DANIELMESQUITTA.COM

# MASTERING POSTGRES TEXT SEARCH

## LIKE, LEVENSHTEIN & TRIGRAMS

BY DANIEL MESQUITA

## Introduction

**Text search** was a **recurring problem** for me, but it took a while before I decided to dive deep into all the available options and determine the **best approach**.

This post is about that, all my **experiences**, pros and cons, and my **recommendations** for handling text search using PostgreSQL.

## Using ILIKE

The ILIKE operator is the **classic method** for **pattern matching** in SQL. It allows you to perform wildcard searches with % and _ characters

```
SELECT *
FROM products
WHERE name ILIKE '%cafe%';
```

# Unnacent

Using just LIKE can quickly **lead to trouble** because if users start using accents, the words won’t match. That’s when you should **use unaccent**.

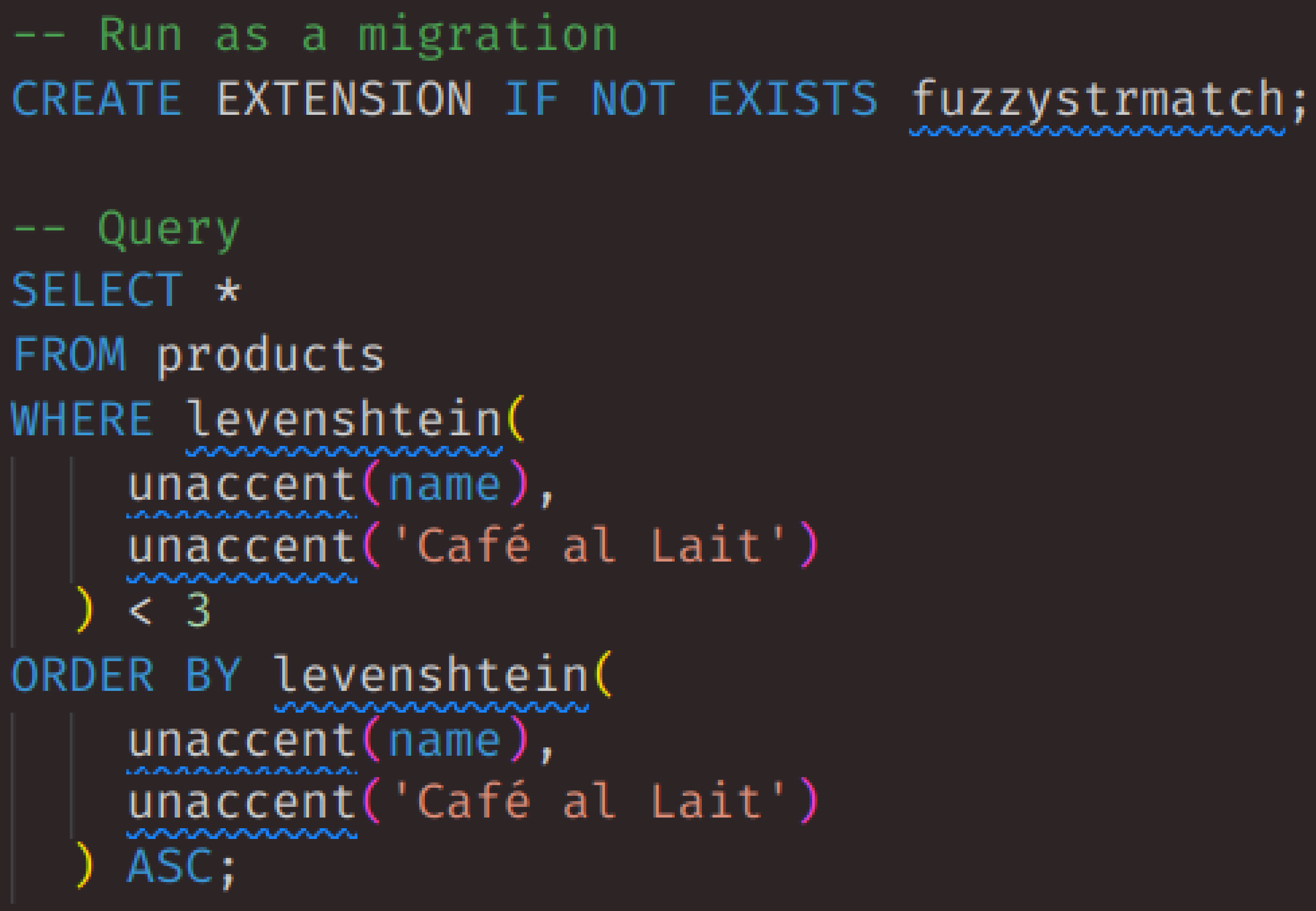
```sql
-- Run as a migration
CREATE EXTENSION IF NOT EXISTS unaccent;

-- Query
SELECT *
FROM products
WHERE unaccent(name) ILIKE unaccent('%cafe%');
```

## Levenshtein

However, using only **ILIKE** and **unaccent won't be enough**. What if someone is searching for a "Café au Lait" but **mistypes** it as "Café al Lait"? In that case, the user wouldn't find **anything**—even though they should.

