Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Vektordatenbanken

Eine kurze Einführung zur Suche mit Qdrant und Neo4j

Karsten Keßler

20.02.2026

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Überblick: Vektordatenbanken und ihre Bedeutung

Vektordatenbanken sind spezialisierte Datenbanksysteme, die für die Speicherung, Verwaltung und Abfrage von Vektoren (hochdimensionale numerische Arrays) optimiert sind. Im Zeitalter von künstlicher Intelligenz und Machine Learning haben Vektordatenbanken eine zentrale Rolle eingenommen, da sie es ermöglichen, semantische Ähnlichkeiten zwischen Dokumenten, Bildern, Audio und anderen Datentypen effizient zu finden.

Qdrant ist eine moderne, hochperformante Vektordatenbank, die speziell für großmaßstäbliche Vektor-Suche entwickelt wurde. Mit ihrer Kombination aus schneller Suche, Skalierbarkeit und Flexibilität hat sich Qdrant zu einer führenden Lösung in diesem Bereich entwickelt.

Einsatzbereiche von Vektordatenbanken:

- Semantic Search: Intelligente Suche über semantische Ähnlichkeit statt exakter Keyword-Matching
- Recommendation Systems: Personalisierte Empfehlungen basierend auf Vektorähnlichkeit
- RAG (Retrieval Augmented Generation): Verbindung von LLMs mit externem Wissen
- Image and Video Search: Inhaltsbasierte Suche in Bildern und Videos
- Anomaly Detection: Erkennung von Abweichungen basierend auf Vektorähnlichkeit

Vergleich: Traditionelle Datenbanken vs. Vektordatenbanken

Traditionelle Datenbanken (SQL)	Vektordatenbanken
Tabellenbasiert mit festen Schemata	Vektorbasiert, optimiert für n-dimensionale Daten
Exakte Matching-Abfragen	Similaritäts- und distanzbasierte Abfragen
ACID-Transaktionen, starke Konsistenz (atomar, Konsistenz, Isolation, dauerhaft)	Je nach System: von eventual consistency bis (teilweise) starker Konsistenz; in verteilten Setups häufig konfigurierbar
Vertikale Skalierung	Horizontale Skalierung
Z.B.: PostgreSQL, MySQL, Oracle	Z.B.: Qdrant, Pinecone

Traditionelle Datenbanken eignen sich hervorragend für strukturierte Daten mit komplexen Abfragen und starken Konsistenzanforderungen. Vektordatenbanken hingegen sind spezialisiert auf die schnelle Suche nach ähnlichen Vektoren in hochdimensionalen Räumen, was sie unverzichtbar für moderne KI-Anwendungen macht.

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Definition und Einsatz von Qdrant

Qdrant ist eine in Rust geschriebene, hochperformante Vektordatenbank, die speziell für die Speicherung und schnelle Abfrage von Vektoren optimiert ist.

Hauptmerkmale von Qdrant:

- Hohe Suchgeschwindigkeit: Millisekunden-Abfragen auf Millionen von Vektoren
- Skalierbarkeit: Unterstützung für verteilte Systeme und große Datenmengen
- REST und gRPC APIs: Flexible Integration in verschiedene Programmiersprachen
- Payload-Speicherung: Zusätzliche Metadaten neben Vektoren speichern
- Filter und Hybrid Search: Kombination von Vektorsuche mit Metadaten-Filtern
- Persistente Speicherung: Datenintegrität auch nach Systemabstürzen

Ideale Einsatzszenarien für Qdrant:

- Semantic Search in Dokumenten und Webseiten
- AI-powered Chatbots und RAG-Systeme
- Personalisierte Empfehlungssysteme
- Bildbasierte Suche und visuelle Ähnlichkeit
- Anomalieerkennung in Zeitreihen
- Automatische Duplikat-Erkennung

Grundkonzepte von Qdrant

Vektoren und Embeddings

Ein Vektor ist ein Array von Zahlen, das eine mehrdimensionale Darstellung von Daten repräsentiert. Embeddings sind numerische Darstellungen von Text, Bildern oder anderen Datentypen, die von Machine-Learning-Modellen erstellt werden. Sie kodieren semantische Bedeutung in numerischer Form, sodass ähnliche Konzepte ähnliche Vektoren produzieren.

