PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
HIGHER SCHOOL OF COMPUTER SCIENCE

2nd Year Superior Cycle (2CS)

2022-2023

Advanced Databases Project

Milvus Vector Database

Realized By :

- Mohammed Abderrahmane Bensalem
- Safa Zakaria Abdellah
- Oussama Hadj Aissa Fekhar
- Abderrahmane Boucenna

Supervised By :

- M.Soumia BENKRID

Year : 2022-2023

Abstract

The rapid growth of Artificial Intelligence (AI) and Big Data has revolutionized various sectors, leading to groundbreaking advancements in technology, research, and decision-making processes. AI algorithms and models, combined with massive amounts of data, have enabled organizations to extract valuable insights, make accurate predictions, and automate complex tasks. However, the increasing volume, velocity, and variety of data have posed significant challenges in terms of storage, retrieval, and analysis. This has resulted in the need for more efficient and scalable solutions,usually the use of modern no-SQL databases with big data architectures is enough for this task but due to the properties of data handled by AI , many inefficiencies remain leading to the emergence of vector databases.

Vector databases are specialized data storage systems designed to handle high-dimensional vector data, which is prevalent in AI and Big Data applications. These databases employ advanced indexing techniques and algorithms optimized for vector operations, enabling fast and accurate similarity searches, clustering, and recommendation tasks. By efficiently organizing and managing vectors, vector databases facilitate the exploration and exploitation of complex data patterns, enabling businesses and researchers to gain deeper insights and improve decision-making processes.

for the purpose of this project ,we will use Milvus, an open-source vector database designed to efficiently store, index, and search high-dimensional vector data. It is specifically built to address the challenges associated with managing large-scale vector data in AI and Big Data applications. this is why we will be focusing more on the vector database properties that distinguishes from the usual DBMS rather than the usual DBMS features.

Contents

1 Vector Databases Overview

1.1 Motivation of creation

With the booming of the application of AI in several domains like IOT,medical imaging,large language models that have large volumes of unstructured data as According to IDC, 80% of data will be unstructured by 2025[**1**],New type of Highly used data manipulations are rising , this would be handled conveniently with No-SQL databases combined with big data but problems arises with the details of these new types of usage or querying for this data .Some of the new types of emerging operations are :

- **Recommandation systems :** Many modern large scale companies like Facebook and YouTube use recommendation systems that try to find similar high dimensionality items to recommend, this goes beyond the classic approach of filtering with predefined sets of values because it is conducted on none structured data .one problem here is that sometimes we don't even know these predefined sets and we need to load the entire database and feed it to the AI algorithm to treat it ,wouldn't it be better if we could have a database that can save the patterns and quickly extract the similar items without loading the entire database ?
- **Image Searching :** image searching platforms like google lens and image filtering try to find the closest image to the searched one or the best image that fits certain constraints, an image of course being treated as a 2d vector of real values .the problem here is that images are often assigned with a string title or id to recognize them , performing a large recognition operation would mean extracting the images off of the database and then feeding them to the AI model , which is largely time consuming. many traditional databases store unstructured data in a binary representation, this doesn't say anything about the features of the data nor it's patterns. it's basically creating a read only memory useful for just storing the data without any other advantage.wouldn't it better if we store unstructured data (images) as a format that actually can have operations performed on it, a format that keeps a semantic meaning for the data and shows it's features, a format that liberates us of needing to load the entire data to the model so that when a query comes it can be handled efficiently ?

- **Long Term Memory:** A general AI problem, Imaging the following scenario. we have a database storing unstructured data for example large sequences of text, an application wants to classify the texts depedning on a criteria such as "harmful or none harmful text" , we load the data , we feed the ai algorithm and we get the results. what if this operation is redandunt, we cannot extract the entire database everytime so what do we do, we could partition our database into to databases (one is harmful text the other is not) or simply add a new field (text class) and it stores the class. the second solution is better however if we have millions of lines (or documents) of text adding this is highly costly but there's worst what if in a day we get alot of queries that all have a different criteria and the number of the criteria is infinite (searching positive tweets, searching racist tweets,....) , it's not convenient to add thousands of fields to the database.the basic problem here is that once we conduct our AI search we don't get any long term benefit of it, as storing it's result is costly and useless . what if there's a data format that is convenient to search patterns , a data format that can quickly classify the data depending on the criteria and without needing to store the results at each time ?

1.2 Foundations of Vector Databases

As discussed previously, storing unstructured data as a binary representation , eliminates all of its semantic meaning , it also limits the manipulations that can be conducted on our unstructured data .thanks to academic research in mathematics and AI alot of ground breaking and revolutionizing techniques have solved these problems, we now show some of the foundations and key ideas that lead to the creation of these databases.

1.2.1 Vector Embedding

Vectors is a very familiar topic in mathematics and ai , virtually anything can be represented by a vector ,but an object can have infinite vector representations and a vector is not necessarily interpretable meaning it can be useful for just storing data without keeping any feature.thankfully this AI technique called vector embedding which is revolutionary in fields like NLP and deep leaning that has largely solved this problem. vector embedding is about using a class of technics that transforms any given data into a vector while keeping the semantic meaning of it illustrated in figure 1 where unstructured data is transformed into high dimensional vectors in a way that keeps track of the semantic meaning of data which results in facilitation of operations like finding similar data , vector searching.

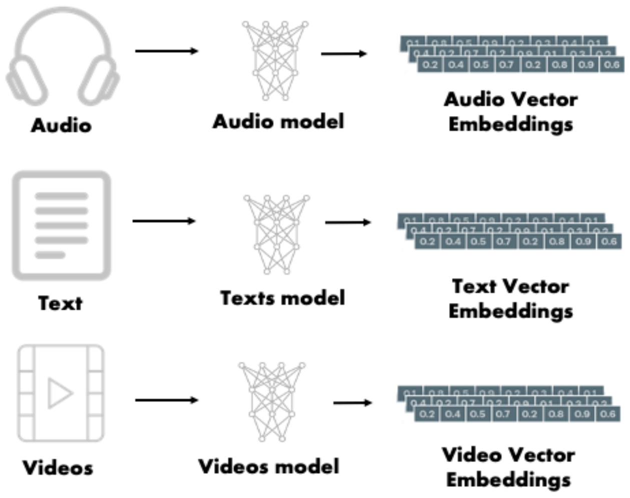


Figure 1: Vector Embedding

some of the techniques are .

- **Word Embeddings:** Word embedding techniques like Word2Vec, GloVe, and FastText represent words as dense vectors in a high-dimensional space. These models learn to capture semantic relationships between words based on their co-occurrence patterns in large text corpora.
- **Document Embeddings:** Document embedding techniques aim to represent entire documents or texts as vectors. Methods like Doc2Vec, Paragraph Vectors, and Universal Sentence Encoder learn representations that encode the semantic meaning of the document, considering the context of words and sentences within it.
- **Image Embeddings:** Image embedding techniques convert images into vector representations. Convolutional Neural Networks (CNNs) are commonly used to extract high-level features from images, which are then used as embeddings. Techniques like Deep Convolutional Embedding Networks (DeCoDe) and Visual Geometry Group (VGG) models are popular for image embedding.

- **Graph Embeddings:** Graph embedding techniques aim to represent nodes or entities in a graph as vectors. These methods learn to capture structural and semantic relationships between nodes in a graph. Techniques like Graph Convolutional Networks (GCNs) and GraphSAGE are widely used for graph embedding.
- **Knowledge Graph Embeddings:** Knowledge graph embedding techniques aim to represent entities and relations in a knowledge graph as vectors. These embeddings capture the semantic relationships between entities and enable reasoning and inference. Techniques like TransE, TransR, and DistMult are commonly used for knowledge graph embedding.
- **Sequential Embeddings:** Sequential embedding techniques capture the sequential patterns in data such as time series, sequences of events, or sequences of actions. Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRUs) are often used to learn sequential representations.

1.2.2 Vector Spaces

A topic we discussed earlier is the ability to perform operations on our data , Vector spaces serve as a fundamental concept in linear algebra (the building block of AI), providing a mathematical framework for studying vectors, vector operations, and vector transformations. They form the basis for understanding linear equations, linear transformations, and various mathematical structures that enables us to apply machine learning algorithms in the most optimized manner.the idea is to map our vectors into the real valued finite dimensional vector space ,this is particulary useful because it keep similar data closer together, it also perserves data features as shown in figure**??** , here features could be seen as axes and vectors as points in our vector space (most appropriately named feature space)

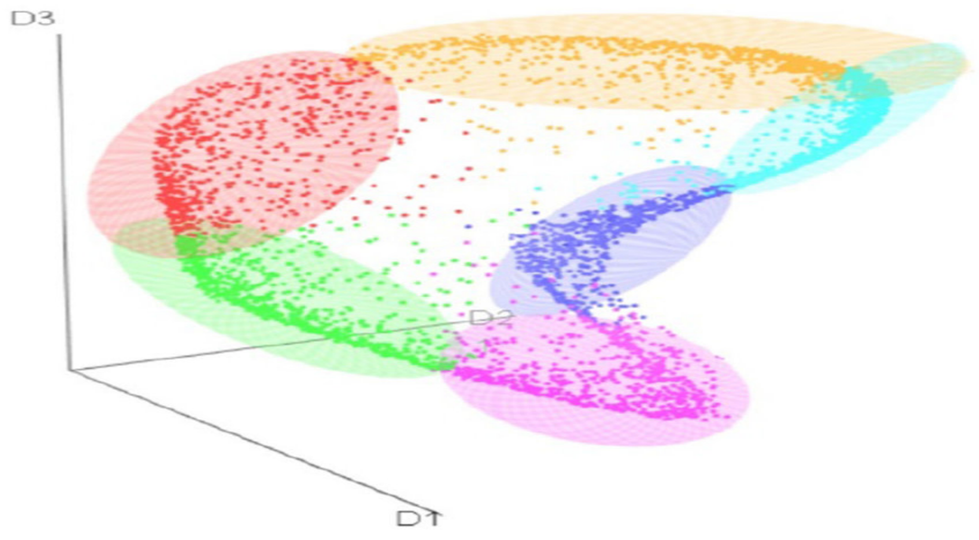


Figure 2: Visual Representation of a vector database items

1.2.3 Ml Algorithms

Having seen the data format and the environment in which these vectors live in , we need to see what type of operations would be conducted in this vector space, having established that the core need behind the vector databases is similarity even in cases where we cannot determine the criteria, Machine learning often revolves around finding the rules instead of following them and many machine learning algorithms have dealt with this problem ,we state the most used algorithms in vector databases :

- KNN short for k nearest neighbors is an unsupervised ML algorithm used to find the nearest neighbors to a data point .this algorithm is heavily used for clustering and is the backbone of vector searches as it provides the nearest vectors to the wanted query.
- dimensionality reduction as most of the times ,many of the data dimensions or features are not useful for the querying, this is especially relevant with high dimensionality data , demensionality reduction is an unsupervised technique to reduce dimensionality , resulting in better performance and faster execution time, this is particularly useful for indexing .

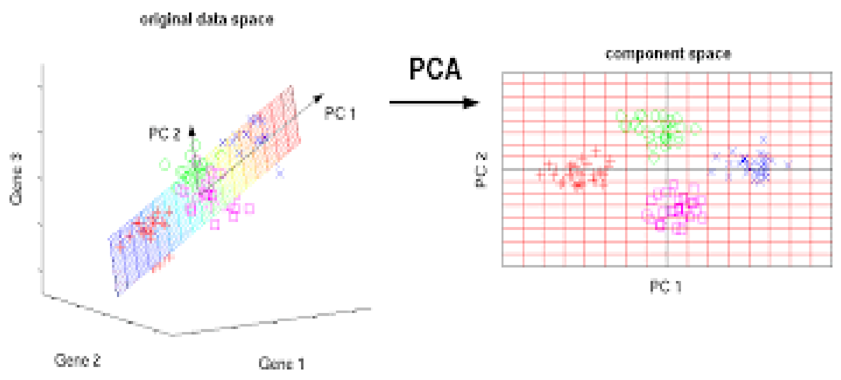


Figure 3: reducing 3d points to 2d with pca

- Neural Networks have shown exceptional accuracy and performance when dealing with extremely large dimensions of data, they are useful for many tasks regarding searching similar images,texts,voice recordings.

1.3 Problems with traditional Databases

Vector databases have the capabilities of a traditional database that are absent in standalone vector indexes and the specialization of dealing with vector embedding, which traditional scalar-based databases lack[**2**] While this type of data is sufficient , modern no-SQL databases have many weaknesses regarding handling high dimensional data which will be stated next.

- **Limited Support for High-Dimensional Vector Indexing:** NoSQL databases often lack specialized indexing techniques optimized for high dimensional vector data. Efficient indexing is crucial for performing similarity searches and other vector-specific operations. Vector databases, such as Milvus, employ advanced indexing algorithms tailored for high-dimensional vectors, enabling faster and more accurate query processing.

- **Lack of Native Vector Operations:** NoSQL databases usually lack built-in support for vector-specific operations, such as distance calculations, cosine similarity, or nearest neighbor searches. Performing these operations efficiently within a NoSQL database might require additional complex workarounds. Vector databases, on the other hand, provide native support for vector operations, making it easier and more efficient to work with vector data.

- **Difficulty in Scaling with High-Dimensional Data:** As the dimension of vector data increases, NoSQL databases can face scalability issues. The indexing structures and storage mechanisms used in NoSQL databases might struggle to handle the increased complexity and computational demands associated with high-dimensional data. Vector databases, specifically designed for efficient management of vector data, are built to handle the challenges of high-dimensional scaling.

