A Deep Dive into Qdrant, the Rust-Based Vector Database

BEGINNER DATABASE GUIDE NLP PYTHON PYTHON RECOMMENDATION VECTOR DATABASE

Vector Databases have become the go-to place for storing and indexing the representations of unstructured and structured data. These representations are the vector embeddings generated by the Embedding Models. The vector stores have become an integral part of developing apps with Deep Learning Models, especially the Large Language Models. In the ever-evolving landscape of Vector Stores, Qdrant is one such Vector Database that has been introduced recently and is feature-packed. Let's dive in and learn more about it.

Learning Objectives

- Familiarizing with the Qdrant terminologies to better understand it
- Diving into Qdrant Cloud and creating Clusters
- Learning to create embeddings of our documents and store them in Qdrant Collections
- Exploring how the querying works in Qdrant
- Tinkering with the Filtering in Qdrant to check how it works

This article was published as a part of the Data Science Blogathon.

Table of contents

What are Embeddings?

Vector Embeddings are a means of expressing data in numerical form—that is, as numbers in an n-dimensional space, or as a numerical vector—regardless of the type of data—text, photos, audio, videos, etc. Embeddings enable us to group together related data in this way. Certain inputs can be transformed

into vectors using certain models. A well-known embedding model created by Google that translates words into vectors (vectors are points with n dimensions) is called Word2Vec. Each of the Large Language Models has an embedding model that generates an embedding for the LLM.

What are Embeddings Used for?

One advantage of translating words to vectors is that they allow for comparison. When given two words as numerical inputs, or vector embeddings, a computer can compare them even though it cannot compare them directly. It is possible to group words with comparable embeddings together. Because they are related to one another, the terms King, Queen, Prince, and Princess will appear in a cluster.

In this sense, embeddings help us locate words that are related to a given term. This can be used in sentences, where we enter a sentence, and the supplied data returns related sentences. This serves as the foundation for numerous use cases, including chatbots, sentence similarity, anomaly detection, and semantic search. The Chatbots that we develop to answer questions based on a PDF or document that we provide make use of this embedding notion. This method is used by all Generative Large Language Models to obtain content that is similarly connected to the queries that are supplied to them.

What are Vector Databases?

As discussed, embeddings are representations of any kind of data usually, the unstructured ones in the numerical format in an n-dimensional space. Now where do we store them? Traditional RDMS (Relational Database Management Systems) cannot be used to store these vector embeddings. This is where the Vector Store / Vector Dabases come into play. Vector Databases are designed to store and retrieve vector embeddings in an efficient manner. There are many Vector Stores out there, which differ by the embedding models they support and the kind of search algorithm they use to get similar vectors.

What is Qdrant?

Qdrant is the new Vector Similarity Search Engine and a Vector DB, providing a production-ready service built in Rust, the language known for its safety. Qdrant comes with a user-friendly API designed to store, search, and manage high-dimensional Points (Points are nothing but Vector Embeddings) enriched with metadata called payloads. These payloads become valuable pieces of information, improving search precision and providing insightful data for users. If you are familiar with other Vector Databases like Chroma, Payload is similar to the metadata, it contains info about the vectors.

Being written in Rust makes Qdrant a fast and reliable Vectore Store even under heavy loads. What differentiates Qdrant from the other databases is the number of client APIs it provides. At present Qdrant supports Python, TypeSciprt/JavaScript, Rust, and Go. It comes with. Qdrant uses HSNW (Hierarchical Navigable Small World Graph) for Vector indexing and comes with many distance metrics like Cosine, Dot, and Euclidean. It comes with a recommendation API out of the box.

Know the Qdrant Terminology

To get a smooth start with Qdrant, it's a good practice to get familiar with the terminology / the main Components used in the Qdrant Vector Database.

Collections

Collections are named sets of Points, where each Point contains a vector and an optional ID and payload. Vectors in the same Collection must share the same dimensionality and be Evaluated with a single chosen Metric.

