Engineer

Full-text search

searching documents by their words, ranked by relevance

What it is

Full-text search finds the documents that contain the words of a query, in whatever form those words take, and can put the best matches first. It is also written full text search, without the hyphen. A document here is whatever the application treats as one unit of searching: an email, an article, a support ticket, or the title and body of one row put together.

What makes it different from a pattern match is the work done before anything is compared. The text is first broken into tokens: words, numbers, email addresses and other classes. Each token is then normalised into a lexeme, a form chosen so that variants of the same word come out alike. Upper case is folded to lower case, a stemmer strips endings such as the English plural, and stop words, the words so common that they carry no signal, are dropped. The result is stored together with the position of each lexeme, which later lets a ranking function prefer documents where the query words stand close together. The query goes through the same process, so a search for “payments” and a document that says “payment” meet in the middle.

PostgreSQL builds this into the database. Its `tsvector` type holds a document in the normalised form, `tsquery` holds a query, and the operator `@@` answers whether one matches the other. A text search configuration decides how normalisation happens: which parser splits the text and which dictionaries turn tokens into lexemes. The standard distribution ships configurations for many languages, among them `english`, `danish`, `norwegian` and `swedish`, and the psql command `\dF` lists the ones installed. SQLite offers the same idea through its FTS5 extension, where a virtual table holds the index and the `MATCH` operator queries it.

Use the concept in a decision

Put “What is full-text search?” to work in a real decision. We can help relate the concept to the system you are building or buying. Finding the queries and the missing indexes behind a system that has got heavier every year, and fixing the ones that matter.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. 30 days of electricity data in 180 ms — 180 ms to aggregate a month of meter readings

30 days of electricity data in 180 ms Get in touch →

Why it matters

The PostgreSQL documentation explains what the older operators lack. `LIKE`, `ILIKE` and regular expressions have no linguistic support, so a search for “satisfy” misses a document that says “satisfies” unless someone spells out every derived form. They have no ranking, so a thousand matches come back in no useful order. And they cannot use an ordinary index for this kind of search, so every query reads every document. A pattern such as `%cat%` also matches “category” and “concatenate”, which is noise rather than recall.

People type words into a search box and expect to find the documents about those words. On a contract archive, a help desk or a product catalogue, full-text search is the difference between a search box that people trust and one they stop using after the third miss. The ranking matters as much as the matching: someone looking for “late payment” wants the reminder letter at the top, not the hundred invoices that mention payment once.

Having it inside the database you already run has a practical side. The search index lives in the same transaction as the data, so a row is searchable the moment it is committed, and a deleted row disappears from results at the same moment. Filters on dates, customers or permissions are ordinary `WHERE` clauses next to the text condition. A separate search server can do more, but it is a second system to fill and to keep in step with the first, and every gap between them shows up as a result that is missing or one that should be gone.

Full-text search also has a limit worth knowing from the start: it matches words, not meaning. A query for “car” does not find “automobile” unless a synonym dictionary maps one to the other. Vector search, which compares numerical representations of meaning, covers that case, and the two are often combined.

What it looks like in practice

Take a table of documents with a title, a body and the place each one came from. A stored generated column keeps the searchable form up to date on every insert and update, and a GIN index makes it fast to search:

```
CREATE TABLE documents (
    id         bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    title      text,
    body       text,
    source     text NOT NULL,
    search_vec tsvector GENERATED ALWAYS AS (
                   setweight(to_tsvector('english', coalesce(title, '')), 'A')
                || setweight(to_tsvector('english', coalesce(body, '')), 'D')
               ) STORED
);

CREATE INDEX documents_search_idx ON documents USING GIN (search_vec);
```

Three details carry the weight. The configuration is named, `'english'`, rather than left to the session's `default_text_search_config`; the two-argument form of `to_tsvector` is the one PostgreSQL can store and index. `coalesce` turns a missing title or body into an empty string, because a single NULL would otherwise make the whole document NULL. And `setweight` labels each lexeme with where it came from: A for the title, D for the body. The ranking functions use the weights {0.1, 0.2, 0.4, 1.0} for D, C, B and A unless told otherwise, so a query word in the title counts ten times as much as the same word in the body.

GIN is the index type the documentation prefers for text search: an inverted index with one entry per lexeme and a compressed list of the rows that contain it. GiST also works, but it is lossy and has to check the table row to rule out false matches.