In diesem Skript verwenden wir für Text-Embeddings das Sentence-Transformers-Modell "all-MiniLM-L6-v2" (384 Dimensionen). Dokumente und Suchanfragen werden mit demselben Modell in denselben Vektorraum abgebildet; dadurch können wir semantische Ähnlichkeiten effizient suchen.

Beispiel eines 3-dimensionalen Vektors: [0.25, -0.12, 0.89]

Collections (Sammlungen)

Eine Collection ist ein Container für eine Gruppe von Vektoren mit konsistenter Dimension. Sie entsprechen konzeptionell Tabellen in relationalen Datenbanken, werden aber speziell für

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Vektordaten optimiert. Jeder Vektor in einer Collection muss die gleiche Anzahl von Dimensionen haben.

Points (Datenpunkte)

Ein Point ist das grundlegende Datenelement in Qdrant und besteht aus:

- ID: Eindeutige Kennung für den Punkt
- Vector: Der eigentliche Vektor mit numerischen Werten
- Payload: Optionale Metadaten (JSON-ähnliche Struktur)

Distance Metrics (Distanzmetriken) – Vector Similarity Measures

In der Praxis sind besonders verbreitet: Kosinus (Cosine), Skalarprodukt (Dot) und euklidische Distanz (Euclid). Qdrant unterstützt zusätzlich Manhattan-Distanz.

Cosine Similarity

Cosine Similarity ist ein populäres Ähnlichkeitsmaß, das den Kosinus des Winkels zwischen zwei Vektoren im multidimensionalen Raum misst. Es ist eines der am häufigsten verwendeten Maße für Text-Embeddings und wird besonders bei der semantischen Suche eingesetzt.

$$\text{Kosinus-Ähnlichkeit: } \cos(A, B) = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$

Wobei $A \cdot B$ das Skalarprodukt der beiden Vektoren ist und $\|A\|$ und $\|B\|$ die Größe (Länge) der Vektoren darstellen.

Rechenbeispiel

Seien a = (1, 2, 3) und b = (4, 0, 8).

Skalarprodukt: $a \cdot b = 1 \cdot 4 + 2 \cdot 0 + 3 \cdot 8 = 28$.

Normen: $\|a\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$, $\|b\| = \sqrt{4^2 + 0^2 + 8^2} = \sqrt{80}$.

Cosinus-Ähnlichkeit: $\cos(a, b) = \frac{28}{\sqrt{14} \cdot \sqrt{80}} \approx 0{,}837$.

- Orthogonal: $a = (1{,}0), b = (0{,}1) \rightarrow a \cdot b = 0 \rightarrow \cos(a, b) = 0$.
- Gleiche Richtung: $b = 2a \rightarrow \cos(a, b) = 1$ (maximale Ähnlichkeit).
- Entgegengesetzte Richtung: $a = (1{,}0), b = (-1{,}0) \rightarrow a \cdot b = -1 \rightarrow \|a\| = \|b\| = 1 \rightarrow \cos(a, b) = -1$.

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

- Der Wert liegt zwischen -1 und 1, wobei 1 bedeutet, dass die Vektoren vollständig ähnlich sind
- Sie misst nur die Richtung der Vektoren, nicht ihre Größe. Zwei proportionale Vektoren unterschiedlicher Länge werden als identisch bewertet
- Ideal für Text und NLP-Anwendungen, da semantische Ähnlichkeit oft winkelbasiert ist

```
import numpy as np

# Define two example vectors
vector_a = np.array([1, 2, 3])
vector_b = np.array([4, 5, 6])

# Calculate Cosine Similarity
cosine_sim = np.dot(vector_a, vector_b) / (np.linalg.norm(vector_a) *
np.linalg.norm(vector_b))

print('Cosine Similarity:', cosine_sim)
```

Euclidean Distance

Die euklidische Distanz ist die gerade Linienentfernung zwischen zwei Datenpunkten im multidimensionalen Raum. Sie bietet ein greifbares Maß, um die absoluten Unterschiede zwischen Datenpunkten zu bestimmen und eignet sich besonders für Clustering und Anomalieerkennung, wenn der Größenunterschied aussagekräftig ist.

Euklidischer Abstand: $d(A,B) = \|A - B\|_2 = \sqrt{\sum_i (A_i - B_i)^2}$

Dies ist die Quadratwurzel der Summe der quadrierten Differenzen aller Dimensionen.