Distance Metrics

Essential for measuring how close are the vectors to each other, distance metrics are selected during the creation of a Collection. Qdrant provides the following Distance Metrics: Dot, Cosine, and Euclidean.

Points

The fundamental entity within Qdrant, points consists of a vector embedding, an optional ID, and an associated payload, where
id: A unique identifier for each vector embedding
vector: A high-dimensional representation of data, which can be either structured or unstructured formats like images, text, documents, PDFs, videos, audio, etc.
payload: An optional JSON object containing data associated with a vector. This can be considered similar to metadata and we can work with this to filter the search process

Storage

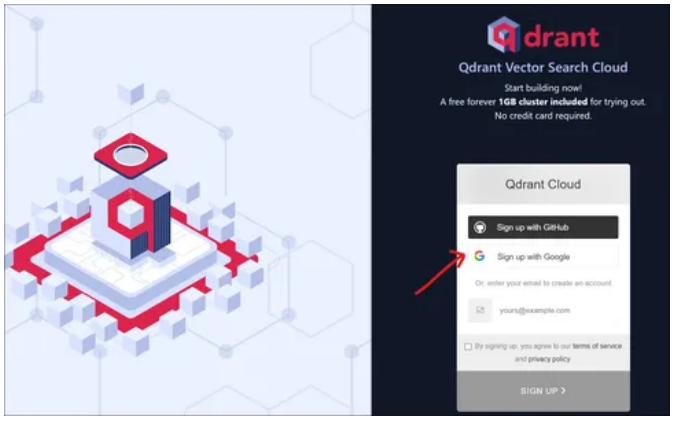
Qdrant provides two storage options:

- **In-Memory Storage**: Stores all vectors in RAM, optimizing speed by minimizing disk access to persistence tasks.
- **Memmap Storage**: Creates a virtual address space linked to a file on disk, balancing speed and persistence requirements.

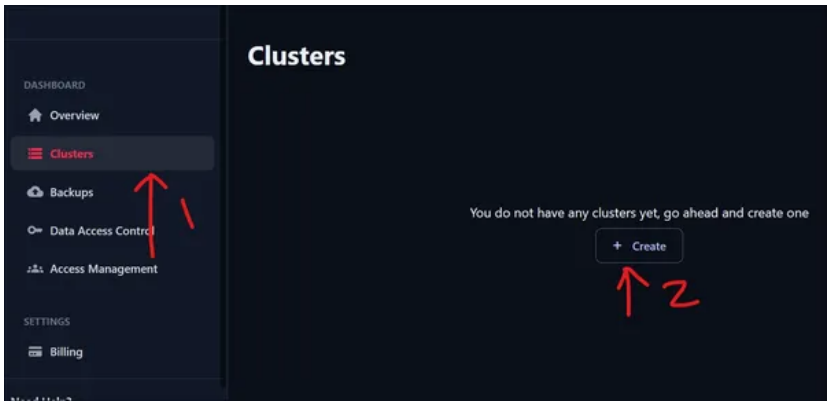
These are the main concepts that we need to be aware of so we can get quickly started with Qdrant

Qdrant Cloud – Creating our First Cluster

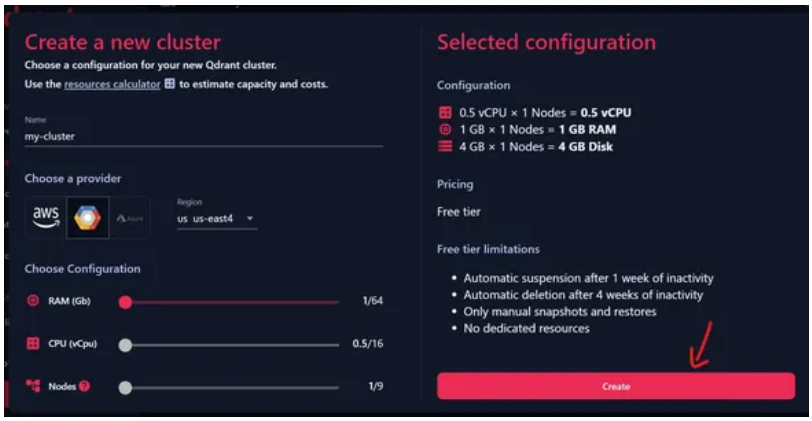
Qdrant provides a scalable cloud service for storing and managing vectors. It even provides a free forever 1GB Cluster with no credit card information. In this section, we will go through the process of creating an Account with Qdrant Cloud and creating our first Cluster.