- **Limited Optimization for Vector Search:** NoSQL databases often prioritize general-purpose features and scalability over specific optimizations for vector search operations. This can result in sub optimal query performance and longer response times when working with vector data. Vector databases, on the other hand, are purpose-built for vector-related tasks, offering specialized optimizations that significantly improve the efficiency of vector search operations.

this is why having just a library wasn't enough to handle vectors, the need for an entire database management system regarding vector manupulation became a must

1.4 Vector Search

Traditionally is we store our unstructured data in a relational or document database, we often have to assign A string representing a title to the images, often this is done manually and searching this image would be through searching it's title, this is unbelievably tedious and useless when trying to insert and search large volumes of unstructured data.vector databases tackle this directly by using a new type of queries called vector search , also known as a similarity search or nearest neighbor query, is an operation performed on a vector database to retrieve vectors that are most similar or nearest to a given query vector. The goal of a vector query is to find vectors in the database that have the closest resemblance or proximity to the query vector based on a specified distance metric as recommendation systems often look for approximate answers and not well bounded ones. This has proven to be so useful when dealing with textual input describing products and retrieving them , getting relevant answers for queries ,proposing similar videos syntactical or semantically for users. this approach includes several steps

1. **Query Vector** this steps revolves around transforming the given query into a query vector which is the target , the goal is to construct the vector that provides the best semantic representation of the searched features .for this there are various techniques taken from NLP,DeepLearning and ML such as TF-IDF(Term Frequency-Inverse Document Frequency),Bag-of-Words (BoW),Word Embeddings the latter being the most common technique for vector databases.[**3**]

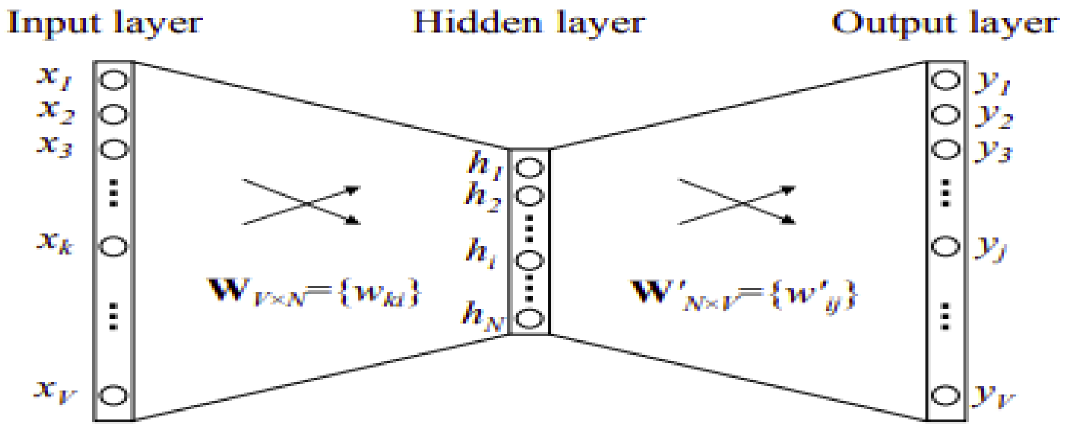


Figure 4: Word Embedding illustration

2. **Distance Calculation** this is often the heaviest step in terms of execution time[**4**] , as it calculates all the distances between target vector which is our query vector and the database items which are vectors using KNN.one of the optimizations proposed to accelerate this calculation is to calculate distance between only the near neighbor in a specified range instead of all elements which is approximate nearest neighbor(ANN). there are several distances used for this like Euclidean distance $\sqrt{\Sigma_{i=1}^{n}(u_i - v_i)^2}$, inner product $1 - u.v$, cosine similarity $1 - \frac{u.v}{||u||.||v||}$ ....

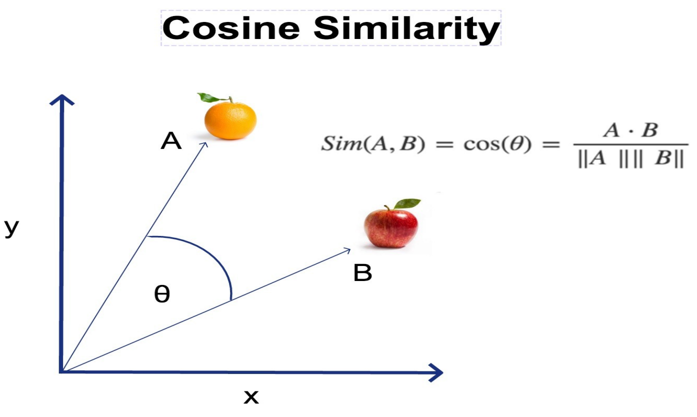

Cosine Similarity
A
B
y
x
θ
$Sim(A, B) = \cos(\theta) = \frac{A \cdot B}{||A\ |||\ B||}$

Figure 5: using cosine similarity for images illustration

3. **K Search** having calculated the distances the next approach would be to simply select the k nearest neighbors representing the result of our query .in figure 6 we have two dimensional items p_i and the query vector .$\{\mathrm{p}_2, p_3, p_6, p_9, p_{10}\}$ are the nearest neighbors and the query solution.

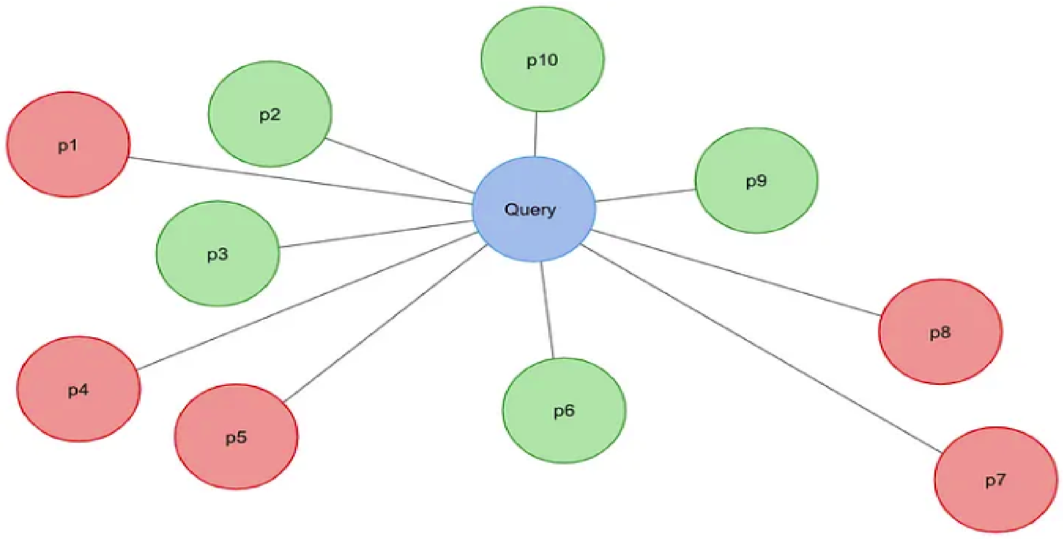


Figure 6: Result search

4. **Filtering** in queries with filters , the selected neighbors get filtered out , for example if we apply some type of filtering that only accepts p_6 and p_3

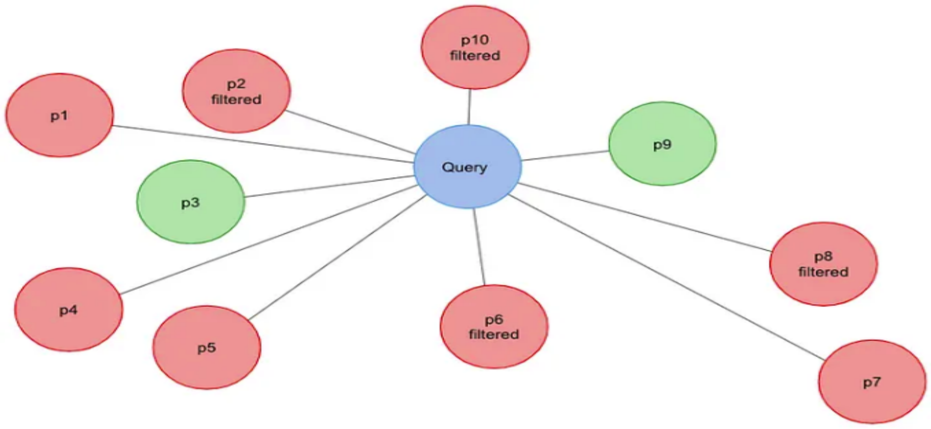


Figure 7: Filtered Result search

1.5 Indexes

While most vector databases support normal indexes used in traditional databases, the nature of the vector database specific queries calls need to new types of indexes not used in traditional databases. some databases have their own variations of the following mentioned indexes but generally many vector databases support these in a way or other.

- **Inverted File IVF** Inverted index is a data structure used in information retrieval systems to efficiently retrieve documents or web pages containing a specific term or set of terms. In an inverted index, the index is organized by terms (words), and each term points to a list of documents or web pages that contain that term[5]. this is heavily used by search engines ,NLP and information retrieval systems.

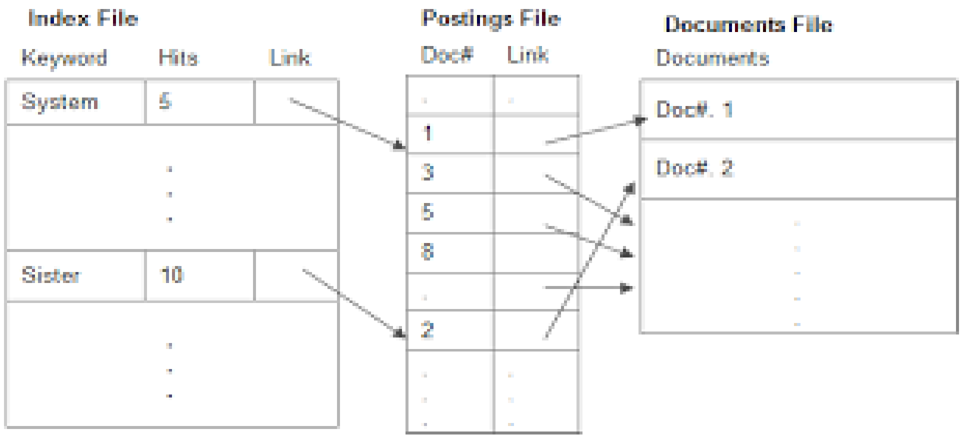


Figure 8: IVF index example

- **Product Quantization** PQ is the approach of representing a range of vectors to a finite smaller set of vectors , in the context of nearest neighbour search, the vector is first split into segments (also named subspaces) and then each segment is quantized independently.to explain it we use an example [6], we begin by a vector of 128 elements, we devide it into 8 subspaces, we perform this for every vector on our collection , so we have 1000 vector in a collection we end up with 8 subspaces each containg 1000 subvector, the magic starts by performing clustering on each subspace,bellow figure **??** is an illustration of a clustering in our context. if we choose the number of clusters to be 256 this means transforming each of our subspaces

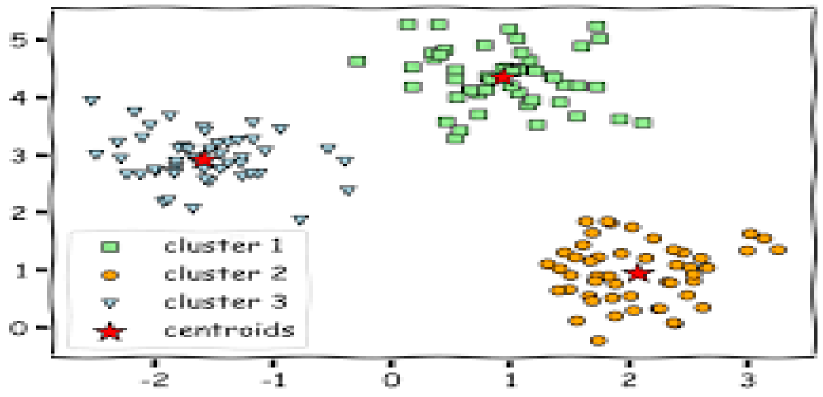
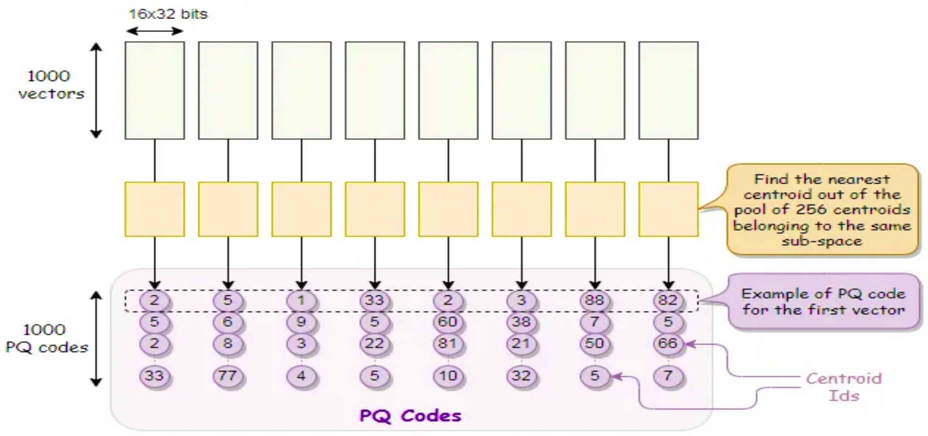


Figure 9: clustering illustration

into 256 clusters, we end up with 8*256 clusters each have the central element which we call centroid.the clusters centroids are grouped into a table called Codebook which is the table of centroids, each centroid has a scalar valued id , this scalar valued id gets inserted into a PQ code table. so



16x32 bits
1000 vectors
Find the nearest centroid out of the pool of 256 centroids belonging to the same sub-space
Example of PQ code for the first vector
1000 PQ codes
Centroid Ids
PQ Codes

Figure 10: creation of PQ codes

what we did essentially is that we reduced a 1000 vector of 128 elements into 256 array of 8 centroids a very huge memory and performance upgrade.this new table containing centroids id is called PQ codes and the one containing centroids is called codebook and it represents our index collection.now the question is how do we use them ? so given a query vector $\vec{q}$,normally we would perform knn and return k nearest neighbors, this is a bit tedious when the db is very large what we do is we devide our query vector into 8 just like we did to our db vectors, for each subspace of the query vector we calculate the distance between it and the corresponding segment from the PQ table , and we save the results in whats called a distance table shown bellow ,this table gets then sorted out ascendenly , and we then get the

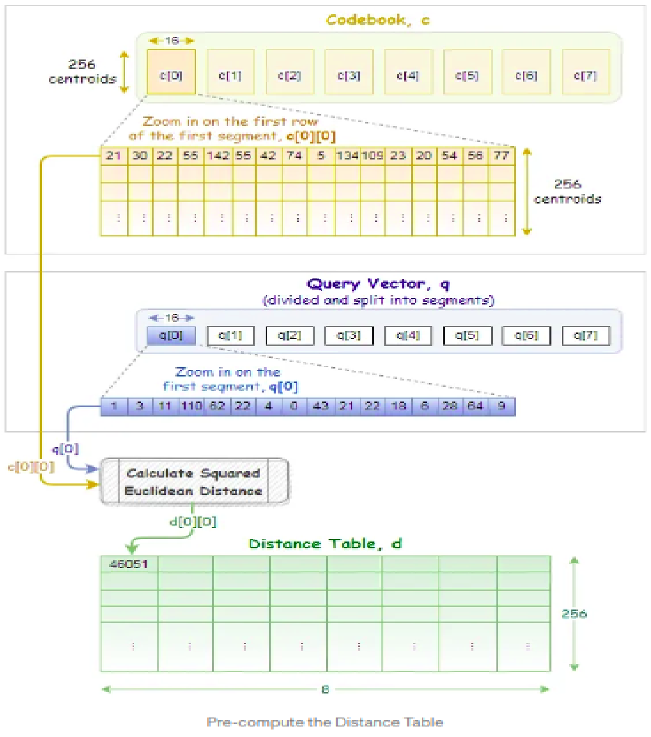


Figure 11: distance table creation

top result, this is especially fast because instead of reaching all vectors, we only look at our codebook and pq table and extract the top elements of the distance table .

