scieneers
DRIVEN BY DATA

Vektordatenbank-Optimierung

Balance zwischen Speicher, Geschwindigkeit und Genauigkeit

25.04.2024 - Jan Höllmer

Vektordatenbanken

Vektordatenbanken ermöglichen das effiziente Speichern und Durchsuchen von Vektoren.

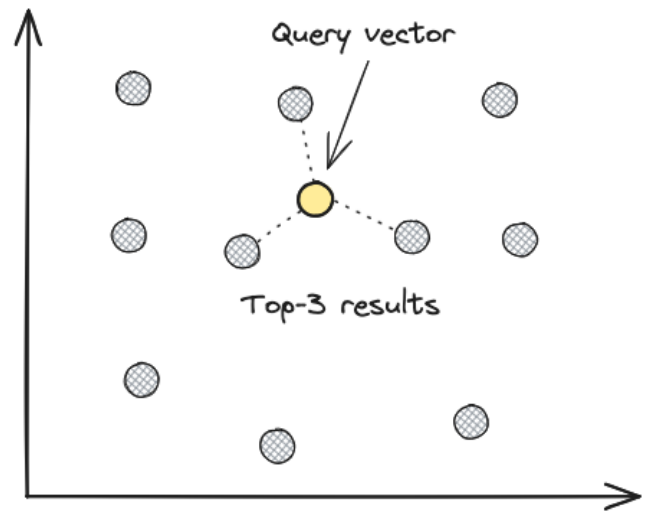
- Für **Retrieval-Augmented-Generation (RAG)** können für Textabschnitte **Embeddings** berechnet und in einer Vektordatenbank gespeichert werden.
- Vektoroperationen ermöglichen **semantische Suchen** und Ähnlichkeitsvergleiche.
- Viele Vektordatenbanken unterstützen zusätzlich Keyword-Suchen für **hybride Suchen**.

Metriken für Vektordatenbanken

Abfragen an Vektordanbanken können anhand der Genauigkeit, der Geschwindkeit und des dafür benötigten Speichers bewertet werden.

- **Genauigkeit**: Sind die vermeintlich ähnlichsten Vektoren auch wirklich die ähnlichsten?
- **Speicher**: Wie viel RAM und Festplattenspeicher wird benötigt?
- **Geschwindigkeit**: Wie schnell ist das Ergebnis verfügbar?

Es gibt einen **Trade-Off** zwischen Genauigkeit, Speicher und Geschwindigkeit.



Bildquelle: LanceDB - Vector Search

Welche Stellschrauben gibt es?

Vektordatenbanken bieten verschiedene Einstellungsmöglichkeiten, um den Trade-Off zwischen Genauigkeit, Speicher und Geschwindigkeit zu beeinflussen

- **Dimensionalität** bzw. **Precision** der Vektoren
- **Speicherort** der Vektoren, des Payloads und des Index
- **Index**: Konfiguration des Index z.B. Hierarchical Navigable Small World Graph (HNSW), Flat IndexIVFADC, Approximate Nearest Neighbors Oh Yeah (ANNOY)

Matryoshka Representation Learning (MRL)

MRL ermöglicht als spezielle Trainingsmethode unterschiedliche Output-Dimensionen für Embedding-Modelle.

Die neusten OpenAI Modelle (text-embedding-3-small & text-embedding-3-large) nutzen MRL.

	ada v2	text-embedding-3-small		text-embedding-3-large		
Embedding Size	1536	512	1536	256	1024	3072
Average MTEB score	61.0	61.6	62.3	62.0	64.1	64.6

Niedrige Embedding-Dimensionen senken den Speicherbedarf und erhöhen die Geschwindigkeit, jedoch auf Kosten der Genauigkeit.

Quellen: OpenAI Blog, MRL Paper

Vector Quantization

Vector Quantization (VQ) ist ein (externes) Verfahren zur Reduzierung der Precision einzelner Vektorkomponenten.

Anders als bei MRL wird hier nicht die Dimensionalität, sondern die Precision der einzelnen Dimensionen verringert (z. B. Verringerung Speicher-Bits). Dies spart Speicherplatz und erhöht die Geschwindigkeit. Abhängig von der verwendeten Quantisierungsmethode sinkt jedoch die Genauigkeit (drastisch).