- Sie ist empfindlich gegenüber der Skalierung der Daten – wenn Features in sehr unterschiedlichen Bereichen variieren, ist eine Normalisierung oft erforderlich
- Sie berücksichtigt sowohl die Richtung als auch die Größe (Länge) der Vektoren
- Geeignet für räumliche Daten und Clustering-Aufgaben

```
import numpy as np

# Define two example vectors
vector_a = np.array([1, 2, 3])
vector_b = np.array([4, 5, 6])
```

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

```
# Calculate Euclidean distance
euclidean_dist = np.linalg.norm(vector_a - vector_b)


print('Euclidean Distance:', euclidean_dist)
```

Payload und Filtration

Payloads sind strukturierte Metadaten, die zusammen mit Vektoren gespeichert werden. Sie ermöglichen Hybrid Search, bei der Vektorsuche mit Metadaten-Filtern kombiniert wird. Beispiele: Dokumenttitel, Zeitstempel, Kategorien oder benutzerdefinierte Tags.

Architektur von Qdrant

Qdrant folgt einer modernen, verteilten Architektur mit folgenden Hauptkomponenten:

Hauptkomponenten:

REST API und gRPC:

Qdrant bietet zwei Kommunikationsprotokolle - REST für einfache HTTP-Anfragen und gRPC für hochperformante binäre Kommunikation über HTTP/2.

Storage Engine:

Die Speicher-Engine ist für persistente Speicherung von Vektoren und Payloads verantwortlich. Qdrant verwendet eine speichereffiziente, auf Festplatte optimierte Struktur.

Vector Index (HNSW - Hierarchical Navigable Small World):

HNSW ist ein spezieller Graph-basierter Index-Algorithmus, der schnelle Vektorsuche in hochdimensionalen Räumen ermöglicht. Er baut eine hierarchische Struktur auf, die es erlaubt, in logarithmischer Zeit zu suchen.

Um eine schnelle Suche nach ähnlichen Vektoren zu ermöglichen, verwendet Qdrant diesen speziellen Vektorindex auf Basis des Algorithmus **HNSW (Hierarchical Navigable Small World)**.

Im Gegensatz zu klassischen B-Bäumen oder k-d-Bäumen organisiert HNSW die Vektoren nicht als Baum, sondern als **mehrstufigen Graphen**. Ähnliche Vektoren sind dabei über Kanten miteinander verbunden. Die Suche beginnt in einer groben oberen Ebene mit wenigen Knoten und nähert sich schrittweise den besten Treffern, während sie in tiefere Ebenen des Graphen wechselt. Dadurch lassen sich auch in sehr großen Datenbeständen ähnliche Vektoren effizient finden.

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Der HNSW-Index ist ein sogenannter **Approximate Nearest Neighbor (ANN)**-Index. Das bedeutet, dass nicht garantiert immer der mathematisch exakt nächste Vektor gefunden wird, die Ergebnisse in der Praxis jedoch sehr genau sind – bei deutlich höherer Performance als eine vollständige Durchmusterung aller Vektoren.

Siehe Abschnitt „HNSW“

Zentrale Konfigurationsparameter

Beim Aufbau des HNSW-Index können folgende Parameter beeinflusst werden:

- **m**
  Anzahl der Verbindungen (Kanten) pro Vektor im Graphen.
  Ein höherer Wert erhöht die Trefferqualität, benötigt aber mehr Speicher.

- **ef_construct**
  Suchbreite beim Aufbau des Indexes.
  Ein höherer Wert führt zu einem qualitativ besseren Index, verlängert jedoch die Aufbauzeit.

Diese Parameter erlauben einen gezielten **Trade-off zwischen Genauigkeit, Speicherbedarf und Performance**.

In Qdrant wird der HNSW-Index automatisch beim Erstellen einer Collection aufgebaut und bildet die Grundlage für die schnelle semantische Vektorsuche.

Sharding:

Für Skalierbarkeit unterstützt Qdrant das Aufteilen von Collections auf mehrere Server, wodurch größere Datenmengen verwaltet werden können.

Replication:

Replikation erhöht die Verfügbarkeit und Fehlertoleranz, indem Daten auf mehreren Knoten kopiert werden.

