Suche gefunden: Text-Suche ohne Overengineering

Stefan Keller

Fr. 26. Juni 2026

OST Campus Rapperswil (Switzerland)

© Slides CC-BY 4.0 unless otherwise stated

Über mich

- Professor für Data Engineering und GISTech an der FH OST
- Schwerpunkte
  - GISTech
  - Databases
  - OpenStreetMap
- … vor allem als Open Source und Open Data
- «Data Scout»

Agenda

- Lexikalische Suche mit BM25
- BM25 direkt in PostgreSQL
- Semantische Text-Suche mit Embeddings und Cosinus-Metrik
- PostgreSQL-Erweiterungen
- Hybride Text-Suche
- Ausblick LLM in der DB

Ziele

- Relevantere Suchergebnisse und besserer Recall bei der Text-Suche
- Aufzeigen, dass Hybride (Text-)Suche auch mit PostgreSQL geht mittels pg_search und pgvector
  - Hybride Text-Suche = Kombination lexikalischer Treffsicherheit mit semantischer Vektorsuche
- Use Cases: Robuste Suche für Objekte, Ortsnamen und Adressen z.B. einer Webkarte

Overengineering bei der Suche

- Klassischer Stack (mit External Search Engine)
  - Architektur: PostgreSQL als transaktionale Hauptdatenbank + Elasticsearch/Solr als redundanter Suchindex
  - Datenfluss: Komplexe Synchronisations-Pipelines (Logstash, Debezium, Kafka)
  - Konsistenz: Synchronisations-Verzögerungen & Gefahr von Zeitversatz und Dateninkonsistenzen (Eventual Consistency)
  - Betrieb: Hoher RAM- und Wartungs-Overhead (JVM-Instanzen, Cluster-Management)
- Die schlankere Alternative (all-in-the-dbms)
  - Architektur: PostgreSQL als Single Source of Truth
  - Datenfluss: Echtzeit-Indexierung direkt in der Datenbank (via Extensions)
  - Konsistenz: 100% transaktionale ACID-Garantie ohne Zeitversatz
  - Betrieb: Minimaler Overhead, einheitliches Backup- und Rechtemanagement

Postgres FTS vs. BM25

- Volltextsuche (Full Text Search, FTS):
  - Text-Suche in PostgreSQL mit Datentypen tsvector und tsquery, ts_rank-Funktion und GIN/GiST-Index
  - Integriert, robust, gut für Keyword-Suche, mehrsprachig (konfigurierbar)
  - Grenzen: Ranking-Qualität und Top-k-Performance je nach Workload
- BM25:
  - Relevanzbasiertes Ranking per PostgreSQL-Extension
  - Vorteile: Bessere Trefferqualität als FTS und schnellere Top-k-Suche je nach Workload
  - Grenzen: Die Grenzen lexikalischer gegenüber semantischer Suche

Volltextsuche Stand (1)

- “Volltextsuche in Echtzeitdaten mit pg_search” von Marco Hugentobler, Vortrag an FOSSGIS 2026
- Problem
  - Volltextsuche ist stark bei exakten Namen, Token, Hausnummern und Tippfehlerstrategien wie Trigramm oder Fuzzy Search.
  - Sie versteht aber keine Bedeutung: Synonyme, umschriebene Suchintentionen oder fachliche Ähnlichkeit sind schwierig.
  - Semantische Suche ist wiederum schwächer bei harten Fakten wie “Rüti”, “Bahnhofstrasse 15” oder IDs.
- Lösung
  - pg_search liefert mit BM25, Inverted Index, Operatoren, Boosting und Fuzzy-Funktionen den präzisen lexikalischen Suchkern innerhalb von PostgreSQL.
  - Ein semantischer Layer ergänzt Treffer, die nicht über dieselben Wörter, aber über ähnliche Bedeutung gefunden werden.
  - Das Ranking kombiniert beide Welten: Textscore, semantische Nähe, Objektwichtigkeit, Aktualität.
- Fazit
  - Hybrid Text Search bedeutet: exakt finden, was exakt gesucht wird, und zusätzlich verstehen, was gemeint ist.
  - Für Web-GIS ist das wertvoll, weil Ortsnamen, Adressen, POIs und Nutzersprache gleichzeitig präzise und mehrdeutig sein können.
- Quelle: https://blog.sourcepole.ch/assets/2026/volltestsuche_in_echtzeit_mit_pg_search.pdf

