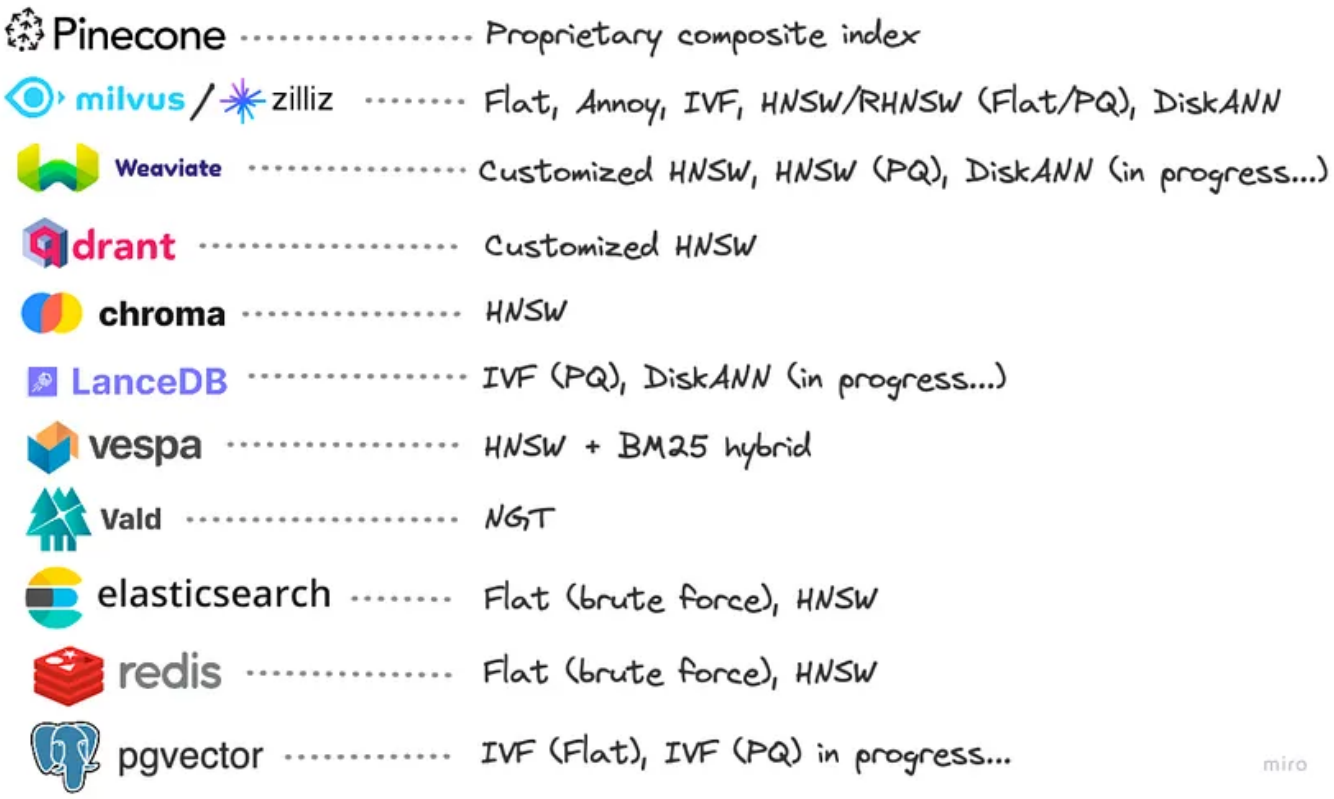
Tree-based methods are efficient for low-dimensional data and offer exact nearest neighbor search. However, their performance generally degrades in high-dimensional spaces due to the “curse of dimensionality”. They also require substantial memory and are less efficient for large datasets, leading to longer build times and higher latency.

Quantization methods are memory-efficient and offer fast search times by compressing vectors into compact codes. However, this compression can lead to a loss of information, potentially reducing search accuracy. Additionally, these methods can be computationally expensive during the training phase, increasing build time.

Hashing methods are fast and relatively memory-efficient, mapping similar vectors to the same hash bucket. They perform well with high-dimensional data and large-scale datasets, offering high throughput; However, the quality of search results can be lower due to hash collisions leading to false positives and negatives. Choosing the right number of hash functions and the number of hash tables are imperative as they significantly affect performance.

Clustering methods can expedite search operations by narrowing down the search space to a specific cluster, but the accuracy of the search results can be influenced by the quality of the clustering. Clustering is typically a batch process, which means it's not well-suited to dynamic data where new vectors are constantly being added, leading to frequent re-indexing.

Graph methods provide a good balance between accuracy and speed. They are efficient for high-dimensional data and can provide high-quality search results. However, they can be memory-intensive, as they need to store the graph structure, and the construction of the graph can also be computationally expensive.

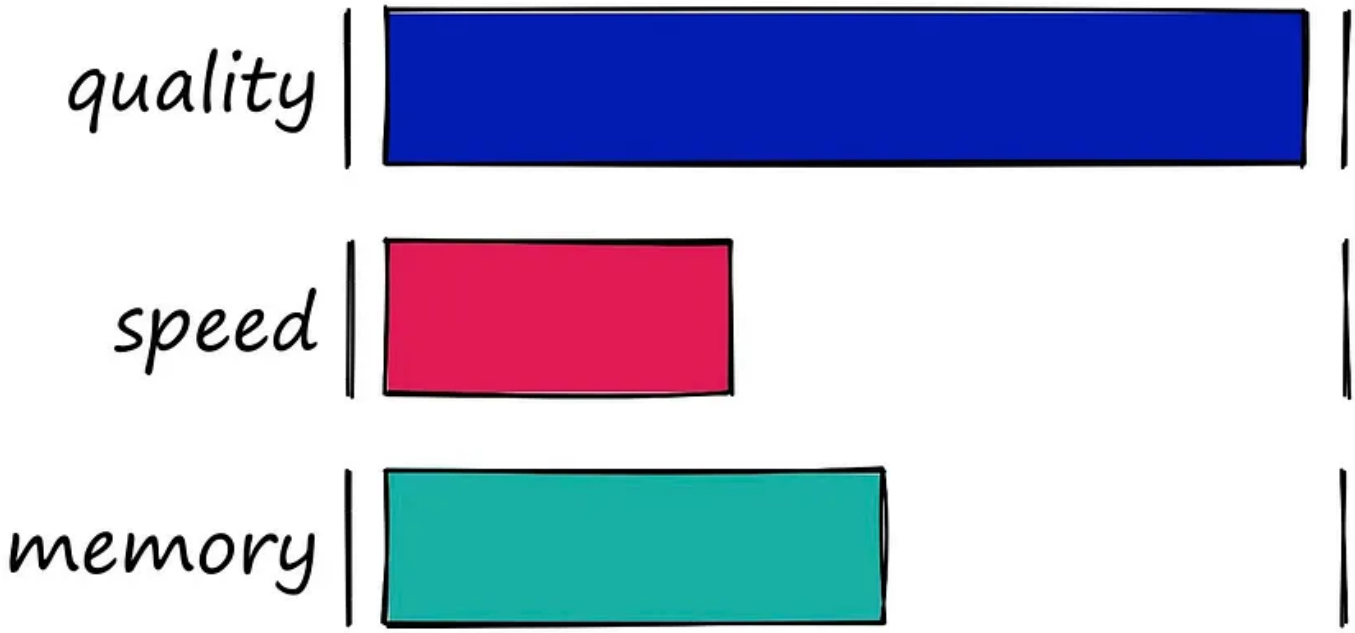


Various Databases with Indexing Methods

The choice of algorithm in a vector database depends on the specific requirements of the task, including the size and dimensionality of the dataset, the available computational resources, and the acceptable trade-off between accuracy and efficiency. It's also worth noting that many modern vector databases use hybrid methods, combining the strengths of different methods to achieve high speed and accuracy. Understanding these algorithms and their trade-offs is crucial for anyone looking to get the best performance out of their vector database. Now let's take a look at all the popular algorithms in each one of these categories.

6. Exact Match

6.1. Flat Indexing (Brute Force)



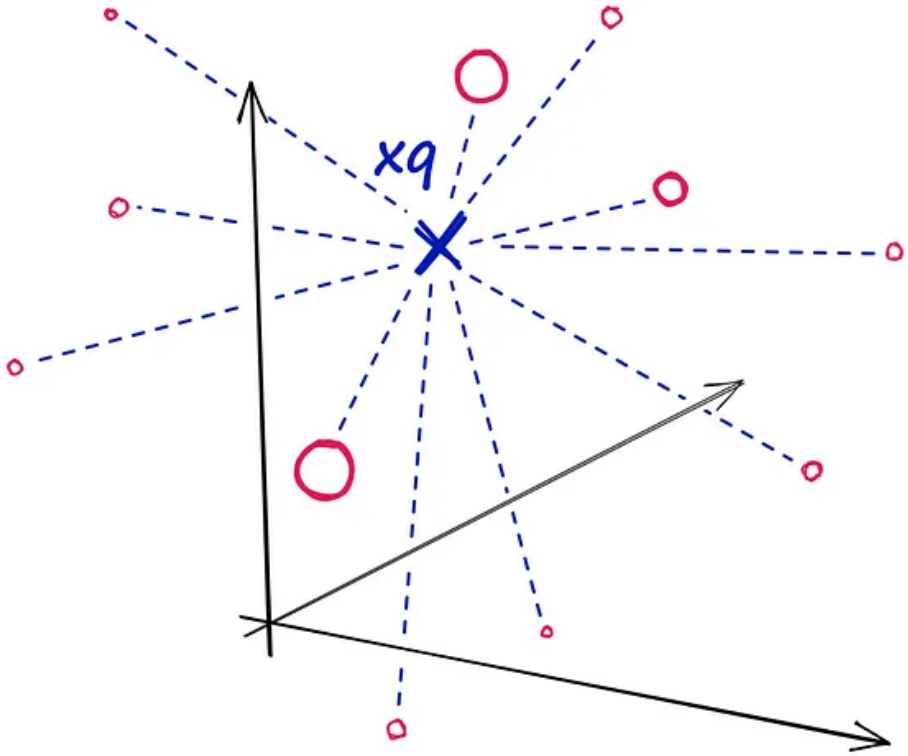
Flat indexes come with perfect search quality at the cost of slow search speeds. Memory utilization of flat indexes is reasonable.

The very first indexes we should look at are the simplest — flat indexes.

Flat indexes are 'flat' because we do not modify the vectors that we feed into them.

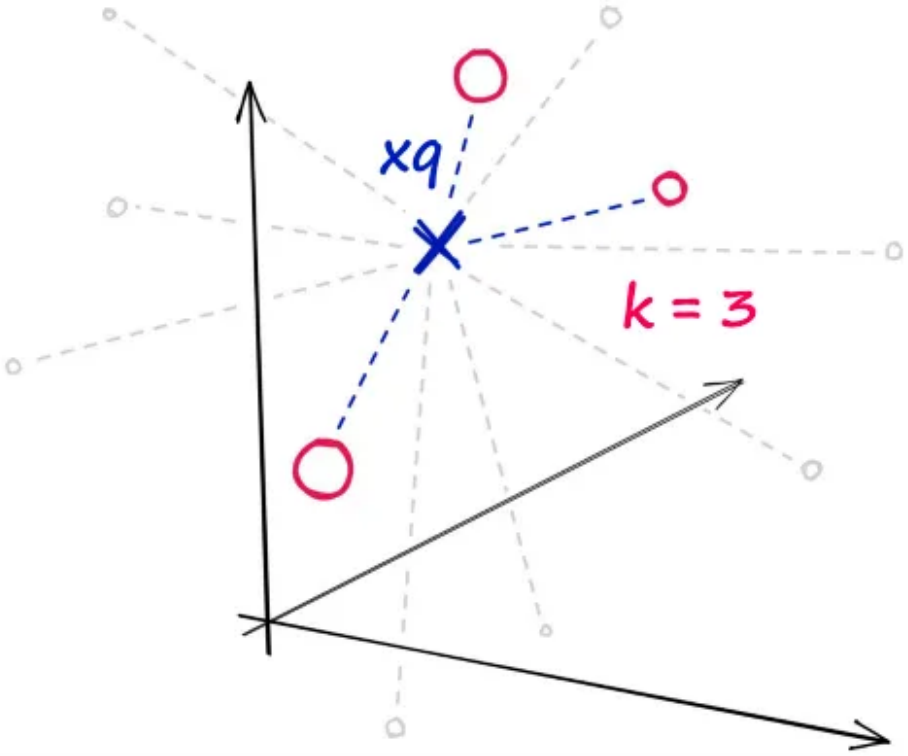
Because there is no approximation or clustering of our vectors — these indexes produce the most accurate results. We have perfect search quality, but this comes at the cost of significant search times.