Datenfluss:

1. Client sendet Query über REST/gRPC
2. Query wird an den Vector Index (HNSW) weitergeleitet
3. Index findet die nächsten ähnlichen Vektoren
4. Metadaten (Payloads) werden abgerufen
5. Ergebnisse werden sortiert und an Client zurückgegeben

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Installation und erste Schritte

Qdrant installieren (mit Docker)

Für den Betrieb von Qdrant wird **Docker** benötigt.
Docker ermöglicht es, Anwendungen in isolierten Containern auszuführen, ohne sie direkt auf dem Host-System installieren zu müssen.

Docker Desktop kann hier heruntergeladen werden:
https://www.docker.com/products/docker-desktop/

Nach der Installation kann die erfolgreiche Einrichtung mit folgendem Befehl überprüft werden:

```
docker --version
```

Die einfachste Methode ist, Qdrant mit Docker zu starten:

```
docker run -p 6333:6333 -p 6334:6334 \
-v "$(pwd)/qdrant_storage:/qdrant/storage:z" \
qdrant/qdrant
```

Hinweis: Der Volume-Mount sorgt dafür, dass die Daten auch nach einem Container-Neustart erhalten bleiben. Die Web-UI erreichst man unter http://localhost:6333/dashboard.

Dies startet Qdrant auf Port 6333 (REST) und 6334 (gRPC).
(gRPC ist ein schnelleres Kommunikationsprotokoll als REST. Statt JSON-Text sendet es komprimierte Binärdaten über HTTP/2.)

Collection erstellen:

Mit REST kann man eine neue Collection mit PUT erstellen:

```
PUT http://localhost:6333/collections/documents1

{

  "vectors": {
    "size": 3,
    "distance": "Cosine"
 }

}
```

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Vektor einfügen:

```
PUT http://localhost:6333/collections/documents/points

{
  "points": [
    {
      "id": 1,
      "vector": [0.1, 0.2, 0.768],
      "payload": {
        "text": "Python ist eine beliebte Programmiersprache",
        "category": "programming"
      }
    },
    {
      "id": 2,
      "vector": [0.2, 0.3, 0.4],
      "payload": {
        "text": "Vektordatenbanken sind wichtig für KI",
        "category": "databases"
      }
    }
  ]
}
```

Ähnliche Vektoren suchen:

Alternative (neuere API): Qdrant bietet zusätzlich die universelle Query-API unter POST http://localhost:6333/collections/<collection_name>/points/query.

```
POST http://localhost:6333/collections/documents/points/search

{
  "vector": [0.15, 0.25, 0.35],
  "limit": 5,
  "with_payload": true
}
```

Payload filter (ohne echte Vektorsuche):

Alternative (neuere API): Qdrant bietet zusätzlich die universelle Query-API unter POST http://localhost:6333/collections/<collection_name>/points/query.

```
POST http://localhost:6333/collections/documents/points/scroll
{
```

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

```
  "filter": {
    "must": [
      { "key": "category", "match": { "value": "programming" } }
    ]
  },
  "limit": 20,
  "with_payload": true
}
```

Praktische Beispiele mit Python und C#

Python-Beispiel mit Qdrant-Client

```
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, PointStruct
from sentence_transformers import SentenceTransformer

# Verbindung zur Qdrant-Instanz (Docker: REST-Port 6333)
client = QdrantClient(url="http://localhost:6333")

# Collection mit 384 Dimensionen erstellen (all-MiniLM-L6-v2)
client.create_collection(
    collection_name="documents",
    vectors_config=VectorParams(size=384, distance=Distance.COSINE),
)

# Text-Embedding-Modell laden
model = SentenceTransformer("all-MiniLM-L6-v2")

documents = [
    "Python ist eine beliebte Programmiersprache",
    "Vektordatenbanken sind wichtig für KI",
    "Qdrant bietet schnelle Vektorsuche",
]

embeddings = model.encode(documents)

# Punkte einfügen
points = [
    PointStruct(id=i, vector=emb.tolist(), payload={"text": doc})
    for i, (doc, emb) in enumerate(zip(documents, embeddings), start=1)
]

client.upsert("documents", points=points)
```

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

```
# Suche durchführen
query = "Programmiersprachen und KI"
query_embedding = model.encode(query).tolist()

results = client.search(
    collection_name="documents",
    query_vector=query_embedding,
    limit=2,
)

for r in results:
    print(f"Score: {r.score:.3f}, Text: {r.payload['text']}")
```