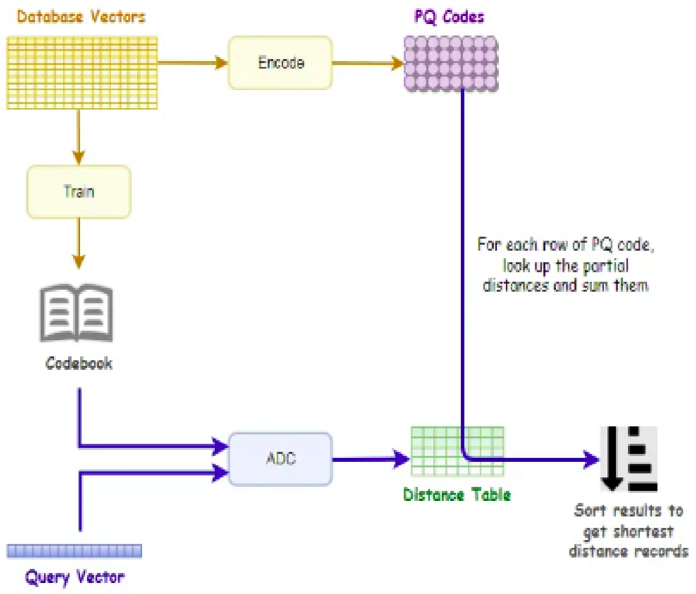

Database Vectors
Encode
PQ Codes
Train
Codebook
For each row of PQ code, look up the partial distances and sum them
ADC
Distance Table
Sort results to get shortest distance records
Query Vector

Figure 12: query with PQ index

- **Hierarchical Navigable Small Worlds (HNSW)** Hierarchical Navigable Small World (HNSW) graphs are among the top-performing indexes for vector similarity search. HNSW is a hugely popular technology that time and time again produces state-of-the-art performance with super fast search speeds and fantastic recall.[7] this unique and very new indexing technique is taking world of ai by storm,it is mainly used for image searching as an alternative to the very costly neural network, to understand it we have to understand first the two algorithms behind it .**Probabilistic skip list** a very simple algorithm , we have a vector of layers ordered in descending order, and a collection of vector items . let's say we search 11 , we

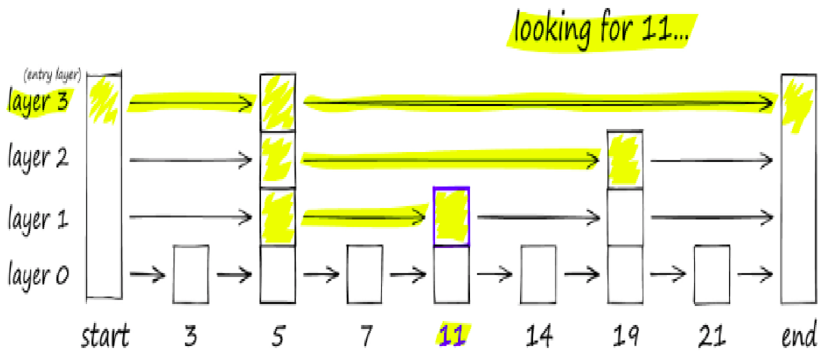


Figure 13: probabilistic skip list example

start off by visiting item 5 then end item, both of these gets marked, we move down the layers, we skip 5 , and visit 14 then skip the end item, we move to layer1 we skip 5 and we find item 11.HNSW inherits the same layered format with longer edges in the highest layers (for fast search) and shorter edges in the lower layers (for accurate search).the second algorithm is **Navigable Small World Graphs (NSW)** developed around 2011 to 2014, this was done to speed up the proximity search for proximity graphs,A proximity graph, also known as a similarity graph or neighborhood graph, is a graph-based data structure that represents the pairwise similarity or proximity relationships between a set of objects. In

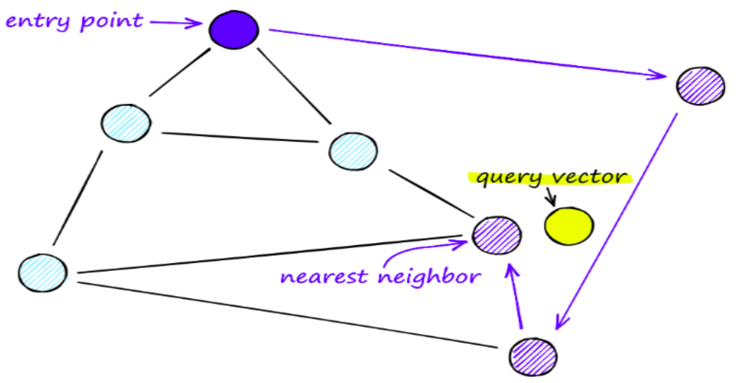


Figure 14: NSW routing

a proximity graph, each object is represented as a node, and edges are added between nodes that are considered similar or close to each other based on some distance or similarity metric.the classic proximity graphes only allowed for short range connections and eliminated long range ones, the NSW decided to keep the long range ones and this did wonders as it led to logarithmic search complexity $O(log(n))$ When searching an NSW graph, we begin at a pre-defined entry-point. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there using the greedy search.the principal routing mechanism is to start by a random entry point usually the one with the lowest number of connections, we take the long path to the connected lowest degree node, we take short paths for high degree nodes.we repeat this process until we reach the nearest neighbor.by combining these two algorithms we get the following structure where each graph in the layer represents the items vector.the connections at the top layer tend to be the longest with nodes of lowest degree and as we move down we get to the nodes with highest degree and lowest connections.

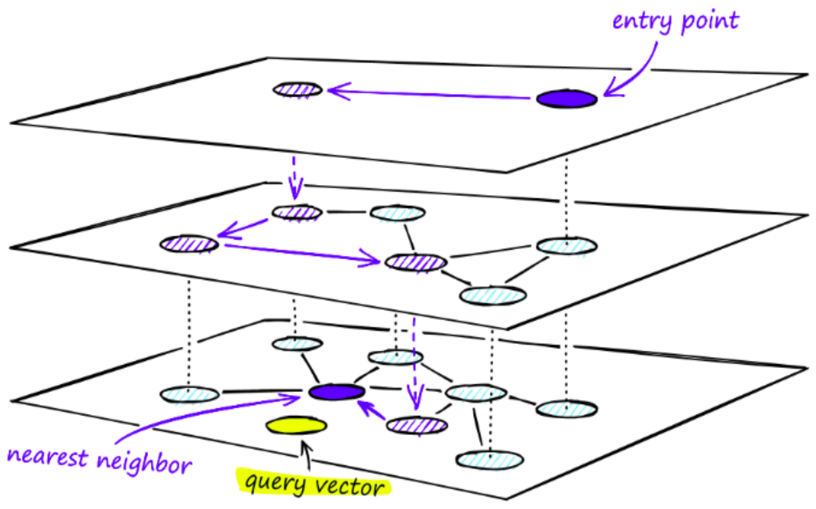


Figure 15: HNSW example

1.6 Known Vector Databases and systems

1.6.1 Weaviate

Weaviate is an open-source vector database and search engine specifically designed for working with and querying high-dimensional vector data. It is developed by the company SeMI Technologies and is built on top of the GraphQL standard for flexible and efficient data querying.it landed 16 million users this april

1.6.2 Facebook Faiss

(Facebook AI Similarity Search) is a library for efficient similarity search and clustering of large-scale vector datasets. While Faiss is not a full-fledged database like Weaviate, it can be integrated with other storage solutions to provide vector search functionality.

1.6.3 Elasticsearch

Elasticsearch is a popular distributed search and analytics engine that can handle vector data through its "dense_vector" field type.

2 Milvus overview

2.1 Introduction

Milvus, is a purpose-built open source database management system to efficiently store and search large scale vector data for data science and AI applications. It is a specialized system for high-dimensional vectors following the design practice of one-sizenot-fits-all [8] in contrast to generalizing relational databases to support vectors. Milvus provides many application interfaces (including SDKs in Python/Java/Go/C++ and RESTful APIs) that can be easily used by applications. Milvus is highly tuned for the heterogeneous computing architecture with modern CPUs and GPUs (multiple GPU devices) for the best efficiency. It supports versatile query types such as vector similarity search with various similarity functions, attribute filtering, and multi-vector query processing. It provides different types of indexes (e.g., quantization-based indexes and graph-based indexes and develops an extensible interface to easily incorporate new indexes into the system. Milvus manages dynamic vector data (e.g., insertions and deletions) via an LSM-based structure while providing consistent real-time searches with snapshot isolation. Milvus is also a distributed data management system deployed across multiple nodes to achieve scalability and availability.Milvus has become one of the most widely used vector Database management systems with its advanced features .

Figure 16: Some of Milvus users

2.2 Why Milvus ?

The majority of the work regqrding software that dealt with vector data was mainly algorithms in the form of libraries or subsystems belonging to relational databases such as facebooks faiss and microsoft SPTAG. Other works revolved around creating real vector databases but some of had problems when the amount of data was massive ,systems like Alibaba AnalyticDB-V and Alibaba PASE (PostgreSQL) had specialized vector column to support vector data,however these systems being not fully dedicated to vector data renderd many of the needed operations on vectors very limited. so in here comes milvus , it is fully dedicated to vector data with a many advanced and new techniques to handle them,it architecture takes advantage of GPU and CPU parallelization to provide the best performance , the focus on vectors renders GPU optimization a wonder for these types of operations , Milvus is also known to be the only database along side weaviate to support multivector queries.A table taken from a research paper shows how Milvus surpasses many vector system.

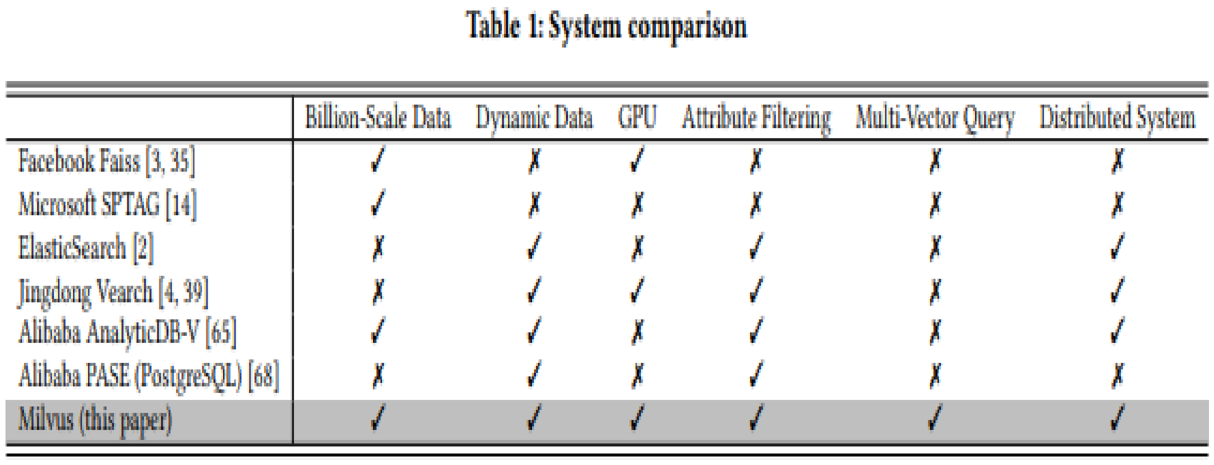


Table 1: System comparison

	Billion-Scale Data	Dynamic Data	GPU	Attribute Filtering	Multi-Vector Query	Distributed System
Facebook Faiss [3, 35]	✓	✗	✓	✗	✗	✗
Microsoft SPTAG [14]	✓	✗	✗	✗	✗	✗
ElasticSearch [2]	✗	✓	✗	✓	✗	✓
Jingdong Vearch [4, 39]	✗	✓	✓	✓	✗	✓
Alibaba AnalyticDB-V [65]	✓	✓	✗	✓	✗	✓
Alibaba PASE (PostgreSQL) [68]	✗	✓	✗	✓	✗	✗
Milvus (this paper)	✓	✓	✓	✓	✓	✓

Figure 17: Comparaison between milvus and other databases

3 Milvus Architecture

Milvus was designed for similarity search on dense vector datasets containing millions, billions, or even trillions of vectors. Before proceeding, familiarize yourself with the basic principles of embedding retrieval. Milvus also supports data sharding, data persistence, streaming data ingestion, hybrid search between vector and scalar data, time travel, and many other advanced functions. The platform offers performance on demand and can be optimized to suit any embedding retrieval scenario. We recommend deploying Milvus using Kubernetes for optimal availability and elasticity. Milvus adopts a shared-storage architecture featuring storage and computing disaggregation and horizontal scalability for its computing nodes. Following the principle of data plane and control plane disaggregation, Milvus comprises four layers: access layer, coordinator service, worker node, and storage. These layers are mutually independent when it comes to scaling or disaster recovery

3.1 General Design

Milvus was built on top of facebooks faiss, it is implemented using C++ with use of OpenMp libraries, it's composed of three major components the query engine,the GPU engine, and the storage engine. The query engine support efficient vector searching and similarity search along with the vector operations, The GPU engine is a co-processing engine that accelerates performance with vast parallelism taking full of advantage of the vast amount of gpu cores that are dedicated to vector operations. It also supports multiple GPU devices for efficiency. The storage engine enables data durability and incorporates an LSM-based structure (merge trees) for dynamic data management. It runs on various file systems (including local file systems, Amazon S3, and HDFS) with a bufferpool in memory.