With flat indexes, we introduce our query vector **xq** and compare it against every other full-size vector in our index — calculating the distance to each.



With flat indexes, we compare our search query xq to every other vector in the index.

After calculating all of these distances, we will return the nearest k of those as our nearest matches. A k-nearest neighbors (kNN) search.

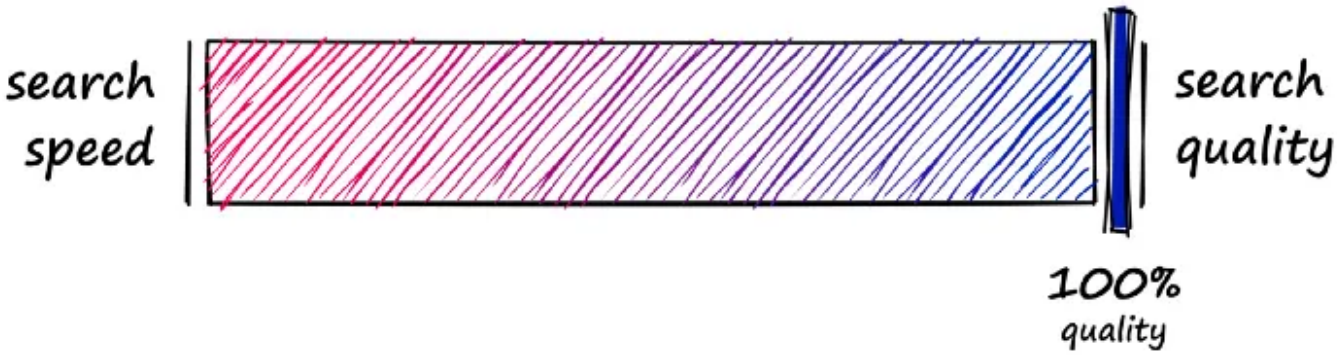


Once we have calculated all of the distances, we return the k nearest vectors.

6.2. Balancing Search Time

Flat indexes are brilliantly accurate but terribly slow. In similarity search, there is always a trade-off between search-speed and search-quality (accuracy).

What we must do is identify where our use-case sweet spot lies. With flat indexes, we are here:



Flat indexes are 100% search-quality, 0% search-speed.

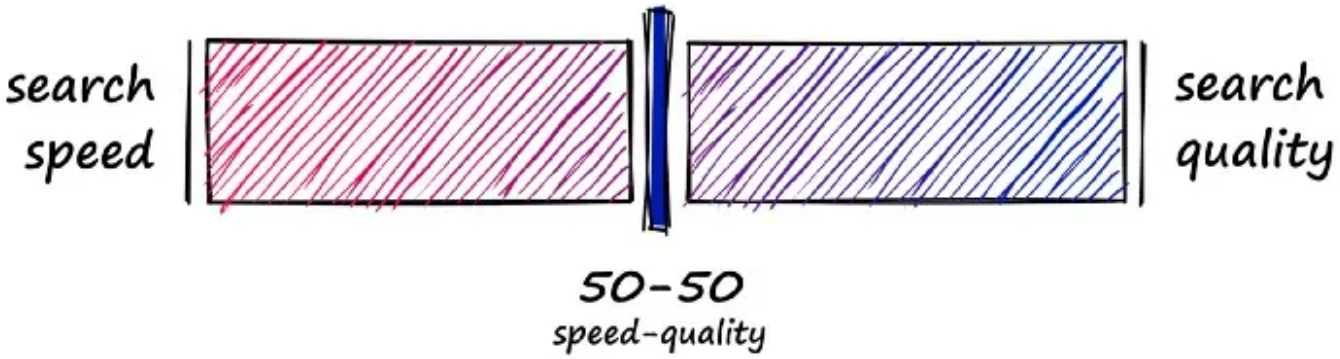
Here we have completely unoptimized search-speeds, which will fit many smaller index use-cases — or scenarios where search-time is irrelevant. But, other use-cases require a better balance speed and quality.

So, how can we make our search faster? There are two primary approaches:

1. Reduce vector size — through dimensionality reduction or reducing the number of bits representing our vectors values.
2. Reduce search scope — we can do this by clustering or organizing vectors into tree structures based on certain attributes, similarity, or distance — and restricting our search to closest clusters or filter through most similar branches.

Using either of these approaches means that we are no longer performing an exhaustive nearest-neighbors search but an approximate nearest-neighbors (ANN) search — as we no longer search the entire, full-resolution dataset.

So, what we produce is a more balanced mix that prioritizes both search-speed and search-time:



Often, we will want a more balanced mix of both search-speed and search-quality.

Often, we will want a more balanced mix of both search-speed and search-quality.

7. Approximate Match

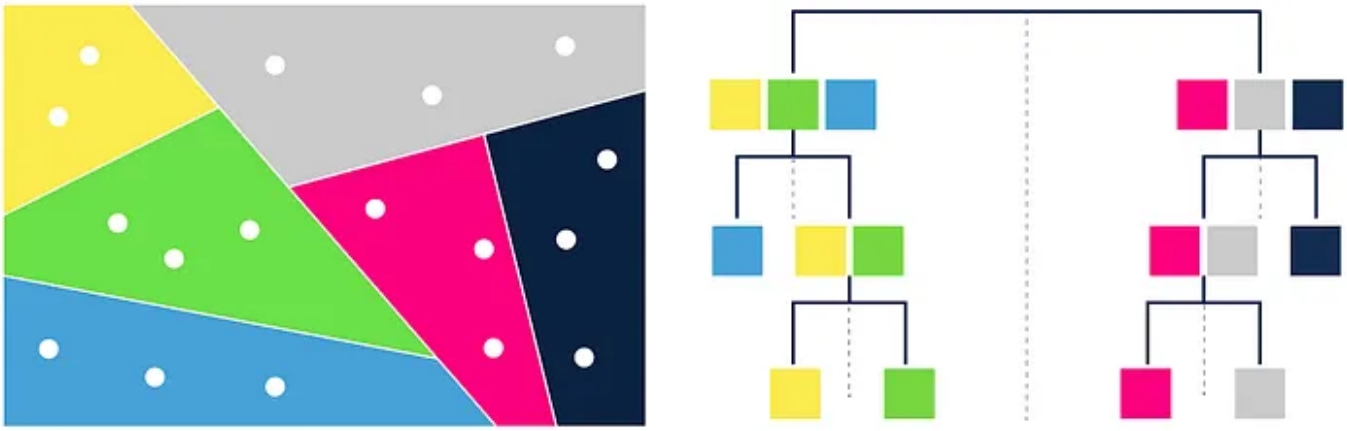
7.1. Annoy (Approximate Nearest Neighbor Oh Yeah)

Among the **tree-based algorithms** used in vector databases, one stands out for its efficiency and simplicity: Annoy, or "Approximate Nearest Neighbors Oh Yeah". **Annoy** is a C++ library with Python bindings that is designed by Spotify to search for points in space that are close to a given query point(**approximate match**). It creates large read-only, file-based data structures that are memory-mapped into memory, which allows multiple processes to share the same data.

Annoy operates by using a forest of **Random Projection** trees to perform an efficient ANN search. The algorithm projects points onto random hyperplanes and partitions the space according to which side of the hyperplane the points fall on. This process is repeated recursively, resulting in a binary tree of partitions. A forest of trees is created each with a different random seed. When a query point is received, the algorithm traverses down each tree in the forest to find the leaf node where the point would belong. The nearest neighbors are then approximated by collecting a list of points in the leaf nodes found in all trees and returning the top-k points from this list that are closest to the query point.

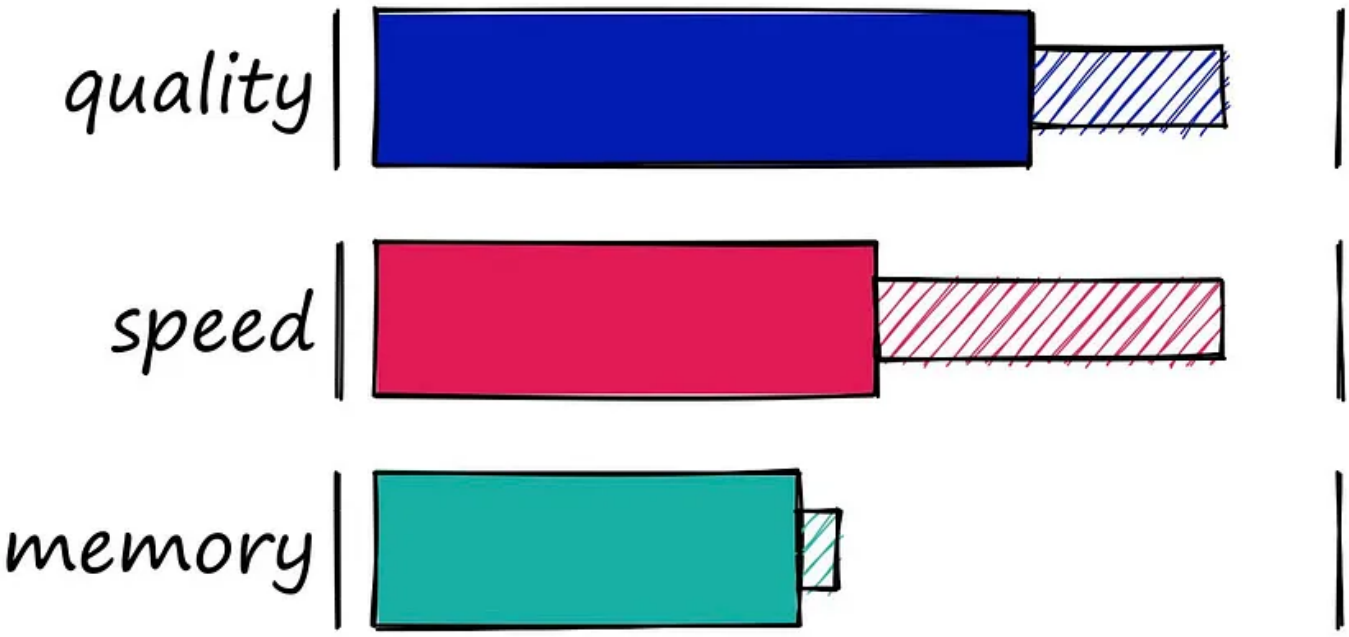
Annoy is particularly well-suited for high-dimensional data, where an exact nearest-neighbor search can be prohibitively expensive. It is used in production at Spotify for music recommendations, where it helps find similar songs based on their audio features. Thus, accuracy can be influenced by the number of trees in the forest and the number of points inspected during the search, both of which can be tuned and adjusted based on the specific requirements of the task.

The efficiency and memory efficiency of Annoy makes it a strong choice for handling high-dimensional data and large databases. There are a few trade-offs to consider. Building the index can take a significant amount of time, especially for large datasets. Because Annoy uses a random forest partitioning algorithm, the indices cannot be updated with new data and must be rebuilt from scratch. Depending on the size of your data set or how often your data changes, retraining the index might be prohibitively expensive.



