Elasticsearch-Quality Full Text Search in Postgres with Tantivy

Philippe Noël

Outline

- Current support for search in Postgres
- What is missing and why it needs to be better
- How pg_search is built to solve these limitations
- What pg_search can be used for (hybrid, full-text, faceting, etc.)

Useful Jargon

- **Tokenization**: splitting text into searchable chunks
- **Stemming**: reducing words to their root form
- **Inverted Index**: data structure used for efficient full text search
- **Faceting/aggregations**: computing metrics/buckets over FTS results
- **Elastic DSL**: domain-specific query language used by Elastic for FTS

Who am I?

- Philippe Noel, CEO of ParadeDB
- Originally from Rivière-du-Loup, Québec
- Previously worked on browser security and product at Microsoft Azure
- My Postgres Life interview: https://postgresql.life/post/philippe_noel/

What is ParadeDB?

- Elasticsearch alternative built on Postgres
- Packaged as two Postgres extensions
  - pg_search: Full text search with BM25
  - pg_analytics: Read data lakes (e.g. S3) and table formats (e.g. Iceberg)
- Built in Rust

Why use ParadeDB?

- Users migrate from Elastic to ParadeDB for
  - Data reliability (Transaction safe search)
  - Data freshness & operational simplicity (No ETL)
  - No schema changes or denormalization
- “Just use Postgres”

Who is ParadeDB?

- Ming Ying
- Neil Hansen
- Eric Ridge
- Myself (hi!)




What is Full Text Search (FTS)?

- Query documents by the presence of specific keywords or phrases
- Can be simple or very complex
- Two components: indexing and querying
    - Indexing: Preprocessing documents for rapid searching later
    - Querying: Searching the index to retrieve some information

Full Text Search vs Vector Search

- Also known as similarity search
- Is a complement to, **not** a substitute for, full text search
- Matches documents by semantic meaning, **not** specific keywords
- `pgvector` is a Postgres extension for vector search

Full Text Search in Postgres

- Three main tools to do FTS in Postgres:
  - LIKE operator
  - ts_vector + GIN index
  - pg_trgm

LIKE Operator

- `column_name LIKE pattern` syntax
- e.g. `SELECT * FROM users WHERE name LIKE 'John%'`
- Limitations:
  - Slow performance over large datasets
  - Very limited FTS functionality
  - No relevance scoring

ts_vector + GIN index

- The “real” implementation of full text search uses the `ts_vector` data type
- Stores the tokenized, stemmed representation of text
- Results can be ranked with the `ts_rank` function using TF-IDF
- GIN index constructs an inverted index over `ts_vector` columns, which improves query performance

pg_trgm

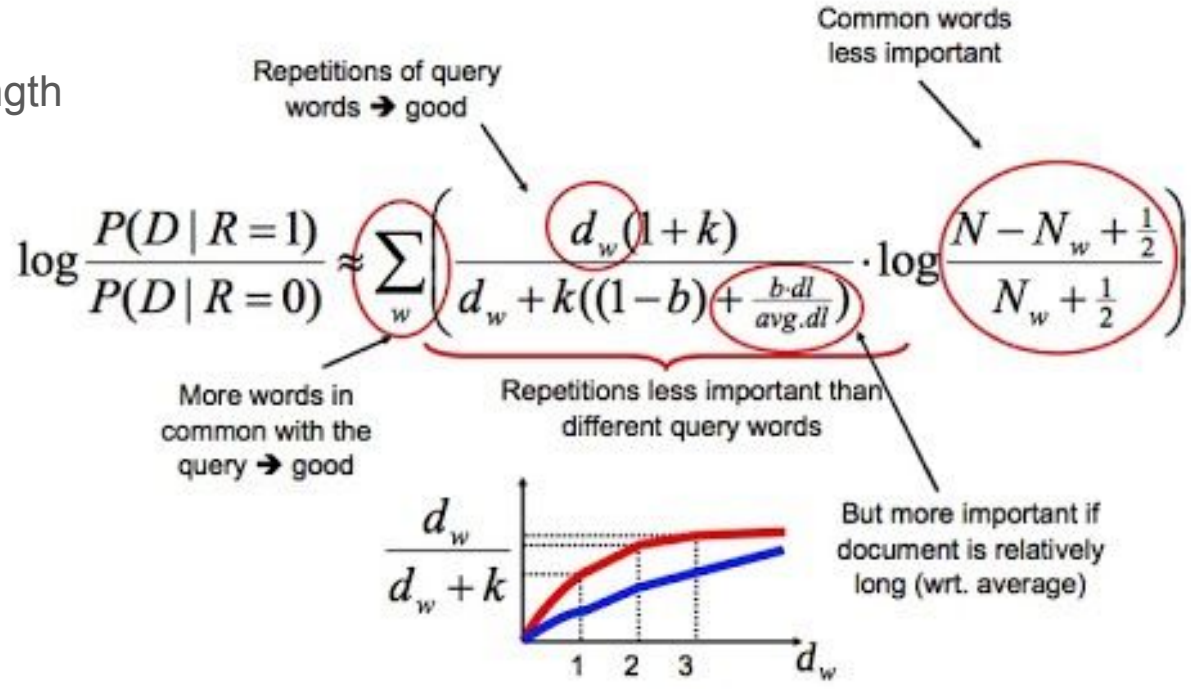
- A built-in Postgres extension that tokenizes text into tri-grams
- Tri-grams split text into groups of 3 characters. For instance, the tri-grams of “cheese” are “che”, “hee”, “ees”, and “ese”.
- Useful for basic autocomplete
- Would return for search like “chees”

