pg_search: Bringing Elasticsearch-Grade Search to PostgreSQL

Philippe Noël

Outline

1. What is “search” and why is it important?
2. Postgres’ existing search functionalities
3. What’s missing
4. How `pg_search` improves full-text search
5. How `pg_search` improves aggregates/facets
6. [In progress] How `pg_search` improves vector/hybrid search

Who am I?

- Philippe Noel, CEO of ParadeDB
- Originally from Rivière-du-Loup, Québec
- Studied CS @ Harvard, but never took a database class (oops)
- Previously worked on a browser startup and on Windows Server at Microsoft Azure
- My Postgres Life interview: https://postgresql.life/post/philippe_noel/

What is ParadeDB?

- Elasticsearch alternative built as a PostgreSQL extension
- Written in Rust (via `pgrx`)
- Users migrate from Elastic to ParadeDB for
  - Data reliability (Transaction safe search)
  - Data freshness & operational simplicity (No ETL)
  - No schema changes or denormalization
  - Faster reindexing in UPDATE-heavy scenarios
- “Just use Postgres”

Useful Jargon

- **Tokenization**: splitting text into searchable chunks
- **Stemming**: reducing words to their root form
- **Inverted Index**: data structure used for efficient search
- **Aggregation**: an analytical query that computes statistics over the entire dataset, e.g. counts, sums, averages
- **Facet**: an aggregation over the output of a search, i.e. “how many results match the word `foo`”
- **Elastic DSL**: domain-specific query language used by Elastic for writing search queries

What is Search?

- User-facing dashboards (e.g. filtering, sorting, bucketing)
  - Log search, security monitoring, etc.
- Search boxes / search UIs
  - Website search, docs search
- E-commerce platforms
  - Catalog search
- RAG / AI Agents

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`:** Tokenizing and storing text in an inverted index
  - **`Querying`:** Finding text in the index that matches a query

Full-Text Search in Postgres

- Three main tools to do FTS in Postgres:
  - `LIKE` operator
  - `ts_vector/ts_rank` + 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 typo/autocomplete-like matching
- Would return for search like “chees”

What is Vector Search?

- Also known as similarity or semantic search
- Is a complement to, **not** a substitute for, full-text search
- Matches documents by semantic meaning, **not** specific keywords
- `pgvector` is the default Postgres extension for vector search
- `SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;` syntax

pgvector

- A popular Postgres extension that enables storing and searching **vector embeddings** in `vector` columns
- Input text is fed to large language models to create embeddings, which encode the “semantic” meaning of the text
- Useful for synonym, similarity matching, recommender systems, and contextual understanding without exact keyword matches
- Performs poorly on inputs without semantic meaning (e.g. stock tickers, medical codes, proper nouns, etc.)

What Postgres 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
- Native hybrid search combining both FTS and vector search in a single index pass

Introducing pg_search

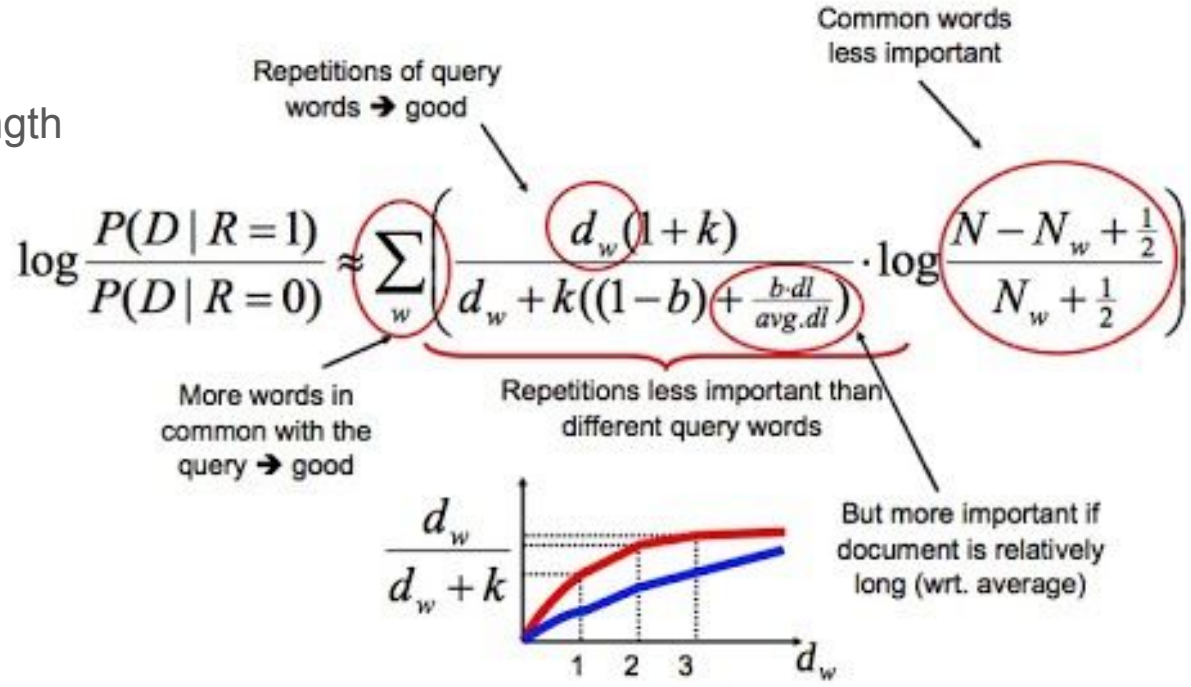
- Native FTS: simple but limited
- Elasticsearch: powerful but operationally heavy
- `pg_search`**: search-engine features inside PostgreSQL**
- One database, SQL-native workflow, stronger relevance
- Built in Rust with `pgrx`
- Uses a FTS library called **`Tantivy`**
- Uses an OLAP library called **`DataFusion`** (out of scope for this talk)

