Qdrant Data Model Masterclass

Points, Vectors, and Payloads

Created by Raj Ghosh

February 15, 2026

Contents

---

1 Points: The Core Entity

Understanding Qdrant's core data model is essential for building effective vector search applications. Points are the central entity that Qdrant operates with.

If IDs are not provided, Qdrant Client will automatically generate them as random UUIDs.

2 Vector Types in Qdrant

Qdrant supports different types of vectors to provide various approaches to data exploration.

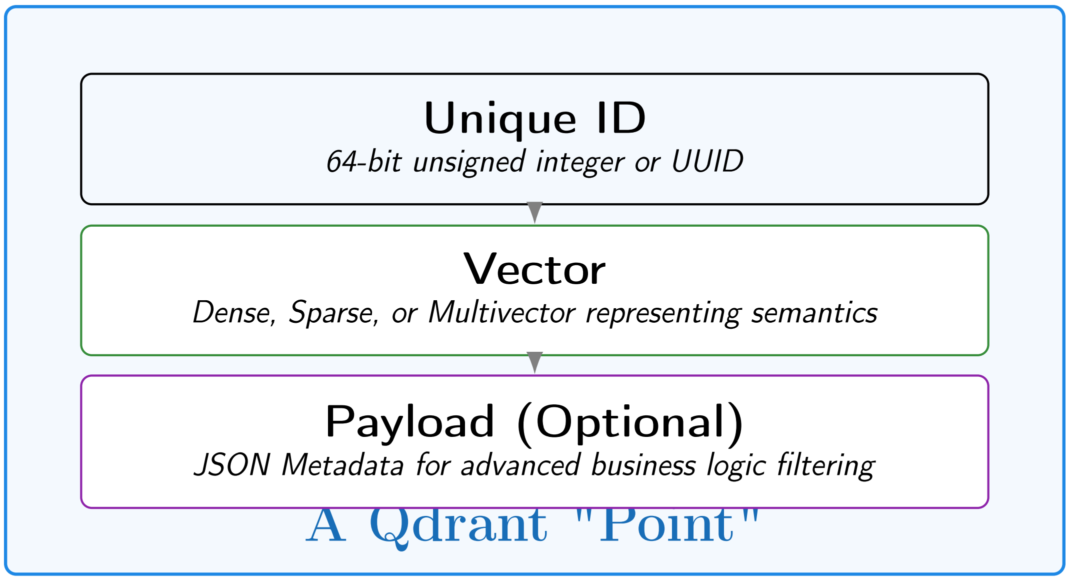
Unique ID
64-bit unsigned integer or UUID
Vector
Dense, Sparse, or Multivector representing semantics
Payload (Optional)
JSON Metadata for advanced business logic filtering
A Qdrant "Point"

Figure 1: Visual Structure of a Qdrant Point

2.1 Dense Vectors

Dense vectors are generated by neural networks (embeddings) to capture complex relationships and semantics. For example, when comparing two similar sentences, their dense embeddings will be mathematically similar because they share linguistic elements.

2.2 Sparse Vectors

Mathematically identical to dense vectors, but containing many zeros. They use optimized storage. Instead of storing zeros, they are represented as a list of `(index, value)` pairs. They are perfect for exact keyword matching.

```
# Dense vector: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 0.0, 0.0]
# Qdrant JSON format (Sparse):
{
 "indices": [6, 7],
 "values": [1.0, 2.0]
}
```

Listing 1: Sparse Vector Representation

2.3 Multivectors

Advanced techniques like late-interaction models (e.g., ColBERT) generate a set of vectors (often one for each text token). Qdrant allows you to store this entire matrix on a single point!

2.4 Named Vectors

You can attach several embeddings of *any type and structure* to a single point using Named Vectors.