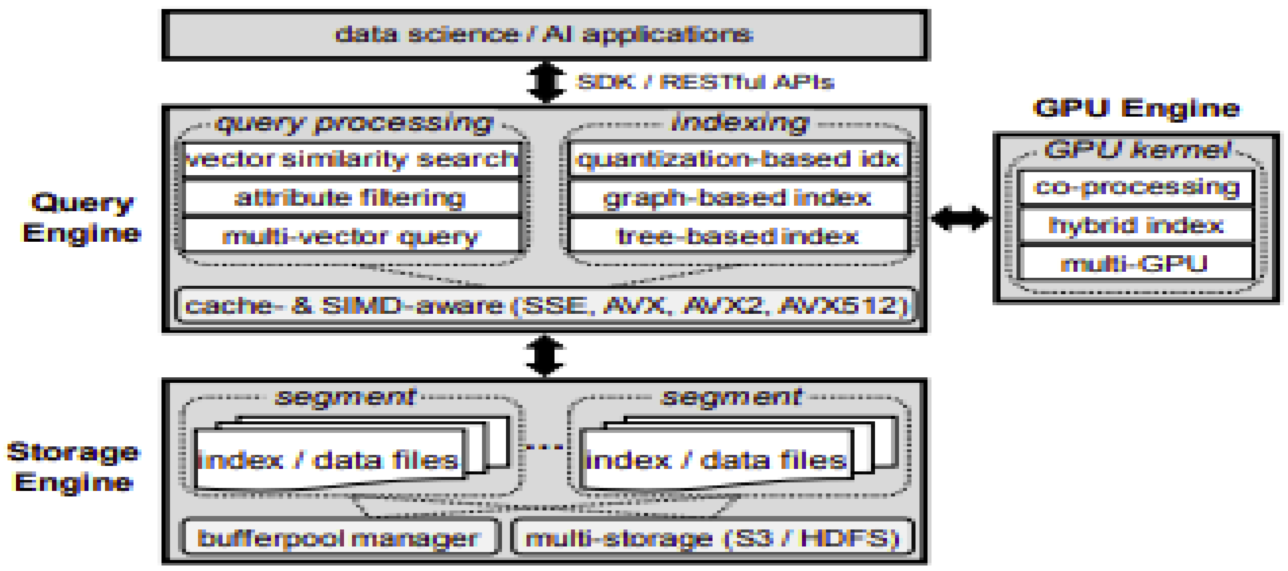


Figure 1: System architecture of Milvus

Figure 18: Overview of the architecture of milvus

3.2 Distributed Architecture

Milvus adopts a shared-storage architecture featuring storage and computing disaggregation and horizontal scalability for its computing nodes. Following the principle of data plane and control plane disaggregation, Milvus comprises four layers: access layer, coordinator service, worker node, and storage. These layers are mutually independent when it comes to scaling or disaster recovery.[**9**] it also supports data sharding, data persistence, streaming data ingestion, hybrid search between vector and scalar data, time travel, and many other advanced functions. The platform offers performance on demand and can be optimized to suit any embedding retrieval scenario. We recommend deploying Milvus using Kubernetes for optimal availability and elasticity.

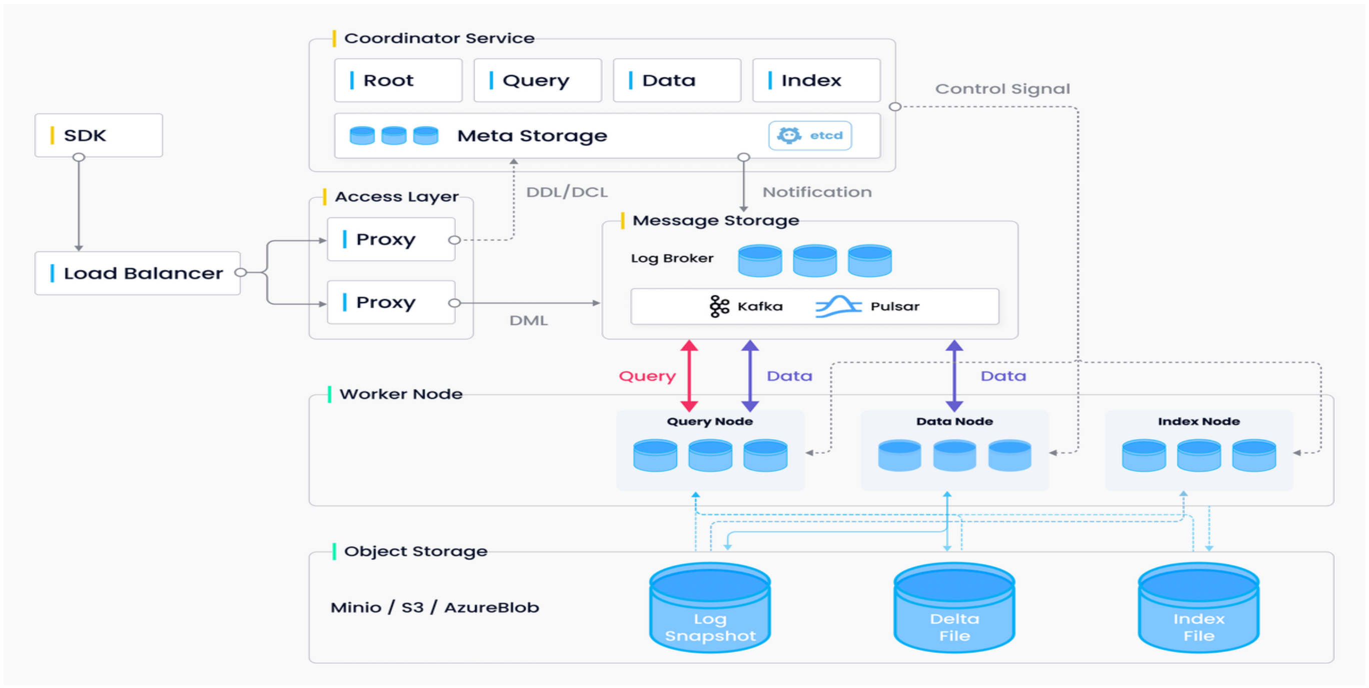


Figure 19: Milvus Distributed Architecture

3.3 main components

3.3.1 SDK

Milvus provides SDKs in various programming languages, including Python, Java, C++, Go, and more. These SDKs enable developers to integrate Milvus into their applications and leverage its vector similarity search capabilities. The SDKs provide an interface to interact with the Milvus database, allowing users to store vectors, perform similarity searches, and manage the database efficiently.

3.3.2 Load balancer

the query nodes which we will talk about later , are responsible for executing and optimizing incoming queries, the load balancer maes sure that the work load remains balanced for maximum performance and scalability.

3.3.3 Access layer

this serves as Proxy that serves as a portal of data insertion requests.Initially, proxy accepts data insertion requests from SDKs, and allocates those requests into several buckets using hash algorithm. Then the proxy requests data coord to assign segments, the smallest unit in Milvus for data storage.Afterwards, the proxy inserts information of the requested segments into message store so that these information will not be lost.

3.3.4 coordinator service

Responsible for allocation of segments and work distribution. it has many functionalities such as allocating space for segments, and it manages their expiration date, it also allocates the the channels to their suitable data node. it also ensures persistence between the cached data and the data node. the nodes

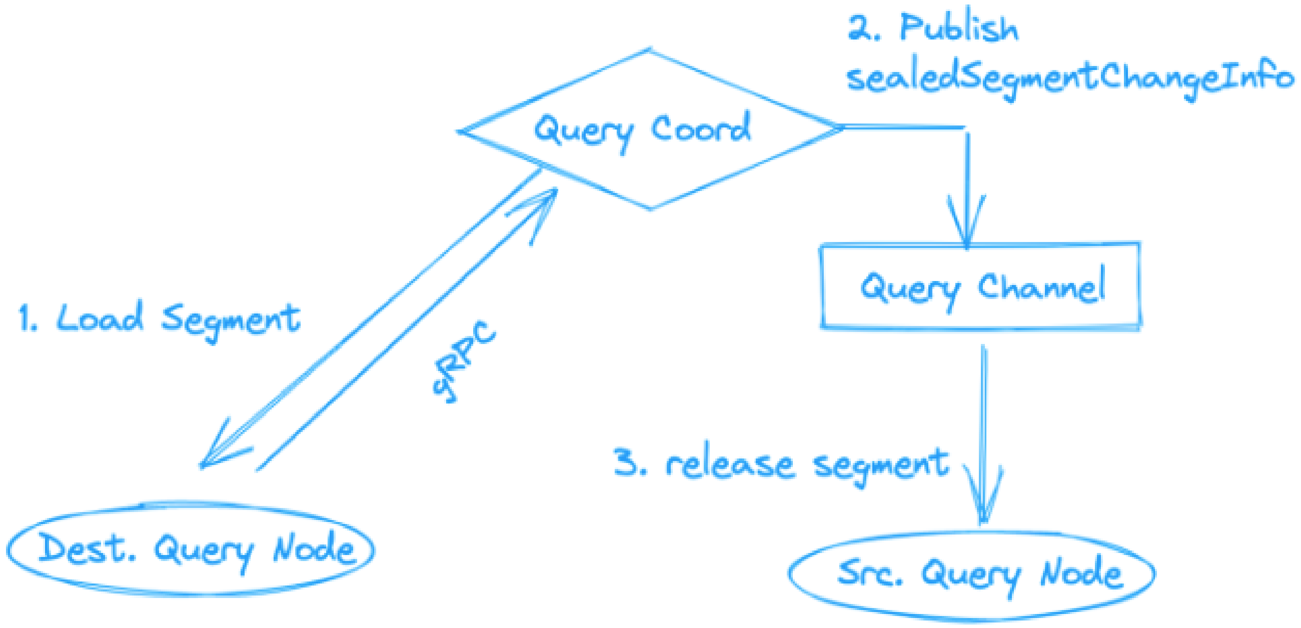


Figure 20: Coordinator illustration

belonging to the coordinator service are of four types :

- **Query node and coordinator:** Query node retrieves incremental log data and turn them into growing segments by subscribing to the log broker, loads historical data from the object storage, and runs hybrid search between vector and scalar data.it coordinator manages topology and load balancing for the query nodes, and handoff from growing segments to sealed segments.

- **Root coordinator and node:** handles data definition language (DDL) and data control language (DCL) requests, such as create or delete collections, partitions, or indexes, as well as manage TSO (timestamp Oracle) and time ticker issuing.
- **query coordinator and node** Query node retrieves incremental log data and turn them into growing segments by subscribing to the log broker, loads historical data from the object storage, and runs hybrid search between vector and scalar data.
- **Index coordinator and node :** manages topology of the index nodes, builds index, and maintains index metadata.its coordinator builds indexes. Index nodes do not need to be memory resident, and can be implemented with the serverless framework.
- **Data coordinator and node** Data coordinator manages topology of the data nodes, maintains metadata, and triggers flush, compact, and other background data operations. Data node retrieves incremental log data by subscribing to the log broker, processes mutation requests, and packs log data into log snapshots and stores them in the object storage.

3.3.5 Meta Data

Milvus uses etcd for storing metadata. This topic introduces how to configure meta storage dependency when you install Milvus with Milvus Operator.the ETCD is configurable using

3.3.6 Message Storage

It's responsible for storing and managing messages or events within the milvus db . It involves persisting messages generated by applications or systems for various purposes such as communication, logging, auditing, event sourcing, or real-time data processing.Milvus uses RocksMQ, Pulsar or Kafka for managing logs of recent changes, outputting stream logs, and providing log subscriptions.

3.3.7 Log Broker

The log broker is a pub-sub system that supports playback. It is responsible for streaming data persistence, execution of reliable asynchronous queries, event

notification, and return of query results. It also ensures integrity of the incremental data when the worker nodes recover from system breakdown. Milvus cluster uses Pulsar as log broker; Milvus standalone uses RocksDB as log broker. Besides, the log broker can be readily replaced with streaming data storage platforms such as Kafka and Pravega.
Milvus is built around log broker and follows the "log as data" principle, so Milvus does not maintain a physical table but guarantees data reliability through logging persistence and snapshot logs.

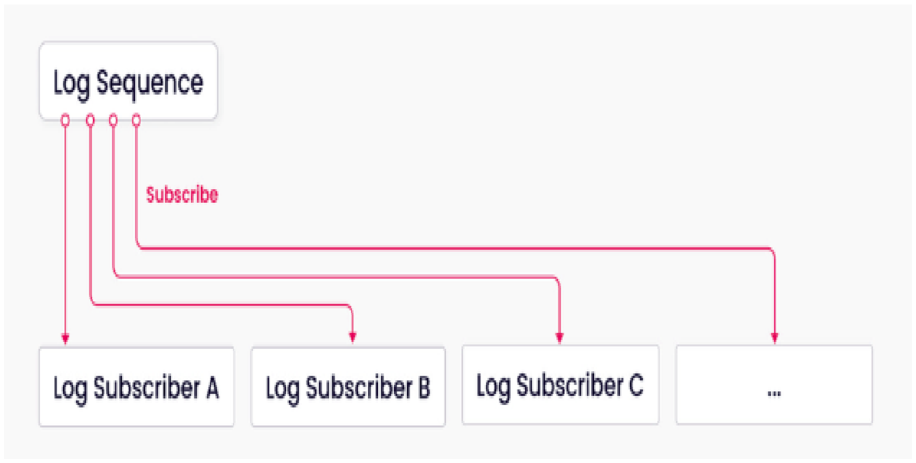


Figure 21: Log broker illustration

3.3.8 Worker Node

this represents a single entity of the distributed node systems , the backbone of the system responsible for charging the data from the data storage to the main memory .the index nodes store manage the created indexes, the data node handles the loaded data and the query node takes care of calculations .