ANNOY

7.2. Inverted File (IVF) Indexing



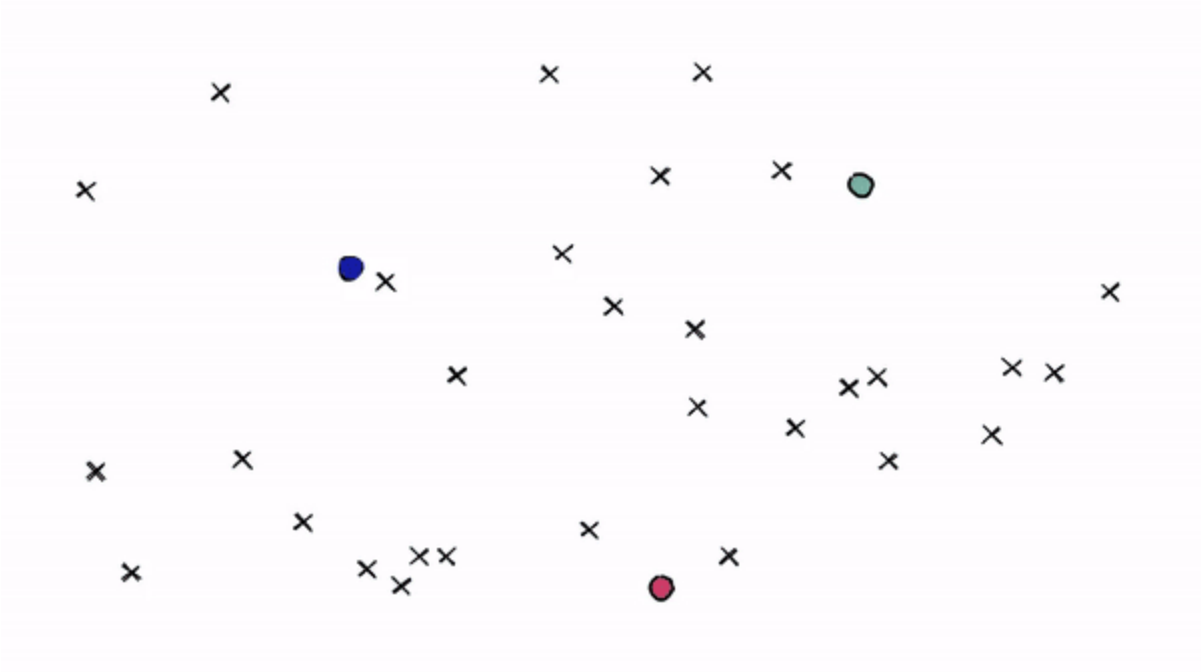
IVF — great search-quality, good search-speed, and reasonable memory usage. The 'half-filled' segments of the bars represent the range in performance encountered while modifying index parameters.

The Inverted File Index (IVF) index consists of search scope reduction through clustering. It's a very popular index as it's easy to use, with high search quality and reasonable search speed.

It works on the concept of Voronoi diagrams — also called Dirichlet tessellation (a much cooler name).

To understand Voronoi diagrams, we need to imagine our highly-dimensional vectors placed into a 2D space. We then place a few additional points in our 2D space, which will become our 'cluster' (Voronoi cells in our case) centroids.

We then extend an equal radius out from each of our centroids. At some point, the circumferences of each cell circle will collide with another — creating our cell edges:



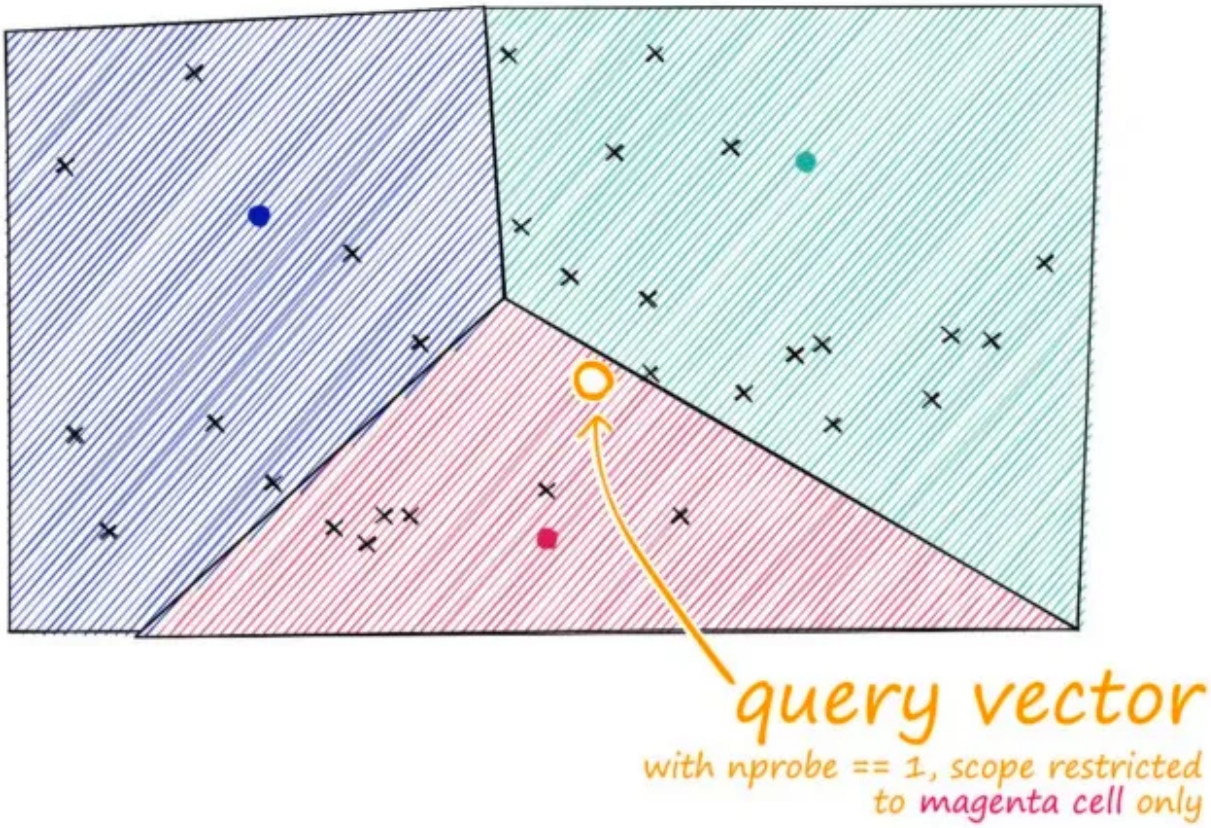
By Pinecone

Now, every datapoint will be contained within a cell — and be assigned to that respective centroid.

Just as with our other indexes, we introduce a query vector **xq** — this query vector must land within one of our cells, at which point we restrict our

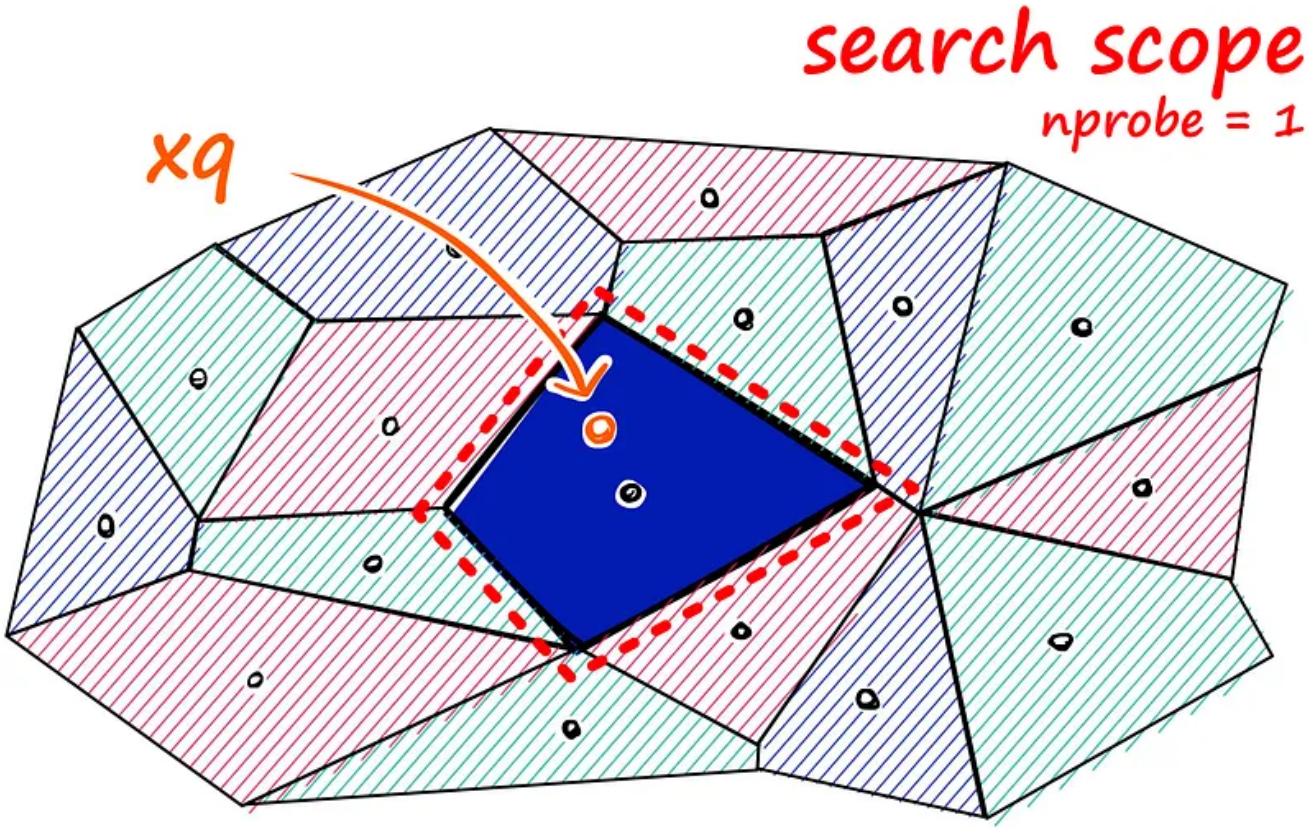
search scope to that single cell.

But there is a problem if our query vector lands near the edge of a cell — there's a good chance that its closest other datapoint is contained within a neighboring cell. We call this the *edge problem*:

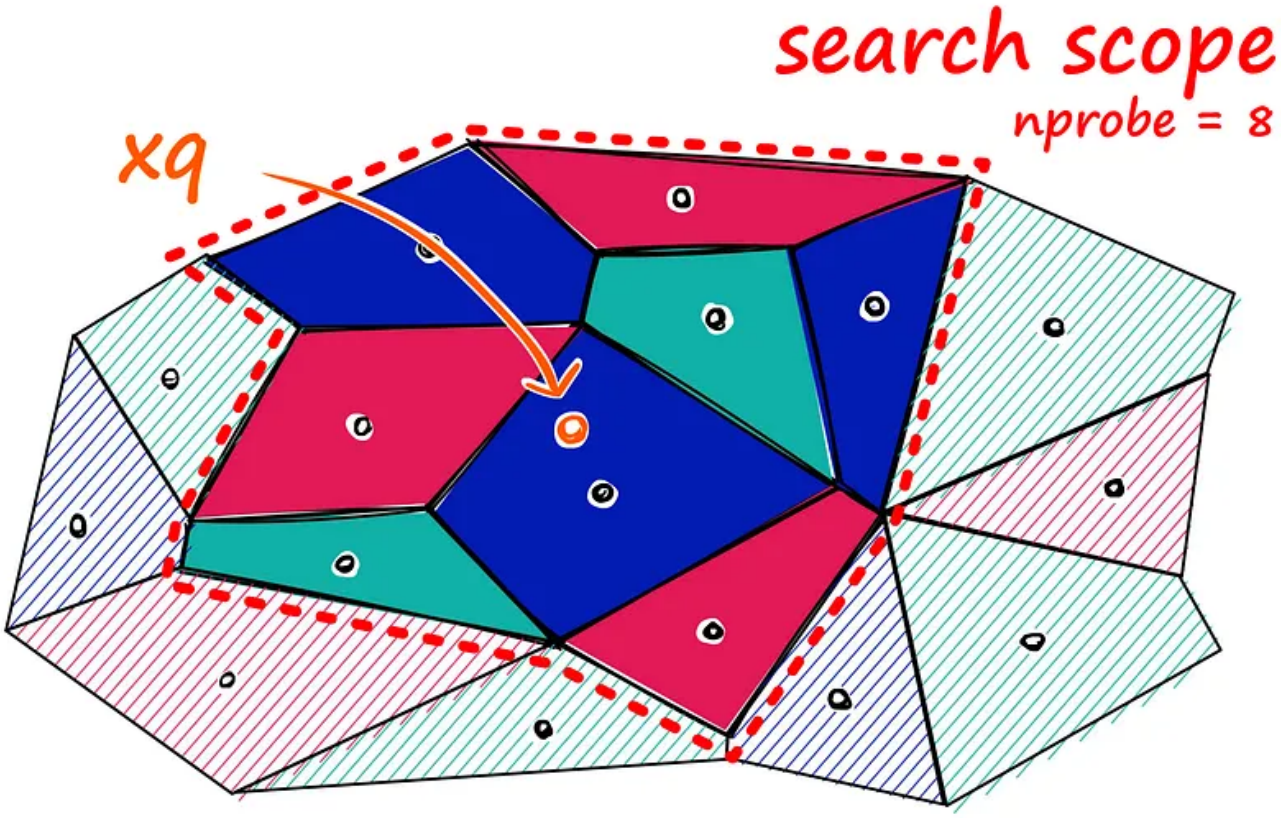


By Pinecone

Now, what we can do to mitigate this issue and increase search-quality is increase an index parameter known as the **nprobe** value. With **nprobe** we can set the number of cells to search.