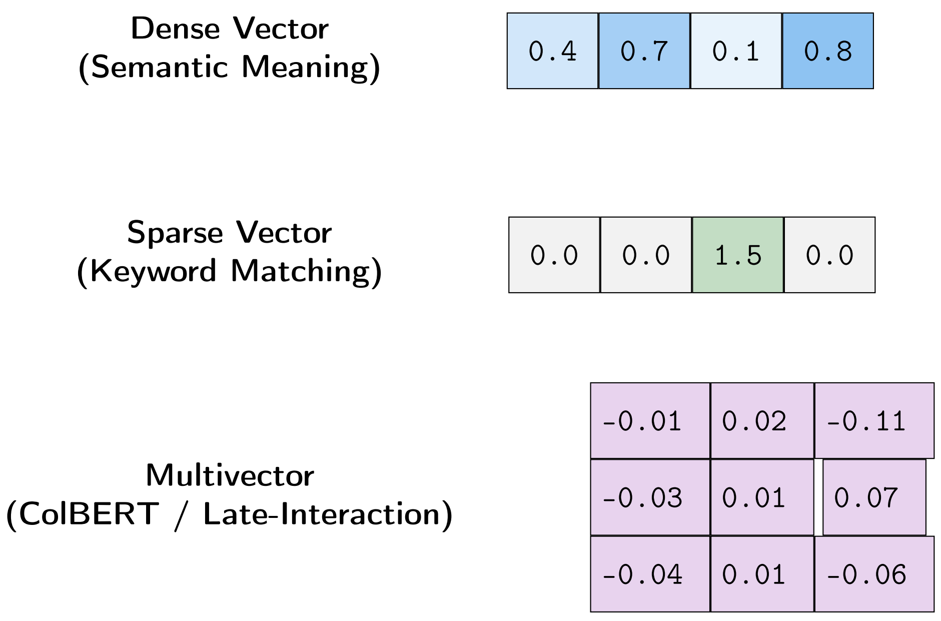

Figure 2: Visualizing the three primary Vector Types

```python
client.upsert(
    collection_name="{collection_name}",
    points=[
        models.PointStruct(
            id=1,
            vector={
                "image": [0.9, 0.1, 0.1, 0.2], # Dense
                "text": [0.4, 0.7, 0.1, 0.8, 0.1], # Dense
                "text-sparse": { # Sparse
                    "indices": [1, 3, 5, 7],
                    "values": [0.1, 0.2, 0.3, 0.4],
                },
            },
        ),
    ],
)
```

Listing 2: Inserting a Point with Named Vectors

3 Vector Dimensionality

Dimensionality directly impacts search efficiency, memory, and accuracy.

Memory Impact Rule of Thumb

A 1536-dimension Float32 vector requires **6KB**. Scale that to 1 Million vectors, and you need **6GB of memory**. 3072-dimension vectors double the requirement!

Model	**Dimensions**	**Use Case**
all-MiniLM-L6-v2	384	Fast, lightweight semantic search. Prototyping.
BAAI/bge-base-en-v1.5	768	High-quality baseline for RAG.
OpenAI text-embedding-3-small	1536	Excellent for production semantic search.
OpenAI text-embedding-3-large	3072	Maximum detail. Ideal for high-accuracy RAG.

Table 1: Common Model Dimensions

4 Common Embedding Sources

Choosing the right source balances cost, performance, and accuracy.

1. **FastEmbed by Qdrant (On-Premise, Optimized):** CPU-friendly generation using quantized weights. Up to 50% faster than PyTorch. Default model `BAAI/bge-small-en-v1.` is lightweight (67MB).

2. **Cloud Providers (Managed):** Qdrant Cloud Inference or third-party APIs (OpenAI, Anthropic). Eliminates network latency if using Qdrant's internal inference.

3. **Open Source Models (Customizable):** Hugging Face models running locally. Maximum control for fine-tuning.

Feature	FastEmbed	Cloud Providers	Open Source
Execution	On-premise (CPU)	Cloud API	On-premise (GPU/CPU)
Speed	Optimized low-latency	Varies by API	Varies by hardware
Control	High	Low	Maximum
Best for	Fast, lightweight CPU	Ease of use, managed	Domain customization

Table 2: Embedding Sources Comparison

5 Payloads (Metadata)

Payloads hold structured metadata for filtering and refinement. They combine the semantic relevance of vectors with strict business logic.

5.1 Payload Types & Structures

Payloads can store standard types, but also complex structures like **Arrays** (`tags: ["vegan", "organic"]`) and **Nested JSON Objects** (`user: {"id": 123, "city": "Berlin"}`).

Type	Example	Description
Keyword	`category: "electronics"`	Exact string matching
Integer	`stock_count: 120`	64-bit signed numerical filtering
Float	`price: 19.99`	Prices, ratings, etc.
Bool	`in_stock: true`	True/false logical checks
Geo	`{"lon": 13.4, "lat": 52.5}`	Radius or bounding box queries
Datetime	`"2024-03-10T12:00:00Z"`	Timestamps in RFC 3339 format
UUID	`"550e8400-e29b..."`	Memory-efficient UUID storage

Table 3: Qdrant Payload Types

6 Filtering Logic: Building Complex Queries

Qdrant uses boolean clauses:

- `must`: All conditions must be satisfied (AND)
- `should`: At least one condition satisfied (OR)
- `must_not`: None should be satisfied (NOT)

6.1 Condition Types Reference

Condition	Use Case	Example Query Structure
Match	Exact value	`"match": {"value": "electronics"}`
Range	Numerical boundaries	`"range": {"gte": 50, "lte": 200}`
Full Text	Substring matching	`"match": {"text": "amazing service"}`
Geospatial	Location-based	`"geo_radius": {"center": ..., "radius": 10000}`
Nested	Array object filtering	`"nested": {"key": "reviews", "filter": ...}`
Has ID	Specific IDs	`"has_id": [1, 5, 10]`

Table 4: Filtering Capabilities Reference

6.2 Advanced: Nested Objects Filtering

For complex arrays of objects, nested filtering ensures conditions are evaluated within *individual* array elements.

```
# Finding products where the *same* review is both 5-stars AND verified
models.Filter(
    must=[
        models.NestedCondition(
            nested=models.Nested(
                key="reviews",
                filter=models.Filter(must=[
                    models.FieldCondition(key="rating", match=models.MatchValue(value=5)),
                    models.FieldCondition(key="verified", match=models.MatchValue(value=True))
                ])
            )
        )
    ]
)
```

Listing 3: Nested Filtering Example

Performance Optimization

To maximize filtering performance, always create payload indexes for frequently filtered fields. Qdrant will automatically bypass vector indexing entirely and use payload indexes for faster results if the filters are highly selective!

7 Summary

Points combine unique IDs, vectors, and metadata into a flexible foundation. Dimensionality choice balances accuracy against performance, and payloads enable rich filtering and structured metadata alongside semantic search.