C#-Beispiel mit Qdrant.Client

```
using Qdrant.Client;
using Qdrant.Client.Grpc;

// Client (gRPC, Standard-Port 6334)
var client = new QdrantClient("localhost", 6334);

// Collection erstellen (Beispiel: 384 Dimensionen, Kosinus-Metrik)
await client.CreateCollectionAsync(
    "documents",
    new VectorParams { Size = 384, Distance = Distance.Cosine }
);

// Beispiel-Punkte einfügen
var points = new List<PointStruct>
{
    new PointStruct
    {
        Id = 1,
        Vectors = new float[384], // TODO: hier echtes Embedding einfügen
        Payload =
        {
            ["text"] = "Python ist eine beliebte Programmiersprache",
            ["category"] = "programming"
        }
    },
    new PointStruct
    {
        Id = 2,
        Vectors = new float[384], // TODO: hier echtes Embedding einfügen
        Payload =
```

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de


```
        {
            ["text"] = "Vektordatenbanken sind wichtig für KI",
            ["category"] = "databases"
        }
    }
};

await client.UpsertAsync("documents", points);

// Suche durchführen (Top-5 ähnlichste Punkte)
var queryVector = new float[384]; // TODO: Query-Embedding erzeugen
var hits = await client.SearchAsync("documents", queryVector, limit: 5);

foreach (var hit in hits)
{
    Console.WriteLine($"Score: {hit.Score}");
    Console.WriteLine($"Text: {hit.Payload["text"]}");
}
```

Vector Search in der Praxis

Semantic Search für RAG-Systeme

RAG (Retrieval Augmented Generation) kombiniert Vektordatenbanken mit großen Sprachmodellen. Der Prozess läuft folgendermaßen ab:

6. Benutzer stellt Frage
7. Frage wird in Vektor umgewandelt
8. Qdrant sucht ähnliche Dokumente
9. Relevante Dokumente werden an LLM übergeben
10. LLM generiert kontextuelle Antwort

Hybrid Search mit Filtern

Qdrant unterstützt die Kombination von Vektorsuche mit Metadaten-Filtern:

```
POST collections/docs1/points/search
{
  "vector": [0.1, 0.2, 0.768],
  "filter": {
    "must": [
      {
        "key": "category",
        "match": { "value": "databases" }
      }
    ]
```

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

```
  },
  "limit": 10,
  "with_payload": true
}
```

Dies erlaubt es, nur Vektoren zu suchen, die bestimmte Metadaten-Bedingungen erfüllen.

Empfehlungssysteme

In Empfehlungssystemen werden User- und Item-Embeddings in Qdrant gespeichert. Die ähnlichsten Items zu einem Benutzer können schnell gefunden werden, indem der User-Vektor gegen alle Item-Vektoren durchsucht wird.

Best Practices bei der Arbeit mit Qdrant

Richtige Embedding-Modelle wählen

Wählen Sie Embedding-Modelle basierend auf Ihrem Use-Case. Für allgemeine Text-Ähnlichkeit sind Modelle wie "all-MiniLM-L6-v2" (384 Dimensionen) oder "sentence-transformers/all-mpnet-base-v2" (768 Dimensionen) geeignet. Größere Modelle bieten bessere Qualität, benötigen aber mehr Speicher.

Payload-Struktur optimieren

Payloads sollten sparsam mit Metadaten gefüllt werden. Zu große Payloads verlangsamen die Suche. Speichern Sie nur notwendige Metadaten und externe Referenzen für zusätzliche Daten.

Indizierung und Quantisierung

Qdrant unterstützt Quantisierung, um Speicher zu sparen. Product Quantization (PQ) kann die Vektorgröße je nach Konfiguration deutlich reduzieren, mit minimalen Auswirkungen auf die Suchergebnisse.

Partitionierung und Sharding

Für große Systeme sollten Collections partitioniert werden. Qdrant unterstützt automatisches Sharding, das Daten gleichmäßig auf mehrere Knoten verteilt.