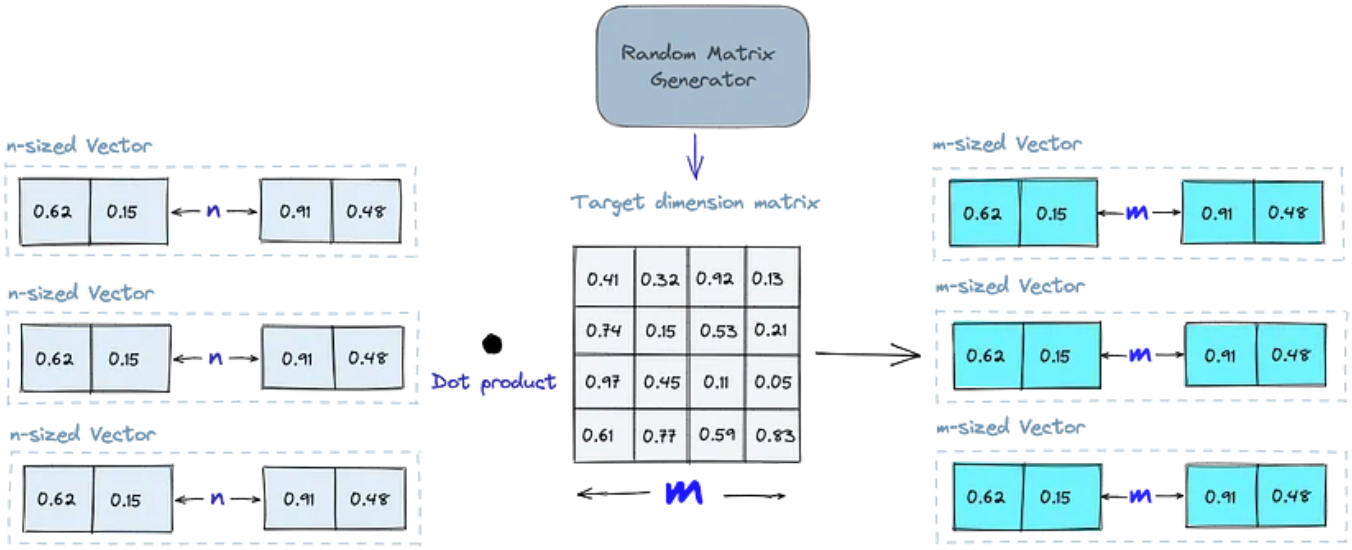
By Pinecone



By Pinecone

7.3. Random Projection (RP)

The basic idea behind random projection is to project the high-dimensional vectors to a lower-dimensional space using a **random projection matrix**. We create a matrix of random numbers. The size of the matrix is going to be the target low-dimension value we want. We then calculate the dot product of the input vectors and the matrix, which results in a **projected matrix** that has fewer dimensions than our original vectors but still preserves their similarity.



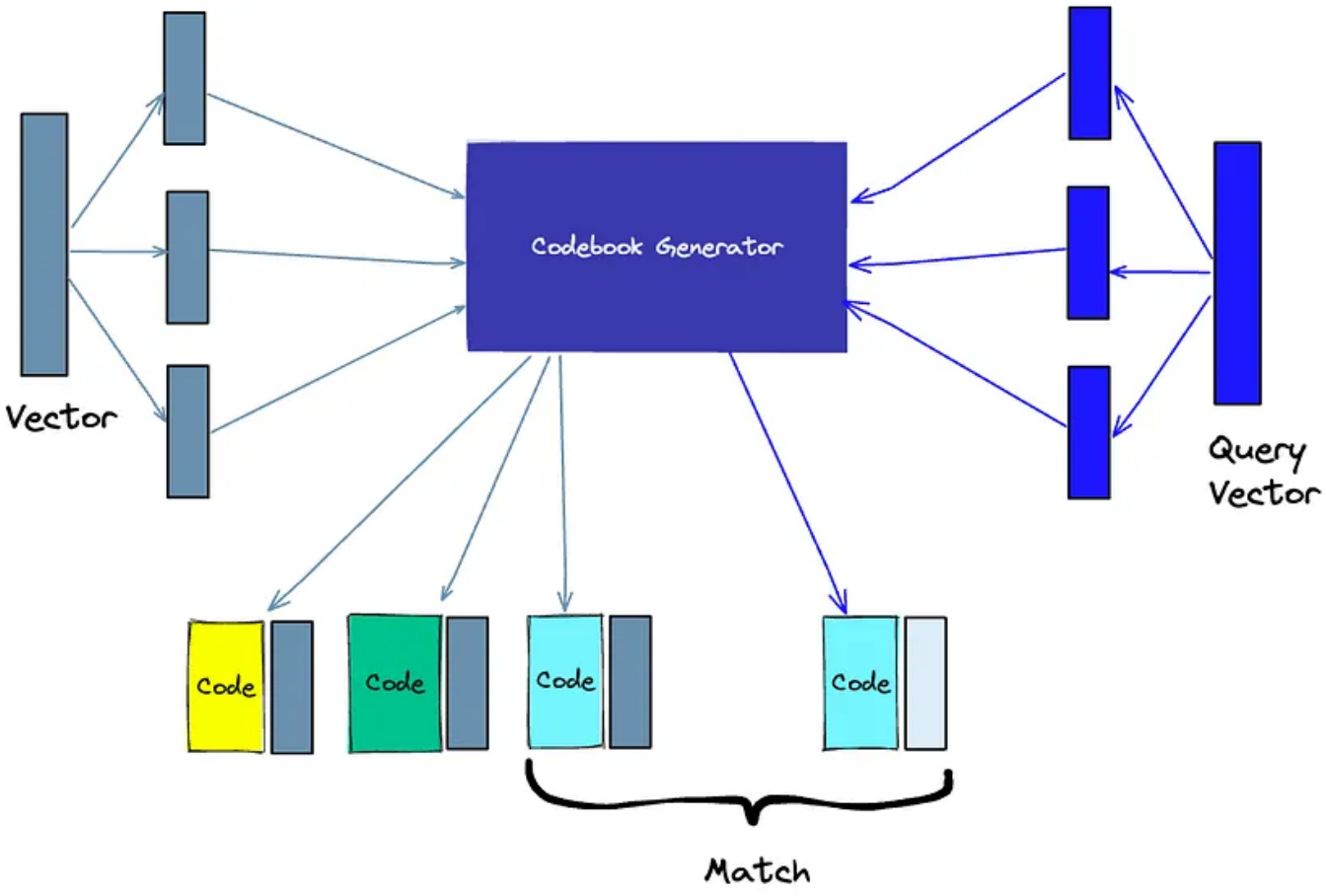
When we query, we use the same projection matrix to project the query vector onto the lower-dimensional space. Then, we compare the projected query vector to the projected vectors in the database to find the nearest neighbors. Since the dimensionality of the data is reduced, the search process is significantly faster than searching the entire high-dimensional space.

Just keep in mind that random projection is an approximate method, and the projection quality depends on the properties of the projection matrix. In general, the more random the projection matrix is, the better the quality of

the projection will be. However, generating a truly random projection matrix can be computationally expensive, especially for large datasets. Learn more about random projection.

7.4. Product Quantization (PQ)

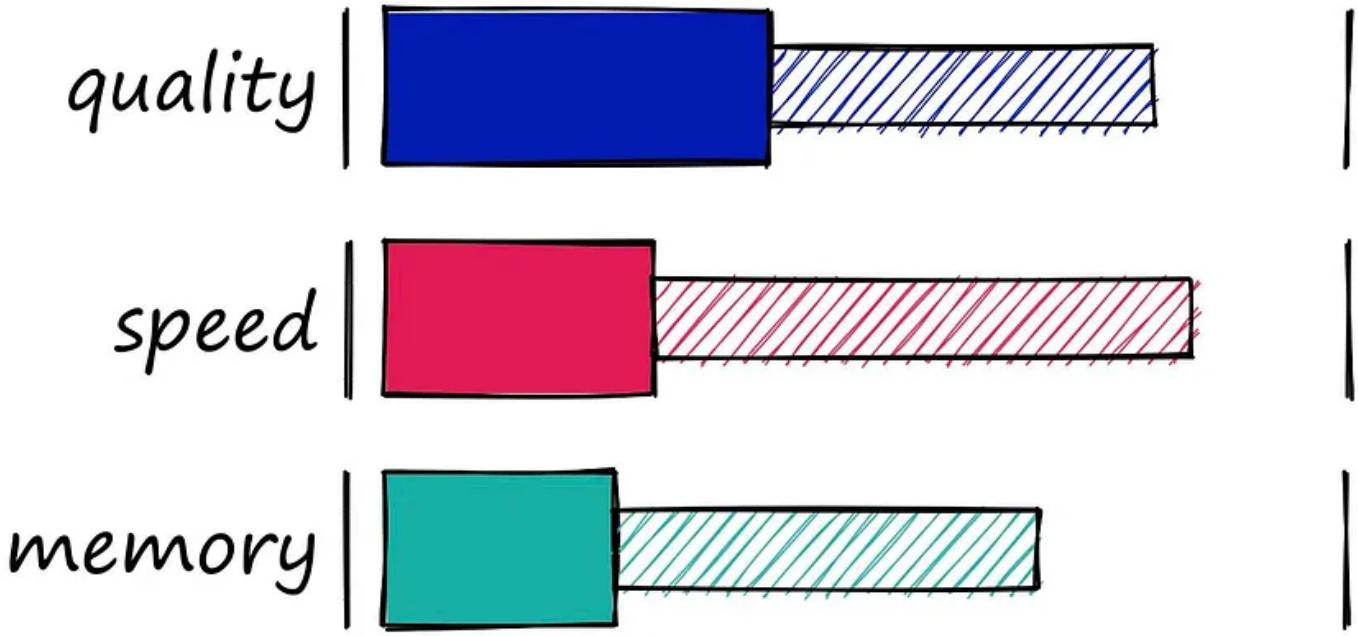
Another way to build an index is product quantization (PQ), which is a *lossy* compression technique for high-dimensional vectors (like vector embeddings). It takes the original vector, breaks it up into smaller chunks, simplifies the representation of each chunk by creating a representative "code" for each chunk, and then puts all the chunks back together — without losing information that is vital for similarity operations. The process of PQ can be broken down into four steps: splitting, training, encoding, and querying.



