Qdrant Tutorial: Key Concepts and Python Examples

# 1. Introduction to Qdrant

Qdrant is a **vector database** designed for storing high-dimensional vectors (like embeddings from text, images, or audio) and performing **similarity search** efficiently.

**Key Terms:**

- **Collection:** Like a table in a database, holds vectors and payloads.

- **Point:** A single vector with metadata (payload).

- **Payload:** Metadata attached to a point (e.g., city name, category).

- **Vector:** Numeric representation (embedding) of your data.

# 2. Python Requirements

Install the following packages:

qdrant-client==1.3.2  
fastembed==0.0.6  
python-dotenv==1.0.0

Use pip install -r requirements.txt to install them.

# 3. Full Python Example with Explanation

# Import necessary modules  
from qdrant\_client import QdrantClient  
from qdrant\_client.models import (  
 VectorParams, Distance, PointStruct, Filter, FieldCondition, MatchValue, HnswConfig  
)  
from fastembed import TextEmbedding

**Explanation:**

- QdrantClient is used to connect to Qdrant (local or cluster).

- VectorParams and Distance define collection vector size and similarity metric.

- PointStruct represents a vector + payload.

- Filter, FieldCondition, MatchValue allow metadata-based filtering.

- HnswConfig is for indexing (fast ANN search).

- TextEmbedding creates vector embeddings for text.

# Connect to Qdrant (in-memory for testing)  
client = QdrantClient(":memory:")  
embedder = TextEmbedding()

**Explanation:**

- :memory: runs Qdrant in-memory (good for testing).

- embedder will convert text to vectors.

# Create collection with HNSW indexing  
client.create\_collection(  
 collection\_name="articles",  
 vectors\_config=VectorParams(size=384, distance=Distance.COSINE),  
 hnsw\_config=HnswConfig(m=16, ef\_construct=200, full\_scan\_threshold=10000)  
)

**Explanation:**

- size=384 corresponds to embedding dimensions.

- Distance.COSINE uses cosine similarity.

- HnswConfig optimizes approximate nearest neighbor search for speed.

# Insert documents with text and category  
docs = [  
 {"id": 1, "text": "Eiffel Tower is in Paris", "category": "travel"},  
 {"id": 2, "text": "Python is a programming language", "category": "tech"},  
 {"id": 3, "text": "Berlin is the capital of Germany", "category": "travel"},  
 {"id": 4, "text": "AI is transforming healthcare", "category": "tech"},  
]  
  
vectors = list(embedder.embed([d["text"] for d in docs]))  
  
points = [  
 PointStruct(id=doc["id"], vector=vec, payload=doc)  
 for doc, vec in zip(docs, vectors)  
]  
  
client.upsert(collection\_name="articles", points=points)

**Explanation:**

- embedder.embed converts text to numeric vectors.

- upsert inserts points into the collection. If a point ID exists, it updates.

- payload stores metadata like category.

# Similarity search for a query  
query = "capital of France"  
query\_vec = list(embedder.embed([query]))[0]  
  
results = client.query\_points(  
 collection\_name="articles",  
 query=query\_vec,  
 limit=3,  
 with\_payload=True  
).points  
  
for r in results:  
 print(r.payload["text"], "| Score:", round(r.score, 3))

**Explanation:**

- query\_points finds the nearest vectors to query\_vec.

- with\_payload=True includes metadata in results.

- r.score shows similarity (higher = more similar).

# Similarity search with filtering (category=travel)  
filter\_condition = Filter(  
 must=[FieldCondition(key="category", match=MatchValue(value="travel"))]  
)  
  
results\_filtered = client.query\_points(  
 collection\_name="articles",  
 query=query\_vec,  
 limit=3,  
 with\_payload=True,  
 query\_filter=filter\_condition  
).points  
  
for r in results\_filtered:  
 print(r.payload["text"], "| Score:", round(r.score, 3))

**Explanation:**

- Only returns points where category is travel.

- Combines **semantic search** with **structured filtering**.

# Update a point  
client.upsert(  
 collection\_name="articles",  
 points=[PointStruct(id=2, vector=vectors[1], payload={"text":"Python programming", "category":"tech"})]  
)  
  
# Delete a point  
client.delete(collection\_name="articles", points\_selector=[4])

**Explanation:**

- upsert with same ID updates the point.

- delete removes a point by ID.

# 4. Key Concepts Summarized

* **Collections:** Store vectors and metadata.
* **Points:** Vector + payload.
* **Payload:** Metadata for filtering.
* **Similarity Search:** Find closest vectors.
* **Filtering:** Narrow results using payload conditions.
* **Indexing:** HNSW, PQ for speed/memory optimization.
* **Production Scaling:** Use Qdrant Cluster with shards, replicas, and replication factor.

# 5. Production Scaling Tips

* **Cluster setup** for high availability & horizontal scaling.
* **Shard large collections** for load distribution.
* **Replication factor ≥2** for redundancy.
* **Backups**: schedule snapshots to cloud/local storage.
* **Monitoring:** Use Prometheus/Grafana for metrics.
* **Client SDK** usage stays the same; just point to cluster endpoint.