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

What is BM25?

- Term saturation
- Factors in document length



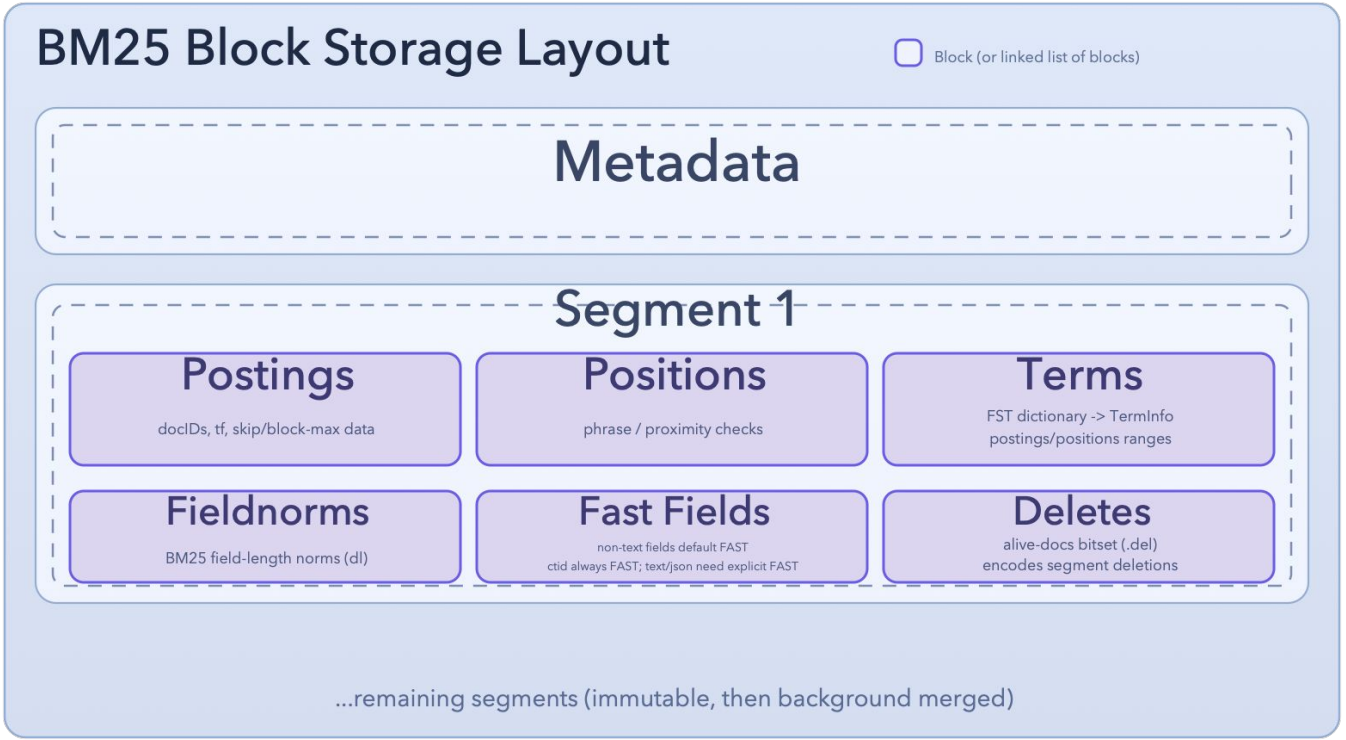
How is pg_search Built?