Going to the Qdrant website, we will a landing page like the above. We can sign up to the Qdrant either with a Google Account or with a GitHub Account.



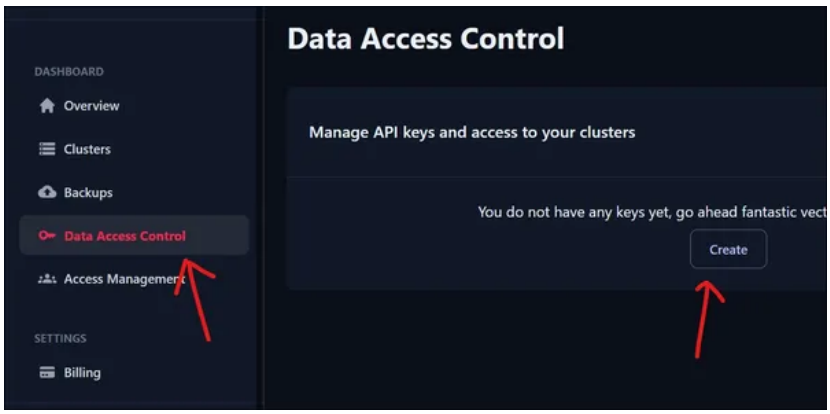
After logging in, we will be presented with the UI shown above. To create a Cluster, go to the left pane and click on the Clusters option under the Dashboard. As we have just signed in, we have zero clusters. Click on the Create Cluster to create a new Cluster.



Now, we can provide a name for our Cluster. Make sure to have all the Configurations set to the starting position, because this gives us a free Cluster. We can choose one of the providers shown above and choose one of the regions associated with it.

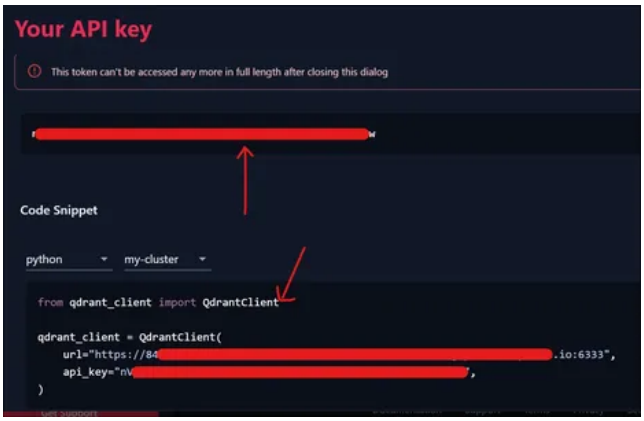
Check the Current Configuration

We can see on the left the current Configuration, i.e. 0.5 vCPU, 1GB RAM, and 4 GB Disk Storage. Click on Create to create our Cluster.



To access our newly created Cluster we need an API Key. To create a new API key, head to Data Access Control under the Dashboard. Click on the Create Button to create a new API key.

As shown above, we will be presented with a drop-down menu where we select what Cluster we need to create the API for. As we have only one Cluster, we select that and click on the OK button.



Then you will presented with the API Token shown above. Also if we see the below part of the image, we are even provided with the code snippet to connect our Cluster, which we will be using in the next section.