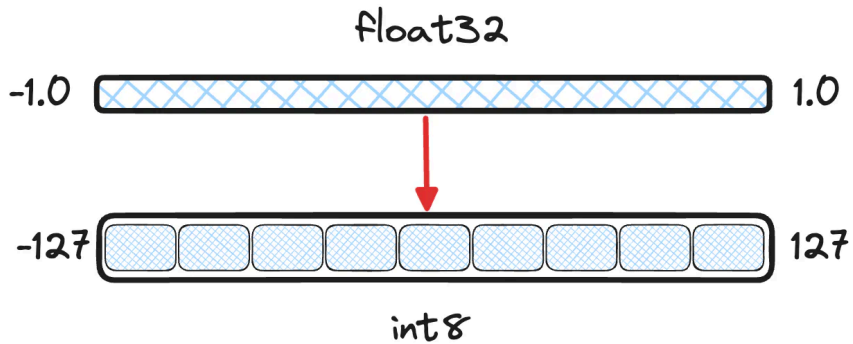
Quantization Method	Genauigkeit	Geschwindigkeit	Kompression
Scalar	0.99	bis zu x2	4
Product	0.7	0.5	bis zu 64
Binary	0.95*	bis zu x40	32

*für kompatible Modelle (z.B. ada v2)

Quelle: Qdrant - Quantization

Scalar Quantization & Binary Quantization

Angenommen, einzelne Vektorkomponenten werden mit 32-Bit Floats repräsentiert. **Scalar Quantization** konvertiert die Vektorkomponenten in 8-Bit Integers (Faktor 4).

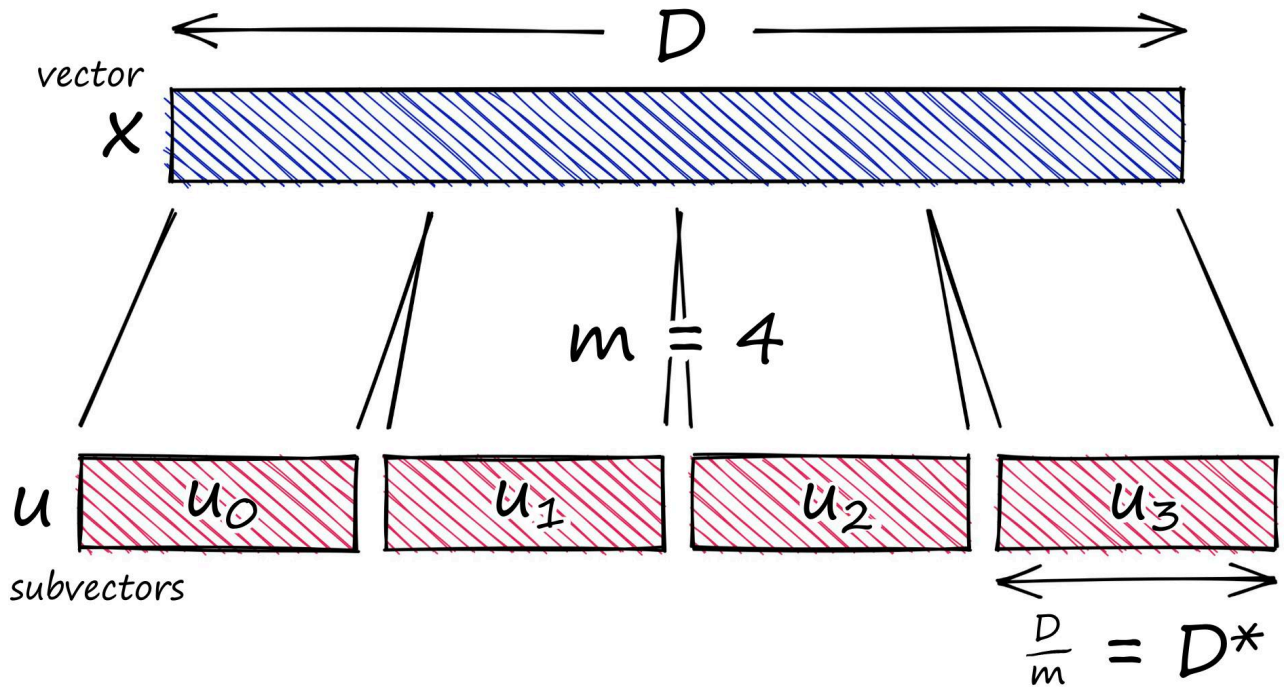


Bei der **Binary Quantization** werden die Vektorkomponenten in binäre Werte konvertiert. Das Embedding-Modell muss dabei eine Normalverteilung der Vektorkomponenten aufweisen, damit die Genauigkeit nicht zu stark sinkt.