1. **Splitting** -The vectors are broken into segments.
2. **Training** — we build a "codebook" for each segment. Simply put — the algorithm generates a pool of potential "codes" that could be assigned to a vector. In practice — this "codebook" is made up of the center points of clusters created by performing k-means clustering on each of the vector's segments. We would have the same number of values in the segment codebook as the value we use for the k-means clustering.
3. **Encoding** — The algorithm assigns a specific code to each segment. In practice, we find the nearest value in the codebook to each vector segment after the training is complete. Our PQ code for the segment will be the identifier for the corresponding value in the codebook. We could use as many PQ codes as we'd like, meaning we can pick multiple values from the codebook to represent each segment.
4. **Querying** — When we query, the algorithm breaks down the vectors into sub-vectors and quantizes them using the same codebook. Then, it uses the indexed codes to find the nearest vectors to the query vector.

The number of representative vectors in the codebook is a trade-off between the accuracy of the representation and the computational cost of searching the codebook. The more representative vectors in the codebook, the more accurate the representation of the vectors in the subspace, but the higher the computational cost to search the codebook. By contrast, the fewer representative vectors in the codebook, the less accurate the representation, but the lower the computational cost. Learn more about PQ.

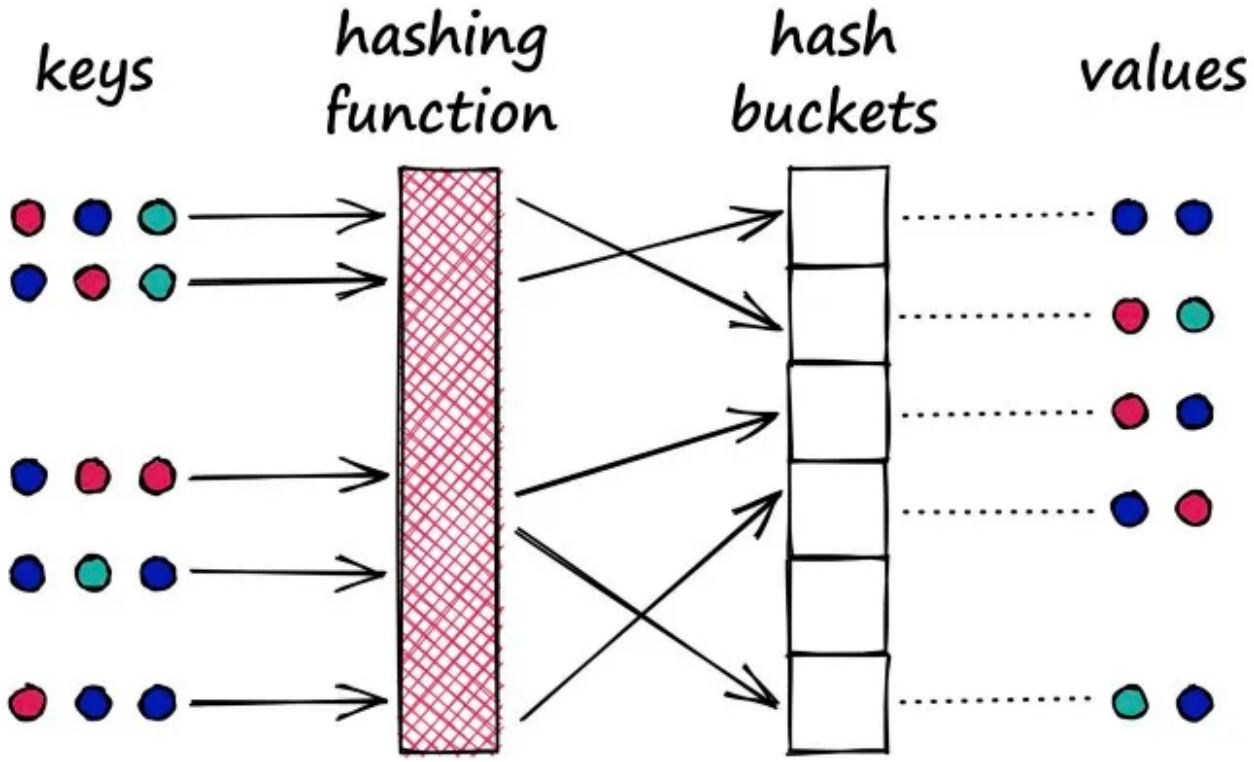
7.5. Locality-Sensitive Hashing (LSH)



LSH — a wide range of performances heavily dependent on the parameters set. Good quality results in slower search, and fast search results in worse quality. Poor performance for high-dimensional data. The 'half-filled' segments of the bars represent the range in performance encountered while modifying index parameters.

Locality Sensitive Hashing (LSH) works by grouping vectors into buckets by processing each vector through a hash function that maximizes hashing collisions — rather than minimizing as is usual with hashing functions.

What does that mean? Imagine we have a Python dictionary. When we create a new key-value pair in our dictionary, we use a hashing function to hash the key. This hash value of the key determines the 'bucket' where we store its respective value:

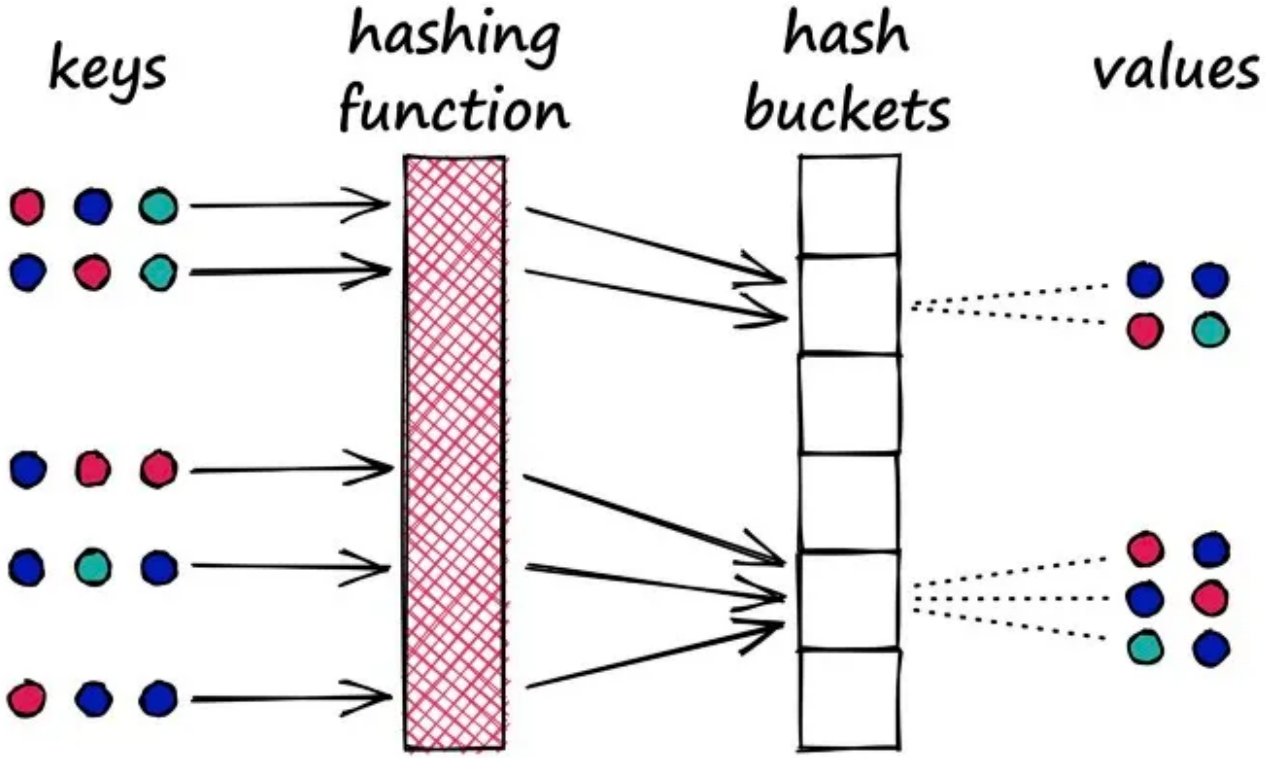


A typical hash function for a dictionary-like object will attempt to minimize hash collisions, aiming to assign only one value to each bucket.

A Python dictionary is an example of a hash table using a typical hashing function that *minimizes* hashing collisions, a hashing collision where two different objects (keys) produce the same hash.

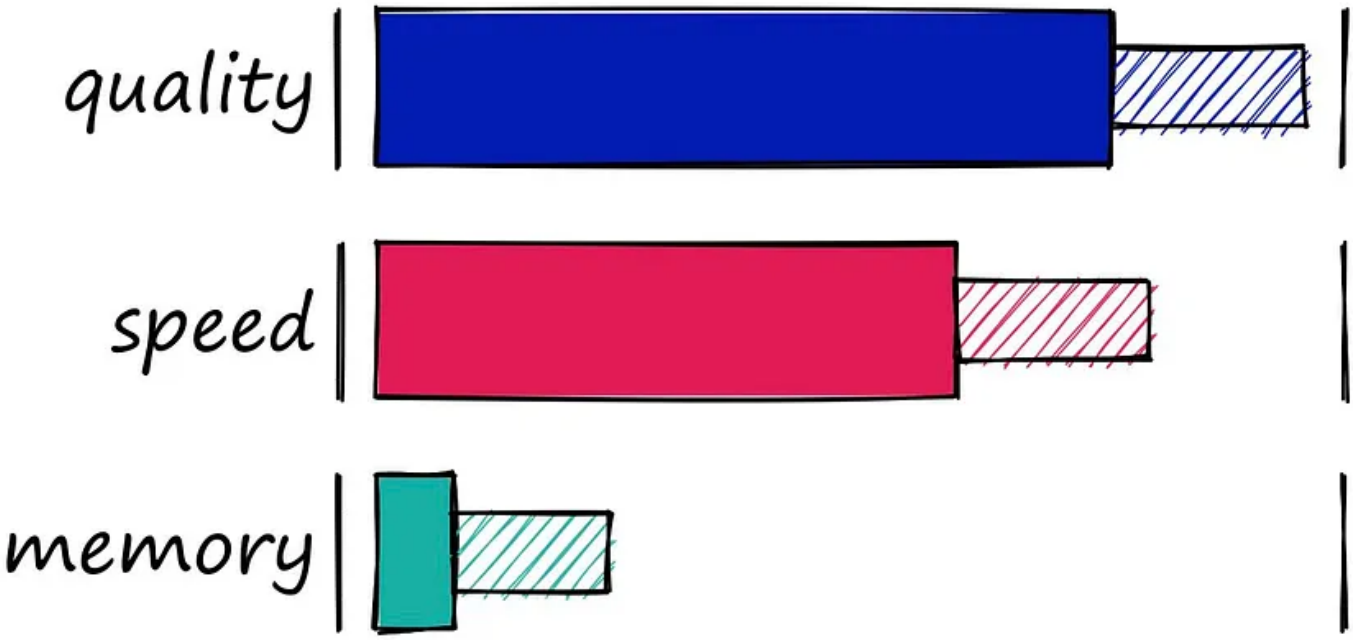
In our dictionary, we want to avoid these collisions as it means that we would have multiple objects mapped to a single key — but for LSH, we want to *maximize* hashing collisions.