3.3.9 Object Storage

Milvus uses an Object storage which is a type of data storage architecture that manages and organizes data as discrete units called objects. Unlike traditional file systems or block storage, which organize data into a hierarchical structure or fixed-size blocks, object storage stores data as self-contained entities with unique identifiers and metadata. Milvus uses MinIO or S3 as object storage to persist large-scale files, such as index files and binary logs.

4 Knowhere system Indexes

Having seen the classic indexes used in the regular vector databases and libraries, Milvus takes a bit of a enovative approach with the nowhere concept. Knowhere is an operation interface for accessing services in the upper layers of the system and vector similarity search libraries like Faiss, Hnswlib, Annoy in the lower layers of the system. In addition, Knowhere is also in charge of heterogeneous computing. More specifically, Knowhere controls on which hardware (eg. CPU or GPU) to execute index building and search requests. This is how Knowhere gets its name - knowing where to execute the operations. More types of hardware including DPU and TPU will be supported in future releases.In a broader sense, Knowhere also incorporates other third-party index libraries like Faiss. Therefore, as a whole, Knowhere is recognized as the core vector computation engine in the Milvus vector database.Computation in Milvus mainly involves vector and scalar operations. Knowhere only handles the operations on vectors in Milvus. The figure above illustrates the Knowhere architecture in Milvus.The bottom-most layer is the system hardware. The third-party index libraries are on top of the hardware. Then Knowhere interacts with the index node and query node on the top via CGO.

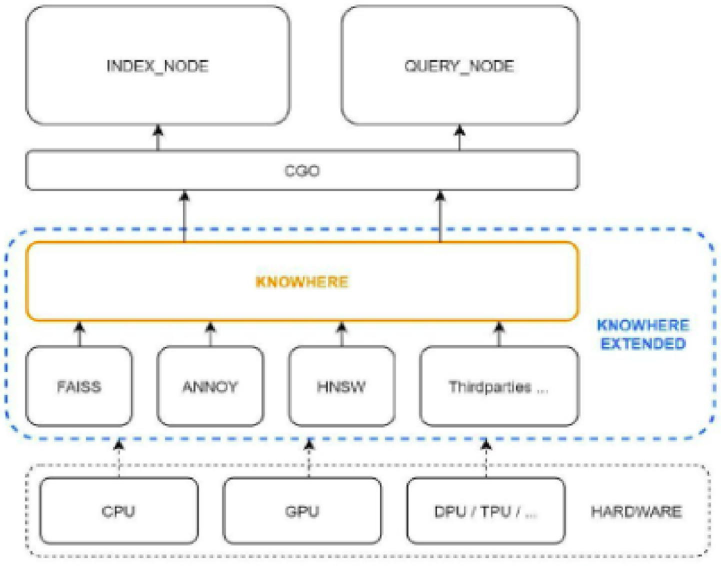


Figure 22: knowhere concept in milvus

Knowhere provides index like :

- The Faiss index has two sub classes: FaissBaseIndex for all indexes on float point vectors, and FaissBaseBinaryIndex for all indexes on binary vectors.
- GPUIndex is the base class for all Faiss GPU indexes.to optimize the search performance for specific use cases.
- OffsetBaseIndex is the base class for all self-developed indexes. Only vector ID is stored in the index file. As a result, an index file size for 128-dimensional vectors can be reduced by 2 orders of magnitude. We recommend taking the original vectors into consideration as well when using this type of index for vector similarity search.
- IDMAP is not exactly an index, but rather used for brute-force search. When vectors are inserted to the vector database, no data training and index building is required. Searches will be conducted directly on the inserted vector data.
- The IVF (inverted file) indexes are the most frequently used. The IVF class is derived from VecIndex and FaissBaseIndex, and further extends to IVFSQ and IVFPQ. GPUIVF is derived from GPUIndex and IVF. Then GPUIVF further extends to GPUIVFSQ and GPUIVFPQ.
- IVFSQHybrid is a class for self-developed hybrid index that is executed by coarse quantize on GPU. And search in the bucket is executed on CPU. This type of index can reduce the occurrence of memory copy between CPU and GPU by leveraging the computing power of GPU. IVFSQHybrid has the same recall rate as GPUIVFSQ but comes with a better performance.

5 Data Model and Organization

Milvus stores data in 4 diffrent levels of nesting which segments,partitions,shards and collections.

5.1 Collection

A collection in Milvus ,can be seen as the equivalent to a table in a relational storage system. Collection is the biggest data unit in Milvus.it composed of multiple shards.

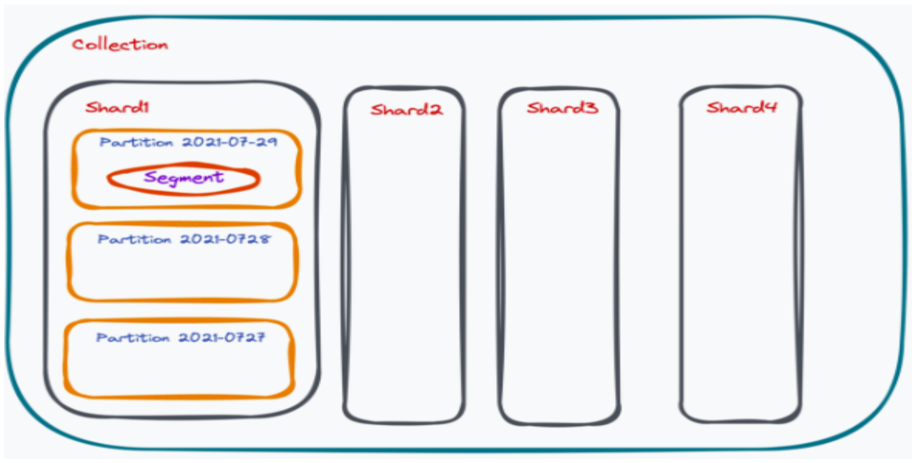


Figure 23: Milvus data model

5.2 shards

Sharding also known as channeling is a technique used in distributed database systems to horizontally partition data across multiple nodes or servers called shards. It is employed to improve the scalability, performance, and availability of a database by distributing the data and workload across multiple machines. In a sharded database, the dataset is divided into smaller subsets called shards, and each shard is hosted on a separate server or node. Each shard is responsible for storing a specific portion of the data. This division can be based on different criteria, such as ranges of values, hash functions, or predefined rules.To take full advantage of the parallel computing power of clusters when writing data, collections in Milvus must spread data writing operations to different nodes. By default, a single collection contains two shards. Depending on your dataset volume, we can have more shards in a collection. Milvus uses a master key hashing method for sharding.

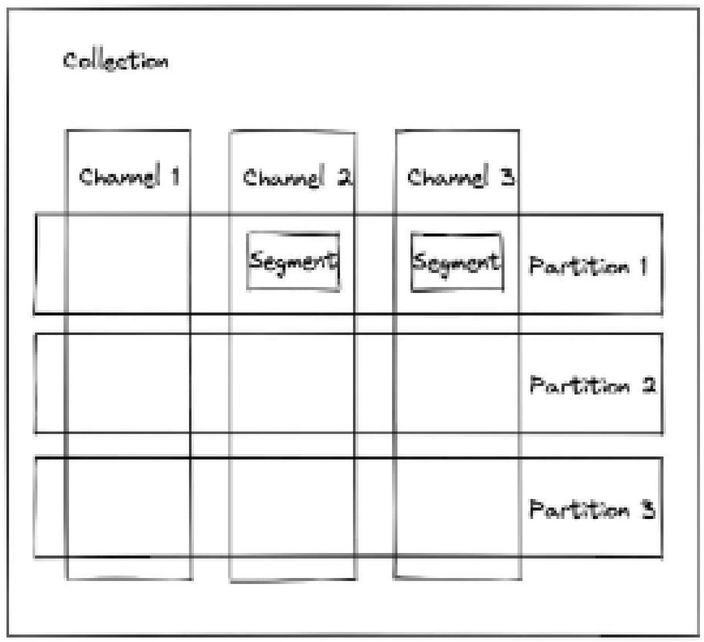


Figure 24: Data organization in milvus

a shard is allocated When data node starts or shuts down or When segment space allocated is requested by proxy. Then there are several strategies of sharding allocation. Milvus supports 2 of the strategies:

- **Consistent hashing:** The default strategy in Milvus, This strategy leverages the hashing technique to assign each channel a position on the ring, then searches in a clock-wise direction to find the nearest data node to a channel. Thus, in the illustration above, channel 1 is allocated to data node 2, while channel 2 is allocated to data node 3. However, one problem with this strategy is that the increase or decrease in the number of data nodes

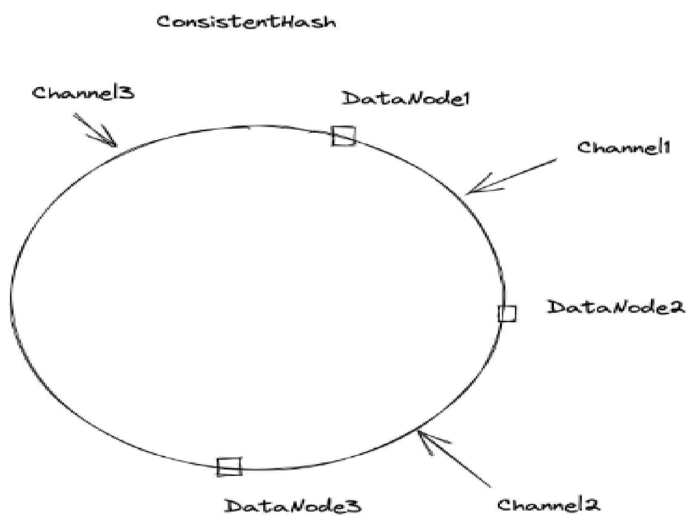


Figure 25: consistent hashing

- **Load balancing** The second strategy is to allocate channels of the same collection to different data nodes, ensuring the channels are evenly allocated. The purpose of this strategy is to achieve load balance.

5.3 Partitions

Milvus divides the data in the collection into multiple parts on physical storage based on certain rules. Such operation is called partitioning. Each partition can contain multiple segments. A partition is identified by a tag. When inserting vector data, you can use the tag to specify which partition to insert the data into. When querying vector data, you can use the tag to specify the partition where the query should be executed. Milvus supports both the exact matching and regular expression matching for partition tags.

5.4 Segments

The smallest storage entity in milvus ,There are three types of segments with different status in Milvus: growing, sealed, and flushed segment:

5.4.1 Growing Segment

A growing segment is a newly created segment that can be allocated to the proxy for data insertion. The internal space of a segment can be used, allocated, or free.it is consumed by the data node meaning its data is moved from the storage to the main memory and requested by the proxy and allocated by data coordinator meaning theres space allocated for it in the memory. Allocated space will expire after a certain period time.

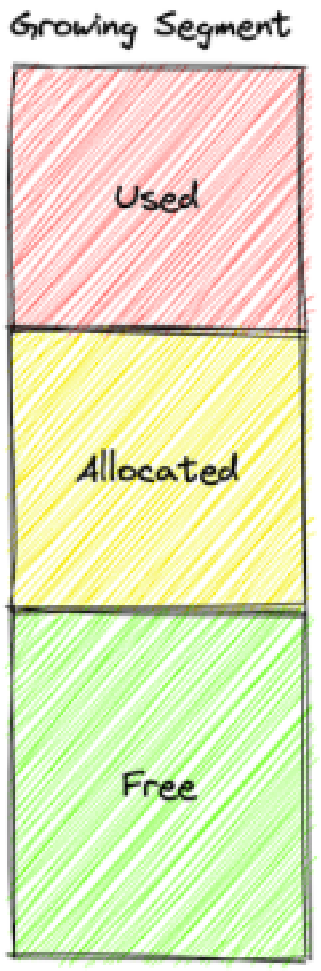


Figure 26: Growing segment

5.4.2 Sealed Segment

A sealed segment is a closed segment that can no longer be allocated to the proxy for data insertion.usually by default the maximum size of a segment in milvus is 512MB ,when the operations of memory wrting into disk are manuel then we get the curcumstance of when reaching 75% of this the segment usually stops growing meaning it's sealed.

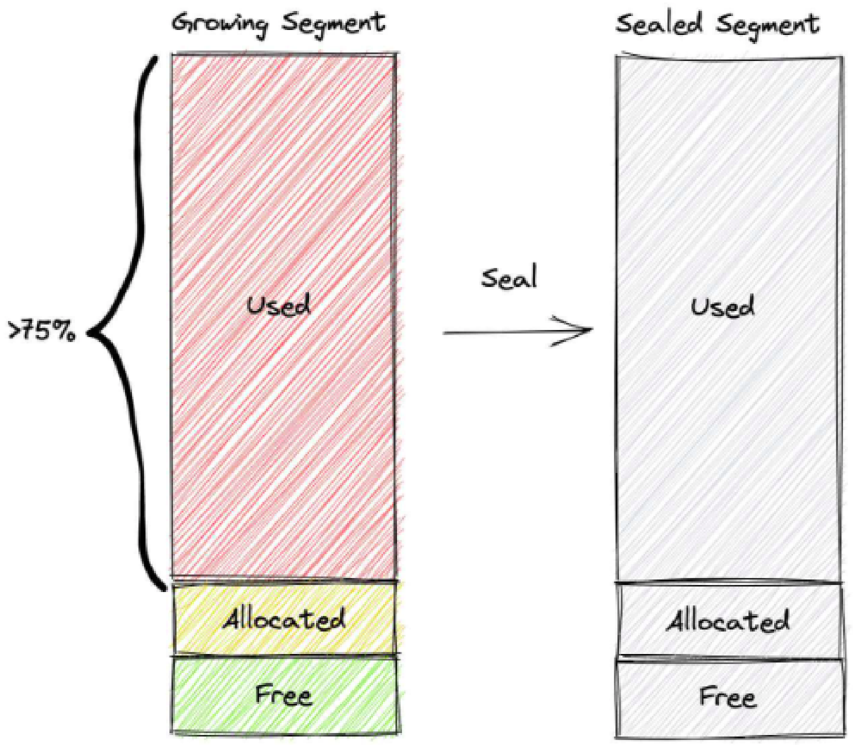


Figure 27: Sealed segment

5.4.3 Flushed Segment

A flushed segment is a segment that has already been written into disk. Flush refers to storing segment data to object storage for the sake of data persistence. A segment can only be flushed when the allocated space in a sealed segment expires. When flushed, the sealed segment turns into a flushed segment.this could be seen as the equivelent of the commit operation in relational databases.

