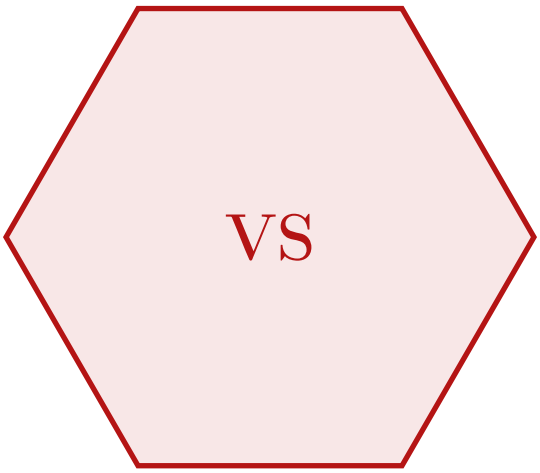


Implementing
Basic Vector Search

A Practical Guide to Building Your First Qdrant Collection

In this guide, you will build:

1. A Connection to Qdrant Cloud
2. A Vector Collection
3. A Similarity Search Engine

Created by Raj Ghosh

Contents

Chapter 1

Environment Setup

1.1 Installing and Importing

To speak to the Qdrant database, we need the official Python translator (the Client).

Step 1: Install Client

Run this command in your terminal or notebook to install the library.

```
!pip install qdrant-client
```

Step 2: Import Libraries

We need to bring in two key components:

- `QdrantClient`: The "phone" we use to call the database.
- `models`: The "dictionary" containing definitions for Points, Vectors, and Distances.

```
from qdrant_client import QdrantClient, models
import os
```

1.2 Connecting to the Cloud

Step 3: Connect

Using the API key and URL you retrieved in Day 0, establish the link.

```
# Load secrets from environment (Best Practice)
qdrant_url = os.getenv("QDRANT_URL")
qdrant_key = os.getenv("QDRANT_API_KEY")

client = QdrantClient(url=qdrant_url, api_key=qdrant_key)

# PRO TIP: For quick testing without internet, use Memory mode:
# client = QdrantClient(":memory:")
```

Chapter 2

The Collection (Your Database Table)

2.1 Understanding Collections

A **Collection** in Qdrant is equivalent to a Table in SQL. It holds your vectors. When you make one, you must define the "Rules of the Game" (Vector Size and Distance Metric).

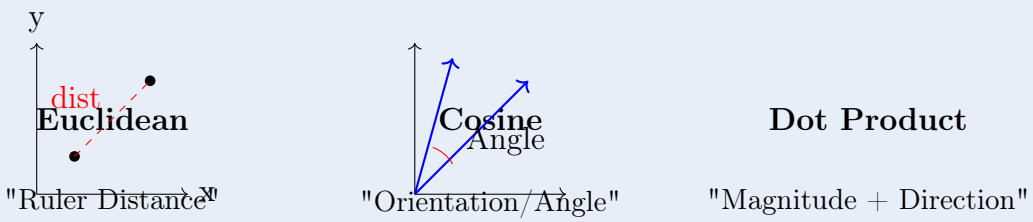
Step 4: Create Collection

```python
collection_name = "my_first_collection"

client.create_collection(
    collection_name=collection_name,
    vectors_config=models.VectorParams(
        size=4,                              # We are using 4D vectors
        distance=models.Distance.COSINE      # How to measure similarity
    )
)
```

Deep Dive: Distance Metrics Explained

How do we know if Vector A is "close" to Vector B?



Step 5: Verify Creation

Always double-check that the operation succeeded.

```python
collections = client.get_collections()
print(collections)
# Output should contain "my_first_collection"
```

Chapter 3

Inserting Points

3.1 Anatomy of a Point

In Qdrant, a data entry is called a **Point**.

- **ID:** Unique Number or UUID.
- **Vector:** The list of numbers (coordinates).
- **Payload:** JSON data (metadata like names, dates, categories).

Step 6: Upsert (Update/Insert)

We use ‘upsert‘ to put data in. If the ID exists, it updates it. If not, it creates it.

```
points = [
    models.PointStruct(
        id=1,
        vector=[0.1, 0.2, 0.3, 0.4],
        payload={"category": "example"}
    ),
    models.PointStruct(
        id=2,
        vector=[0.2, 0.3, 0.4, 0.5],
        payload={"category": "demo"}
    )
]

client.upsert(
    collection_name=collection_name,
    points=points
)
```

Step 7: Collection Info

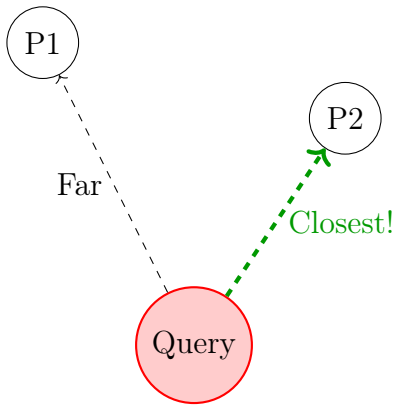
Verify the data is actually there.

```
info = client.get_collection(collection_name)
print(f"Points Count: {info.points_count}")
# Should print: Points Count: 2
```

Chapter 4

Running Similarity Search

This is the moment of truth. We provide a **Query Vector**, and Qdrant finds the mathematically closest match.



Semantic Search
Finds the vector with the highest similarity score.

Step 8: Query Points

```
query_vector = [0.08, 0.14, 0.33, 0.28]

search_results = client.query_points(
    collection_name=collection_name,
    query=query_vector,
    limit=1   # How many results do you want?
)

print(search_results)
```

Understanding the Output

You will receive a `ScoredPoint` object:

- **ID:** The ID of the matching point (e.g., 1).
- **Score:** A number between 0 and 1 (for Cosine). Closer to 1.0 means more similar.
- **Payload:** The metadata we stored ("category": "example").

Tutorial Completed

You have successfully implemented your first Vector Search engine.

Created by Raj Ghosh