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

Vektorsuche vs. SQL-Suche

Wie Qdrant zu seinem Ergebnis kommt und warum semantische Suche der SQL-Suche überlegen ist

Karsten Keßler

20.02.2026

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

Inhaltsverzeichnis

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

---

1. Wie Qdrant zu seinem Ergebnis kommt

Qdrant ist eine Vektordatenbank, die Dokumente nicht als Text, sondern als hochdimensionale Zahlenvektoren (Embeddings) speichert. Die Suche erfolgt über mathematische Ähnlichkeitsberechnung im Vektorraum statt über Textvergleiche. Der gesamte Prozess lässt sich in drei Phasen unterteilen.

1.1 Phase 1: Indexaufbau (Offline)

Schritt 1: Produkte aus MySQL laden

Die Rohdaten werden aus der MySQL-Datenbank geladen. Der IndexService ruft dazu die MySQL-Repository-Methode auf:

```
# Produkte aus MySQL laden
products = self.mysql_repo.load_products_for_index(
    limit=limit, include_tags=True
)
```

Schritt 2: Produkttext zusammenbauen

Jedes Produkt wird in einen zusammenhängenden Textblock konvertiert. Die Methode product_to_document() fügt alle relevanten Felder zusammen — Titel, Beschreibung, Marke, Kategorie, Tags, Anwendung, Belastungsklasse und Temperaturbereich:

```
# Produktfelder zu einem Textblock zusammenfügen
@staticmethod
def product_to_document(product: dict) -> str:
    parts: list[str] = []
    if product.get("title"):
        parts.append(product["title"])
    if product.get("description"):
        parts.append(product["description"])
    if product.get("brand"):
        parts.append(f"Marke: {product['brand']}")
    if product.get("category"):
        parts.append(f"Kategorie: {product['category']}")
    if product.get("tags"):
        parts.append("Tags: " + ", ".join(product["tags"]))
    if product.get("application"):
        parts.append(f"Anwendung: {product['application']}")
    return "\n".join(parts).strip()
```

Dieser kombinierte Textblock ist entscheidend: Er bestimmt, wonach Qdrant suchen kann. Alle Felder fließen in einen einzigen Text ein, der anschließend als Ganzes embeddet wird.

Schritt 3: Embeddings erzeugen

Das Sentence-Transformer-Modell all-MiniLM-L6-v2 wandelt jeden Textblock in einen 384-dimensionalen Vektor um. Die Normalisierung stellt sicher, dass alle Vektoren die Länge 1 haben:

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

```
# Texte in Vektoren umwandeln
def embed_texts(self, texts: list[str]) -> list:
    model = self._get_embedding_model()
    return model.encode(texts, normalize_embeddings=True)
```

Das Ergebnis: Jeder Produkttext wird zu einem Zahlenvektor der Form [0.032, -0.118, 0.241, ...] mit exakt 384 Dimensionen. Semantisch ähnliche Texte erhalten dabei ähnliche Vektoren — das ist die Kernidee der Vektorsuche.

Schritt 4: In Qdrant speichern

Die Collection wird mit HNSW-Indexierung und Kosinus-Distanz erstellt:

```
# Qdrant-Collection konfigurieren
self.client.create_collection(
    collection_name=collection_name,
    vectors_config=VectorParams(
        size=vector_size,          # 384
        distance=Distance.COSINE   # Kosinus-Aehnlichkeit
    ),
    hnsw_config=HnswConfigDiff(
        m=16,              # 16 Nachbarverbindungen pro Knoten
        ef_construct=128   # Suchbreite beim Aufbau
    ),
)
```

Jeder Punkt in der Collection besteht aus drei Teilen:

Bestandteil	Inhalt	Beispiel
id	MySQL-Produkt-ID	3803
vector	384-dim Embedding-Vektor	[0.032, -0.118, 0.241, ...]
payload	Metadaten (Titel, Marke, etc.)	{"title": "Schaeffler STRASSE-3803", ...}

Der HNSW-Algorithmus (Hierarchical Navigable Small World) baut dabei einen mehrstufigen Graphen auf, in dem jeder Punkt mit seinen nächsten Nachbarn verbunden ist. Dies ermöglicht eine effiziente Suche in $O(\log n)$ statt $O(n)$ bei einer Brute-Force-Suche.

