ChatGPT

Fast Global Search in PostgreSQL for 25M Employees

A performant global search over 25 million employee records requires a carefully designed schema, indexes, and query strategy. We propose an **Employees** table with core columns for `name`, `location`, and structured fields for multi-valued attributes (e.g. `skills`, `certifications`, `projects`). For example:

```
CREATE TABLE employees (
    id BIGSERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    location TEXT,
    skills TEXT[],             -- array of skill names
    certifications TEXT[],     -- array of certificate names
    projects JSONB,            -- JSON data for project details
    search_vector TSVECTOR     -- combined full-text search vector (generated)
);
```

Here, `search_vector` is a tsvector column aggregating all text-searchable fields. We can maintain it via a **generated column** or trigger. For example, PostgreSQL 12+ supports a *stored generated column* (avoiding manual triggers) [1]:

```
ALTER TABLE employees
  ADD COLUMN search_vector TSVECTOR
    GENERATED ALWAYS AS (
      to_tsvector('english',
        coalesce(name,'') || ' ' ||
        coalesce(array_to_string(skills,' '), '') || ' ' ||
        coalesce(array_to_string(certifications,' '), '') || ' ' ||
        coalesce(projects ->> 'description','')
      )
    ) STORED;
```

This expression concatenates and vectorizes `name`, all `skills` and `certifications` (joined as strings), and a JSON field (e.g. `projects->>'description'`). We use `coalesce` / `array_to_string` to handle NULLs [1]. Alternatively, built-in triggers like `tsvector_update_trigger` can auto-update the column [2]. Weighting can be applied with `setweight`: e.g., boosting `name` lexemes higher than others improves relevance [3].

Data model considerations: keep textual fields reasonably sized (e.g. TEXT or VARCHAR), and use arrays or JSONB for lists. JSONB offers flexibility for complex project data, with GIN indexing (see below) [4] [5]. Fixed attributes (name, location) use native columns. If strict normalization is needed, join tables can be used, but denormalized arrays/JSONB often simplify full-text indexing.

Indexing Strategy

Efficient indexes are key for sub-second search. We recommend:

- **GIN Full-Text Index:** Create a GIN index on the combined `search_vector` column to accelerate searches across all text fields. For example:

```
CREATE INDEX ON employees USING GIN(search_vector);
```

This leverages PostgreSQL's full-text engine, indexing each lexeme. As per the docs: `CREATE INDEX ... USING GIN (to_tsvector('english', body))` is the standard way to speed up text search [6]. A GIN index contains one entry per distinct lexeme, enabling fast lookups [7] [6].

- **GIN (pg_trgm) Indexes for LIKE:** For substring or "fuzzy" search on specific columns (like partial name/location), use the **pg_trgm** extension. Example:

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE INDEX ON employees USING GIN (name gin_trgm_ops);
```

This splits names into trigrams and indexes them [8]. It allows indexed searches on `name ILIKE '%substring%'`, as illustrated by the GIN trigram example in [61]. Similarly, a GIN index on `location gin_trgm_ops` helps prefix or wildcard location queries.

- **GIN on JSONB/Array:** Use GIN for multi-valued fields. For `skills` and `certifications` (TEXT[] arrays), a GIN index (default operator class) will index each element for containment queries:

```
CREATE INDEX ON employees USING GIN (skills);
CREATE INDEX ON employees USING GIN (certifications);
```

This lets queries like `skills @> ARRAY['Python']` use the index. For the JSONB `projects` column, a GIN index also applies. By default (`jsonb_ops`), it indexes all keys and values [9]. If most queries test containment of specific paths, use the `jsonb_path_ops` class for efficiency:

```
CREATE INDEX ON employees USING GIN (projects jsonb_path_ops);
```

This is recommended for key/value containment queries [5]. For example, `WHERE projects @> '{"role": "Manager"}'` can use the index.

- **B-tree Indexes:** A simple B-tree index on `location` (text) can speed exact-location filters (e.g. `WHERE location = 'Bangalore'`). Numeric or foreign-key fields (not shown) should have B-tree indexes as usual. But for full-text combos, the GIN indexes above handle most searches.

- **Materialized Views / Partial Indexes:** In some cases, precomputing or partially indexing frequent queries can help. For example, if many searches filter by a certain region or department, create a partial index on those filtered rows. Materialized views can aggregate text from related tables. However, with 25M rows, a well-tuned GIN index on `search_vector` is usually sufficient.

Search Function

We implement a PL/pgSQL function `search_employees(json_query JSONB)` that accepts JSON input and returns matching rows (as JSON or record). The function parses the JSON and constructs a tsquery and filters. For example:

```
CREATE OR REPLACE FUNCTION search_employees(params JSONB)
RETURNS TABLE (
    id BIGINT,
    name TEXT,
    location TEXT,
    skills TEXT[],
    certifications TEXT[],
    rank REAL
) AS $$
DECLARE
    q TEXT := COALESCE(params->>'query','');
    tsq TSQUERY;
BEGIN
    -- Build a tsquery from the input text (websearch syntax allows AND/OR, etc.)
    IF q <> '' THEN
        tsq := websearch_to_tsquery('english', q);
    ELSE
        tsq := NULL;
    END IF;

    RETURN QUERY
    SELECT
        e.id, e.name, e.location, e.skills, e.certifications,
        ts_rank(e.search_vector, tsq) AS rank
    FROM employees e
    WHERE (tsq IS NULL OR e.search_vector @@ tsq)
      -- Optionally apply field-specific filters from JSON:
      AND (params->>'location' IS NULL OR e.location ILIKE '%' || params->>'location' || '%')
      AND (params->'skills' IS NULL OR e.skills @> (params->'skills')::text[])
      -- Add more filters (certifications, projects) as needed
    ORDER BY rank DESC
    LIMIT COALESCE((params->>'limit')::INT, 50);
END;
$$ LANGUAGE plpgsql;
```

In this function, we use `websearch_to_tsquery` (or `plainto_tsquery`) to convert the user query string into a `tsquery` [10] [11]. The `@@` operator matches the precomputed `search_vector`. We

also demonstrate JSON-derived filters (e.g. matching `location` or array containment on `skills`). Results are ordered by `ts_rank` (relevance) and limited.

Example Search Scenarios

- **Full-text keyword search:** Find any employee with keyword "database" in *any* field.

```
SELECT id,name,location FROM employees
WHERE search_vector @@ plainto_tsquery('english', 'database');
```

This uses the GIN index on `search_vector` for a global keyword match [6].

- **Name search (prefix):** Find employees whose name begins with "Alice".

```
SELECT id,name FROM employees
WHERE name ILIKE 'Alice%';
```

A trigram GIN index on `name` makes even `ILIKE '%Alice%'` fast [8]. Alternatively, using full-text: `search_vector @@ to_tsquery('english','Alice:*')` to match prefix.

- **Skill search:** Find employees skilled in "Python".

```
SELECT id,name FROM employees
WHERE skills @> ARRAY['Python'];
```

This uses the GIN index on the `skills` array for containment. We could also do full-text: `@@ to_tsquery('english','Python')`.

- **Name + skill:** Find "John" who knows "Python":

```
SELECT id,name FROM employees
WHERE search_vector @@ websearch_to_tsquery('english', 'John & Python');
```

Here both terms must appear in the combined text vector. This utilizes the GIN FTS index to filter by both name and skill terms [6].

- **Skill + location:** Python developers in Mumbai:

```
SELECT id,name FROM employees
WHERE search_vector @@ websearch_to_tsquery('english', 'Python & Mumbai');
```

The vector contains both skill and location keywords. Optionally an explicit filter: `AND location ILIKE '%Mumbai%'`.

- **Certifications and projects:** AWS-certified employees on “Project X”:

```
SELECT id,name FROM employees
WHERE certifications @> ARRAY['AWS Certified']
  AND projects @> '{"project_name": "Project X"}'::jsonb;
```

This uses the GIN index on both the `certifications` array and the JSONB `projects` column (the latter via `jsonb_path_ops`) [5] [9].

- **Multi-term full-text:** Any field containing “machine learning AI”:

```
SELECT id,name FROM employees
WHERE search_vector @@ to_tsquery('english','machine & learning & ai');
```

Each word is required. The query planner will Bitmap-AND the index entries for each lexeme.

- **Partial substring (trigram):** Last name containing “son”:

```
SELECT id,name FROM employees
WHERE name ILIKE '%son%'
ORDER BY similarity(name,'son') DESC;
```

The GIN trigram index enables fast wildcard search on `name` [8].

- **Relevance ranking:** Return top 5 by relevance for “java developer”:

```
SELECT id,name, ts_rank(search_vector, q) AS rank
FROM employees, plainto_tsquery('english','java developer') AS q
WHERE search_vector @@ q
ORDER BY rank DESC
LIMIT 5;
```

PostgreSQL applies a *top-N heapsort* for ranking [12]. (Using `LIMIT` avoids a full sort.)

- **Exact JSONB containment:** Find employees whose project data has `"role":"Manager"`:

```
SELECT id,name FROM employees
WHERE projects @> '{"role": "Manager"}';
```

This uses the GIN JSONB index (`jsonb_path_ops`) for efficient matching [5] [9].

Performance Tuning

For sub-second responses on 25M rows, tune both queries and the database:

- **Proper Configuration:** Allocate enough memory. For example, set `shared_buffers` to ~15–25% of RAM [13] and `effective_cache_size` to ~50% [14] so planner can optimize. Increase `work_mem` to allow sorting/ranking in-memory. Tune `maintenance_work_mem` for index rebuilds.

- **Analyze & Maintenance:** Regularly run `ANALYZE` / `VACUUM` (or autovacuum) so the planner has up-to-date stats. Autovacuum also flushes the GIN *pending list*. (GIN indexes buffer insertions in a pending list; if it fills or on VACUUM, it merges—this flush can be expensive [15]. Keep `gin_pending_list_limit` tuned or occasionally call `gin_clean_pending_list()` to manage overhead.)

- **Use Index Conds:** Ensure queries are sargable. Avoid leading wildcards unless covered by trigram index. Compare columns directly (`column @@ tsquery`) to use index rather than applying functions.

- **EXPLAIN and Trace:** Use `EXPLAIN (ANALYZE)` to confirm that search queries use index scans (Bitmap Index Scan on GIN) rather than Seq Scans. The examples above show Bitmap Index Scans and Bitmap Heap Scans using the GIN idx [16] [17].

- **Limit Sort Work:** Always apply `LIMIT` after ordering by rank. PostgreSQL performs a Top-N heapsort (with small memory) which avoids sorting all matches [12]. Minimizing the result set accelerates response.

- **Hardware and Parallelism:** Deploy on modern hardware (SSD for WAL/index write/read). PostgreSQL can use parallel queries on GIN scans, especially if multiple CPUs are available. Ensure `max_parallel_workers_per_gather` is set reasonably high.

- **Avoid Bloated Indexes:** Periodically `REINDEX` or `VACUUM FREEZE` if write-heavy, as GIN indexes can bloat with frequent updates. If full-text data is mostly read-only (e.g. employee directory), the overhead is mainly on bulk refresh.

- **Cache Layer:** For very frequent identical searches, an external cache (e.g. Redis or an application cache) can store recent result sets. This reduces database load, at the cost of cache invalidation logic.

Extensions and Enhancements

- **pg_trgm (Trigram):** Already used for substring matching, it's part of standard PG and accelerates `ILIKE '%...%'` patterns [17].

- **pg_vector:** For advanced similarity or recommendation search (e.g. semantic skill matching), PostgreSQL's **pgvector** extension allows storing embedding vectors. It supports KNN and distance searches in SQL [18]. One could compute a text embedding of employee profiles (or skills) externally and index with pgvector for "more like this" queries.

- **RUM Index:** RUM (an open extension) is an enhanced GIN that can index and rank more efficiently. It supports faster `ORDER BY ts_rank` queries than plain GIN [19]. If search relevance ranking is critical, a RUM index on `search_vector` is worth considering.

- **Full-Text Configs:** You can customize dictionaries or stop-words if needed (Chapter 12 in docs). Different languages can be indexed by specifying config (see PG docs). We assume English here.

- **Elasticsearch Integration:** While this solution uses PG alone, one could integrate or offload to Elasticsearch if needed. The **ZomboDB** extension, for example, makes Postgres use Elasticsearch under the hood: *"ZomboDB is a Postgres extension that enables efficient full-text searching via the use of indexes backed by Elasticsearch"* [20]. For very large scale or more complex search features, a hybrid approach (PG + ES) is an option, but PG's native FTS often suffices.

Caching and Hybrid Approaches

Optionally, implement caching or external search as follows:

- **Result Caching:** Use a query cache layer for repeated popular queries. For example, store recent search inputs and their result sets in Redis, with a short TTL. This can drastically cut response time for hot searches.

- **Read Replicas:** If search load is high, direct reads to replica servers. Each replica has the same indexes and can answer sub-second queries under high concurrency.

- **Elasticsearch/PG Sync:** Maintain an Elasticsearch index in parallel (via logical decoding or ETL) for alternate search API, while keeping PostgreSQL as the system-of-record.

In summary, a combination of a good schema with a precomputed TSVECTOR, GIN (and trigram/JSONB) indexes, and a well-tuned PostgreSQL instance can deliver real-time, sub-second global search over 25M records. Proper query design (using `to_tsquery`/`websearch_to_tsquery`, ranking, and limiting) plus system tuning makes this feasible [21] [6]. The above schema and indexes provide the foundation for fast keyword-based search across names, skills, locations, certifications, and project details.

Sources: We leverage PostgreSQL's documented FTS features [6] [2] and community benchmarks [21] [8]. Extensions like `pg_trgm` and `pgvector` are used as recommended in expert blogs [19] [18]. ZomboDB and Elasticsearch serve as optional scaling paths [20]. These practices ensure efficient full-text search at scale.

---

[1] [6] **PostgreSQL: Documentation: 17: 12.2. Tables and Indexes**
https://www.postgresql.org/docs/current/textsearch-tables.html

[2] **PostgreSQL: Documentation: 17: 12.4. Additional Features**
https://www.postgresql.org/docs/current/textsearch-features.html

[3] [10] [11] **PostgreSQL: Documentation: 17: 12.3. Controlling Text Search**
https://www.postgresql.org/docs/current/textsearch-controls.html

4 5 7 8 9 15 17 19 **Understanding Postgres GIN Indexes: The Good and the Bad**
https://pganalyze.com/blog/gin-index

12 16 21 **PostgreSQL Full-Text Search: A Powerful Alternative to Elasticsearch for Small to Medium Applications | by Miftahul Huda | Medium**
https://iniakunhuda.medium.com/postgresql-full-text-search-a-powerful-alternative-to-elasticsearch-for-small-to-medium-d9524e001fe0

13 14 **How to tune PostgreSQL for memory | EDB**
https://enterprisedb.com/postgres-tutorials/how-tune-postgresql-memory?lang=en

18 **Vector Similarity Search with PostgreSQL's pgvector - A Deep Dive | Severalnines**
https://severalnines.com/blog/vector-similarity-search-with-postgresqls-pgvector-a-deep-dive/

20 **Integrate Postgresql and Elasticsearch - ZomboDB**
https://www.zombodb.com/