Monitoring und Performance-Tuning

- Überwachen Sie Query-Latenzen und Speichernutzung
- Nutzen Sie Batch-Operationen für Multiple Inserts
- Setzen Sie angemessene HNSW-Parameter (M und ef)
- Verwenden Sie Connection-Pooling in Produktionsumgebungen

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Sicherheit

- Verwenden Sie API-Keys für Zugriffskontrolle
- Wichtig: Standardmäßig startet Qdrant ohne Authentifizierung/Encryption – in Produktion unbedingt absichern.
- Verschlüsseln Sie Daten in Transit (TLS/SSL)
- Sichern Sie Collections regelmäßig
- Verwenden Sie Read-Replicas für Desaster Recovery

Testing und Validierung

Testen Sie die Qualität von Embeddings und Suchergebnissen mit Ground-Truth-Datasets. Verwenden Sie Metriken wie Mean Reciprocal Rank (MRR) und Recall@K zur Evaluierung.

Zusammenfassung und Ausblick

Vektordatenbanken wie Qdrant sind eine Schlüsseltechnologie für moderne KI-Anwendungen. Ihre Fähigkeit, hochdimensionale Vektoren effizient zu speichern und zu durchsuchen, ermöglicht es Entwicklern, intelligente Anwendungen zu bauen, die semantisches Verständnis bieten.

Qdrant bietet dabei eine ausgezeichnete Balance zwischen Leistung, Skalierbarkeit und Benutzerfreundlichkeit. Mit REST- und gRPC-APIs und unterstützten Python- und C#-Bindings ist es eine vielseitig einsetzbare Lösung für Universität und Industrie.

Qdrant und Neo4j – Ein Vergleich der Datenmodelle

Während Qdrant sich auf semantische Ähnlichkeit durch Vektorberechnungen konzentriert, verfolgt Neo4j einen grundlegend anderen Ansatz. Der Schlüssel zum Verständnis dieses Unterschieds liegt in der Bedeutung der Pfeile in beiden Systemen.

Neo4j – Semantische Beziehungen

Neo4j ist eine Graphdatenbank, die eine völlig andere Philosophie verfolgt als Vektordatenbanken. Während Qdrant numerische Ähnlichkeiten berechnet, modelliert Neo4j explizite Beziehungen zwischen Entitäten.

- Pfeile sind persistente Kanten: In Neo4j sind Pfeile (Relationen) als explizite Datenstrukturen im Graphen gespeichert, nicht nur berechnet.
- Teil des Datenmodells: Die Beziehungen sind zentral für die Datenorganisation und haben einen semantischen Bedeutungswert.
- Mit Typ und Richtung: Jede Beziehung hat einen Typ (z.B. HAS_BRAND, BELONGS_TO) und eine Richtung, die die Art der Verbindung genau definiert.

Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

- Explizit modellierter fachlicher Zusammenhang: Die Beziehungen repräsentieren echtes Wissen über die Domäne, nicht nur numerische Ähnlichkeit.

Die zentrale Frage bei Neo4j lautet daher: „Was gehört zusammen?"

Vektoren versus Graphen

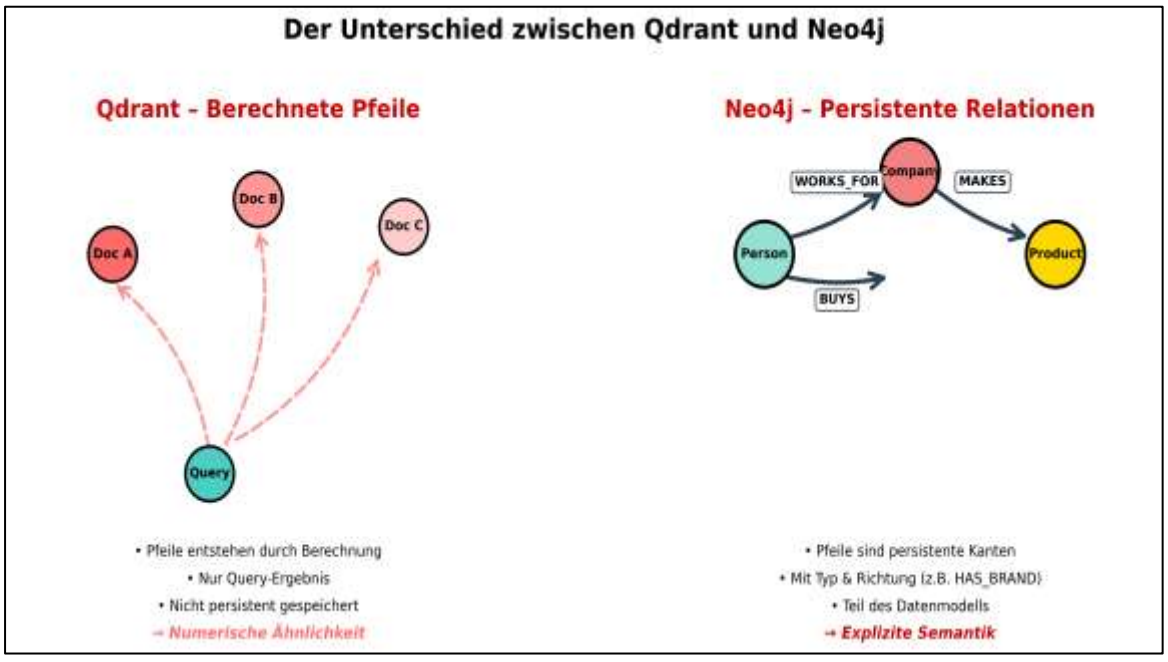
Der Unterschied lässt sich prägnant zusammenfassen: „Vektoren rechnen Ähnlichkeit – Graphen speichern Bedeutung."