This issue can be resolved with **Levenshtein distance**.

# Levenshtein example



```sql
-- Run as a migration
CREATE EXTENSION IF NOT EXISTS fuzzystrmatch;

-- Query
SELECT *
FROM products
WHERE levenshtein(
        unaccent(name),
        unaccent('Café al Lait')
    ) < 3
ORDER BY levenshtein(
        unaccent(name),
        unaccent('Café al Lait')
    ) ASC;
```

## Explaining Levenshtein

Levenshtein calculates the **number of single-character edits** (insertions, deletions, or substitutions) needed to transform one string into another.

So, in this example, “Café au Lait” will be at the top of the list, since the **character distance** is only 1.

# Trigram Comparison

But what if the user just types "Café Laite," expecting the system to find "Café au Lait"?
**Neither LIKE nor Levenshtein can handle that**, because the character distance is too high and "Café Laite" isn't contained in "Café au Lait."

However, **trigram comparison would find** what the user is looking for.

## Trigram Comparison Example

```sql
-- Run as a migration
CREATE EXTENSION IF NOT EXISTS pg_trgm;

-- Query
SELECT *
FROM products
WHERE
  name % 'Café Laite'
ORDER BY
  similarity(name, 'Café Laite') DESC;
```

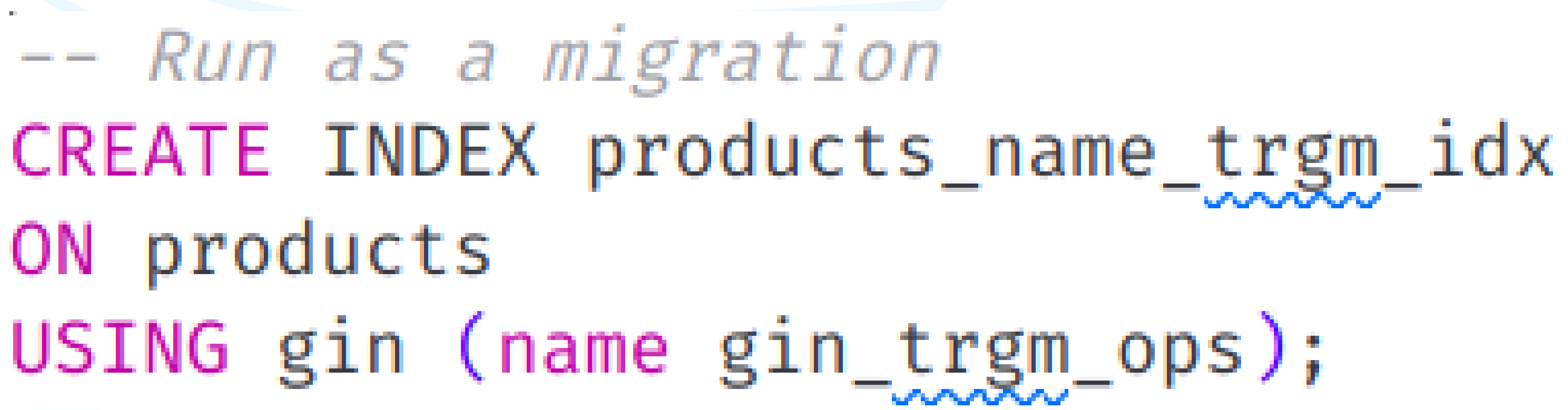
# Explaining Trigram Comparison

Trigram comparison is a **text similarity method** that **splits text** into overlapping **three-character sequences**.

So "Café au Lait" becomes "Caf", "afé", "fé ", "é a", " au", "au ", "u L", " La", "Lai", and "ait". It then **measures similarity** by counting **how many trigrams two texts share**.