Volltextsuche Stand (2)

- "Hybrid search with PostgreSQL and pgvector“, Post von Jonathan Katz, September 16, 2024
- Kombiniert Vektorsuche mit PostgreSQL Full-Text-Search
- Rankings werden per Reciprocal Rank Fusion (RRF) zusammengeführt
- Beispiel nutzt Produkttabelle mit 50.000 Datensätzen, Embeddings sowie GIN- und HNSW-Indizes
- Fazit: Hybrid-Suche verbessert im Beispiel das Ranking deutlich
- Einschränkung: Künstliches Dataset, daher keine belastbare Aussage zu Performance oder Qualität
- https://jkatz05.com/post/postgres/hybrid-search-postgres-pgvector/

Volltextsuche Stand (3)

- "Semantic Search and Information Retrieval with Transformers", Tutorial von Richard Hightower, Juli 2025
- Inhalt:
  - Von Keyword-Matching zu Bedeutungsverständnis
  - Transformer Embeddings erfassen Kontext und Nutzerintention
  - Vector Databases / FAISS / pgvector ermöglichen skalierbare semantische Suche
  - Hybrid Search kombiniert semantische Suche mit BM25/Full-Text-Search
  - RAG nutzt Suchergebnisse als Kontext für LLM-Antworten
  - Qualität wird über F1, Mean Reciprocal Rank (MRR) und Normalized Discounted Cumulative Gain (NDCG) bewertet
- Quelle: https://medium.com/@richardhightower/semantic-search-and-information-retrieval-with-transformers-rag-fundamentals-15f62073a95a

BM25-Rankingschema

Verbessert das klassische TF-IDF-Schema durch bessere

- Term-Sättigung und
- Dokumentenlängen-Normierung.

$$\text{Score}_{D,Q} = \sum_{q_i \in Q} \text{IDF}(q_i) \cdot \frac{\text{tf}(q_i, D) \cdot (k_1 + 1)}{\text{tf}(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

$f(q_i, D)$ **(Termfrequenz):** Die Häufigkeit des Suchbegriffs q_i im Dokument D. Ein häufigeres Vorkommen steigert die Relevanz, allerdings nicht linear, sondern gedämpft durch den Parameter k_1.

$\frac{|D|}{\text{avgdl}}$ **(Relative Dokumentenlänge):** Das Verhältnis der Länge des spezifischen Dokuments $|D|$ zur durchschnittlichen Länge (avgdl) aller Dokumente im gesamten Index.

k_1 **(TF-Sättigung — Standard: 1.2):** Regelt die Sättigung der Termfrequenz. Dieser Wert bestimmt, ab wann eine zusätzliche Nennung desselben Suchbegriffs kaum noch zu einem höheren Score führt. Ein höherer Wert verschiebt diese Sättigungsgrenze nach oben.

b **(Längen-Normalisierung — Standard: 0.75):** Steuert die Strafgewichtung für lange Dokumente. Je näher b bei 1 liegt, desto stärke werden lange Dokumente abgewertet, wenn sie den Suchbegriff nicht in einer entsprechend hohen Dichte enthalten. Bei $b = 0$ wird die Dokumentenlänge komplett ignoriert.

BM25 nativ in PostgreSQL: pg_search

- PostgreSQL-Erweiterung in Rust
- Teil des ParadeDB-Ökosystems
- Integriert die Such-Engine Tantivy nativ in Postgres.
- inspiriert von Apache Lucene
- Github: https://github.com/paradedb/paradedb/blob/main/pg_search/README.md