Qdrant – Hands On

In this section, we will be working with the Qdrant Vector Database. First, we will start off by importing the necessary libraries.

```
!pip install sentence-transformers !pip install qdrant_client
```

The first line installs the sentence-transformer Python library. The sentence transformer library is used for generating sentence, text, and image embeddings. We can use this library to import different embedding models to create embeddings. The next statement installs the qdrant client for Python. Let's start off by creating our client.

```
from qdrant_client import QdrantClient client = QdrantClient( url="YOUR CLUSTER URL", api_key="YOUR API KEY",
)
```

QdrantClient

In the above, we instantiate the **client** by importing the **QdrantClient** class and giving the Cluster URL and the **API Key** that we just created a while ago. Next, we will bring in our embedding model.

```
# bringing in our embedding model from sentence_transformers import SentenceTransformer model = SentenceTransformer('sentence-transformers/all-mpnet-base-v2')
```

In the above code, we have used the **SentenceTransformer** class and instantiated a model. The embedding model we have taken is the **all-mpnet-base-v2**. This is a widely popular general-purpose vector embedding model. This model will take in text and output a **768-dimensional** vector. Let's define our data.

```
# data documents = [ """Elephants, the largest land mammals, exhibit remarkable intelligence and \ social bonds, relying on their powerful trunks for communication and various\ tasks like lifting objects \and gathering food.""", """ Penguins, flightless birds adapted to life in the water, showcase strong \ social structures and exceptional parenting skills. Their sleek bodies \ enable efficient swimming, and they endure \ harsh Antarctic conditions in tightly-knit colonies. """, """Cars, versatile modes of transportation, come in various shapes and \ sizes, from compact city cars to powerful sports vehicles, offering a \ range of features for different preferences and \ needs.""", """Motorbikes, nimble two-wheeled machines, provide a thrilling and \ liberating riding experience, appealing to enthusiasts who appreciate \ speed, agility, and the open road.""", """Tigers, majestic big cats, are solitary hunters with distinctive \ striped fur. Their powerful build and stealthy movements make them \ formidable predators, but their populations are threatened \ due to habitat loss and poaching.""" ]
```

In the above, we have a variable called documents and it contains a list of 5 strings(let's take each of them like a single document). Each string of data is related to a particular topic. Some data is related to elements and some data is related to automobiles. Let's create embeddings for the data.

```
# embedding the data embeddings = model.encode(documents) print(embeddings.shape)
```

We use the **encode()** function of the model object to encode our data. To encode, we directly pass the documents list to the **encode()** function and store the resultant vector embeddings in the embeddings variable. We are even printing the shape of the embeddings, which here will print (5, 768). This is because we have 5 Data Points, that is 5 documents and for each document, a vector embedding of 768 Dimensions is created.

Create your Collection

Now we will create our Collection.

```
from qdrant_client.http.models import VectorParams, Distance client.create_collection( collection_name = "my-collection", vectors_config = VectorParams(size=768,distance=Distance.COSINE) )
```

- To create a Collection, we work with the create_collection() function of the client object, and to the "**Collection_name**", we pass in our Collection name i.e. "my-collection"
- **VectorParams**: This class from **qdrant** is for vector Configuration, like what is the vector embedding size, what is the distance metric, and such
- **Distance**: This class from **qdrant** is for defining what distance metric to use for querying vectors
- Now to the **vector_config** variable we pass our Configuration, that is the size of vector embeddings i.e. **786**, and the distance metric we want to use, which is **COSINE**

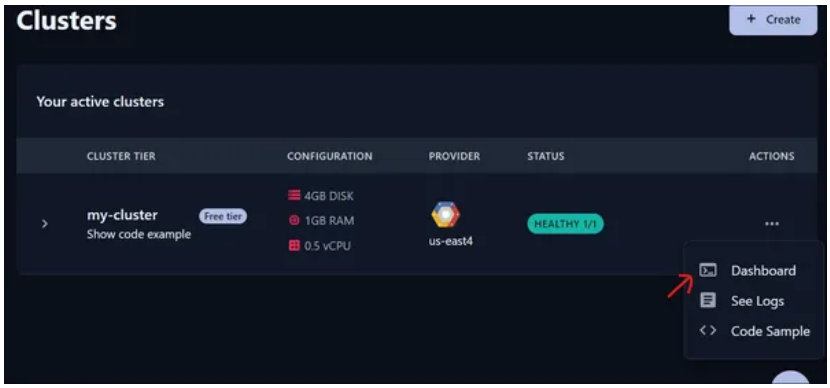
Add Vector Embeddings

We have now successfully created our Collection. Now we will be adding our vector embeddings to this Collection.