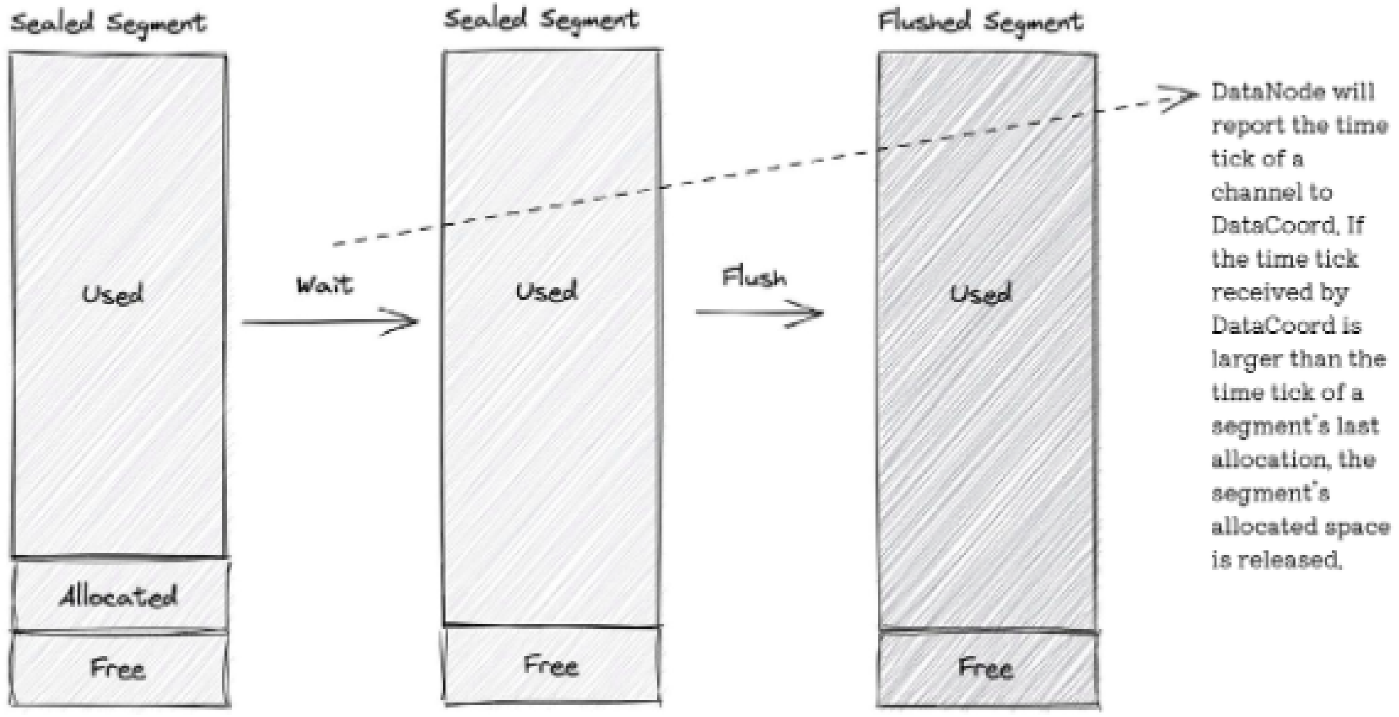

Sealed Segment
Used
Allocated
Free
Wait
Sealed Segment
Used
Free
Flush
Flushed Segment
Used
Free
DataNode will report the time tick of a channel to DataCoord, If the time tick received by DataCoord is larger than the time tick of a segment's last allocation, the segment's allocated space is released.

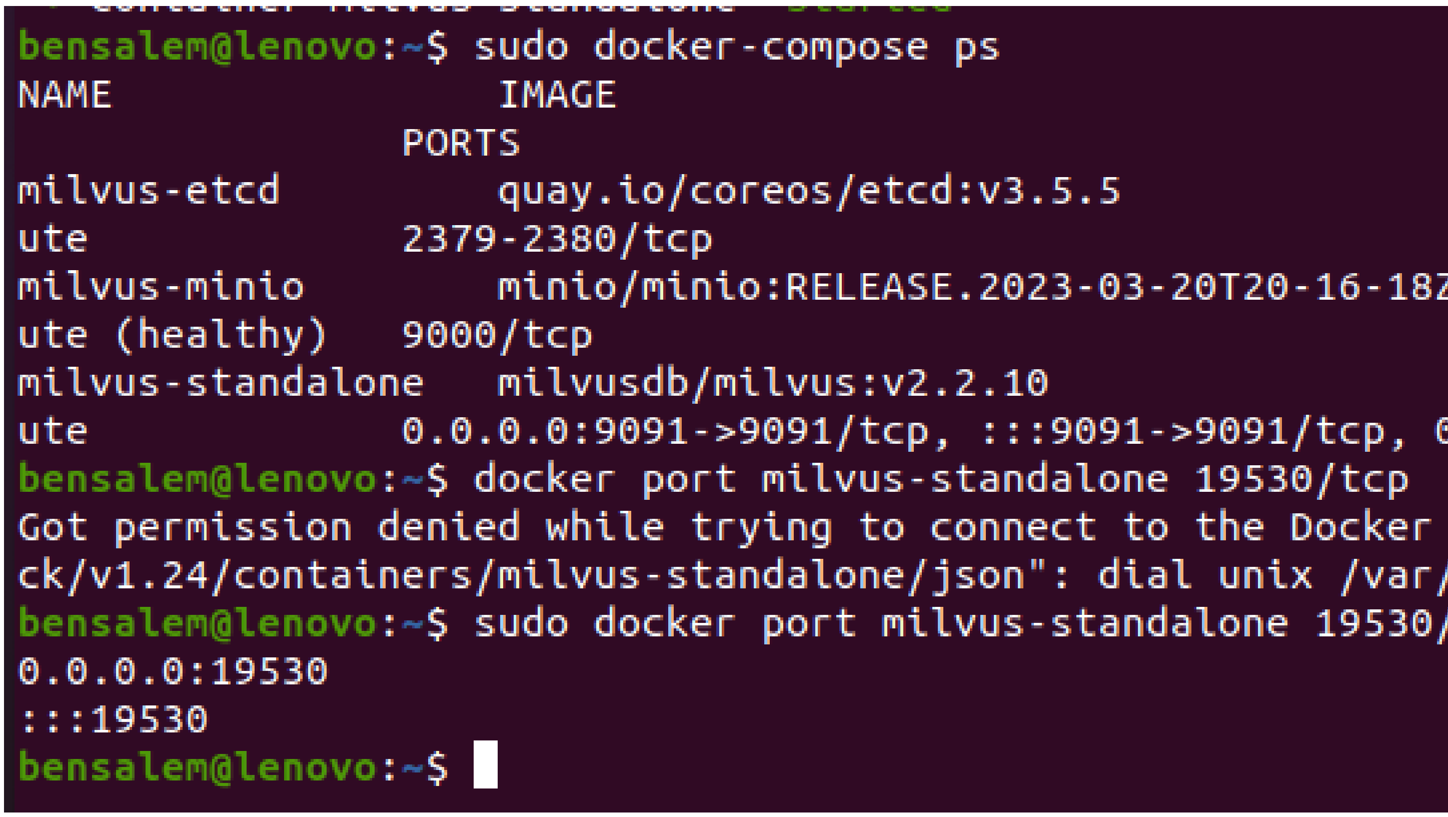
Figure 28: Flushed Segment

6 Milvus Installation and connection

6.1 Docker Installation

the installation process in docker is fairly straightforward and easy, we go through the following commands

```
//installing milvus yaml file
    wget https://github.com/milvusio/milvus/releases/download/v2.2.10/milvu
//start milvus docker
sudo docker-compose ps
//connect to the milvus server through port 19530
sudo docker port milvus-standalone 19530/tcp
```



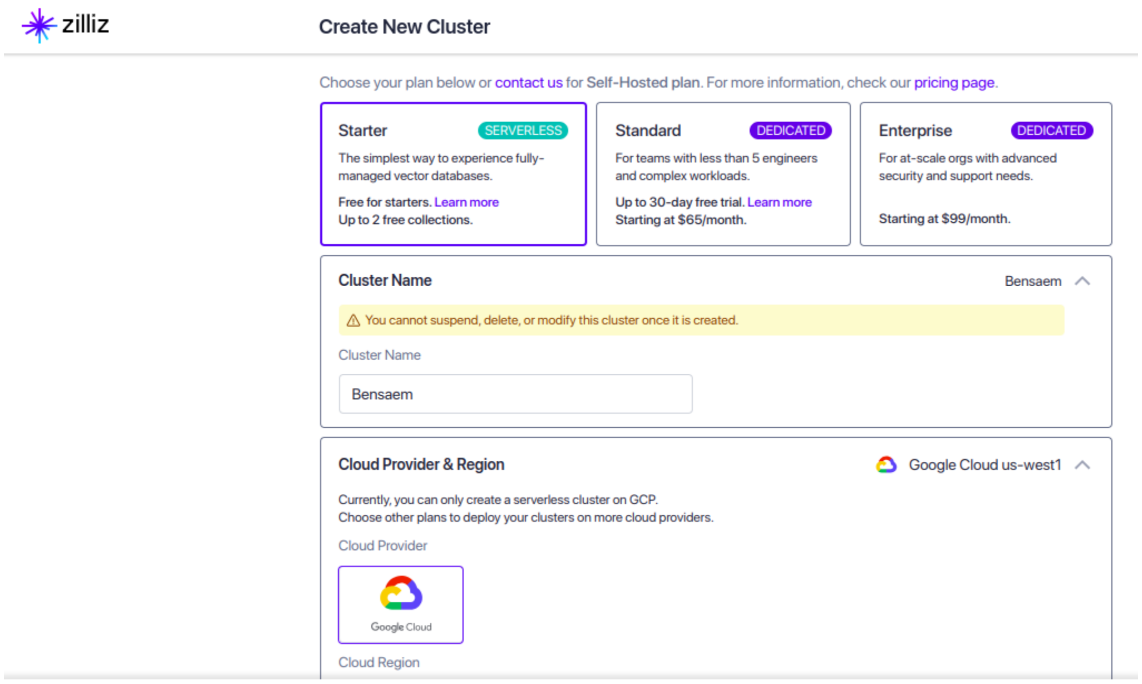
```
bensalem@lenovo:~$ sudo docker-compose ps
NAME                    IMAGE
                PORTS
milvus-etcd             quay.io/coreos/etcd:v3.5.5
ute             2379-2380/tcp
milvus-minio            minio/minio:RELEASE.2023-03-20T20-16-18Z
ute (healthy)   9000/tcp
milvus-standalone       milvusdb/milvus:v2.2.10
ute             0.0.0.0:9091->9091/tcp, :::9091->9091/tcp, 0
bensalem@lenovo:~$ docker port milvus-standalone 19530/tcp
Got permission denied while trying to connect to the Docker
ck/v1.24/containers/milvus-standalone/json": dial unix /var/
bensalem@lenovo:~$ sudo docker port milvus-standalone 19530/
0.0.0.0:19530
:::19530
bensalem@lenovo:~$
```

Figure 29: Milvus installation docker

Milvus can also be installed inside clusters through the Milvus Operator which is a solution that helps you deploy and manage a full Milvus service stack to target Kubernetes (K8s) clusters. The stack includes all Milvus components and relevant dependencies like etcd, Pulsar and MinIO. This topic introduces how to deploy a Milvus cluster with Milvus Operator on K8s.

6.2 Cloud Zilliz

Cloud zilliz is a platform that allows for the creation and hosting milvus vector databases.it allows for the creation of a single cluster and 2 collections at max. the registration processes go as follows,after login in we choose our usage plan



zilliz

Create New Cluster

Choose your plan below or contact us for Self-Hosted plan. For more information, check our pricing page.

Starter SERVERLESS The simplest way to experience fully-managed vector databases. Free for starters. Learn more Up to 2 free collections.	Standard DEDICATED For teams with less than 5 engineers and complex workloads. Up to 30-day free trial. Learn more Starting at $65/month.	Enterprise DEDICATED For at-scale orgs with advanced security and support needs. Starting at $99/month.

Cluster Name — Bensaem

⚠ You cannot suspend, delete, or modify this cluster once it is created.