What Postgres Full Text Search is Missing

- BM25 relevance
- More powerful tokenizers and token filters
- Elastic DSL-style, advanced FTS queries (i.e. relevance tuning, dismax, etc.)
- Fast facets and aggregations

What is BM25?

- Term saturation
- Factors in document length



Introducing pg_search

- An extension that brings Elasticsearch-quality FTS to Postgres
- Built in Rust with `pgrx`
- Uses a FTS library called Tantivy

What is Tantivy?

- Rust-based search engine library
- Heavily inspired by Lucene (the search library used by Elasticsearch)
- Support for fast FTS and faceting
- BM25 scoring by default
- Inverted index and columnar storage

How is pg_search Built?

- Four key components
  - Custom FTS operator `@@@`
  - Custom Postgres index
  - Query builder API
  - Custom scan

Custom FTS Operator

- `@@@` is our FTS operator that resolves a query against a text string, returning `true` if the text is a match
- Can be dropped into **any Postgres query**
- i.e. `SELECT * FROM mock_items WHERE id < 10 AND description @@@ 'keyboard'`
- Friendly to JOINs, ORDER BY, GROUP BY, etc.

Custom Index

- Running `@@@` on every row is slow – this is called a sequential scan
- Our custom index, the BM25 index, constructs an inverted index over the text field
- Works exactly like other built-in Postgres indexes (i.e. B-tree) for index construction, updates, vacuums, and scans
- One exception: the BM25 index is a **covering index**

Query Builder API

- Beyond simple text queries, queries can take the form of complex JSON objects
- The right-hand side of `@@@` can also accept JSON
- Our query builder functions make it easy to construct this JSON

```
SELECT * FROM mock_items WHERE id => paradedb.boolean(
  should => ARRAY[
    paradedb.boost(query => paradedb.parse('description:shoes'), boost => 2.0),
      paradedb.term(field => 'description', value => 'running')
    ]
  )
);
```

Custom Scan

- The Postgres custom scan API allows us to take control of other parts of the query beyond `WHERE…@@@`
- Enables three key use cases:
  - Predicate pushdown
  - BM25 scoring
  - Fast facets/aggregations

Predicate Pushdown

- Consider `SELECT * FROM mock_items WHERE description @@@ 'keyboard' AND rating < 5`
- Without a custom scan, Postgres will perform separate scans over `description` and `rating`, even if `rating` and `description` are in the BM25 index

BM25 Scoring

- Consider `SELECT * FROM mock_items WHERE description @@@ 'keyboard'`
- How do we return BM25 scores to the user?
- The custom scan can “project” a `score_bm25` column into the result

```
SELECT *, paradedb.score_bm25(id) AS score_bm25
FROM mock_items WHERE description @@@ 'keyboard'
ORDER BY score_bm25;
```

