Full Documentation: Implementing Fuzzy Search and Type-Ahead Suggestions using PostgreSQL pg_trgm

---

1. Overview

This document walks you through a complete implementation of a **fast, typo-tolerant search** on PostgreSQL tables using the `pg_trgm` extension and GIN–trgm indexes. You will learn:

- Why and when to use trigram-based indexing for large datasets
- How to enable and configure `pg_trgm` in your database
- How to create GIN–trgm indexes on each searchable column
- How to define two RPC functions:
    1. **Full-table substring search** (case-insensitive match across multiple columns)
    2. **Type-ahead suggestions** (fuzzy, deduplicated suggestions prioritized by relevance)
- How to call these functions from your application (using Supabase as an example)
- Best practices for maintenance, schema changes, and handling new data

This guide is written to be **self-contained**—any developer or DBA can copy, paste, and customize the SQL and client code for their own tables.

---

2. Why pg_trgm and GIN–trgm Indexes?

2.1 The Challenge of Large-Scale Text Search

- **Standard `LIKE` or `ILIKE '%term%'`** scans on large tables (100k+ rows) can become prohibitively slow, because each query requires a full table scan.

- **Client-side libraries** like Fuse.js work well for small datasets (<400k rows) but can’t scale to millions of records without heavy memory and CPU overhead.

2.2 Trigrams to the Rescue

- PostgreSQL’s `pg_trgm` extension decomposes text into 3-character sequences (“trigrams”), allowing similarity-based matching and fast indexing.
- The **`%` operator** checks whether a string is similar to a pattern based on trigram overlap.
- The **`similarity()`** function computes a numeric score (0.0 to 1.0), letting you rank candidates by closeness.

2.3 Why GIN–trgm Indexes?

- **GIN** (Generalized Inverted Index) with the `gin_trgm_ops` operator class allows sub-linear lookup for both:
  - **Wildcards** (`ILIKE '%term%'`) by using the same trigram index behind the scenes
  - **Fuzzy matching** (`column % term`) directly against trigrams
- Building these indexes **concurrently** avoids long table locks in production environments.

---

3. Prerequisites and Permissions

Before proceeding, ensure that:

1. Your PostgreSQL version is **≥ 9.6** (for `pg_trgm`).
2. The database user has privileges to:
   - Create extensions
   - Create and drop indexes

    - Create and replace functions

3. You have a table (e.g. `lmia` or `hot_leads_new`) with at least one text or non-text column you wish to search.

---

4. Step 1: Enable the pg_trgm Extension

```
-- This only needs to run once per database:
CREATE EXTENSION IF NOT EXISTS pg_trgm;
```

- **Purpose**: Registers trigram operators, the `%` match operator, and the `similarity()` function in your database.
- **Idempotent**: Safe to run multiple times; only the first invocation has effect.

---

5. Step 2: Create GIN–trgm Indexes on Searchable Columns

For each column you plan to search, create a GIN–trgm index:

```
-- Example for a text or castable column `city`:
CREATE INDEX CONCURRENTLY IF NOT EXISTS idx_yourtable_city_trgm
  ON public.your_table
  USING GIN ((city::text) gin_trgm_ops);
```

- **CONCURRENTLY**: Builds without locking writes, so your production traffic is unaffected.
- **IF NOT EXISTS**: Prevents errors if the index already exists.
- **Casting**: Use `column::text` if the column type is not already `text` or `varchar`.

Repeat the above for every column you include in your search functions.

---

6. Step 3: Define the Full-Table Substring Search RPC

This SQL function returns **all rows** where **any** of your specified columns contains the search term, case-insensitive:

```
CREATE OR REPLACE FUNCTION public.rpc_search_your_table(term text)
  RETURNS TABLE (
    col1 text,
    col2 text,
    -- ... other columns ...
  )
LANGUAGE sql
STABLE
AS $func$
  SELECT
    col1::text,
    col2::text,
    -- ... cast other columns ...
  FROM public.your_table
  WHERE
    col1::text ILIKE '%' || term || '%'
    OR col2::text ILIKE '%' || term || '%'
    -- ... OR other columns ...
;
$func$;
```

Explanation

1. **RETURNS TABLE(...)**: Lists which columns the function will output (all cast to text).
2. **STABLE**: Indicates the function does not modify data and returns consistent results within a query.
3. **ILIKE**: Performs a case-insensitive substring match.
4. The **trigram indexes** you created earlier accelerate each `ILIKE '%term%'` clause, so the query avoids full table scans.