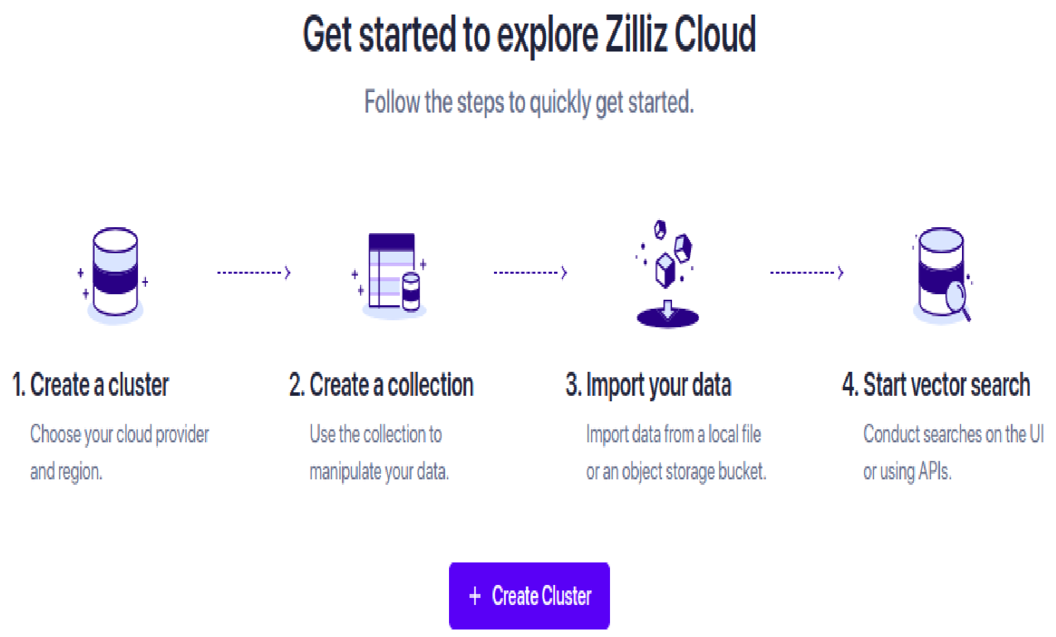
Cluster Name: Bensaem

Cloud Provider & Region — Google Cloud us-west1

Currently, you can only create a serverless cluster on GCP.
Choose other plans to deploy your clusters on more cloud providers.

Cloud Provider: Google Cloud

Cloud Region



Get started to explore Zilliz Cloud

Follow the steps to quickly get started.

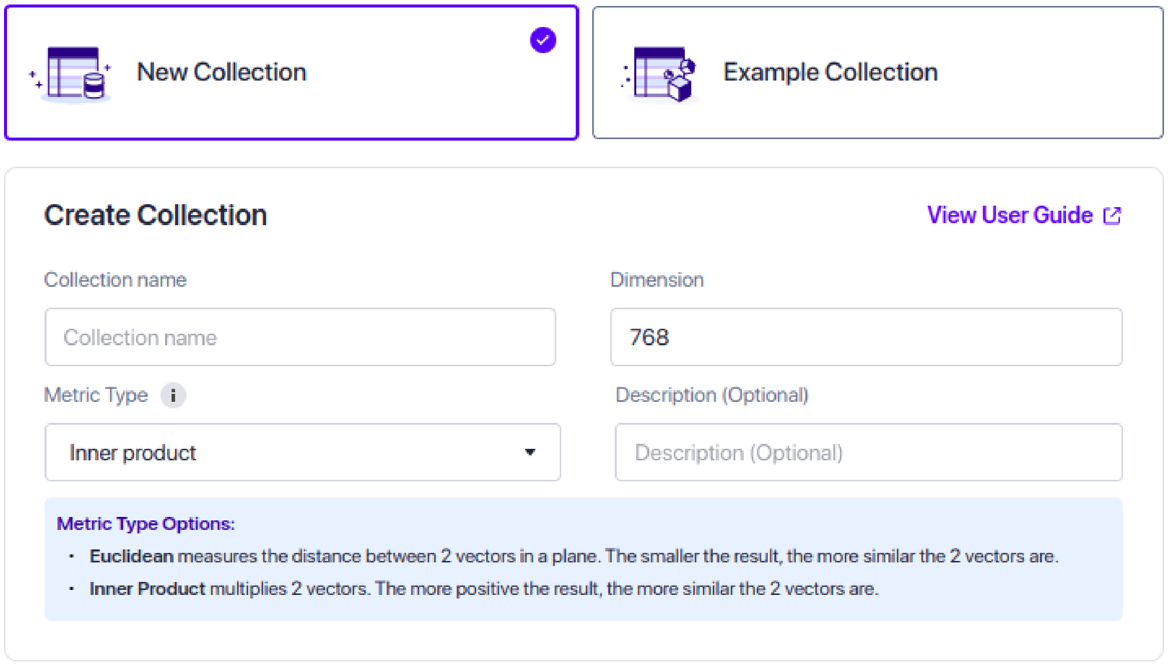
1. Create a cluster
Choose your cloud provider and region.

2. Create a collection
Use the collection to manipulate your data.

3. Import your data
Import data from a local file or an object storage bucket.

4. Start vector search
Conduct searches on the UI or using APIs.

then we create our cluster. the metric type refers to the distance measurement seen back in the vector databases overview



Collection name: Collection name

Dimension: 768

Metric Type: Inner product

Description (Optional): Description (Optional)

Metric Type Options:

- **Euclidean** measures the distance between 2 vectors in a plane. The smaller the result, the more similar the 2 vectors are.
- **Inner Product** multiplies 2 vectors. The more positive the result, the more similar the 2 vectors are.

6.3 SDK Connection

Milvus supports a wide range of SDK (python,NodeJs,Java...) the method of connection differs depending on whether we are using the cloud or using it localy,we will be using python for the manupulation of the collection.we start by installing pymilvus a library that supports and links between milvus and python sdk.using the command

```
pip3 install -b pymilvus
```

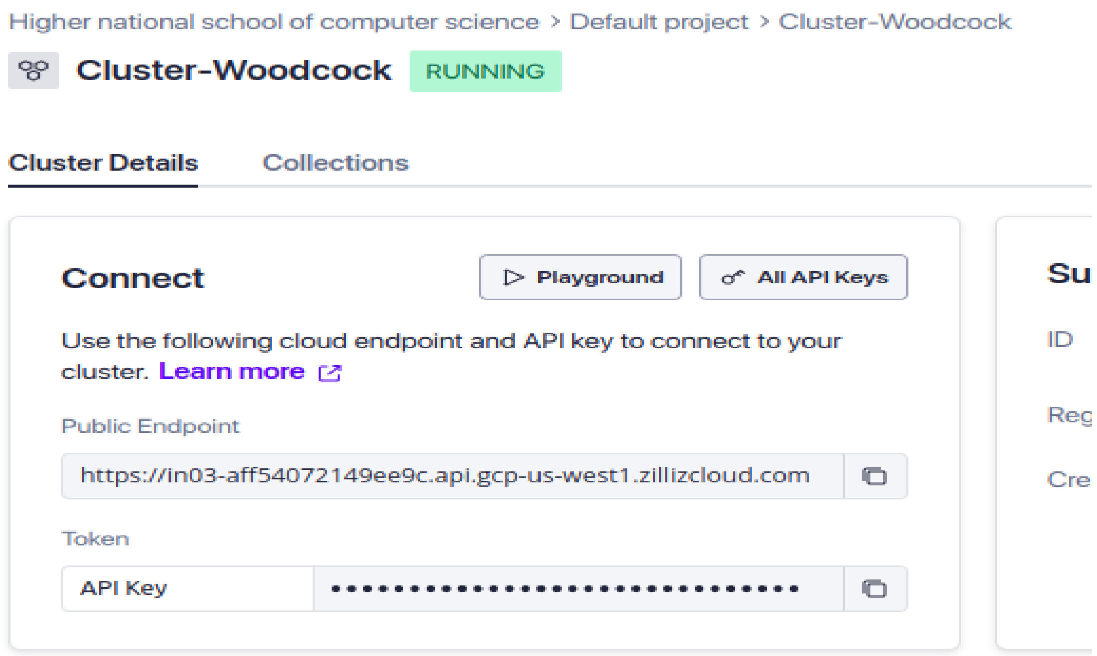
once this is ready we pass on to the connection.

6.3.1 API connection

6.3.2 Cloud connection

the connection to the cloud is done through API key and the Endpoint, of these are found at the first page in the cluster. the connection would be of the form the token is the api key and the uri is our endpointthen inside the code we connect through the command

```
connections.connect("default",
                    uri=milvus_uri,
                    token=token)
   print(f"Connecting to DB: {milvus_uri}")
```



Higher national school of computer science > Default project > Cluster-Woodcock

Cluster-Woodcock RUNNING

Connect

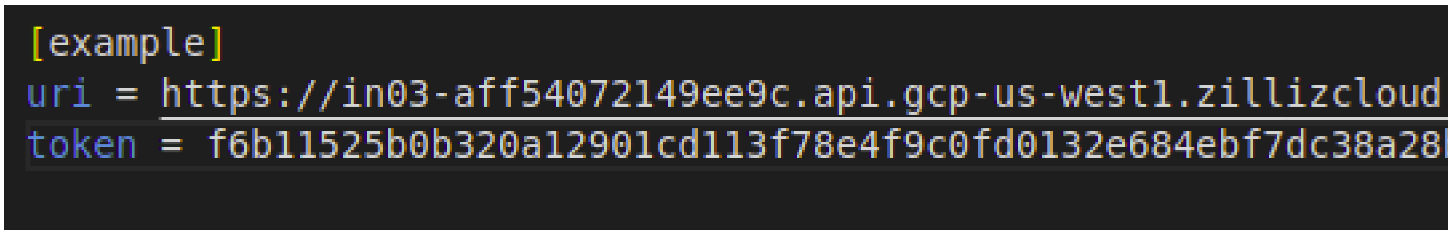
Use the following cloud endpoint and API key to connect to your cluster. Learn more

Public Endpoint

https://in03-aff54072149ee9c.api.gcp-us-west1.zillizcloud.com

Token

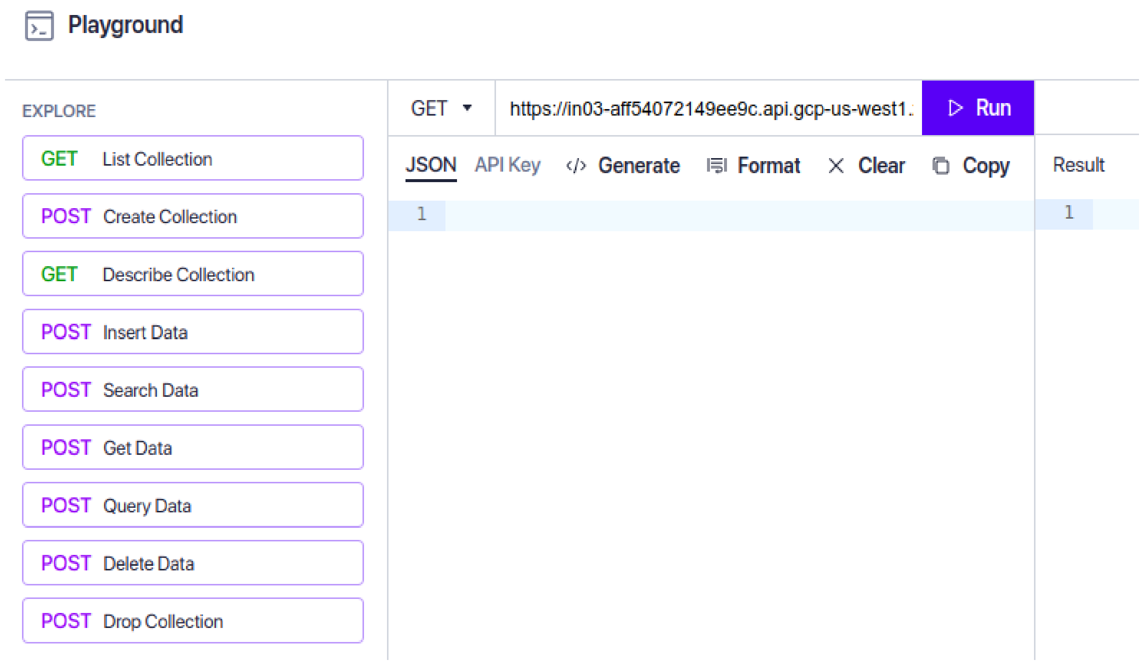
API Key ••••••••••••••••••••••••••••••



```
[example]
uri = https://in03-aff54072149ee9c.api.gcp-us-west1.zillizcloud
token = f6b11525b0b320a12901cd113f78e4f9c0fd0132e684ebf7dc38a28
```

6.3.3 API connection

Milvus provides endpoint for direct http requests for the cloud platform , this is done through the window



6.3.4 Local connection

when the milvus is run locally for example in docker then the connection to the server would be through the endpoint and instead of an api key we store the username and password.since we don't have a password and user yet we basicly connect using

```
connections.connect(
  alias="default",
  user='',
  password='',
  host='localhost',
  port='19530'
)
```

7 Milvus Database Manupulation

to create a database we have to already have a user and password or an api key

```
  from pymilvus import connections, db
conn = connections.connect(host="127.0.0.1", port=19530)
database = db.create_database("books")
```