Why would we want to maximize collisions? Well, for search, we use LSH to group similar objects together. When we introduce a new query object (or vector), our LSH algorithm can be used to find the closest matching groups:



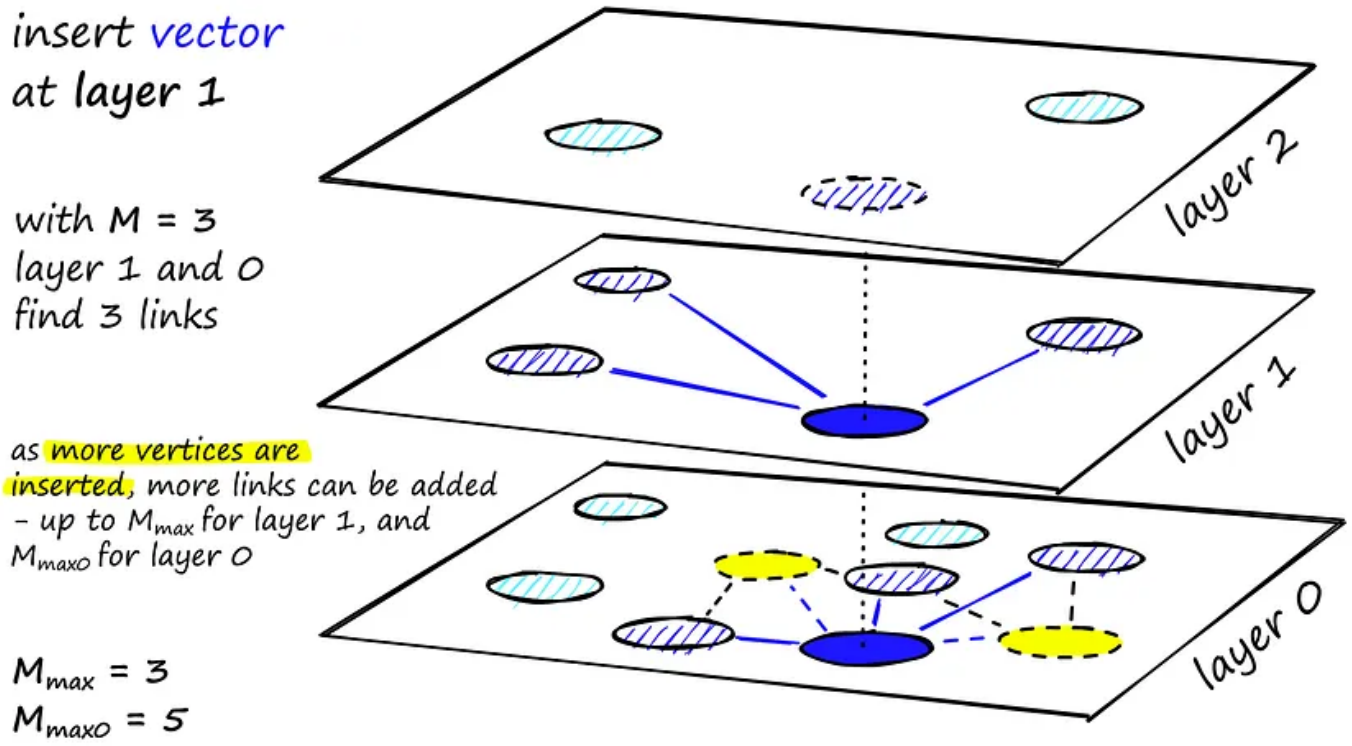
Our hash function for LSH attempts to maximize hash collisions, producing groupings of vectors.

7.6. Hierarchical Navigable Small World (HNSW)



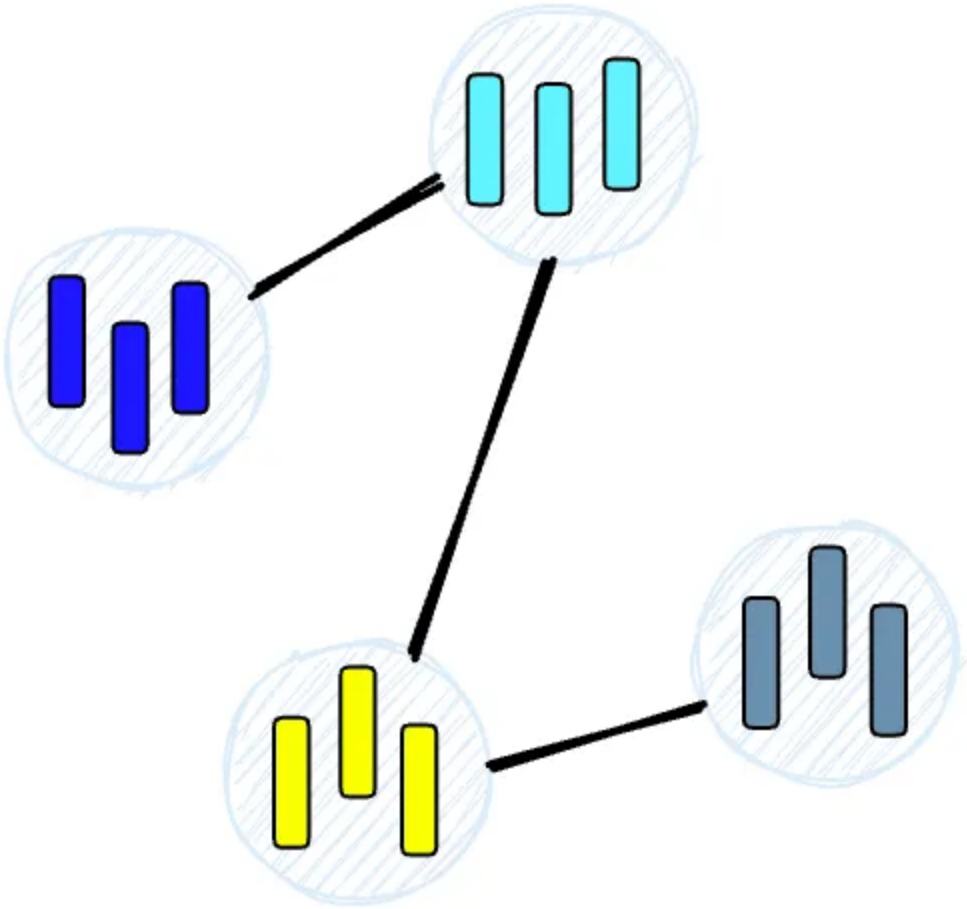
HNSW — great search-quality, good search-speed, but substantial index sizes. The 'half-filled' segments of the bars represent the range in performance encountered while modifying index parameters.

HNSW creates a hierarchical, tree-like structure where each node of the tree represents a set of vectors. The edges between the nodes represent the **similarity** between the vectors. The algorithm starts by creating a set of nodes, each with a small number of vectors. This could be done randomly or by clustering the vectors with algorithms like k-means, where each cluster becomes a node.



Explanation of the number of links assigned to each vertex and the effect of M, M_max, and M_max0.

The algorithm then examines the vectors of each node and draws an edge between that node and the nodes that have the most similar vectors to the one it has.

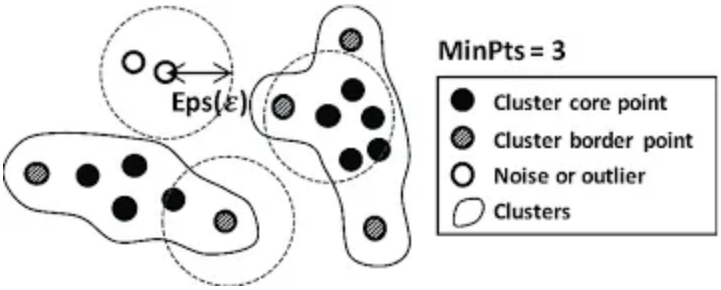


When we query an HNSW index, it uses this graph to navigate through the tree, visiting the nodes that are most likely to contain the closest vectors to the query vector. Learn more about HNSW.

7.7. Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

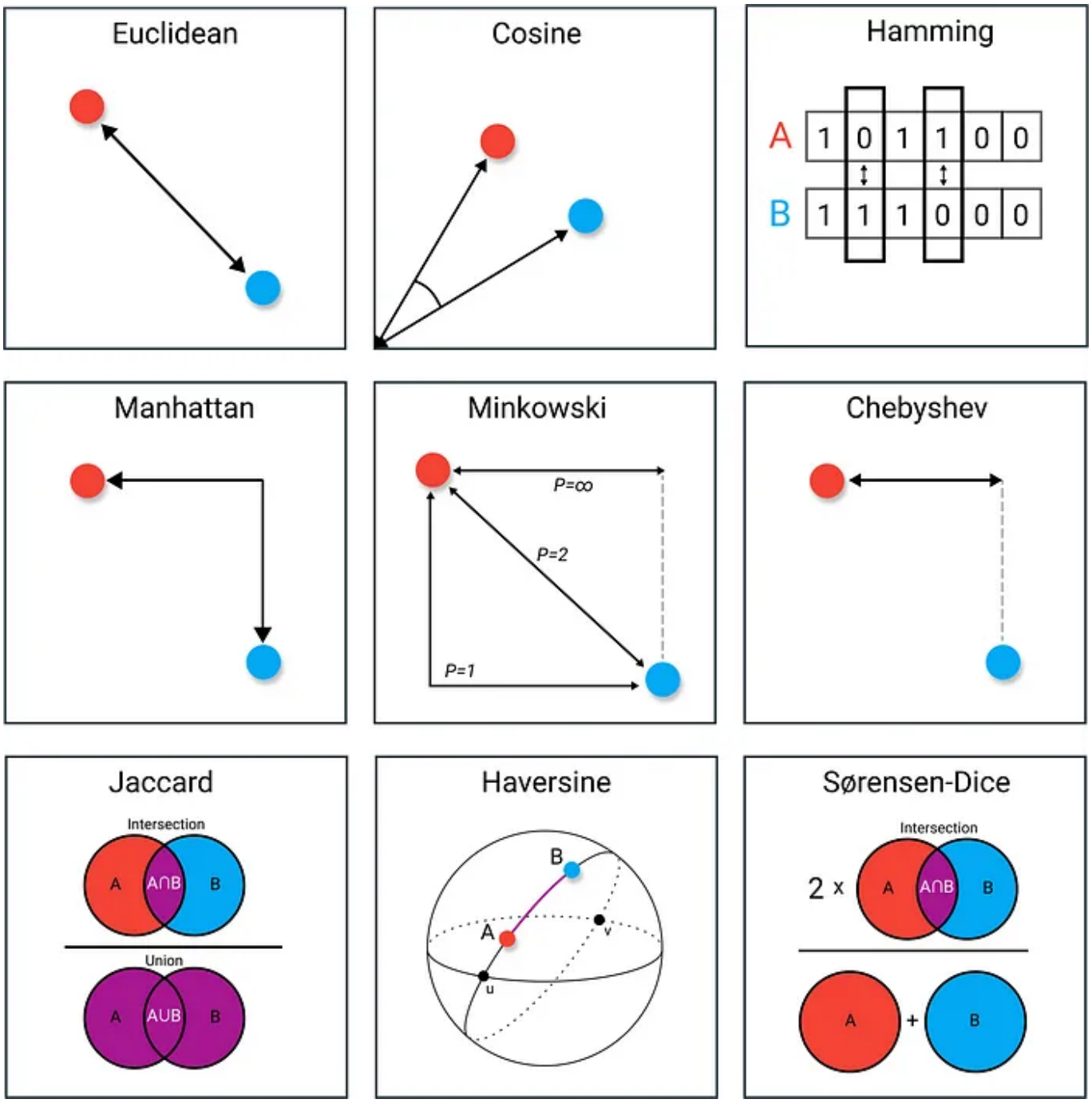
DBSCAN operates on the idea of density reachability and density connectivity. It starts with an arbitrary point in the dataset, and if there are at least 'minPts' within a given radius 'eps' from that point, a new cluster is created. Eps stands for epsilon which is a user-defined input denoting the maximum distance between two points to be considered in a cluster, while minPts refer to the minimum number of data points required to form a cluster. The algorithm then iteratively adds all directly reachable points within the 'eps' radius to the cluster. This process continues until no further points can be added to the cluster. Then, the algorithm proceeds to the next

unvisited point in the dataset and repeats the process until all points have been visited. The key parameters in DBSCAN are ‘eps’ and ‘minPts’, which define the cluster of points and the minimum density of points required to form a cluster respectively.