1.2 Phase 2: Suchanfrage (Online)

Schritt 1: Query-Embedding erzeugen

Die Suchanfrage des Nutzers wird mit dem gleichen Modell in einen Vektor umgewandelt:

```
# Suchanfrage in Vektor umwandeln
query_embedding = self.embed_texts([query])[0].tolist()

# Beispiel: "Schiff" -> [0.087, -0.203, 0.156, ...] (384 Dimensionen)
```

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

Schritt 2: Approximate Nearest Neighbor Search

Qdrant sucht die ähnlichsten Vektoren im HNSW-Graphen:

```
# Ähnlichste Vektoren in Qdrant suchen
points = self.qdrant_repo.search(
    collection_name=collection,
    query_vector=query_embedding,
    limit=topk,
    with_payload=True,
)
```

Intern in der Qdrant-Repository wird die Query an den Qdrant-Server gesendet:

```
# Query an den Qdrant-Server senden
resp = self.client.query_points(
    collection_name=collection_name,
    query=query_vector,
    limit=limit,
    with_payload=with_payload,
)
```

Der HNSW-Algorithmus arbeitet dabei wie folgt:

1. Einstieg auf der obersten Ebene des Graphen bei einem zufälligen Startknoten
2. Greedy-Navigation: Sprung zum jeweils ähnlichsten Nachbarn auf jeder Ebene
3. Auf der untersten Ebene: Durchsuchen von ef=64 Kandidaten (konfigurierbar; ef ist der Suchparameter zur Queryzeit, nicht zu verwechseln mit ef_construct, der nur beim Indexaufbau verwendet wird)
4. Für jedes Paar: Berechnung der Kosinus-Ähnlichkeit
5. Rückgabe der Top-K Punkte mit dem höchsten Score

Die Kosinus-Ähnlichkeit berechnet sich als:

$$\cos(A, B) = \frac{A \cdot B}{\|A\| \times \|B\|}$$

Da alle Vektoren normiert sind (normalize_embeddings=True), vereinfacht sich dies zum reinen Skalarprodukt: $\cos(A, B) = A \cdot B$. Ein Score von 1.0 bedeutet identische Bedeutung, 0.0 bedeutet keine semantische Überschneidung.

1.3 Phase 3: Ergebnisaufbereitung

Die Ergebnisse werden mit ihrem Ähnlichkeits-Score formatiert und dem Nutzer präsentiert:

```
# Ergebnisse mit Score aufbereiten
for p in points:
```

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

```
    payload = (p.payload or {}) if hasattr(p, "payload") else {}
    score = getattr(p, "score", None)
    results.append({
        "product_id": payload.get("mysql_id"),
        "name":       payload.get("title"),
        "brand":      payload.get("brand"),
        "category":   payload.get("category"),
        "tags":       payload.get("tags", []),
        "price":      payload.get("price"),
        "doc_preview": payload.get("doc_preview"),
        "score":      round(float(score), 4),
        "score_type": "cosine similarity (Qdrant)",
    })
```

Jedes Ergebnis enthält also neben den Produktdaten einen kontinuierlichen Ähnlichkeits-Score, der ein automatisches Ranking ermöglicht.

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

---

2. Wie eine SQL-Suche funktioniert

Im Gegensatz zur Vektorsuche arbeitet die SQL-Suche mit exaktem Zeichenkettenvergleich. Der Nutzer schreibt die SQL-Query selbst und bestimmt damit genau, in welchen Feldern und nach welchem Muster gesucht wird.

2.1 Architektur der SQL-Suche

Der Weg einer SQL-Suchanfrage durch den Code:

1. Route: Route — nimmt die Query entgegen und prüft den Typ

2. Service: Service — leitet an das Repository weiter

3. Repository: Repository — führt Sicherheitsprüfungen durch und exekutiert die Query

```
# Query-Typ prüfen und weiterleiten
elif search_type in ("sql", "script"):
    if not query.strip().upper().startswith("SELECT"):
        answer = "Nur SELECT-Queries sind erlaubt!"
    else:
        results = product_service.execute_sql_query(query)
```

2.2 Funktionsweise von LIKE

Der LIKE-Operator in MySQL führt einen zeichenweisen Mustervergleich durch. Das Prozentzeichen (%) steht als Wildcard für beliebig viele Zeichen:

```
SELECT * FROM products WHERE name LIKE '%schiff%'

-- MySQL prueft fuer JEDE Zeile:
-- Enthaelt das Feld 'name' den Substring 'schiff'?
-- -> Zeichenweiser Vergleich, KEINE semantische Analyse
```

Dies ist ein rein syntaktischer Vergleich. MySQL durchsucht jede Zeile der Tabelle (oder nutzt einen Index, falls vorhanden) und prüft, ob die Zeichenkette 'schiff' als Teilstring vorkommt. Es gibt kein Verständnis für Bedeutung, Synonyme oder verwandte Begriffe. Hinweis: In MySQL 8.4 mit der Standard-Collation utf8mb4_0900_ai_ci ist LIKE case-insensitive, d.h. '%schiff%' würde auch „Schiff" oder „SCHIFF" finden.