Fast Facets/Aggregations

- Consider SELECT COUNT(id), description FROM mock_items WHERE description @@@ 'keyboard' LIMIT 10
- If millions of results are found, COUNT(id) will be very slow
- Luckily, Tantivy has the concept of **fast fields**

Fast Fields

- Fields indexed as “fast” are stored in a column-oriented fashion
- A custom scan can return id to COUNT in batches (i.e. columns)
- Custom scans can also be parallelized
- **Result: a column-oriented, vectorized, parallelized faceting engine**

Use Cases

- Every software application needs search and analytics
  - Companies who want to stick with Postgres or migrate off Elastic
  - UPDATE-heavy workloads like e-commerce search
  - Faceted search for SaaS applications
  - Hybrid search for improving recall

Deployment

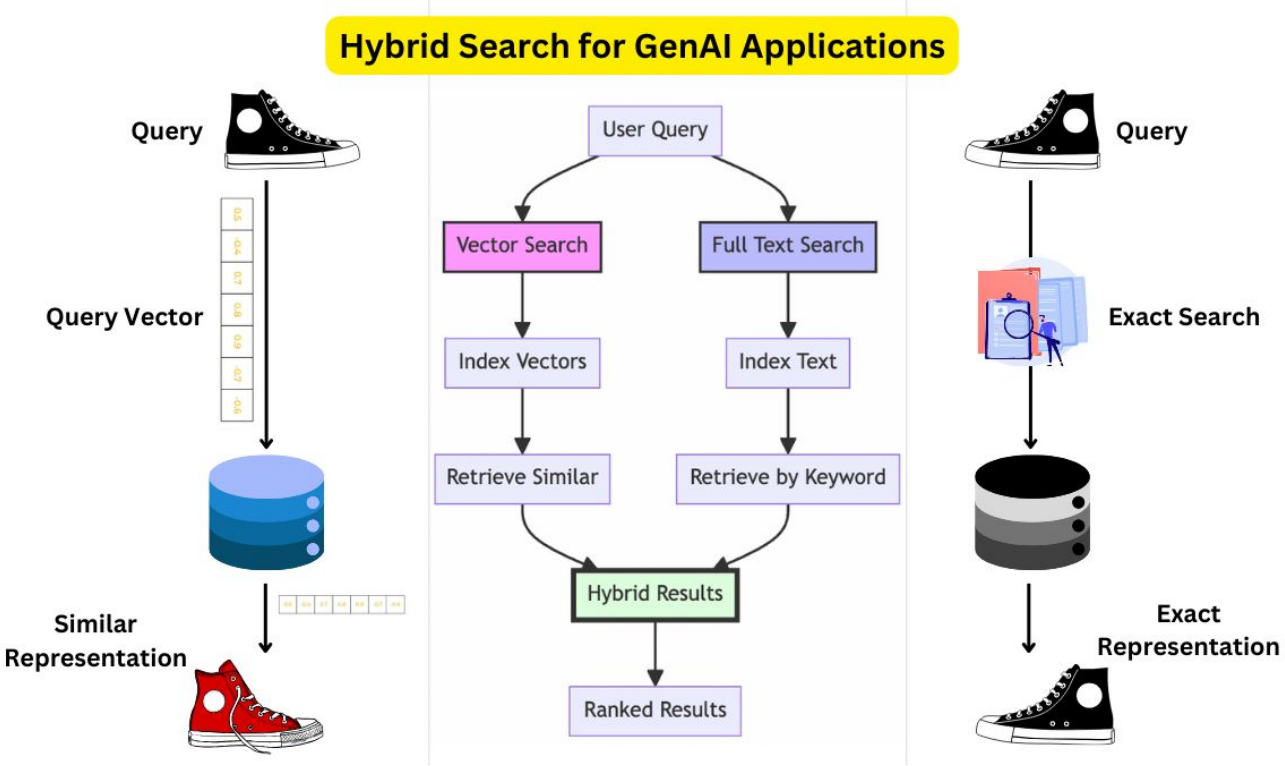
- ParadeDB pg_search integrates with:
  - AWS RDS/Aurora, GCP CloudSQL, etc.. via logical replication
  - CloudNativePG for self-hosted deployments
  - Ubicloud.com for a fully-managed solution

Thank You!