Bildquelle: elastic - Scalar quantization 101

Product Quantization 1/3

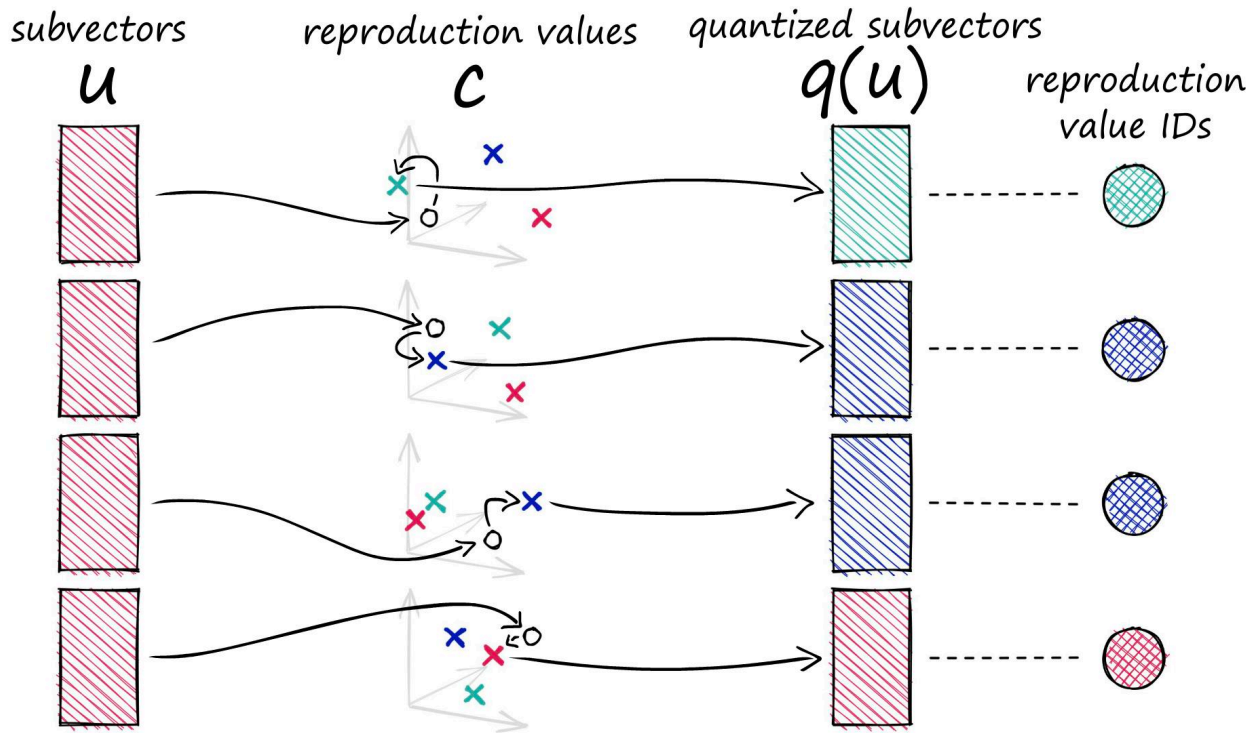
Zunächst wird jeder Vektor in gleich große Subvektoren unterteilt.



Bildquelle: Pinecone - Product Quantization

Product Quantization 2/3

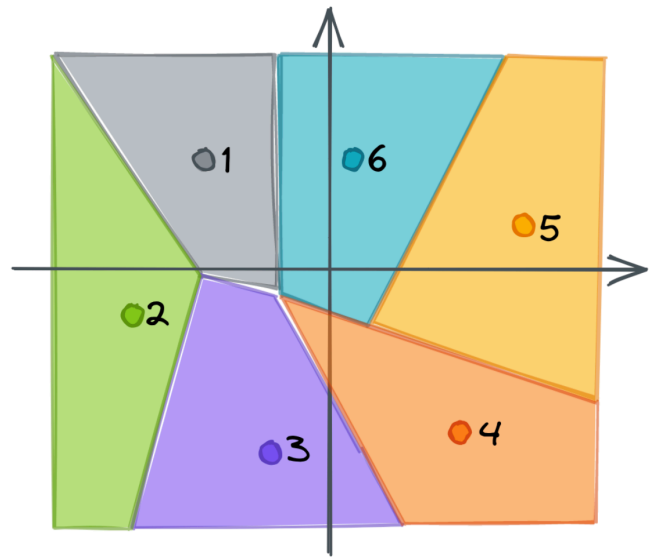
Jedem Subvektor wird die ID (reproduction value) eines Zentroids zugewiesen (ähnlich zu Clustering).