2.3 Sicherheitsmechanismen

Da der Nutzer bei der SQL-Suche beliebige Queries eingeben kann, implementiert der Code umfangreiche Sicherheitsprüfungen:

```
# Sicherheitsprüfungen für SQL-Queries
# 1. Nur SELECT erlaubt
if not query_upper.startswith("SELECT"):
    raise ValueError("Nur SELECT-Queries sind erlaubt!")

# 2. Keine Mehrfach-Statements
```

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

```
if ";" in query_no_strings:
    raise ValueError("Mehrere Statements sind nicht erlaubt.")

# 3. Blocklist gefaehrlicher Keywords
forbidden = ["DROP","DELETE","UPDATE","INSERT","ALTER",...]

# 4. Allowlist erlaubter Tabellen
allowed_tables = {
    "PRODUCTS","BRANDS","CATEGORIES",
    "TAGS","PRODUCT_TAGS","ETL_RUN_LOG"
}
```

Diese Sicherheitsschicht ist notwendig, weil der Nutzer direkt SQL schreibt. Bei der Qdrant-Suche entfällt dies komplett, da der Nutzer nur natürliche Sprache eingibt und keinen Zugriff auf die Abfragesprache hat.

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

---

3. Gegenüberstellung am Beispiel „Schiff“

Das Suchbeispiel „Schiff“ zeigt den fundamentalen Unterschied zwischen beiden Ansätzen besonders deutlich. Keines der Produkte in der Datenbank enthält das Wort „Schiff“ im Titel — und trotzdem liefert Qdrant sinnvolle Ergebnisse.

3.1 SQL-Suche: SELECT * FROM products WHERE name LIKE '%schiff%'

Die SQL-Query lautet:

```
SELECT * FROM products WHERE name LIKE '%schiff%'
```

Ergebnis: 6 Treffer, aber semantisch irrelevant

MySQL durchsucht die Spalte „name" jeder Zeile nach dem Substring „schiff". Tatsächlich findet die Query 6 Produkte, da mehrere Produktbezeichnungen das Kürzel „SCHIFF" enthalten:

- „NSK SCHIFF-8019" — enthält „schiff" als Teil der Produktnummer
- „Schaeffler SCHIFF-9837" — enthält „schiff" als Teil der Produktnummer
- „SKF SCHIFF-9285" — enthält „schiff" als Teil der Produktnummer
- sowie deren jeweilige Variante-B-Versionen

Keines dieser Produkte hat semantisch etwas mit einem Schiff (Wasserfahrzeug) zu tun. Die Zeichenkette „SCHIFF" ist hier ein zufälliger Bestandteil der Produktbezeichnung.

Warum ist das problematisch?

Der LIKE-Operator hat kein Konzept von Bedeutung. Er sieht nur Zeichen. Er findet „SCHIFF" in „NSK SCHIFF-8019", weil die Zeichenkette übereinstimmt — nicht weil das Produkt etwas mit Schiffen zu tun hat. Umgekehrt würde ein Produkt wie „Maritimer Korrosionsschutz für Hochseeanwendungen" nicht gefunden, obwohl es semantisch perfekt zum Suchbegriff „Schiff" passt.

LIKE liefert hier also **falsche Treffer** (false positives) und verpasst gleichzeitig **relevante Ergebnisse** (false negatives).

3.2 Qdrant-Suche: Semantische Suche nach „Schiff“

Die Qdrant-Suche liefert folgende Ergebnisse (aus dem Screenshot):

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

#	Produktname	Beschreibung (Auszug)	Quelle
1	Schaeffler STRASSE-3803 (Variante B)	Hochwertiges Produkt für den Einsatz in hohe Belastungen und lange Lebensdauer. Ausgelegt für precisione Anwendungen mit higher Belastung und 120°C.	Qdrant
2	Schaeffler SCHWIMMEN-9149 (Variante B)	Hochwertiges Produkt für den Einsatz. Ausgelegt für automotivee Anwendungen mit lowe Temperaturbereich von -20–120°C.	Qdrant
3	Schaeffler FLASCHE-7333 (Variante B)	Hochwertiges Produkt für den Einsatz in Belastungen und lange Lebensdauer. Ausgelegt für precisione Anwendungen mit higher Belastung.	Qdrant

Ergebnis: 3 Treffer (und mehr, je nach Top-K)

3.3 Warum findet Qdrant diese Ergebnisse?

Der Schlüssel liegt im Embedding-Modell (all-MiniLM-L6-v2). Dieses Sprachmodell wurde auf großen Textkorpora trainiert und hat dabei gelernt, semantische Beziehungen zwischen Wörtern zu erkennen.

Wenn der Suchbegriff „Schiff“ embeddet wird, erzeugt das Modell einen Vektor, der im semantischen Raum nahe an Konzepten liegt, die mit Schifffahrt, Transport, Wasser, Schwimmen, Straße (als Transportweg) und ähnlichen Begriffen verwandt sind.