In Qdrant sehen Pfeile aus wie Beziehungen, sind aber in Wirklichkeit nur das Ergebnis einer Berechnung. Der Algorithmus findet die ähnlichsten Vektoren zu einem Query-Vektor – diese „Verbindungen" existieren nicht dauerhaft im System.

In Neo4j hingegen sind die Pfeile das Wissen selbst. Sie sind permanent gespeichert, haben explizite Bedeutung und Typ, und werden als erstes Klassenelement des Datenmodells behandelt.

Einsatzszenarien

Qdrant eignet sich ideal für Anwendungen, bei denen es um die Findung von Ähnlichkeit geht: Dokumentenempfehlungen, Image Search, RAG-Systeme. Neo4j hingegen ist die Wahl, wenn die explizit modellierten Beziehungen die Essenz der Anwendung bilden: Soziale Netzwerke, Wissensgraphen, Fraud Detection, Master Data Management. Oft ist es sogar sinnvoll, beide Systeme kombiniert einzusetzen – Graphdatenbanken für strukturierte Beziehungen und Vektordatenbanken für semantische Ähnlichkeit.



Karsten Keßler
tutor@kkessler.de
tutor.kkessler.de
14.02.2026

Neo4j Praktisch: Cypher Queries

```
// Löschen, falls vorhanden
MATCH (n:Kurs) DETACH DELETE n;
MATCH (n:Dozent) DETACH DELETE n;
CREATE (kurs1:Kurs {id: 1, name: "Vektordatenbanken", category: "Datenbanken"})
CREATE (kurs2:Kurs {id: 2, name: "Statistik", category: "Mathematik"})
CREATE (kurs3:Kurs {id: 3, name: "Analysis", category: "Mathematik"})
MATCH (k:Kurs)
RETURN k.id, k.name, k.category
CREATE (dozent:Dozent {name: "Karsten Keßler"})
MATCH (d:Dozent)
RETURN d.name
MATCH (kurs1:Kurs {id: 1}), (kurs2:Kurs {id: 2}), (kurs3:Kurs {id: 3}), (dozent:Dozent {name:
"Karsten Keßler"})
CREATE (kurs1)-[:GEHALTEN_VON]->(dozent)
CREATE (kurs2)-[:GEHALTEN_VON]->(dozent)
CREATE (kurs3)-[:GEHALTEN_VON]->(dozent)
// Nur Datenbank-Kurse anzeigen
MATCH (k:Kurs)-[:GEHALTEN_VON]->(d:Dozent)
WHERE k.category = "Datenbanken"
RETURN k.name as Kurs, d.name as Dozent
// Alle Kurse mit Dozenten
MATCH (k:Kurs)-[:GEHALTEN_VON]->(d:Dozent)
RETURN k.name as Kurs, k.category as Kategorie, d.name as Dozent
ORDER BY k.name
// Graph visualisieren (MIT Beziehungen)
MATCH (k:Kurs)-[g:GEHALTEN_VON]->(d:Dozent)
RETURN k, g, d
```

Weitere Ressourcen:

- LLM-Erklärseite mit didaktischer Aufbereitung zentraler Konzepte
- Qdrant Dokumentation: https://qdrant.tech/documentation/
- GitHub Repository: https://github.com/qdrant/qdrant
- Sentence Transformers: https://www.sbert.net/