- Custom Postgres index access method (**`BM25`** index)
- Custom FTS operators (`|||`, **`&&&`**, **`###`**, **`===`**, `@@@`)
- Query builder which converts SQL statements to **`Tantivy queries`**
- Custom scan path and execution
- **Custom scan enables projection of a BM25 score into the target list**

Custom Index

- The BM25 **index is a native PostgreSQL index access method for search, built inside the database engine.**
- It is WAL-logged, **MVCC-aware**, and integrates with normal Postgres lifecycle operations (INSERT/UPDATE/DELETE, VACUUM, backup/replication).
- It combines **`inverted indexing`** for text with **`columnar storage`**, enabling efficient search, filtering, sorting, and aggregation from the index.
- It acts as a **`covering search index`**: include all fields used by search/filter/order/facets to maximize pushdown and minimize heap work.
- Created with standard CREATE INDEX … syntax

Custom Index



Custom Operators

- Custom full-text search operators for match, term, phrase, etc. (`|||`, `&&&`, `###`, `===`, `@@@`)
- Can be dropped into **any Postgres query**
- `SELECT * FROM mock_items WHERE id < 10 AND description ||| 'running shoes';`
- Friendly to JOINs, ORDER BY, GROUP BY, etc.
- Beyond simple text queries, queries can use `@@@` to define complex search queries

Custom Scans

- The Postgres custom scan API allows us to take control of other parts of the query beyond `WHERE … @@@`
- `pg_search` has 3 different custom scans (search, aggregates, JOINs)
- Enables 3 key use cases:
  - **Predicate pushdown**
  - **Scoring projection**
  - **Fast facets/aggregations**

Custom Scan: Predicate Pushdown

- We want all predicates (WHERE, ORDER BY, etc.) to be executed in a single index scan in Tantivy for maximum performance
- `EXPLAIN SELECT description, rating, category FROM mock_items WHERE description ||| 'running shoes' ORDER BY rating LIMIT 5;`

Custom Scan: Predicate Pushdown

Gets rewritten to:

```
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=10.00..10.02 rows=3 width=552)
   ->  Custom Scan (ParadeDB Base Scan) on mock_items  (cost=10.00..10.02 rows=3 width=552)
         Table: mock_items
         Index: search_idx
         Segment Count: 1
         Exec Method: TopKScanExecState
         Scores: false
            TopK Order By: rating asc
            TopK Limit: 5
         Tantivy Query: {"with_index":{"query":{"match":{
                          "field":"description",
                          "value":"running shoes",
                          "tokenizer":null,
                          "distance":null,
                          "transposition_cost_one":null,
                          "prefix":null,
                          "conjunction_mode":false}}}}
(10 rows)
```

Custom Scan: Scoring Projection

- How do we return BM25 scores to the user, since they're not in the table?
- The custom scan can "project" a `score` column into the result
- `SELECT description, pdb.score(id) FROM mock_items WHERE description ||| 'running shoes' AND rating > 2 ORDER BY score DESC LIMIT 5;`

```
 description       |  score
-------------------+-----------
Sleek running shoes |  6.817111
Generic shoes       |  3.8772602
White jogging shoes |  3.4849067
(3 rows)
```