BM25 nativ in PostgreSQL: pg_textsearch

- PostgreSQL-Erweiterung in C und PL/pgSQL
- Nutzt BM25-Ranking mit konfigurierbaren Parametern wie `k1` und `b`
- Einfache SQL-Syntax: `ORDER BY content <@> 'search terms'`
- Unterstützt Sprachkonfigurationen wie Englisch, Deutsch und Französisch
- Ermöglicht JSONB-, Multi-Column-, Expression- und Partial-Indexes
- Optimiert für schnelle Top-k-Suchen durch «Block-Max-WAND»-Algo.
- PostgreSQL 17 und 18 („production ready“), Postgres OSS licensed
- GitHub: https://github.com/timescale/pg_textsearch

Volltextsuche und Indexierung mit pg_search

1. BM25-Index über die Textspalten legen

```
CREATE INDEX idx_doc_search ON documents
USING pg_search (title, description);
```

2. Abfrage mit lexikalischem Ranking ausführen

```
SELECT id, title, description, pg_search.score(id) AS bm25_score
FROM documents
WHERE pg_search.search(
    query => 'Ausdehnung Ozonloch'
)
ORDER BY bm25_score DESC
LIMIT 5;
```

Semantische Vektorsuche: pgvector

- Problem lexikalischer Suche:
  - Beispiel-Query: "Erhöhtes Krebsrisiko durch Sonneneinstrahlung"
  - Findet keine Synonyme (z.B. ‘Schwimmbad' vs. ‘Swimmingpool’)
  - Versteht keinen “Kontext”
- Lösung:
  - Repräsentation von Texten als hochdimensionale Vektoren (Embeddings), generiert durch vortrainierte Transformer-Modelle
- PostgreSQL-Erweiterung pgvector
  - Ermöglicht das Speichern und hocheffiziente Durchsuchen dieser Vektoren direkt in PostgreSQL.
  - HNSW (Hierarchical Navigable Small World) Index
  - IVFFlat Index für Schnelle Nächste-Nachbarn-Suchen (ANN)

Kosinus-Ähnlichkeit und Vektor-Indexierung

- Tabelle mit Vektorspalte definieren (z.B. 384 Dimensionen)

```
ALTER TABLE documents ADD COLUMN embedding vector(384);
```

- HNSW-Index für Kosinus-Ähnlichkeit erstellen

```
CREATE INDEX idx_docs_vector_hnsw ON documents
USING hnsw (embedding vector_cosine_ops);
```

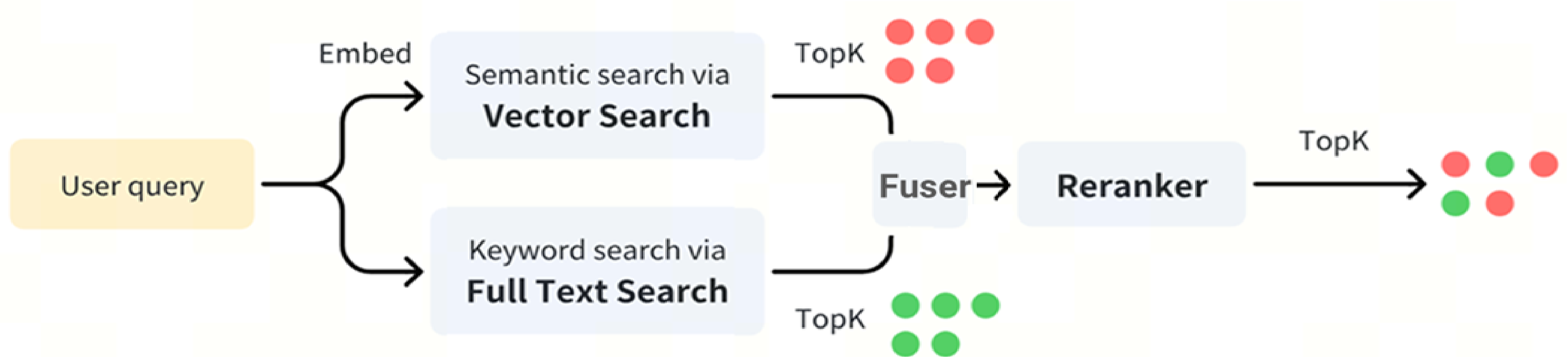
- Semantische Suche via Kosinus-Distanz (<=>)

```
SELECT id, title, (1 - (embedding <=> :query_embedding))
    AS cosine_similarity
FROM documents
ORDER BY embedding <=> :query_embedding
LIMIT 5;
```