The query takes the words a user typed, as they typed them:

```
SELECT id, title, source,
       ts_rank_cd(search_vec, query) AS rank
FROM   documents,
       websearch_to_tsquery('english', 'unpaid invoice -draft') AS query
```

```
WHERE  search_vec @@ query
ORDER  BY rank DESC
LIMIT  10;
```

`websearch_to_tsquery` reads input the way a web search box does. Plain words must all appear, in any form the English stemmer folds together, text in double quotes must appear as a phrase, the word “or” offers an alternative, and a leading minus excludes a word, so this query leaves out every document that mentions a draft. The function never raises a syntax error, which is why it is the safe choice for raw input. `ts_rank_cd` ranks by cover density, a method from a 1999 paper by Clarke, Cormack and Tudhope that also rewards query words standing close together. Its sibling `ts_rank` counts how often the words appear. Both accept a normalisation flag: 1, for instance, divides the rank by one plus the logarithm of the document’s length, so a long document does not win only because it is long.

The same need runs through the company’s work on company data for AI agents. The page describes old PDFs, scans and mail archives turned into searchable text with their source kept beside them, so they can be found rather than only stored. The `source` column above has that job: a match is worth more when the reader can open the original.

SQLite reaches the same result with less ceremony. A virtual table declared with `CREATE VIRTUAL TABLE notes USING fts5(title, body)` is queried with `WHERE notes MATCH 'invoice NOT draft'`, and `ORDER BY bm25(notes)` puts the best matches first: FTS5 returns better matches as numerically lower values, so the default ascending order is the right one. The default tokenizer is `unicode61`; the `porter` tokenizer adds stemming, which its documentation says is designed for English only.

Where it goes wrong

An index the query never touches. An expression index built on `to_tsvector('english', body)` serves only queries that repeat exactly that expression. A query written as `to_tsvector(body)`, relying on the default configuration, reads the whole table. Store the vector in a generated column, as above, and query the column; then the query cannot drift from the index.

One NULL that hides a document. `to_tsvector(NULL)` returns NULL, and concatenating a NULL title with a body gives NULL, so a row without a title silently drops out of every search. Wrap every field in `coalesce`.

Raw input in `to_tsquery`. `to_tsquery` expects its own operator syntax and is less forgiving of its input than the other conversion functions, so a user who types an unbalanced parenthesis or a stray ampersand gets an error page. Pass user input through `websearch_to_tsquery`, or `plainto_tsquery` when operators are not wanted at all.

The wrong language. Danish text indexed with the English configuration is stemmed by English rules and filtered by English stop words, so half the variants never meet. Pick the configuration for the language of the text. A collection in several languages can store the configuration name in a column of its own and index `to_tsvector(config_name, body)`, as long as the queries name the same column.

Ranking every match. Ranking reads the `tsvector` of each matching row, and the documentation warns that this can be I/O bound and slow. A one-word query that matches half the table makes the database rank half the table before `LIMIT` keeps ten. Narrow the candidates with ordinary conditions first, such as a date range or a customer, and keep the vector stored so it is not computed again for every row.

A rank shown as a percentage. The ranking functions use no information about the collection as a whole, so a rank cannot be turned into a fair “87% relevant”. The normalisation flag 32 maps every rank to a value between 0 and 1, but the documentation calls that a cosmetic change. Use the rank to order results and nothing else.

Highlights that carry markup. `ts_headline` returns an excerpt with the query words marked, which is just what a results page wants. PostgreSQL warns that its output is “not guaranteed to be safe for direct inclusion in web pages”. A document that holds HTML can carry a script into the page. Strip the markup from the input or run the output through an HTML sanitiser. The function also works on the original text, not the vector, so call it only for the rows on the page, after `LIMIT`, not for every match.

Documents too large for one vector. A `tsvector` must stay under 1 MB, and it keeps no more than 256 positions per lexeme. A whole book or a year of minutes in one row runs into both. Split long documents into sections or pages, which also gives each result a more precise place to point to.

Fragments and part numbers. Full-text search compares whole lexemes, so “inv” finds nothing about invoices and a part number such as `AB-1042-X` may be split by the parser into pieces. Keep an ordinary index on the columns people search by exact value, or, in SQLite, use the `trigram` tokenizer, which treats every run of three characters as a token and so supports substring matching.

An SQLite index that drifts from its table. An FTS5 table declared with `content=` reads its text from another table, and the documentation leaves it to the user to keep the two consistent. When they drift, queries return rows that seem to contradict each other. Create the insert, update and delete triggers the documentation shows, in the same migration as the virtual table.

Further reading

- Introduction to full text search — PostgreSQL’s definition of documents, tokens, lexemes and configurations, and what `LIKE` and regular expressions lack.
- Tables and Indexes — searching a table, expression indexes against a stored generated column, and why the configuration must be named.
- Controlling Text Search — weights, `websearch_to_tsquery`, `ts_rank` and `ts_rank_cd`, normalisation flags and the `ts_headline` warning.
- Preferred Index Types for Text Search — GIN and GiST compared, including why GiST needs to recheck rows.
- Limitations — the size limits on lexemes, vectors, positions and queries.
- SQLite FTS5 Extension — virtual tables, the `MATCH` operator, the built-in tokenizers, `bm25()` and external content tables.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is full-text search?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/full-text-search/