---

7. Step 4: Define the Type-Ahead Suggestions RPC

This function provides a **limited set** of fuzzy suggestions, ranked by relevance:

```
CREATE OR REPLACE FUNCTION public.rpc_search_your_table_suggestions(
  term       text,
  p_limit    int DEFAULT 20,   -- max suggestions
  branch_lim int DEFAULT 100   -- per-column candidate limit
)
RETURNS TABLE (suggestion text, column_name text)
LANGUAGE sql
STABLE
AS $func$
WITH
  -- Branch per column:
  b_col1 AS (
    SELECT col1::text AS suggestion, 'col1' AS column_name,
           similarity(col1::text, term) AS sim
      FROM public.your_table
     WHERE col1::text % term       -- trigram match
     ORDER BY sim DESC             -- highest similarity first
     LIMIT branch_lim
  ),
  b_col2 AS (...), -- repeat for each column

  -- Combine all branches:
  candidates AS (
    SELECT * FROM b_col1
    UNION ALL SELECT * FROM b_col2
    -- ... other branches ...
  )

SELECT
  suggestion,
  column_name
FROM candidates
GROUP BY suggestion, column_name  -- dedupe identical suggestions
ORDER BY MAX(sim) DESC            -- best matches overall
LIMIT p_limit;
$func$;
```

Detailed Walkthrough

- **Branch CTEs (`b_<col>`)**:

    - Query each column separately for values ‘close’ to the input term using the `%` operator (trigram match).
    - Compute a **similarity score** for each candidate.
    - **Limit** to `branch_lim` per column to avoid overwhelming the union.
- **UNION ALL**:
    - Merges results from all columns into a single pool.
- **GROUP BY & ORDER BY**:
    - **Group** by `(suggestion, column_name)` to remove duplicates.
    - Order by the **highest similarity score** across duplicates.
- **LIMIT**:
    - Return only the top `p_limit` suggestions overall.

---

8. Step 5: Application Integration Examples

Supabase JavaScript Client

```
// 1. Perform full-table search:
const { data: rows, error: searchError } = await supabase
  .rpc('rpc_search_your_table', { term: 'engineer' });

// 2. Fetch type-ahead suggestions:
const { data: suggestions, error: sugError } = await supabase
  .rpc('rpc_search_your_table_suggestions', {
    term: 'engin',     // partial user input
    p_limit: 10,       // maximum suggestions returned
    branch_lim: 50,    // top 50 per column scanned
  });
```

- Replace **`your_table`**, **function names**, and **column lists** with your actual values.

- The returned `rows` and `suggestions` can be directly bound to your UI components.

---

9. Index Creation & Migration Considerations

Error: `CREATE INDEX CONCURRENTLY cannot run inside a transaction block`

- **Cause**: Many migration tools wrap each file in a transaction. Concurrent index builds must be executed outside a transaction.

Solutions:

1. **Interactive Execution**:
   - Connect via `psql` or a SQL client, paste each `CREATE INDEX CONCURRENTLY` statement, and run individually.
2. **One-statement Migration Files**:
   - Split each `CREATE INDEX CONCURRENTLY` into its own migration script so your migration runner can detect no transaction wrapper is needed.
3. **Fallback Non-Concurrent Index**:

Drop the `CONCURRENTLY` keyword if you can tolerate a brief write lock:

```
CREATE INDEX IF NOT EXISTS idx_yourtable_col_trgm
 ON public.your_table USING gin ((col::text) gin_trgm_ops);
```

-

---

10. Maintenance, Schema Changes & New Data

10.1 Handling New Rows

- **Automatic Inclusion**: Once the extension, indexes, and functions are in place, **any new rows** inserted into the table are immediately searchable and will appear in both full-table and suggestion queries. You do **not** need to recreate extensions, rebuild indexes, or redefine functions for standard INSERT/UPDATE operations.

10.2 Schema Modifications

- **Adding Searchable Columns**:

  1. Create a new GIN–trgm index for the added column.

  2. Update both RPC definitions to include the new column in their `SELECT` clauses and `WHERE`/`CTE` branches.

10.3 Performance Tuning

- **`VACUUM ANALYZE`** periodically to update planner statistics so indexes remain effective.

- **`REINDEX INDEX ...`** if GIN indexes become bloated due to heavy churn.

- Adjust **`pg_trgm.similarity_threshold`** (e.g., `SET pg_trgm.similarity_threshold = 0.25;`) to fine-tune what counts as a trigram match.

- Tune `branch_lim` and `p_limit` to control coverage vs. performance in the suggestions RPC.

---

11. Conclusion

You now have a **comprehensive**, production-ready pattern for adding:

- Case-insensitive **full-table searches**

- Typo-tolerant, **ranked suggestions**

on any PostgreSQL table using trigrams and GIN indexes. Copy and adapt the SQL and client snippets to your own schemas, and enjoy fast, scalable search capabilities without external services.

Happy indexing! 🎉