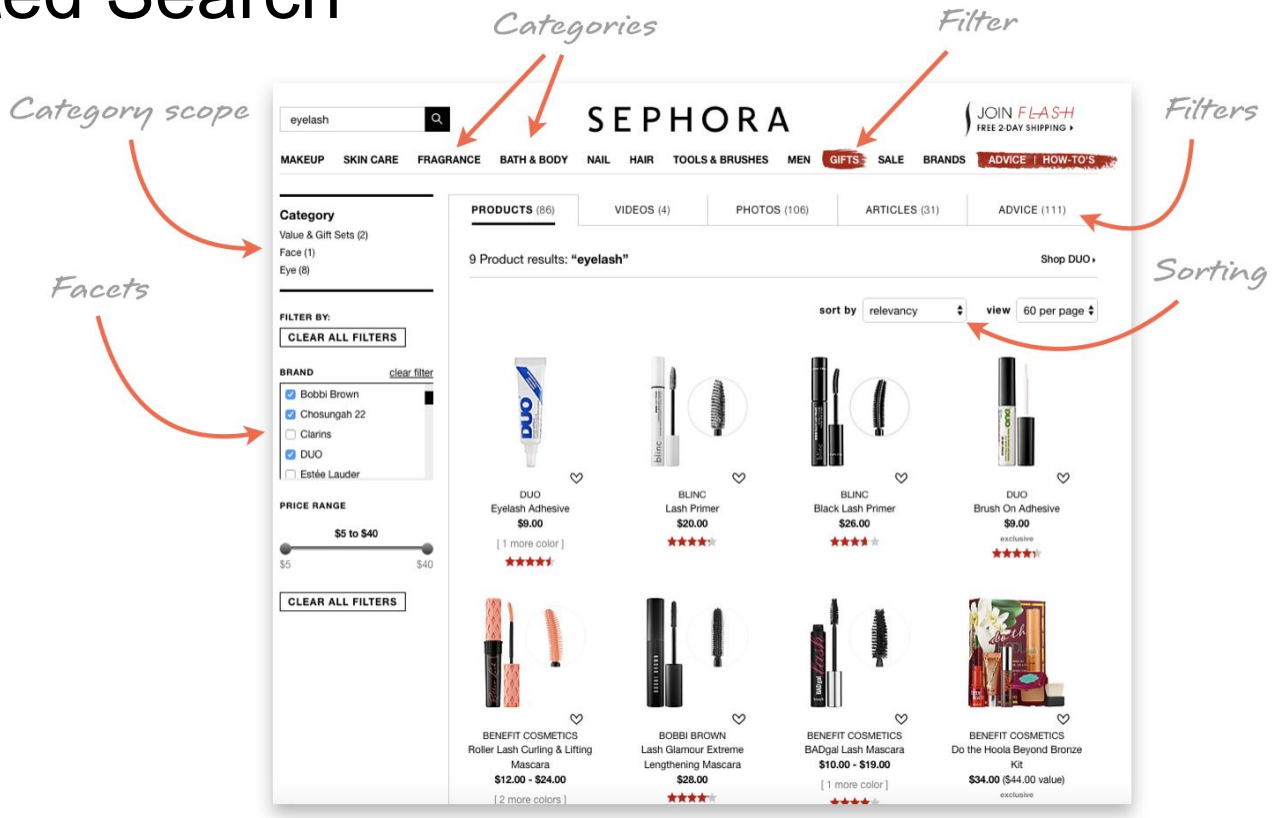
paradedb.com

Appendix

Hybrid Search



Faceted Search



Hierarchical Search

```
SELECT * FROM parts LIMIT 5;
 part_id | parent_part_id |     description
---------+----------------+---------------------
       1 |              0 | Chassis Assembly
       2 |              1 | Engine Block
       3 |              1 | Transmission System
       4 |              1 | Suspension System
       5 |              2 | Cylinder Head
(5 rows)
```