## Database indexing

To **increase performance**, what you can do is **create indexes** for the fields you will use in your search, for example:



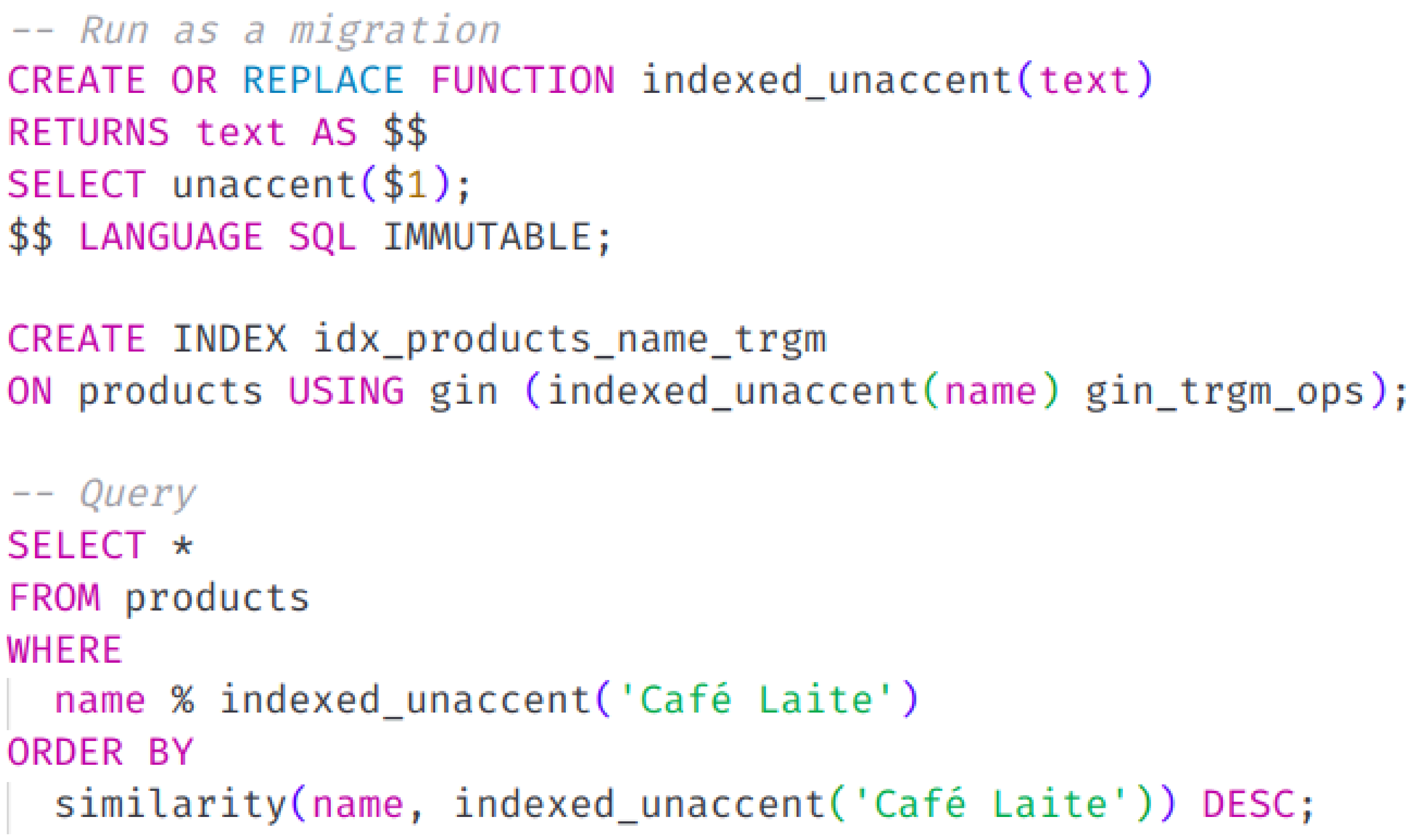
```
-- Run as a migration
CREATE INDEX products_name_trgm_idx
ON products
USING gin (name gin_trgm_ops);
```

# Trigam with Unnacent

**Here's my final tip:** use **trigram with unnacent** since trigram doesn't handle accents.
For me, it's my **go-to approach** and it fits almost every use case I've come across.
It's easy to use, efficient, and finds exactly what users are looking for.

The only issue is that you can't create indexes using unnacent as it is, but I'll show you a **little trick**.

# My go-to approach



```sql
-- Run as a migration
CREATE OR REPLACE FUNCTION indexed_unaccent(text)
RETURNS text AS $$
SELECT unaccent($1);
$$ LANGUAGE SQL IMMUTABLE;

CREATE INDEX idx_products_name_trgm
ON products USING gin (indexed_unaccent(name) gin_trgm_ops);

-- Query
SELECT *
FROM products
WHERE
  name % indexed_unaccent('Café Laite')
ORDER BY
  similarity(name, indexed_unaccent('Café Laite')) DESC;
```

DONT FORGET TO LIKE, SHARE AND SAVE IF YOU LIKE THIS POST

DANIEL MESQUITA