Bildquelle: Pinecone - Product Quantization

Product Quantization 3/3

- Für jede Subgruppe an Vektoren wird ein eigenes Clustering durchgeführt.
- Die meisten Vektordatenbanken (z.B. Qdrant oder Weaviate) verwenden K-means mit $K = 256$ (entspricht einem Byte/Int8).
- Die Suche erfolgt auf den reproduction values.
- Reduziert man bspw. einen Vektor mit 128 Dimensionen und Float32 ($182 * 32 = 4096$) auf 8 Dimensionen mit Int8 ($8 * 8 = 64$), so ergibt sich eine Kompression von 64.



Clustering mit $k = 6$.

Bildquelle: Qdrant - Product Quantization

Quantization mit Qdrant 1/2

Für Qdrant kann die Quantization auf Collection-Ebene konfiguriert werden.

```python
from qdrant_client import QdrantClient, models

client = QdrantClient(**kwargs)

client.create_collection(
  quantization_config=models.ProductQuantization(
    product=models.ProductQuantizationConfig(
      compression=models.CompressionRatio.X16,
      always_ram=True, # store quantized vectors in RAM
    ),
  ),
  **kwargs
)
```

Quantisierte Vektoren können in RAM oder Speicher gespeichert werden.

Quantization mit Qdrant 2/2

Rescoring bei der Suche kann die Genauigkeit erhöhen.

```python
from qdrant_client import QdrantClient, models

client = QdrantClient(**kwargs)

client.search(
  search_params=models.SearchParams(
    quantization=models.QuantizationSearchParams(
      rescore=True,
      oversampling=2.0, # oversampling factor for rescoring
    ),
  ),
  **kwargs
)
```

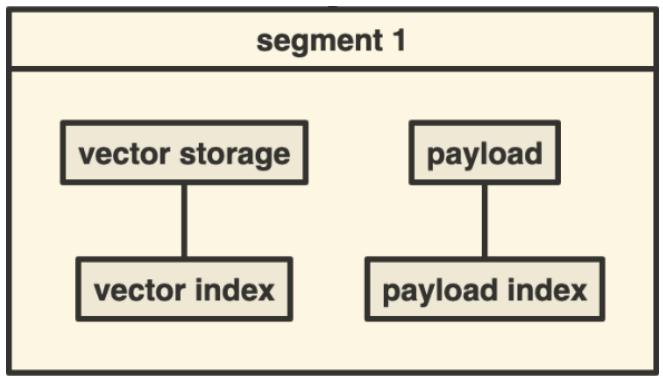
- **Rescoring**: Die Suche auf den quantisierten Vektoren wird anschließend durch die originalen Vektoren verfeinert.
- **Oversampling**-Faktor: Wie viele Vektoren im ersten Schritt (Suche mit quantisierten Vektoren) ausgewählt werden.

Qdrant - Segmente

Alle Daten in Qdrant sind in individuellen Segmenten gespeichert.

Ein Segment beinhaltet:

- Vektor (entspricht Embedding)
- Vektorindex
- Payload (z.B. Inhalt, Metadaten)
- Payloadindex



Bildquelle: Qdrant - Optimizer

Qdrant - Vector Storage

Vektoren können entweder im RAM oder auf der Festplatte gespeichert werden.

Vector Storage kann global oder auf Collection-Ebene konfiguriert werden.

```yaml
storage:
    optimizers:
        memmap_threshold_kb: 1        # maximum size of vectors to store in RAM per segment
        indexing_threshold_kb: 20000  # maximum size of vectors allowed for plain index
```

- **memmap_threshold_kb**: Maximale Größe von Vektoren, die pro Segment im RAM gespeichert werden. Größere Vektoren werden als Memmap-Datei gespeichert.
- **indexing_threshold_kb**: Wenn alle Vektoren diesen Schwellenwert überschreiten, wird ein Index erstellt.
- Vektoren können auch per **on_disk** Parameter manuell in den Hauptspeicher geschoben werden.