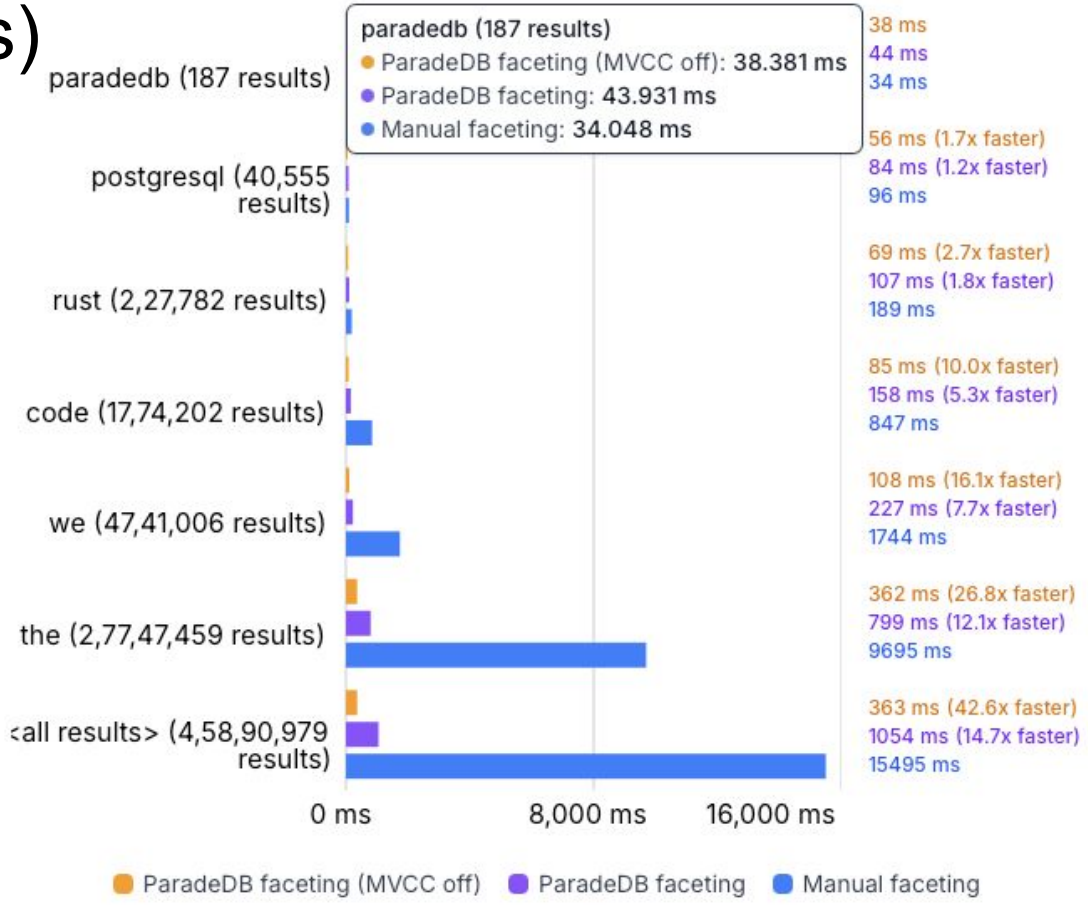
Custom Scan: Aggregates

- Consider `SELECT COUNT(id), description FROM mock_items WHERE description ||| 'keyboard' LIMIT 10`
- This is called a **facet**: Top K results along with the term bucket count in the same query
- If millions of results are found, COUNT(id) will be very slow
- Luckily, Tantivy has the concept of **columnar storage**
- Columnar storage enables us to process the index data in batches rather than on a per-row basis like regular Postgres would
- **Result: state-of-the-art performance for facets and general analytical queries**

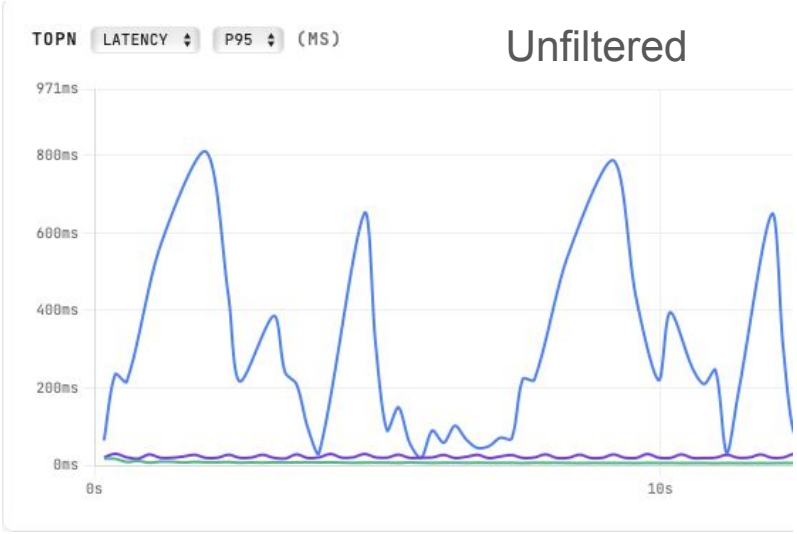
Performance (Facets)