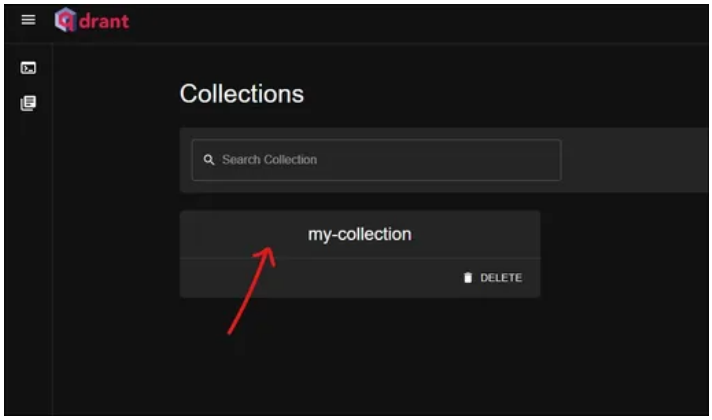
```
from qdrant_client.http.models import Batch client.upsert ( collection_name = "my-collection", points = Batch( ids = [1,2,3,4,5], payloads= [ {"category":"animals"}, {"category":"animals"}, {"category":"automobiles"}, {"category":"automobiles"}, {"category":"animals"} ], vectors = embeddings.tolist() ) )
```

- To add data to qdrant we call the **upsert()** method and pass in the Collection name and Points. As we have learned above, a **Point** consists of vectors, an optional index, and payloads. The **Batch** Class from qdrant lets us add data in batches instead of adding them one by one.
- **ids**: We are giving our documents an ID. At present, we are giving a range of values from 1 to 5 because we have 5 documents on our list.
- **payloads**: As we have seen before, the **payload** contains information about the vectors, like **metadata**. We provide it in key-value pairs. For each document we have provided a **payload** here, we are assigning the category information for each document.
- **vectors**: These are the vector embeddings of the documents. We are converting it into a list from a numpy array and feeding it.

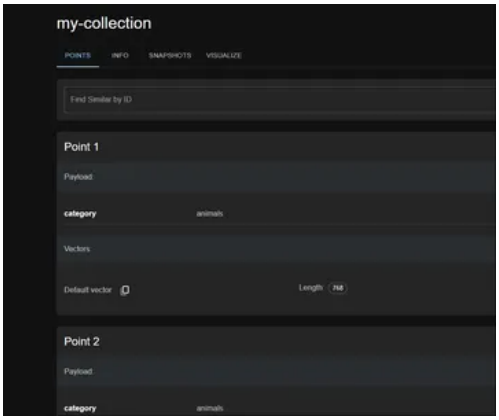
So, after running this code, the vector embeddings get added to the Collection. To check if they have been added, we can visit the cloud dashboard that the Qdrant Cloud Provides. For that, we do the following:



We click on the dashboard and then a new page gets opened.



This is the qdrant dashboard. Check our “**my-collection**” collection here. Click on it to view what’s in it.



In the Qdrant cloud, we observe that our Points (vectors + payload + IDs) are indeed adding to our Collection within our Cluster. In the follow-up section, we will learn how to query these vectors.