Qdrant - Payload Storage

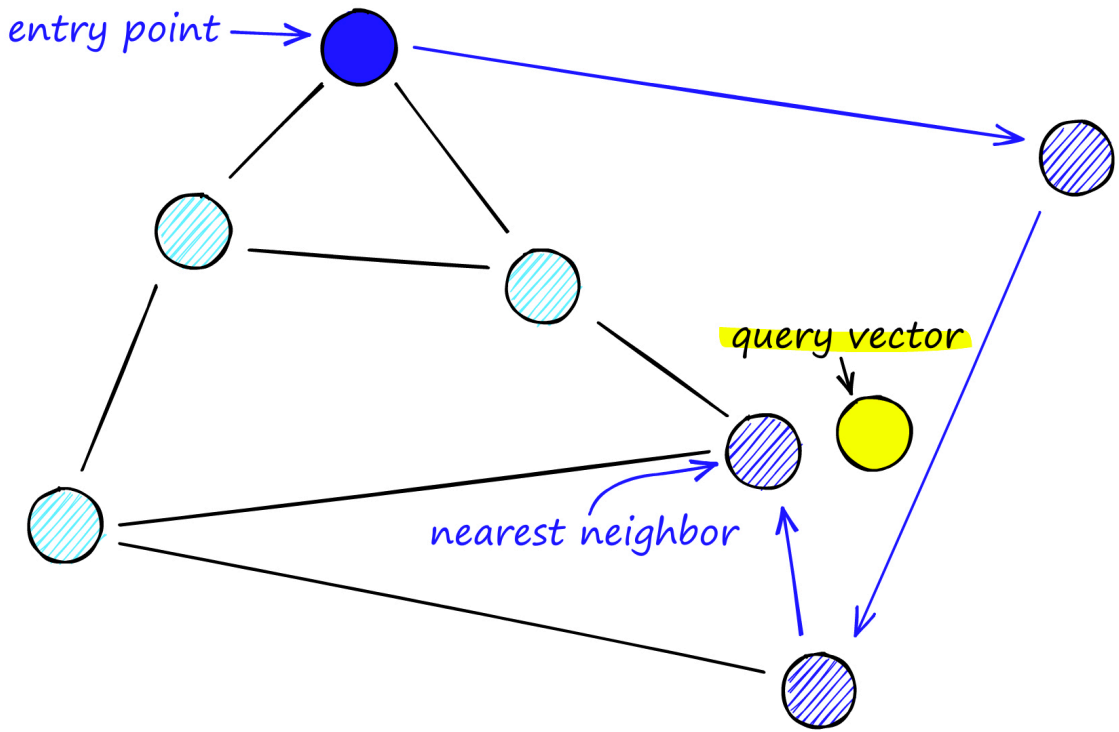
Ähnlich zu Vektoren kann auch der Payload im RAM oder auf der Festplatte gespeichert werden.

```python
from qdrant_client import QdrantClient, models

client = QdrantClient(**kwargs)

client.create_collection(
  on_disk_payload=True,     # store payload on disk
  **kwargs
)
```

- Große Payloads können auf der Festplatte gespeichert werden, um den RAM zu entlasten. Zugriff auf Payloads ist dementsprechend langsamer.
- Die Verwendung von SSDs kann die Zugriffszeiten verbessern.
- Qdrant verwendet für ondisk Payloads RocksDB.