Hybrid Search: Man kombiniere

- Die Präzision lexikalischer Suche: Exakte Treffer für Eigennamen, IDs, Artikelnummern, SKUs und Fehler-Codes
- Mit Recall semantischer Suche: Thematische Abdeckung durch abstrakte Konzepte, Umschreibungen und Synonyme
- Konkret
  - BM25 findet präzise Treffer bei konkreten Suchbegriffen
  - pgvector erweitert die Suche um semantisch ähnliche Inhalte
- Fusion:
  - Merging beider Trefferlisten zu einem gemeinsamen Ranking

Die Hybrid-Sucharchitektur



Die Hybrid-Sucharchitektur

- User Query / Suchanfrage wird parallel an zwei Verarbeitungs-Pfade übergeben.
- Pfad A (lexikalisch):
  - pg_search (BM25) -> Index Scan -> Top-K Lexical Results
- Pfad B (semantisch):
  - Embedding-Generierung -> pgvector -> Top-K Semantic Results
- Fusions-Pipeline via SQL:
  - Berechnung der Merge-Funktion und Zusammenführung.
- Finales Ranking:
  - Ausgabe der Top-K hybrid optimierten Suchresultate ohne Overhead.

Hybrid Search – Fusion (1)

- Reciprocal Rank Fusion (RRF)
  - Bewertet Dokumente unabhängig von ihren absoluten Scores rein basierend auf ihrer Position (Rank) in den jeweiligen Trefferlisten:
  - $RRF(d) = \sum [1 / (k + r_m(d))]$
    - m: Die Menge der Suchsysteme (hier: BM25 und Vektorsuche).
    - $r_m(d)$: Der Rang des Dokuments d im System m (1-basiert).
    - k Konstante: Glättet Einfluss von vorderen Rängen gegenüber hinteren Plätzen (Default 60)
  - Vorteil: keine Normalisierung nötig
  - Nachteil: nicht so «intelligent»

Hybrid Search – Fusion (2)

- Weighted Linear Combination (WLC)
  - Methode, bei der mehrere Roh-Scores mit Gewichten multipliziert und anschliessend addiert
  - Score_hybrid = α * Score_BM25 + (1 - α) * Score_Vector
- Normalized Discounted Cumulative Gain (NDCG)
  - Bewertet geordnete (und vorher genormte) Ergebnisse unter Berücksichtigung mehrstufiger Relevanz und penalisiert späte relevante Treffer
- Vorteil: «intelligenter»
- Nachteil Beide haben das Problem der Normalisierung

Kombinierte RRF-Query in einer Abfrage

```
WITH lexical_search AS (
    SELECT id, ROW_NUMBER() OVER (ORDER BY pg_search.score(id) DESC) AS rank
    FROM products WHERE pg_search.search(query => :user_query) LIMIT 100
),
semantic_search AS (
    SELECT id, ROW_NUMBER() OVER (ORDER BY embedding <=> :query_vector) AS rank
    FROM products ORDER BY embedding <=> :query_vector LIMIT 100
)
SELECT
    COALESCE(l.id, s.id) AS id,
    COALESCE(1.0 / (60 + l.rank), 0.0) +
    COALESCE(1.0 / (60 + s.rank), 0.0) AS rrf_score
FROM lexical_search l
FULL OUTER JOIN semantic_search s ON l.id = s.id
ORDER BY rrf_score DESC
LIMIT 10;
```

Metriken zur Evaluation: F1 Score & MAP

- Qualitätsmetriken für die Suchrelevanz
- Precision (Präzision):
  - Wie viele der zurückgelieferten Dokumente sind tatsächlich für den Nutzer relevant?
- Recall (Wiederauffindungsrate / Vollständigkeit):
  - Wie viele der insgesamt existierenden relevanten Dokumente wurden gefunden?
- F1-Score:
  - Das harmonische Mittel aus Precision und Recall.
  - Gut zur Messung der Balance zwischen Rauschen und Vollständigkeit.
- MAP (Mean Average Precision):
  - Berücksichtigt die exakte Reihenfolge der relevanten Treffer.
  - Relevante Ergebnisse weiter oben führen zu einem signifikant höheren MAP-Wert.