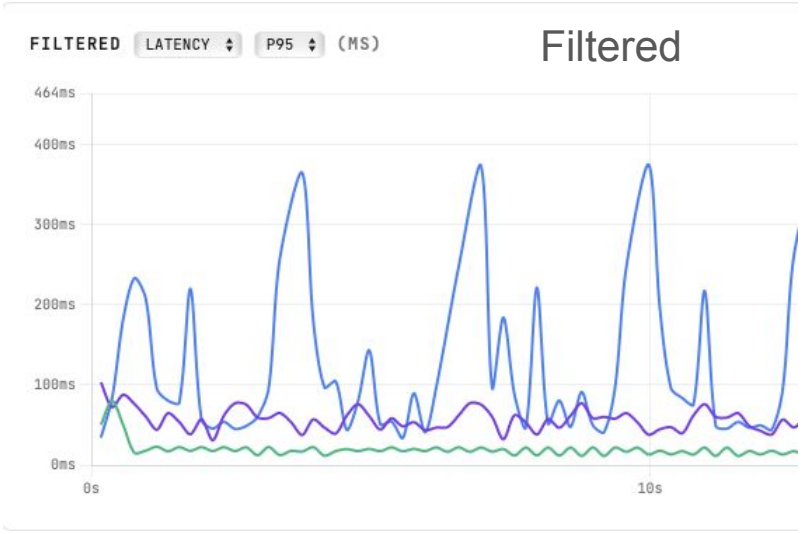
- **Dataset:** 46 million Hacker News posts comments

- **~15x faster with MVCC on**
  ~43x faster with MVCC off



Performance (Top K)






```
shared buffer = 2GB
table size    = 13GB
```

GIN looking up through the heap for ranking — `thrashes` the shared buffer

Looking Ahead

Vector Search

- Vectors are all the hype. So what about `pgvector`?
- It is possible to combine the scores of vector search queries with `pgvector` with `pg_search` full-text search queries
- This process is called **Reciprocal Rank Fusion**
- Two main issues:
  - **Performance:** `pgvector` performs poorly once the index doesn't fit in memory
  - **Recall:** filters destroy `pgvector`'s performance and recall

Vector Search

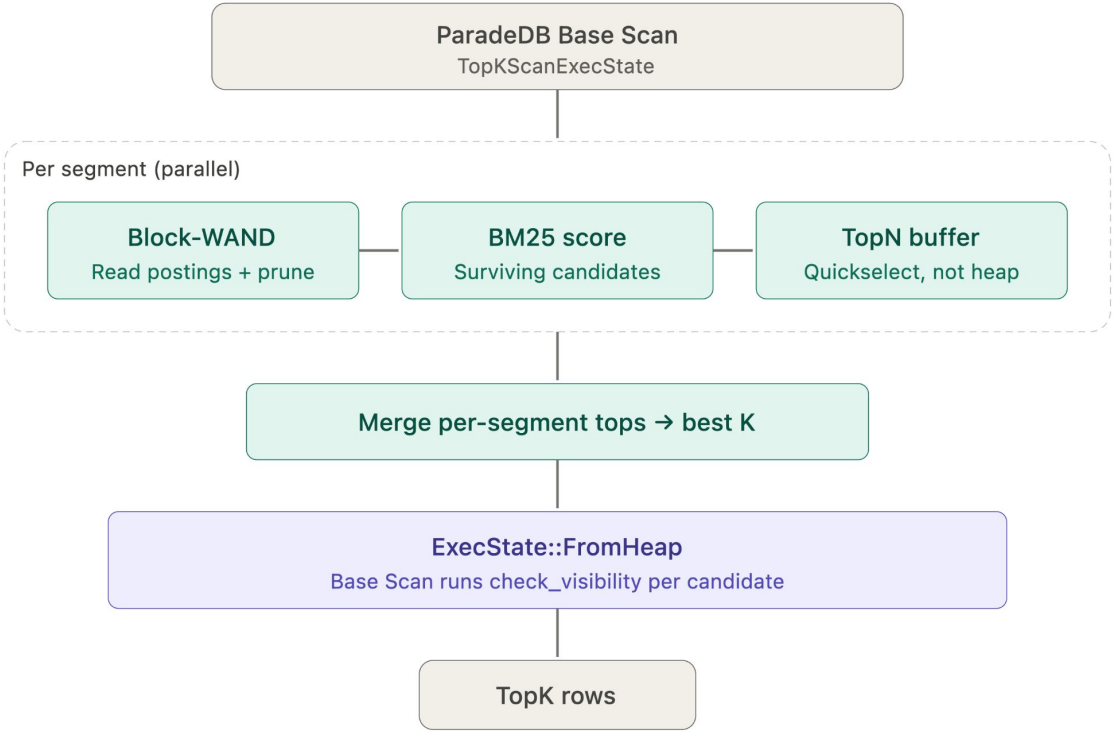
- If only we could apply vector search filters in the **same index scan** as metadata and full-text…
- **Solution:** Vector search natively integrated with Tantivy within the ParadeDB index
- **Result:** State-of-the-art vector and hybrid search in Postgres, competitive with dedicated vector databases and search engines like Elasticsearch
- This is work in progress, stay tuned for announcements over the next few weeks!

Thank You!

paradedb.com

Appendix

ParadeDB Base Scan



ParadeDB Aggregate Scan