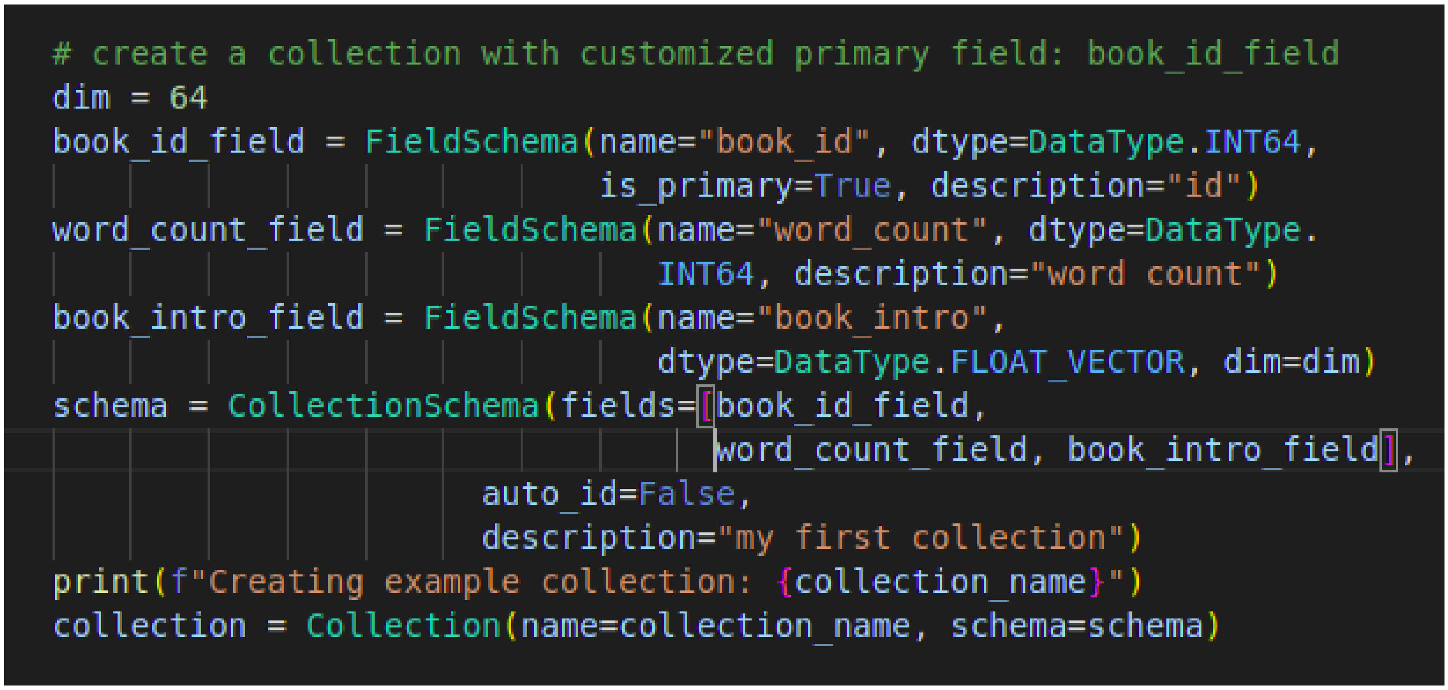
to use ,drop or list the databases we write

```
   db.use_database("book)
   db.list_database()
db.drop_database("book")
```

8 Milvus Collection Manupulation

8.1 collection Creation

to create a collection we need to first create it's schema , we begin by creating the fieldschemas and providing their type , and in the end we create the schema



```
# create a collection with customized primary field: book_id_field
dim = 64
book_id_field = FieldSchema(name="book_id", dtype=DataType.INT64,
                            is_primary=True, description="id")
word_count_field = FieldSchema(name="word_count", dtype=DataType.
                               INT64, description="word count")
book_intro_field = FieldSchema(name="book_intro",
                               dtype=DataType.FLOAT_VECTOR, dim=dim)
schema = CollectionSchema(fields=[book_id_field,
                                  word_count_field, book_intro_field],
                          auto_id=False,
                          description="my first collection")
print(f"Creating example collection: {collection_name}")
collection = Collection(name=collection_name, schema=schema)
```

8.1.1 renaming,dropping,searching

using the utility tool of pymilvus we could easily do the rest of the operations regarding deletion,modification of name and searching

```
    utility.rename_collection("old_collection", "new_collection") # Output
utility.drop_collection("new_collection")
utility.has_collection("new_collection") # Output: False
collection.load(replica_number=2)
```

9 Milvus Partition Manupulation

We have seen that a partition is a useful method to fasten the search process when a collection starts getting overwhelmingly massive .creating a partition only requires to inject it's name. Note that the maximum number of a partitions in a collection is 4096 .now to create,drop or load a partition we do

```
    from pymilvus import Collection
collection = Collection("book")      # Get an existing collection.
collection.create_partition("novel")
collection.load(["novel"], replica_number=2)
```

the load instruction loads the chosen column and it's values from the disk to memory,it could be replicated with the replica parameter, the amount of data to load should not exceed 90% of the available memory.in addition to this a partition can have a key, this is added with the constraint "is$_{p}artition_{k}ey$"

10 Milvus data manipulation

10.1 Data types

10.1.1 Primary Key types

- INT64: numpy.int64
- VARCHAR: VARCHAR

10.1.2 Scalar Valued types

- BOOL: Boolean (true or false)
- INT8: numpy.int8
- INT16: numpy.int16
- INT32: numpy.int32
- INT64: numpy.int64
- FLOAT: numpy.float32
- DOUBLE: numpy.double
- VARCHAR: VARCHAR

10.1.3 Vector field supports:

- BINARY_VECTOR: Binary vector
- FLOAT_VECTOR: Float vector

Note that the dimension for a vector is 32,768 exceeding this number returns an error.

10.1.4 Composite data types(Recently added)

- DYNAMIC: JSON

10.2 Data Manipululation

```
    collection.insert(data)#insert operation
    collection.delete(data)#delete operation
```

Along side these operations we can insert numpy or json files for better AI integration using utility bulk$_insertion$

```
    task_id = utility.do_bulk_insert(
    collection_name="book",
    partition_name="2022",
    files=["test.json","test2.json"]
)
```

10.3 Data compaction

this operation includes data dimensionality or volume reduction ,

```
    collection.compact()
```

11 Index manipulation

11.1 Vector Indexes

having seen how indexes differ from regular ones and how they include new algorithms we show how they get created

```
index_params = {
"metric_type":"L2",
"index_type":"IVF\_FLAT",
"params":{"nlist":1024}
}
utility.index_building_progress("book")
```

this a sample code that builds a index of type IVF_FLAT , a variation of the inverted file that we already saw, it uses the L2 metric which is the inner product to measure distance and it has parameters, the number of elements used in the index

11.2 metric_type

is Type of metrics used to measure the similarity of vectors ,For floating point vectors L2 (Euclidean distance) and IP (Inner product) but For binary vectors we have :JACCARD (Jaccard distance),TANIMOTO (Tanimoto distance),HAMMING (Hamming distance),SUPERSTRUCTURE (Superstructure),SUBSTRUCTURE (Substructure)

11.3 index_type

For floating point vectors:

- *FLAT (FLAT)*
- *IVF_FLAT (IVF_FLAT)*
- *IVF_SQ8 (IVF_SQ8)*
- *IVF_PQ (IVF_PQ)*
- *HNSW (HNSW)*

For binary_vectors we have BIN_FLAT (BIN_FLAT) and BIN_IVF_FLAT (BIN_IVF_FLAT)

11.4 Scalar Indexes

scalar indexes are of the same types as the regulare ones, the type of index is chosen dynamicly by the DB, the field name specifies what column to index on

```
collection.create_index(
  field_name="book_name",
  index_name="scalar_index",
)
collection.drop(index_name)
```

12 Querying and Searching

```
search_params = {"metric_type": "L2", "params": {"nprobe": 10}, "offset": 5}
results = collection.search(
data=[[0.1, 0.2]],
anns_field="book_intro",
param=search_params,
limit=10,
expr=None,
# set the names of the fields you want to retrieve from the search result.
output_fields=['title'],
consistency_level="Strong"
)
```

here we have a search parameters specifications , we have already seen the metric type and params with indexes, the offset here means that the result should be collected from the fifth position meaning that , the first 4 will be ignored this is useful when searching for similaire but not almost exact copies of the vecotr like in image similarity. for the following is a query search

- *collection: Refers to the collection in Milvus where the search is performed.*
- *data=[[0.1, 0.2]]: Specifies the query vector(s) for the search operation. In this example, a single query vector [0.1, 0.2] is provided.*
- *anns_field="book_intro": Indicates the field in the collection that contains the vectors to be searched. In this case, the vectors are stored in the field "book_intro".*

- *param=search_params: Specifies additional search parameters. It refers to the search_params object that holds parameters like the distance metric, number of probes, and offset (as explained in the previous response).*
- *limit=10: Defines the maximum number of search results to be returned. In this case, the limit is set to 10, meaning the search operation will retrieve at most 10 results. expr=None: Allows specifying an expression for more complex queries, but in this example, it is not used, so it is set to None.if the case where we have a hybrid search , the condition would be added here .*
- *output_fields=['title']: Specifies the names of the fields that should be retrieved from the search results. In this case, only the "title" field will be returned for each matching result.*
- *consistency_level="Strong": Determines the consistency level of the search operation. Setting it to "Strong" ensures that the search results are up-to-date and consistent with the latest changes made to the collection.*

13 Security

13.1 User Security

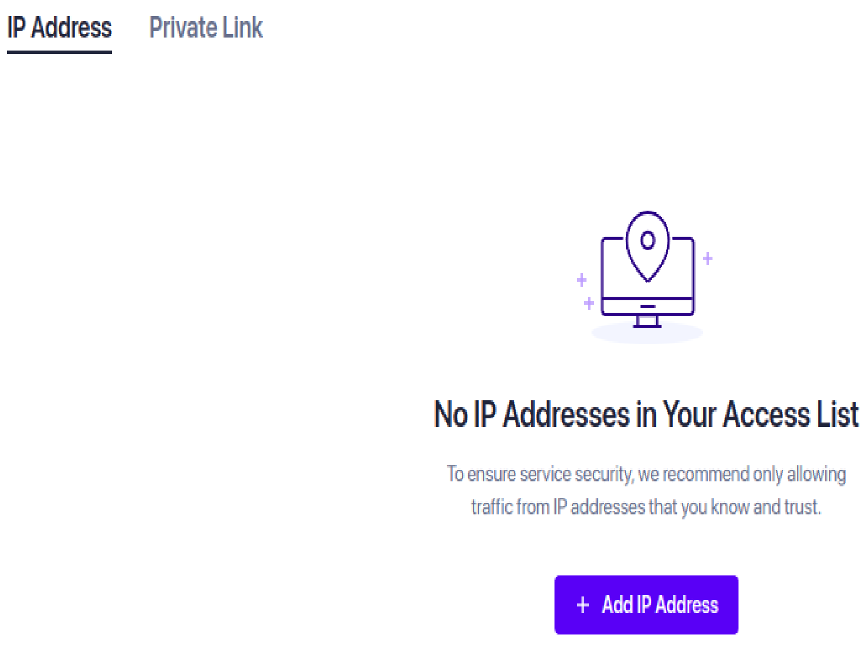
- *Milvus supports RBAC connection with the Grant / Revoke / List Grant operations ,in python it would be done through*

```
from pymilvus import connections, Role

_HOST = '127.0.0.1'
_PORT = '19530'
_ROOT = "root"
_ROOT_PASSWORD = "Milvus"
_ROLE_NAME = "test_role"
_PRIVILEGE_INSERT = "Insert"

def connect_to_milvus(db_name="default"):
    print(f"connect to milvus\n")
    connections.connect(host=_HOST,
    port=_PORT, db_name=db_name)
collection.drop(index_name)
```

- *Ip Permissions and Private links :Milvus allows for Ip permission list , to access collections.this can be done through the cloud platform,*


IP Address Private Link

No IP Addresses in Your Access List

To ensure service security, we recommend only allowing traffic from IP addresses that you know and trust.

Figure 30: Ip permission setting in cloud

13.2 database security

- *TimeTravel ,is a feature that allows you to access historical data at any point within a specified time period, making it possible to query, restore, and back up data in the past. With Time Travel, you can Search or query data that has been deleted,Restore data that has been deleted or updated. ,Back up data before a specific point of time. Unlike traditional databases that use snapshots or retain data to support the Time Travel feature, the Milvus vector database maintains a timeline for all data insert or delete operations and adopts a timestamp mechanism. This means you can specify the timestamp in a search or query to retrieve data at a specific point of time in the past to significantly reduce maintenance costs.*
- *Timestamp ,In the Milvus vector database, each entity has its own timestamp attribute. All data manipulation language (DML) operations including data insertion and deletion, mark entities with a timestamp. For instance, if you inserted several entities all at one go, this batch of data will be marked with timestamps and share the same timestamp value.*
- *DML operations ,When the proxy receives a data insert or delete request,*

it also gets a timestamp from the root coord. Then, the proxy adds the timestamp as an additional field to the inserted or deleted data. Timestamp is a data field just like primary key (pk). The timestamp field is stored together with other data fields of a collection. When you load a collection to memory, all data in the collection, including their corresponding timestamps, are loaded into memory.

- *When data is ingested into Milvus, it is initially stored in memory for faster access and efficient operations. However, for durability and persistence, the data needs to be periodically flushed to disk.*

  The flush operation in Milvus ensures that the data stored in memory, such as indexed vectors and associated metadata, is persisted to disk storage. This ensures that the data remains intact and available even in the event of system failures or restarts.

Conclusion

with the emergence of AI,vector databases have become essential tools for managing and querying high-dimensional vector data efficiently. They address the challenges associated with working with large-scale vector datasets and enable various applications that require similarity search or nearest neighbor operations.

Vector databases, such as Milvus, have emerged as powerful tools for managing and searching high-dimensional vector data efficiently. They are specifically designed to handle the challenges associated with working with large-scale vector datasets and enabling similarity search operations. Milvus, in particular, provides a feature-rich vector database solution with several notable capabilities

Ressources

- *https://www.cs.purdue.edu/homes/csjgwang/pubs/SIGMOD21_Milvus.pdf*

- *https://www.pinecone.io/learn/hnsw/*

References

[1] Timothy King. 2019. 80 Percent of Your Data Will Be Unstructured in Five Years. https://solutionsreview.com/data-management/80-percent-of-your-data-will-be-unstructured-in-five-years/

[2] Roie-Schwaber-Cohen. Vector databases have the capabilities of a traditional database that are absent in standalone vector indexes and the specialization of dealing with vector embeddings, which traditional scalar-based databases lack. https://www.pinecone.io/learn/vector-database/

[3] Figure 2 https://arxiv.org/pdf/1411.2738.pdf.

[4] Cour ANAD SIL .

[5] Inverted index is a data structure used in information retrieval systems to efficiently retrieve documents or web pages containing a specific term or set of terms. In an inverted index, the index is organized by terms (words), and each term points to a list of documents or web pages that contain that term https://www.geeksforgeeks.org/inverted-index/

[6] https://towardsdatascience.com/product-quantization-for-similarity-search-2f1f67c5fddd.

[7] E. Bernhardsson, ANN Benchmarks (2021), GitHub

[8] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In International Conference on Data Engineering (ICDE). 2–11.

[9] https://milvus.io/docs/architecture_overview.md