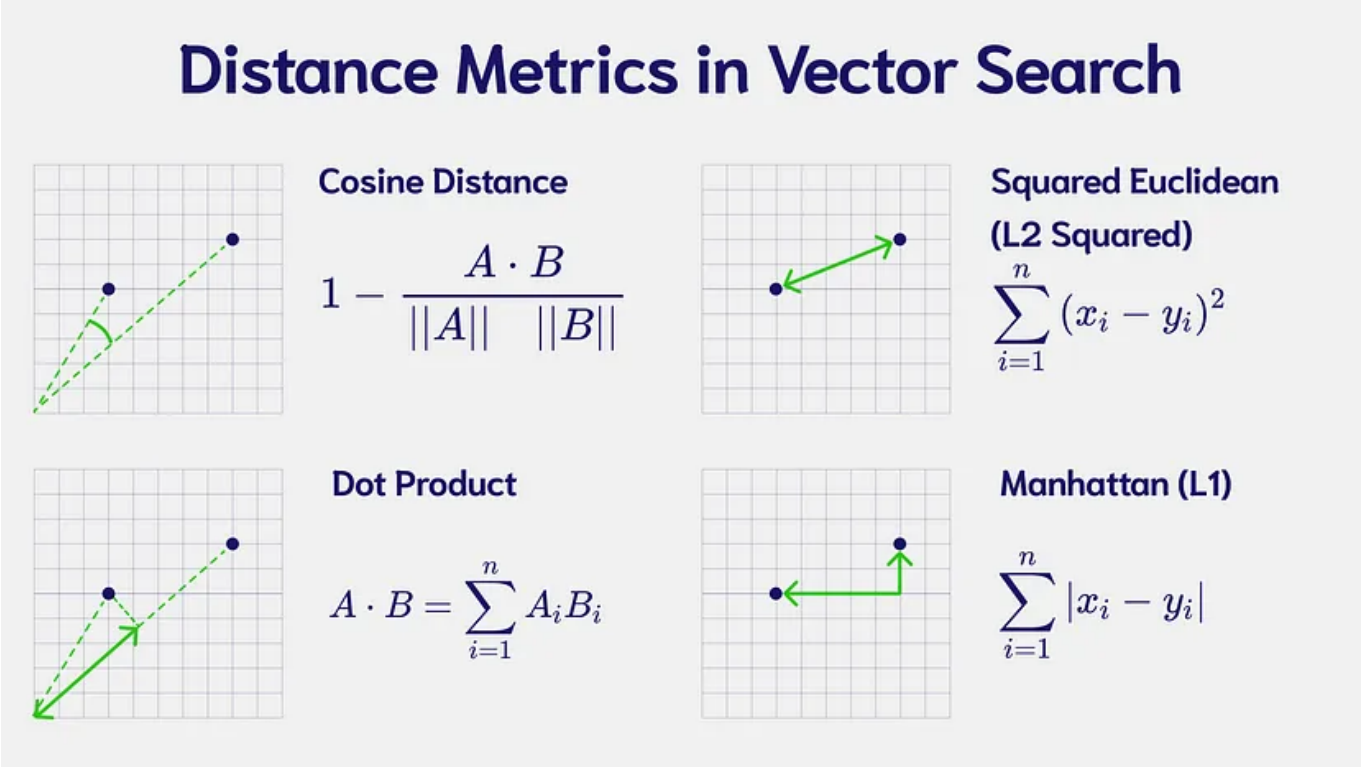


DBSCAN

8. Similarity Measures: Distance Metrics



Distance Metrics



Distance Metrics

Similarity measures are mathematical methods for determining how similar two vectors are in a vector space. Similarity measures are used in vector databases to compare the vectors stored in the database and find the ones that are most similar to a given query vector.

Several similarity measures can be used, including:

- **Cosine similarity:** measures the cosine of the angle between two vectors in a vector space. It ranges from -1 to 1, where 1 represents identical vectors, 0 represents orthogonal vectors, and -1 represents vectors that are diametrically opposed.
- **Euclidean distance:** measures the straight-line distance between two vectors in a vector space. It ranges from 0 to infinity, where 0 represents identical vectors, and larger values represent increasingly dissimilar vectors.

- **Dot product:** measures the product of the magnitudes of two vectors and the cosine of the angle between them. It ranges from -∞ to ∞, where a positive value represents vectors that point in the same direction, 0 represents orthogonal vectors, and a negative value represents vectors that point in opposite directions.

8.1. How to Choose a Similarity Metric

It's a general best practice to use the same similarity measure for the search that the embeddings were trained on; however, the choice of similarity measure also depends on the specific characteristics of the data and the context of the problem you are trying to solve. Here are some top applications for each of the discussed similarity measures:

Euclidean Distance

- **Clustering Analysis:** Clustering, like k-means, groups data points based on their proximity in vector space.
- **Anomaly and Fraud Detection:** In these use cases, unusual data points can be detected through unusually large distances from the centroid of normal transactions.

Dot Product

- **Image Retrieval and Matching:** Images with similar visual content will have closely aligned vectors, resulting in higher dot product values. This makes dot product a good choice when you want to find images similar to a given query image.
- **Neural Networks and Deep Learning:** In neural networks, fully connected layers use the dot product to combine input features with

learnable weights. This captures relationships between features and is helpful for tasks like classification and regression.

- **Music Recommendation:** Dot product similarity helps identify tracks with similar audio characteristics, making it valuable for music recommendation systems.

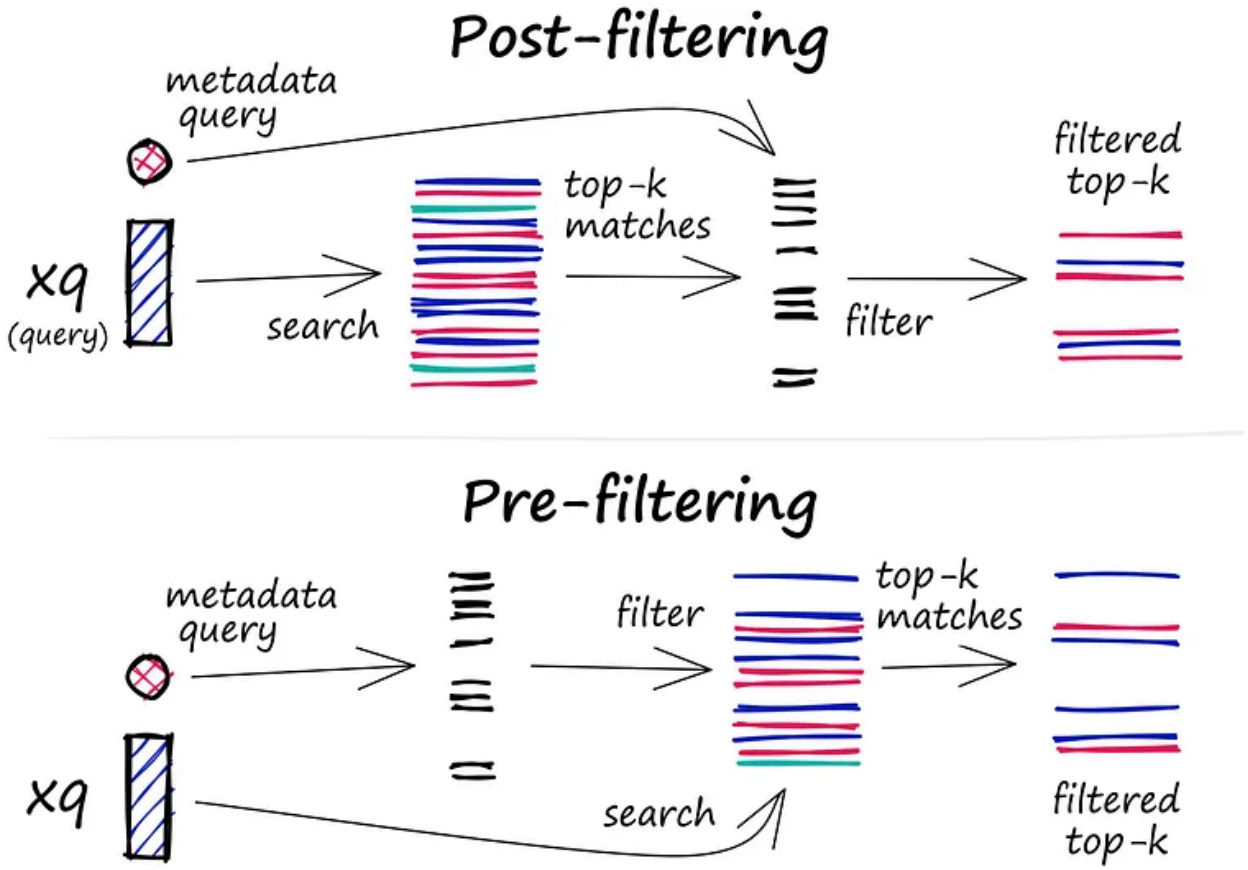
Cosine Similarity

- **Topic Modeling:** In document embeddings, each dimension can represent a word's frequency or TF-IDF weight. However, two documents of different lengths can have drastically different word frequencies yet the same word distribution. Since this places them in similar directions in vector space but not similar distances, cosine similarity is a great choice.
- **Document Similarity:** Another application of Topic Modeling. Similar document embeddings have similar directions but can have different distances.
- **Collaborative Filtering:** This approach in recommendation systems uses the collective preferences and behaviors of users (or items) to make personalized recommendations. Users (or items) are represented as vectors based on their interactions. Since overall ratings and popularity can create different distances, but the direction of similar vectors remains close, cosine similarity is often used.

9. Filtering

Every vector stored in the database also includes metadata. In addition to the ability to query for similar vectors, vector databases can also filter the results based on a metadata query. To do this, the vector database usually maintains

two indexes: a vector index and a metadata index. It then performs the metadata filtering either before or after the vector search itself, but in either case, there are difficulties that cause the query process to slow down.



The filtering process can be performed either before or after the vector search itself, but each approach has its challenges that may impact the query performance:

- **Pre-filtering:** In this approach, metadata filtering is done before the vector search. While this can help reduce the search space, it may also cause the system to overlook relevant results that don't match the metadata filter criteria. Additionally, extensive metadata filtering may slow down the query process due to the added computational overhead.

- **Post-filtering:** In this approach, the metadata filtering is done after the vector search. This can help ensure that all relevant results are considered, but it may also introduce additional overhead and slow down the query process as irrelevant results need to be filtered out after the search is complete.

To optimize the filtering process, vector databases use various techniques, such as leveraging advanced indexing methods for metadata or using parallel processing to speed up the filtering tasks. Balancing the trade-offs between search performance and filtering accuracy is essential for providing efficient and relevant query results in vector databases. Learn more about vector search filtering.

10. Choosing a Vector Database

Here's a succinct summary to aid your decision:

1. **Open-Source and hosted cloud**: If you lean towards open-source solutions, **Weaviate**, **Milvus**, and **Chroma** emerge as top contenders. **Pinecone**, although not open-source, shines with its developer experience and a robust fully hosted solution.
2. **Performance**: When it comes to raw performance in queries per second, **Milvus** takes the lead, closely followed by **Weaviate** and **Qdrant**. However, in terms of latency, **Pinecone** and **Milvus** both offer impressive sub-2ms results. If multiple pods are added for P**inecone**, then much higher QPS can be reached.
3. **Community Strength**: **Milvus** boasts the largest community presence, followed by **Weaviate** and **Elasticsearch**. A strong community often translates to better support, enhancements, and bug fixes.

4. **Scalability, advanced features and security:** Role-based access control, a feature crucial for many enterprise applications, is found in **Pinecone**, **Milvus**, and **Elasticsearch.** On the scaling front, dynamic segment placement is offered by **Milvus** and **Chroma**, making them suitable for ever-evolving datasets. If you need a database with a wide array of index types, **Milvus**' support for 11 different types is unmatched. While hybrid search is well-supported across the board, **Elasticsearch** does fall short in terms of disk index support.

5. **Pricing:** For startups or projects on a budget, **Qdrant's** estimated $9 pricing for 50k vectors is hard to beat. On the other end of the spectrum, for larger projects requiring high performance, **Pinecone** and **Milvus** offer competitive pricing tiers.

In conclusion, there's no one-size-fits-all when it comes to vector databases. The ideal choice varies based on specific project needs, budget constraints, and personal preferences.