Hinweis: Die folgenden Erklärungen sind vereinfachte Interpretationen. Das Modell arbeitet nicht mit expliziten Wortassoziationen, sondern mit gelernten statistischen Mustern im Vektorraum.

Konkret für die gefundenen Produkte:

STRASSE-3803: Das Wort „Straße“ ist semantisch mit Transport und Fortbewegung verbunden — genau wie „Schiff“ ein Fortbewegungsmittel ist. Das Embedding-Modell erkennt diese thematische Verwandtschaft.

SCHWIMMEN-9149: „Schwimmen“ hat eine direkte semantische Nähe zu „Schiff“ — beides findet im Wasser statt. Das Modell platziert diese Begriffe nahe beieinander im Vektorraum.

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

FLASCHE-7333: „Flasche“ hat eine schwächere, aber vorhandene Verbindung — Flaschenpost, Trinkwasser auf Schiffen oder einfach die Assoziation mit Flüssigkeiten.

Wichtig: Das Modell vergleicht nicht einzelne Wörter, sondern den gesamten Textblock aus product_to_document(). Der Kontext (Beschreibung, Kategorie, Tags, Anwendung) fließt in den Vektor ein und beeinflusst die Ähnlichkeitsberechnung.

3.4 Vergleichstabelle

Aspekt	SQL-Suche (LIKE)	Qdrant-Vektorsuche
Eingabe	SQL-Query muss vom Nutzer geschrieben werden	Natürliche Sprache (z.B. einfach „Schiff“)
Matching-Verfahren	Zeichenweiser Substring-Vergleich (Pattern Matching)	Kosinus-Ähnlichkeit im 384-dimensionalen Vektorraum
Ergebnis für „Schiff“	6 Treffer (false positives) („SCHIFF“ als zufälliges Produktkürzel)	Mehrere Treffer (STRASSE, SCHWIMMEN, FLASCHE, ...)
Synonyme / Semantik	Nicht erkannt. „Schiff“ ≠ „Schwimmen“	Automatisch erkannt. „Schiff“ ≈ „Schwimmen“
Ergebnis-Typ	Binär: Match oder kein Match	Kontinuierlicher Score (0.0–1.0)
Ranking	Kein automatisches Ranking (nur via ORDER BY möglich)	Automatisch nach Ähnlichkeits-Score sortiert
Durchsuchte Felder	Nur die explizit in WHERE genannten Spalten	Alle Felder des Produkts (via product_to_document())
Vorwissen	Keins — rein syntaktisch	Sprachmodell mit trainiertem Weltwissen über Wortbedeutungen
SQL-Kenntnisse nötig?	Ja — Nutzer muss SQL beherrschen	Nein — natürliche Sprache genügt
Sicherheitsrisiko	SQL-Injection möglich → umfangreiche Validierung nötig	Kein SQL-Injection-Risiko, da kein User-SQL

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

4. Fazit

Das Beispiel „Schiff“ verdeutlicht den Kernunterschied:

SQL sucht nach Zeichen — Qdrant sucht nach Bedeutung.

Die SQL-Suche mit LIKE ist deterministisch und exakt: Sie findet genau das, was als Zeichenkette übereinstimmt, nicht mehr und nicht weniger. Das macht sie ideal für strukturierte Abfragen mit bekannten Werten (z.B. WHERE brand = 'Schaeffler' AND price < 100).

Die Qdrant-Vektorsuche hingegen nutzt ein vortrainiertes Sprachmodell, um die Bedeutung einer Suchanfrage zu erfassen. Sie findet Produkte, die semantisch verwandt sind, auch wenn kein einziges Wort übereinstimmt. Dies macht sie ideal für explorative Suchen, bei denen der Nutzer nicht genau weiß, wie das gesuchte Produkt heißt.

Zusammengefasst als Entscheidungshilfe:

Anwendungsfall	Empfohlene Suche
Exakte Filter (Preis, Marke, ID)	SQL
Aggregationen (COUNT, SUM, AVG)	SQL
Joins über mehrere Tabellen	SQL
Explorative Suche („Schiff“, „robust“)	Qdrant
Synonym-Erkennung benötigt	Qdrant
Nutzer ohne SQL-Kenntnisse	Qdrant
Mehrsprachige Suche (Embedding-Modelle können mehrsprachige Vektoren erzeugen, z.B. mit multilingual-e5-large)	Qdrant

Im Projekt werden beide Ansätze sinnvoll kombiniert: Die SQL-Suche für präzise, strukturierte Abfragen und die Qdrant-Vektorsuche für semantische, natürlichsprachliche Suchen. Zusätzlich ermöglicht die RAG-Pipeline (Retrieval-Augmented Generation) die Kombination der Vektorsuche mit einem LLM für natürlichsprachliche Antworten.