Evaluation

Search Text	Relevant Docs	Lexical Search		Semantic Search		Hybrid Search	
		Docs found	AP	Docs found	AP	Docs found	AP
Atembeschwerden Ozon	[9, 3, 4]	[9, 5, 3, 4]	0.94	[3, 7, 6, 9]	0.59	[3, 9, 5, 7]	0.72
Ozon-Alarm Schwimmbad	[4]	[4, 5, 3, 9]	1.0	[3, 4, 7, 6]	0.63	[4, 3, 9, 5]	1.0
Ausdehnung Ozonloch	[1, 6, 7, 10]	[1, 6, 10]	0.9	[3, 7, 6, 1]	0.59	[1, 6, 3, 7]	0.88
Geschichte Ozonloch Antarktis	[6, 1, 7, 10]	[10, 1, 6]	0.74	[6, 1, 10, 7]	1.0	[6, 10, 1, 7]	1.0
Ozonloch Ozon Loch	[1, 6, 7, 3]	[4, 10, 1, 8]	0.24	[3, 6, 7, 1]	0.86	[4, 6, 1, 3]	0.64
Ozonverlust Tonnen	[1, 6]	[1]	0.76	[3, 6, 7, 1]	0.57	[1, 3, 6, 7]	0.95
Ozonschicht UV - Bestrahlung	[7, 3, 6]	[7, 3, 10]	0.84	[7, 3, 10, 6]	0.98	[7, 3, 10, 6]	0.98
Ozon O3 Sauerstoff	[3]	[3, 5, 4, 9]	1.0	[3, 7, 6, 1]	1.0	[3, 9, 5, 7]	1.0
Ozonloch UV - Bestrahlung	[6, 10, 7, 1]	[10, 7, 3, 1]	0.58	[7, 3, 10, 6]	0.66	[7, 10, 3, 6]	0.7
Luftverschmutzung Städte	[9, 3]	[]	0.0	[9, 4, 3, 6]	0.95	[9, 4, 3, 6]	0.95
Gefahr für Badegäste im Sommer	[4, 9]	[4, 9]	1.0	[4, 9, 10, 3]	1.0	[4, 9, 10, 3]	1.0
Erhöhte Krebsgefahr durch Sonnenlicht	[7, 10]	[]	0.0	[10, 7, 9, 3]	1.0	[10, 7, 9, 3]	1.0
Abbau der Erdatmosphäre durch Industrieprodukte	[6, 9, 1]	[]	0.0	[2, 9, 7, 6]	0.48	[2, 9, 7, 6]	0.48
Mean Average Precision			**0.56**		**0.74**		**0.80**

Quelle: Eigene Fragen sowie Ozon-Kollektion mit 10 Dokumenten aus www.soekia.ch

Evaluation fortg.

Suchstrategie	MAP	F1-Score	Charakteristik & Verhalten
Lexikalisch (BM25 via pg_search)	0.68	0.65	Gut bei exakten Artikelnummern, Eigennamen. Schwächelt bei Umgangssprache.
Semantisch (pgvector HNSW)	0.72	0.70	Hoher Recall. Versteht Absichten und Synonyme gut, ungenau bei Fachwörtern.
Hybrid Search (RRF, k=60)	0.80	0.79	Etwas verbesserte Gesamtperformance. Kompensiert systematisch die Schwächen beider Systeme.

Ausblick: Erweiterungen pg_trgm, pg_tre

- pg_trgm – Trigramm-Suche
  - Seit PostgreSQL 9; GIN-Index unterstützt
  - Funktionsweise: Zerlegt Text in Dreier-Buchstabenkombinationen zur Ähnlichkeitsmessung.
  - Vorteil: Unterstützt performante, unscharfe Übereinstimmungen und beschleunigt LIKE/ILIKE via GiST- und GIN-Indizes.
- pg_tre – Approximate Regex Search
  - Seit PostgreSQL-18+
  - Index: eigener Access Method mit USING tre auf Textspalten
  - Feature: Regex + Levenshtein-Edit-Distanz in einer indexgestützten Suche
  - Stärken: Tippfehler, OCR-Fehler, Varianten und unvollständige Muster finden
  - Use Cases: Logs, SKUs, Seriennummern, Fehlercodes, technische IDs
  - Positionierung: fuzzy-lexikalische Ergänzung zu pg_trgm, FTS und pgvector