Landscape of Vector Databases

10.1. Comparison Parameters

- **Is open source:** Indicates if the software's source code is freely available to the public, allowing developers to review, modify, and distribute the software.
- **Self-host:** Specifies if the database can be hosted on a user's own infrastructure rather than being dependent on a third-party cloud service.
- **Cloud management:** Offers an interface for database cloud management
- **Purpose-built for Vectors:** This means the database was specifically designed with vector storage and retrieval in mind, rather than being a general database with added vector capabilities.
- **Developer experience:** Evaluates how user-friendly and intuitive it is for developers to work with the database, considering aspects like documentation, SDKs, and API design.
- **Community:** Assesses the size and activity of the developer community around the database. A strong community often indicates good support, contributions, and the potential for continued development.
- **Queries per second:** How many queries the database can handle per second using a specific dataset for benchmarking (in this case, the nytimes-256-angular dataset)
- **Latency:** the delay (in milliseconds) between initiating a request and receiving a response. 95% of query latencies fall under the specified time for the nytimes-256-angular dataset.
- **Supported index types:** Refers to the various indexing techniques the database supports, which can influence search speed and accuracy. Some vector databases may support multiple indexing types like HNSW, IVF, and more.

- **Hybrid Search:** Determines if the database allows for combining traditional (scalar) queries with vector queries. This can be crucial for applications that need to filter results based on non-vector criteria.
- **Disk index support:** Indicates if the database supports storing indexes on disk. This is essential for handling large datasets that cannot fit into memory.
- **Role-based access control:** Checks if the database has security mechanisms that allow permissions to be granted to specific roles or users, enhancing data security.
- **Dynamic segment placement vs. static data sharding:** Refers to how the database manages data distribution and scaling. Dynamic segment placement allows for more flexible data distribution based on real-time needs, while static data sharding divides data into predetermined segments.
- **Free hosted tier:** Specifies if the database provider offers a free cloud-hosted version, allowing users to test or use the database without initial investment.
- **Pricing (50k vectors @1536)** and **Pricing (20M vectors, 20M req. @768):** Provides information on the cost associated with storing and querying specific amounts of data, giving an insight into the database's cost-effectiveness for both small and large-scale use cases.

Conclusion

This blog provided us with a concise exploration of vector databases, focusing on their crucial role in Retrieval-Augmented Generation (RAG) applications. Highlighting popular databases like ChromaDB, Pinecone, and

Weaviate, the author emphasizes the importance of efficient storage and retrieval for optimal RAG performance.

The blog dives into various indexing techniques and algorithms, offering insights into Annoy, Inverted File (IVF) Indexing, Random Projection, Product Quantization, Locality-Sensitive Hashing (LSH), HNSW, and DBSCAN. It further explains similarity measures like cosine similarity, Euclidean distance, and dot product, with practical applications in document similarity, image retrieval, and collaborative filtering.

A significant portion is dedicated to aiding readers in choosing the right vector database. Key factors such as open-source availability, performance, community strength, scalability, advanced features, security, and pricing are considered. The comparison parameters offer a quick summary, empowering readers to make informed decisions tailored to their specific needs and preferences.

In conclusion, the blog serves as a quick guide for those navigating vector databases, providing essential knowledge for seamless integration and optimization of RAG models.

Index	Memory (MB)	Query Time (ms)	Recall	Notes
Flat (L2 or IP)	~500	~18	1.0	Good for small datasets or where query time is irrelevant
LSH	20 - 600	1.7 - 30	0.4 - 0.85	Best for low dimensional data, or small datasets
HNSW	600 - 1600	0.6 - 2.1	0.5 - 0.95	Very good for quality, high speed, but large memory usage
IVF	~520	1 - 9	0.7 - 0.95	Good scalable option. High-quality, at reasonable speed and memory usage

Quick summary of each index's memory, speed, and search-quality performance.

Credits

In this blog post, we have compiled information from various sources, including research papers, technical blogs, official documentations, YouTube videos, and more. Each source has been appropriately credited beneath the corresponding images, with source links provided.

Below is a consolidated list of references:

1. https://www.shakudo.io/blog/introduction-to-vector-databases
2. https://medium.com/@apurvak/vector-databases-which-one-to-choose-3806c5ff67b0
3. https://www.pinecone.io/learn/vector-database/
4. https://thedataquarry.com/posts/vector-db-1/
5. https://www.pinecone.io/learn/series/faiss/vector-indexes/
6. https://www.pinecone.io/learn/series/faiss/vector-indexes/
7. https://medium.com/@david.gutsch0/vector-databases-understanding-the-algorithm-part-3-bc7a8926f27c#:~:text=DBSCAN%20(Density%2DBased%20Spatial%20Clustering,discover%20clusters%20of%20arbitrary%20shape.
8. https://weaviate.io/blog/distance-metrics-in-vector-search
9. https://www.linkedin.com/pulse/choosing-vector-database-your-gen-ai-stack-abhinav-srivastava/

Thank you for reading!

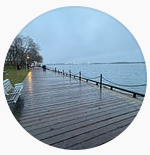
If this guide has enhanced your understanding of Python and Machine Learning:

- Please show your support with a clap 👏 or several claps!
- Your claps help me create more valuable content for our vibrant Python or ML community.
- Feel free to share this guide with fellow Python or AI / ML enthusiasts.
- Your feedback is invaluable — it inspires and guides my future posts.

Connect with me!

Vipra

Vector Database | Indexing | Retrieval Augmented | Large Language Models | Database

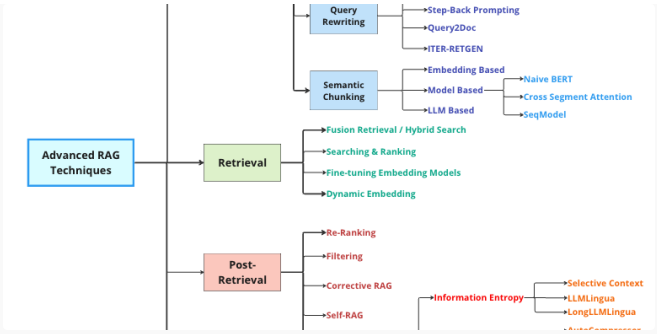


Written by Vipra Singh

1K Followers

Follow

More from Vipra Singh



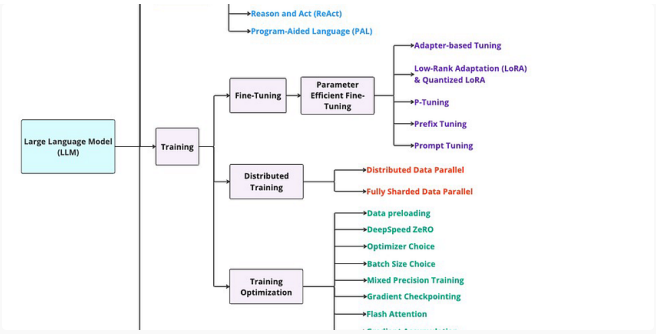
Vipra Singh

Building LLM Applications: Advanced RAG (Part 10)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

✦ · 48 min read · Apr 27, 2024

384 2



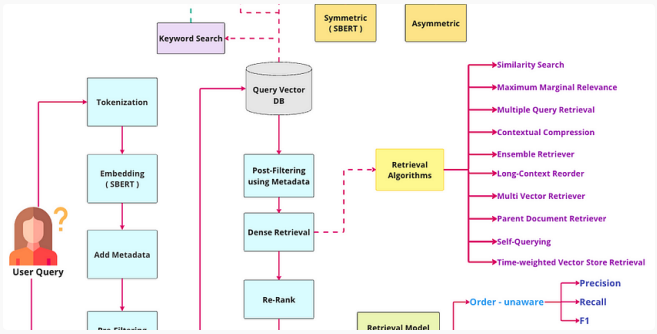
Vipra Singh

Building LLM Applications: Large Language Models (Part 6)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

38 min read · Feb 10, 2024

390 1



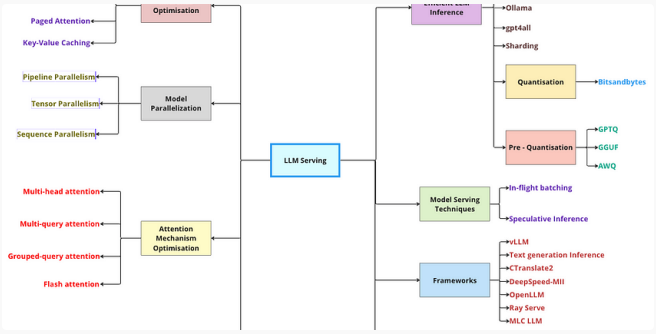
Vipra Singh

Building LLM Applications: Search & Retrieval (Part 5)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

26 min read · Jan 27, 2024

405 1



Vipra Singh

Building LLM Applications: Serving LLMs (Part 9)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

✦ · 50 min read · Apr 17, 2024

546 3

See all from Vipra Singh

Recommended from Medium

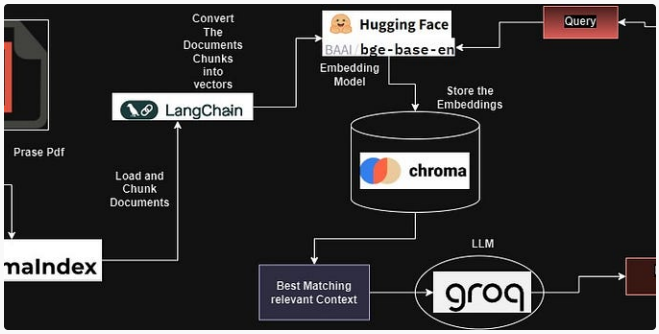
Paul Iusztin in Decoding ML

The 4 Advanced RAG Algorithms You Must Know to Implement

Implement from scratch 4 advanced RAG methods to optimize your retrieval and post-...

15 min read · May 4, 2024

1.4K 10



Plaban Nayak in The AI Forum

RAG on Complex PDF using LlamaParse, Langchain and Groq

Retrieval-Augmented Generation (RAG) is a new approach that leverages Large Languag...

13 min read · Apr 7, 2024

678 9

Lists

Natural Language Processing

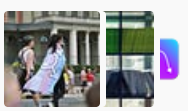
1494 stories · 1010 saves

AI Regulation

6 stories · 473 saves

ChatGPT prompts

47 stories · 1642 saves



Generative AI Recommended Reading

52 stories · 1106 saves

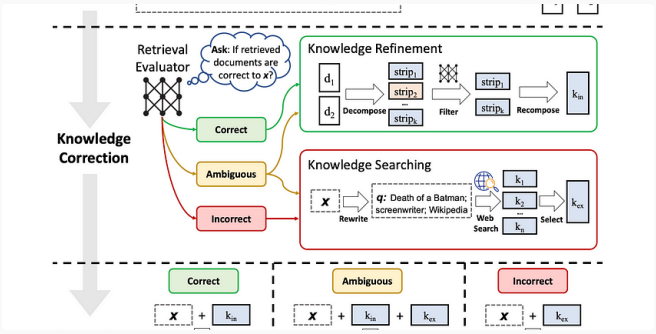
Lars Wiik

Best Embedding Model 🌟 — OpenAI / Cohere / Google / E5 /...

An In-depth Comparison of Multilingual Embedding Models

12 min read · Apr 7, 2024

478 3



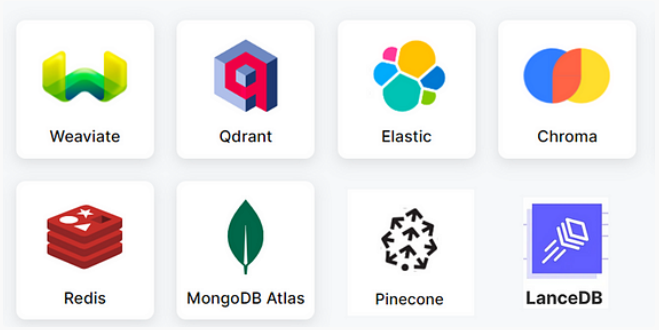
Barsha Rani Swain in GoPenAI

Advanced RAG: Corrective Retrieval Augmented Generation...

CRAG enhances the traditional RAG by introducing a retrieval evaluator to assess th...

10 min read · Apr 23, 2024

320



Jayita Bhattacharyya in CodeX

A Brief Comparison of Vector Databases

In this supercharged era of RAG (Retrieval Augmented Generation) fueled by LLMs...

3 min read · May 6, 2024

Mahesh

How to Productionize Large Language Models (LLMs)

Understand LLMOps, architectural patterns, how to evaluate, fine tune & deploy...

94 min read · Mar 27, 2024

 10    186  2 

See more recommendations