HNSW - Allgemein 1/2

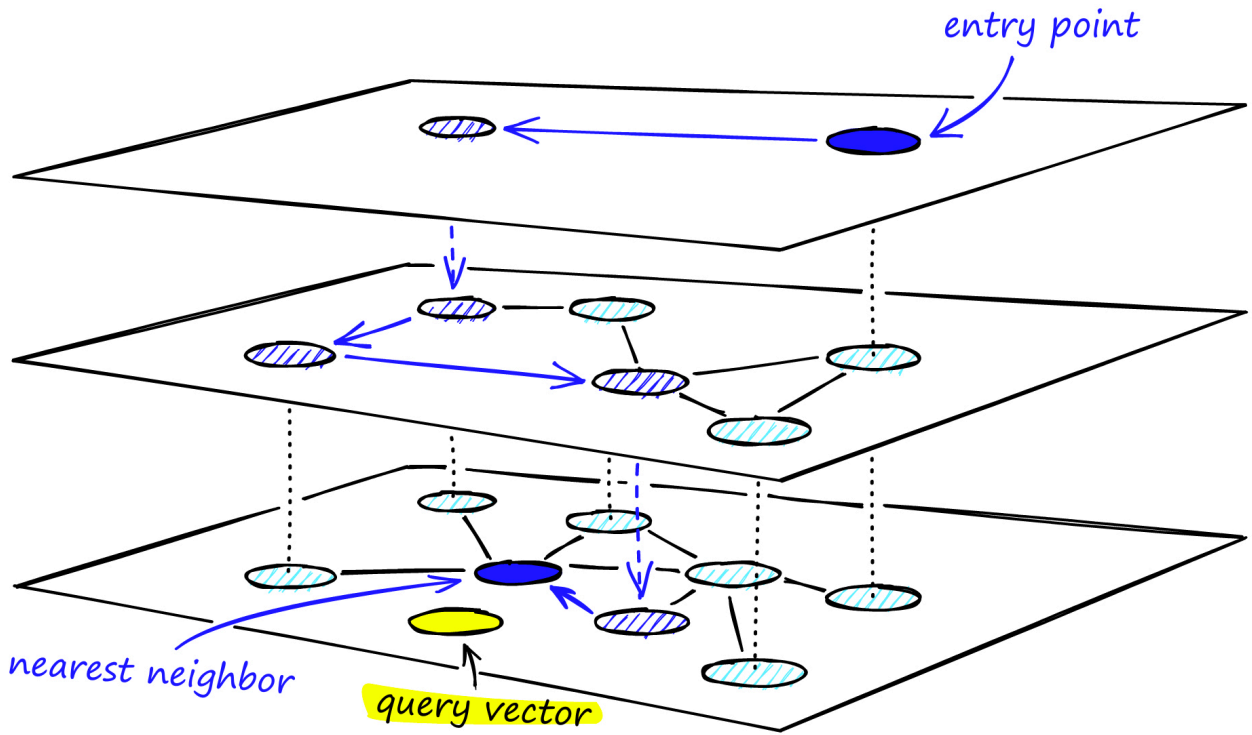
Hierarchical Navigable Small World (HNSW) basiert auf Navigable Small World (NSW) Graphen.



Bildquelle: Pinecone - Hierarchical Navigable Small Worlds (HNSW)

HNSW - Allgemein 2/2

HNSW basiert auf hierarchischen NSW Ebenen, welche die Suche nach ähnlichen Vektoren beschleunigen.



Bildquelle: Pinecone - Hierarchical Navigable Small Worlds (HNSW)

HNSW - Parameter

```
storage:
    hnsw_index:
        m: 16             # number of neighbors for each node in the graph
        ef_construct: 100 # number of neighbors to consider during index construction
```

- **m**: Höheres m führt zu höherer Genauigkeit und Speicherbedarf, aber langsamerer Indexierung.
- **ef_construct**: Höheres ef_construct führt zu höherer Genauigkeit, aber langsamerer Indexierung.
- Bei der Suche nach den k ähnlichsten Vektoren kann der **ef** Parameter gewählt werden:

$$k \leq ef \leq size(dataset)$$

Höheres **ef** resultiert in höherer Genauigkeit, aber langsamerer Suche.

HNSW - Qdrant Konfiguration

HNSW Index kann im RAM oder Hauptspeicher gespeichert werden.

```python
from qdrant_client import QdrantClient, models

client = QdrantClient(**kwargs)

client.create_collection(
  hnsw_config=models.HnswConfigDiff(on_disk=True),  # store HNSW index on disk
  **kwargs
)
```

- Es wird weniger RAM benötigt, aber die Suche ist langsamer.

Fazit

- Matryoshka Representation Learning (MRL) und Quanization verringern den RAM-Bedarf und die Genauigkeit, aber erhöhen die Geschwindigkeit.
  - MRL und Product Quantization reduzieren die Dimensionalität der Vektoren.
  - Quantization allgemein verringert die Precision der einzelnen Vektordimensionen.
- Das Verschieben von Vektoren / Payload / Index in den Hauptspeicher verringert die Geschwindigkeit, aber auch den RAM-Bedarf.
- Index-Parameter können die Genauigkeit auf Kosten der Geschwindigkeit beeinflussen.

```python
from qdrant_client import QdrantClient, models

client = QdrantClient(**kwargs)

client.create_collection(
  quantization_config=models.ProductQuantization(
    product=models.ProductQuantizationConfig(
      compression=models.CompressionRatio.X16,
      always_ram=True,
    ),
  ),
  on_disk_payload=True,
  optimizers_config=models.OptimizersConfigDiff(
    memmap_threshold=20000
  ),
  hnsw_config=models.HnswConfigDiff(on_disk=True),
  **kwargs
)
```

Kontakt

Ich freue mich über einen Austausch mit Euch.

Jan Höllmer

jan.hoellmer@scieneers.de

Mobil: +49 151 551 52 562

scieneers
DRIVEN BY DATA

Danke fürs Zuhören!