Qdrant Collections and Vectors: A Beginner's Guide

What is Qdrant?

Qdrant is a vector database that stores and searches through "vectors" - lists of numbers that represent data like text, images, or any information you want to search through semantically.

Think of a Qdrant collection like a filing cabinet, but instead of storing documents, it stores vectors that represent the meaning of your data.

Understanding Vectors

A vector is like a coordinate that describes something in multi-dimensional space.

Simple Example: Describing a person with numbers:

- Height: 5.8
- Weight: 150
- Age: 25

That's a 3-dimensional vector: [5.8, 150, 25]

In Qdrant: Vectors are usually much longer (100-1500+ numbers) and represent complex concepts like the "meaning" of a sentence or document.

Creating Collections - The Basics

Creating a collection is straightforward:

```
client.create_collection(
    collection_name="my_collection",
    vectors_config=VectorParams(size=384, distance=Distance.COSINE)
)
```

Vector Parameters Explained

When creating a collection, you must specify three key parameters:

1. Size - How Many Numbers in Each Vector?

Important: You don't choose the size arbitrarily - your embedding model determines it.

Common embedding model sizes:

- OpenAI `text-embedding-3-small`: 1536 dimensions
- OpenAI `text-embedding-3-large`: 3072 dimensions
- Sentence Transformers `all-MiniLM-L6-v2`: 384 dimensions
- Google's models: Often 768 dimensions

Rule: The collection size must exactly match your embedding model's output size.

```
# If using OpenAI embeddings
collection_config = VectorParams(
    size=1536,  # Must match OpenAI's output exactly
    distance=Distance.COSINE
)
```

Smaller vs Larger Vectors:

- **Smaller (384):** Faster searches, less storage, less detailed understanding
- **Larger (1536+):** More detailed understanding, slower searches, more storage

2. Distance - How to Measure Similarity

Three main options:

- **COSINE:** Best for text/semantic similarity (most common choice)
- **EUCLIDEAN:** Measures straight-line distance between points
- **DOT:** For when larger numbers indicate higher similarity

3. Vector Name (Optional)

You can store multiple types of vectors in one collection:

- "text_vector" and "image_vector" in the same collection
- Each can have different sizes and distance metrics

Understanding PointStruct

`PointStruct` is Qdrant's way of organizing data - like a database row format:

```
PointStruct(
    id=1,            # Unique identifier (like a primary key)
    vector=[...],    # The embedding numbers from your model
    payload={...}    # Your actual data and metadata
)
```

The Payload - Your Flexible Metadata

The payload can contain unlimited metadata about your data:

```
PointStruct(
    id=1,
    vector=embedding_vector,  # From your embedding model
    payload={
        "review": "Great pizza!",
        "rating": 5,
        "customer_name": "John Smith",
        "phone": "555-1234",
        "visit_date": "2024-06-15",
        "restaurant_name": "Tony's Pizza",
        "location": "Downtown",
        "price_range": "$$",
        "cuisine_type": "Italian"
    }
)
```

Complete Real-World Example

```python
import openai
from qdrant_client import QdrantClient
from qdrant_client.models import PointStruct, VectorParams, Distance

# 1. Create collection (size matches OpenAI's model)
client.create_collection(
    collection_name="restaurant_reviews",
    vectors_config=VectorParams(size=1536, distance=Distance.COSINE)
)

# 2. Convert text to vector using OpenAI
review_text = "Great pizza, friendly service!"
embedding = openai.Embedding.create(
    input=review_text,
    model="text-embedding-3-small"
)
vector = embedding['data'][0]['embedding']  # This will be 1536 numbers

# 3. Store in Qdrant
client.upsert(
    collection_name="restaurant_reviews",
    points=[
        PointStruct(
            id=1,
            vector=vector,  # The 1536 numbers from OpenAI
            payload={
                "review": review_text,
                "rating": 5,
                "customer": "Alice",
                "date": "2024-06-18",
                "restaurant": "Tony's Pizza",
                "location": "Downtown"
            }
        )
    ]
)
```

The Power of Hybrid Search

Qdrant gives you two types of search capabilities:

1. Vector Search Only (Semantic Similarity)

Find content with similar meaning:

```
# Find reviews similar to "great food"
results = client.search(
    collection_name="restaurant_reviews",
    query_vector=embed_text("great food"),
    limit=5
)
```

2. Hybrid Search (Vector + Metadata Filtering)

Combine semantic search with precise filtering:

```
# Find reviews similar to "great food"
# BUT only from Italian restaurants in downtown with 4+ stars
results = client.search(
    collection_name="restaurant_reviews",
    query_vector=embed_text("great food"),
    query_filter=Filter(
        must=[
            FieldCondition(key="cuisine_type", match=MatchValue(value="Italian")),
            FieldCondition(key="location", match=MatchValue(value="Downtown")),
            FieldCondition(key="rating", range=Range(gte=4))
        ]
    ),
    limit=5
)
```

Real-World Use Case Example

Building a restaurant recommendation system:

```python
# User asks: "I want great pasta near downtown with good service"
search_query = "great pasta good service"
user_location = "downtown"

results = client.search(
    collection_name="restaurant_reviews",
    query_vector=embed_text(search_query),  # Semantic similarity
    query_filter=Filter(
        must=[
            FieldCondition(key="location", match=MatchValue(value=user_location)),
            FieldCondition(key="rating", range=Range(gte=4))
        ]
    )
)

# Returns reviews that:
# 1. Are semantically similar to "great pasta good service" (vector search)
# 2. Are from downtown restaurants (metadata filter)
# 3. Have 4+ star ratings (metadata filter)
```

Why Hybrid Search is Powerful

Traditional SQL databases can't understand that "delicious" and "tasty" mean similar things.

Pure vector search can't easily filter by exact criteria like "only Italian restaurants."

Qdrant's hybrid approach gives you:

- **Semantic understanding** from vectors ("great food" matches "delicious meal")
- **Precise filtering** from metadata (price range, location, date, ratings)
- **Fast searches** because Qdrant indexes both vectors and metadata

Key Takeaways

1. **Vector size is determined by your embedding model** - you must match it exactly
2. **Only embed the meaningful text** (like review content) - store everything else as metadata
3. **Use COSINE distance for most text applications**
4. **Combine vector search with metadata filtering** for powerful, precise results
5. **Think of it as a semantic-aware SQL database** that understands meaning while allowing precise filtering

Best Practices

- Choose your embedding model first, then set collection size to match
- Store rich metadata in the payload for flexible filtering
- Use descriptive field names in your payload
- Index frequently filtered fields for better performance
- Test different distance metrics with your specific use case