Ausblick: Embeddings direkt in PostgreSQL

- Kernkonzept:
- Inferenz direkt im Datenbank-Kern.
- Verhindert verwaiste oder asynchrone Vektoren, da Daten und Vektoren atomar konsistent in der DB bleiben (Keine Synchronisations-Gaps).
- Schliesst die Lücke im RAG-Pattern, indem generative KI-Features (Zusammenfassung, Klassifizierung, Extraktion) nativ via SQL auf operativen Tabellendaten ausgeführt werden – komplett unabhängig von externen APIs.

pg_infer

- Für Textklassifizierung und Zusammenfassungen. Nativer SQL-Zugriff auf Decoder-LLMs.
- Nativer Modell-Support: Direkte Ausführung lokaler Open-Source-Modelle auf der DB-Infrastruktur.
- Modell-Typ: Unterstützt ausschliesslich Decoder-LLMs (Generative Modelle wie Qwen, LLaMA, Gemma).
- Dateiformat: Benötigt Modelle zwingend im spezifischen vindex-Format, was die Auswahl extrem einschränkt: Stand Juni 2026 existieren auf HuggingFace nur sehr wenige kompatible vindex-Modelle. De facto läuft aktuell nur Qwen 2.5 0.5B stabil.
- Kein Werkzeug für klassische Encoder-Modelle und kein pgvector-Ersatz: generiert keine standardisierten Embedding-Vektoren. Seine interne Ähnlichkeitssuche (<~>) basiert rein auf Gate-Aktivierungen des Modells und erreicht nicht die Qualität dedizierter Embedding-Modelle (wie E5 / pg_onnx).