Querying the Qdrant Vector Database

In this section, we will go through querying the vector database and even try adding in some filters to get a filtered result. To query our qdrant vector database, we need to first create a query vector, which we can do by:

```
query = model.encode(['Animals live in the forest'])
```

Query Embedding

The following will create our **query** embedding. Then using this, we will query our vector store to get the most relevant vector embeddings.

```
client.search( collection_name = "my-collection", query_vector = query[0], limit = 4 )
```

Search() Query

To query we use the **search()** method of the client object and pass it the following:

- **Collection_name**: The name of our Collection
- **query_vector**: The query vector on which we want to search the vector store
- **limit**: How many search outputs do we want the **search()** function to limit too

Running the code will produce the following output:

```
[ScoredPoint(id=1, version=0, score=0.3731497, payload={'category': 'animals'},
vector=None),
 ScoredPoint(id=5, version=0, score=0.3664084, payload={'category': 'animals'},
vector=None),
 ScoredPoint(id=2, version=0, score=0.14141288, payload={'category': 'animals'},
vector=None),
 ScoredPoint(id=4, version=0, score=0.08643663, payload={'category':
'automobiles'}, vector=None)]
```

We see that for our query, the top retrieved documents are of the category animals. Thus we can say that the search is effective. Now let's try it with some other query so that it gives us different results. The vectors are not displayed/fetched by default, hence it is set to None.

```
query = model.encode(['Vehicles are polluting the world']) client.search( collection_name = "my-collection", query_vector = query[0], limit = 3 )
```

```
[ScoredPoint(id=3, version=0, score=0.41747698, payload={'category':
'automobiles'}, vector=None),
 ScoredPoint(id=4, version=0, score=0.26304838, payload={'category':
'automobiles'}, vector=None),
 ScoredPoint(id=5, version=0, score=0.17894708, payload={'category': 'animals'},
vector=None)]
```

Query Related to Vehicles

This time we have given a **query** related to **vehicles** the vector database was able to successfully fetch the documents of the relevant Category (automobile) at the top. Now what if we want to do some filtering? We can do this by:

```
from qdrant_client.http.models import Filter, FieldCondition, MatchValue query = model.encode(['Animals live in the forest']) custom_filter = Filter( must = [ FieldCondition( key = "category", match = MatchValue( value="animals" ), ) ] )
```

- Firstly, we are creating our query embedding/vector
- Here we import the **Filter**, **FieldCondition**, and **MatchValue** classes from the qdrant library.
- **Filter**: Use this class to create a Filter object
- **FiledCondition**: This class is for creating the filtering, like on what we want to filter our search
- **MatchValue**: This class is for telling on what value for a given key we want the qdrant vector db to filter

So in the above code, we are basically saying that we are creating a **Filter** that checks the **FieldCondition** that the key "**category**" in the **Payload** matches(**MatchValue**) the value "**animals**". This looks a bit big for a simple filter, but this approach will make our code more structured when we are dealing with a **Payload** containing a lot of information and we want to filter on multiple keys. Now let's use the filter in our search.

```
client.search( collection_name = "my-collection", query_vector = query[0], query_filter = custom_filter, limit = 4 )
```

Query_filter

Here, this time, we are even giving in a **query_filter** variable which takes in the **Custom Filter** that we have defined. Note that we have kept a limit of 4 to retrieve the top 4 matching documents. The query is related to animals. Running the code will result in the following output:

```
[ScoredPoint(id=1, version=0, score=0.3731497, payload={'category': 'animals'},
vector=None),
 ScoredPoint(id=5, version=0, score=0.3664084, payload={'category': 'animals'},
vector=None),
 ScoredPoint(id=2, version=0, score=0.14141288, payload={'category': 'animals'},
vector=None)]
```

In the output we have received only the top 3 nearest Docs even though we have 5 documents. This is because we have set our filter to choose only the animal categories and there are only 3 documents with that category. This way we can store the vector embeddings in the qdrant cloud perform vector search on these embedding vectors retrieve the nearest ones and even apply filters to filter the output:

Applications

The following applications can Qdrant Vector Database:

- **Recommendation Systems:** Qdrant can power recommendation engines by efficiently matching high-dimensional vectors, making it suitable for personalized content recommendations in platforms like streaming services, e-commerce, or social media.
- **Image and Multimedia Retrieval:** Leveraging Qdrant's capability to handle vectors representing images and multimedia content, applications can implement effective search and retrieval functionalities for image databases or multimedia archives.
- **Natural Language Processing (NLP) Applications:** Qdrant's support for vector embeddings makes it valuable for NLP tasks, like semantic search, document similarity matching, and content recommendation in applications dealing with large amounts of textual datasets.
- **Anomaly Detection:** Qdrant's high-dimensional vector search can be worked in anomaly detection systems. By comparing vectors representing normal behavior against incoming data, anomalies can be identified in fields, like network security or industrial monitoring.
- **Product Search and Matching:** In e-commerce platforms, Qdrant can improve product search capabilities by matching vectors representing product features, facilitating accurate and efficient product recommendations based on user preferences.
- **Content-Based Filtering in Social Networks:** Qdrant's vector search can be applied in social networks for content-based filtering. Users can get relevant content based on the similarity of vector representations, improving user engagement.

Conclusion

As the demand for efficient representation of data grows, Qdrant stands out being an Open Source feature-packed vector similarity search engine, written in the robust and safety-centric language, Rust. Qdrant includes all the popular Distance Metrics and provides a robust way to Filter our vector search. With its rich features, cloud-native architecture, and robust terminology, Qdrant opens doors to a new era in vector similarity search technology. Even though it is new to the field it provides client libraries for many programming languages and provides a cloud that scales efficiently with size.

Key Takeaways

Some of the key takeaways include:

- Crafted in Rust, Qdrant ensures both speed and reliability, even under heavy loads, making it the best choice for high-performance vector stores.
- What sets Qdrant apart is its support for client APIs, catering to developers in Python, TypeScript/JavaScript, Rust, and Go.
- Qdrant leverages the HSNW algorithm and gives different distance metrics, including Dot, Cosine, and Euclidean, empowering developers to choose the metric that aligns with their specific use cases.

- Qdrant seamlessly transitions to the cloud with a scalable cloud service, providing a free-tier option for exploration. Its cloud-native architecture ensures optimal performance, irrespective of data volume.

Frequently Asked Questions

Q1: What is Qdrant, and how does it differ from other vector databases?

A: Qdrant is a vector similarity search engine and vector store written in Rust. It stands out for its speed, reliability, and rich client support, providing APIs for Python, TypeScript/JavaScript, Rust, and Go.

Q2: How does Qdrant handle distance metrics in vector searches?

A: Qdrant uses the HSNW algorithm and gives different distance metrics like Dot, Cosine, and Euclidean. Developers can choose the metric that aligns with their specific use cases when creating collections.

Q3: What are the Key Components and terminologies in the Qdrant Vector Database?

A: Important Components include Collections, Distance Metrics, Points (vectors, optional IDs, and payloads), and Storage options (In-Memory and Memmap).

Q4: Can Qdrant be integrated with cloud services, and what advantages does it provide?

A: Yes, Qdrant seamlessly integrates with cloud services, providing a scalable cloud solution. The cloud-native architecture ensures optimal performance, making it changes to varying data volumes and computational needs.

Q5: How does Qdrant support filtering in vector searches?

A: Qdrant allows filtering through payload information. Users can define filters using the Qdrant library, by giving conditions based on payload keys and values to refine search results.

Article Url - https://www.analyticsvidhya.com/blog/2023/11/a-deep-dive-into-qdrant-the-rust-based-vector-database/

Ajay

I work as a Developer in the field of Data Science. I constantly spend time learning new things be it related to AI, DataSceine, and CyberSecurity. Deep learning and machine learning are two topics that I find particularly fascinating, and Python is my preferred language for programming. Cyber Security is another field that I'm touching upon recently. I have experience with large-scale data analysis, and I have a solid grasp of a variety of deep learning and machine learning approaches, including neural networks, regression models, and natural language processing. I'm eager to take on new challenges and make a meaningful contribution to the industry, so I'm constantly seeking for ways to enlarge and deepen my knowledge and skills in the subject.