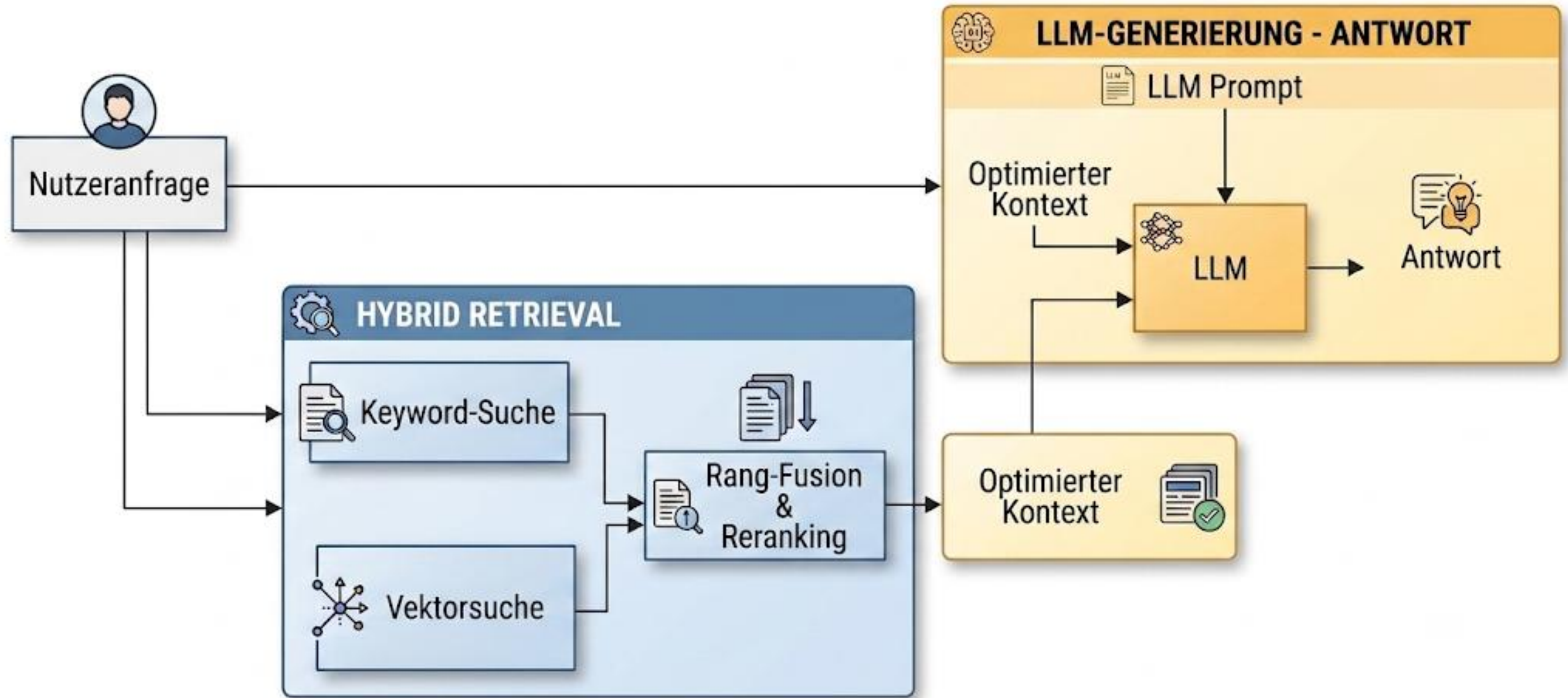
pg_onnx

- Autarke In-DB Embeddings. Integration der ONNX-Runtime als Worker für Echtzeit-Vektorisierung direkt bei INSERT/UPDATE
- ONNX-Runtime: Ausführung lokaler Encoder-Modelle (inkl. E5) im PG-Prozessraum
- E5: Modelle, die kurze Queries und lange Dokumenten/Abschnitten erwarten
- PostgreSQL Background Worker: Vektorgenerierung läuft asynchron
- Modell-Typ: Encoder-Modelle (z.B. BERT, E5, MiniLM) für Vektorgenerierung.
- Dateiformat: Standard .onnx. Modelle werden als BYTEA via SQL importiert.
- Optionale In-DB Tokenisierung: Standardmässig führt es reine Modell-Inferenz aus. Tokenizing/Encoding von Rohtext direkt in SQL erfordert das Laden eines Tokenizer-Modells (tokenizer.onnx) sowie onnxruntime-extensions.
- Optionaler GPU-Support: Kann optional für GPUs gebuildet (CUDA, NVIDIA) werden (C++ Boost, CMake, ONNX Runtime)

Retrieval-Augmented Generation (RAG)

1. Nutzeranfrage (Query): Der Benutzer stellt eine Frage.
2. Retrieval (Datenbank/en)
   - Bisher: Lexikalische Suche
   - Neu: Hybrid Retrieval: Anfrage wird parallel an die Keyword-Suche und die semantische Vektorsuche übergeben. Beide Ergebnisse werden fusioniert und ggf. re-ranked.
3. LLM: Die relevantesten Textabschnitte (Chunks) werden als Kontext zusammen mit der Frage an das LLM weitergeleitet.
4. Antwort (Generation): Das LLM generiert die Antwort basierend auf den Fakten aus dem Retrieval.

Hybride Suche vs. RAG



Fazit

- Stack-Vereinfachung:
  - Durch die Extensions pg_search und pgvector wird eine zusätzliche Elasticsearch-Infrastruktur für viele Use Cases komplett hinfällig.
- Hybrid ist König:
  - Die Fusion beider Welten (BM25 + Vektorsuche) liefert nachweislich bessere Relevanz-Ergebnisse (MAP: 0.80) als Einzelsysteme.
- Datenintegrität:
  - Keine Synchronisationspipelines, kein zeitlicher Versatz, 100% transaktionale ACID-Garantie direkt auf dem primären Datensatz.
- RAG vs. Agentische Systeme:
  - Die Schritte in RAG sind linear vordefiniert und oft ein Werkzeug innerhalb eines Agenten — der Agent entscheidet, wann er die Wissensbasis durchsucht, wann er eine API aufruft, und wann er fertig ist.