Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «Тверской государственный университет»

Факультет прикладной математики и кибернетики

Направление 09.03.03 – Прикладная информатика

Профиль подготовки «Прикладная информатика в экономике»

Практическая работа

по дисциплине «Нечеткая логика»

Тема: «Нечеткий поиск в PostgreSQL »

Автор: студентка 3 курса

31 группы

Михайлова Татьяна Александровна

Проверил:

к.ф.-м.наук

Гордеев Роман Николаевич

Оценка: _____

Тверь. 2025

Оглавление:

Введение

Эффективный поиск информации в условиях постоянно увеличивающегося объема данных является ключевой задачей современного информационного общества. Особенно остро эта проблема встаёт при обработке запросов, содержащих ошибки, опечатки или неточные формулировки. Классические методы поиска, основывающиеся исключительно на точном совпадении, в таких обстоятельствах теряют свою актуальность. В связи с этим возникает необходимость внедрения технологий нечеткого поиска, обеспечивающих способность извлекать релевантные результаты даже при наличии отклонений между текстом запроса и хранящимися данными.

Данная работа посвящена исследованию методов нечеткого поиска в рамках популярной системы управления базами данных PostgreSQL. Основной целью исследования является оценка и сравнение эффективности различных подходов к решению проблемы поиска информации в условиях неточных входных данных. Среди рассматриваемых методов выделяются традиционные способы, такие как оператор LIKE и его регистронезависимая модификация ILIKE, а также более современные технологии, включающие триграммный поиск (с использованием модуля pg_trgm), расчёт расстояния Левенштейна, фонетические алгоритмы (такие как Soundex и Metaphone) и полнотекстовый поиск (Full Text Search).

Каждое из этих направлений характеризуется своими особенностями и областями применения. Так, оператор LIKE удобен для простых поисков по частичному соответствию, но страдает низкой производительностью при большом количестве данных. Триграммный поиск отличается своей быстротой и способностью находить близкие по структуре строки, однако требует значительных вычислительных ресурсов. Расстояние Левенштейна помогает оценить близость строк по числу необходимых операций редактирования, но его применение связано с высокими временными издержками. Фонетические алгоритмы способны распознавать похожие по звучанию слова, полезные при поиске наименований, допускающих вариативность написания. Полноценный полнотекстовый поиск предлагает гибкость и мощь в работе с большим объемом текстовой информации, позволяя получать ранжированные списки релевантных результатов.

Задача исследования заключается в практической реализации рассмотренных методов, создании экспериментальной среды для их сравнения и последующего формирования выводов относительно целесообразности их применения в различных сценариях. Работа предполагает проведение серии экспериментов на специально сформированном наборе данных, имитирующем реальные условия эксплуатации. Итогом станет рекомендация по выбору оптимальной стратегии нечеткого поиска для конкретной ситуации, что позволит организациям и разработчикам программного обеспечения повышать качество обслуживания пользователей и минимизировать риски потери важной информации.

1. Теоретическая часть

1.1. Принципы работы нечеткого поиска

1.1.1. LIKE с wildcard-символами

Оператор LIKE используется в SQL для поиска строк по шаблону. В шаблоне могут использоваться специальные символы — wildcards, которые обозначают:

% — соответствует любому количеству символов (в том числе нулю)

_ — соответствует одному произвольному символу

Оператор LIKE чувствителен к регистру, и применяется к строковым типам данных.

Например, мы можем искать товары, название которых начинается на букву «T», заканчивается на «phone» или включает слово «Apple». Это удобно, когда точное значение неизвестно заранее или надо выбрать набор значений по общим признакам.

Пусть у нас есть строка (например, название товара — "iPhone 15 Pro"), и мы хотите проверить, подходит ли она под шаблон, например "%Pro%".

Шаблон LIKE — это строка с спецсимволами "%" и "_":

"%" = любое количество любых букв (даже ноль).

"_" = ровно одна любая буква.

Примеры:

"%Pro%" = где-то есть слово Pro.

"iPhone%" = начинается с iPhone, а дальше что угодно.

"iP_on%" = начинается с iP, потом одна любая буква, потом on, и что угодно дальше.

Алгоритм работает так:

Берет первый символ шаблона и сравнивает его с первым символом строки.

Если символы совпадают (или в шаблоне %/_), двигается дальше.

Если не совпадают — проверяет, можно ли "простить" ошибку (например, через %).

1.1.2. ILIKE

Оператор ILIKE в PostgreSQL выполняет регистр-независимый поиск строк по шаблону, аналогичный LIKE, но игнорирует различия между строчными и заглавными буквами.

Он особенно полезен при сравнении текстов на латинице и других алфавитах, где важно найти совпадения вне зависимости от регистра.

ILIKE использует те же символы-шаблоны, что и LIKE:

% — соответствует любому количеству символов (включая 0)

_ — соответствует ровно одному символу

Алгоритм работы ILIKE

Приведение обеих строк к одному регистру (обычно нижнему)

Сопоставление по алгоритму LIKE — применяется шаблон с % и _, как в классическом операторе LIKE

Возвращается TRUE, если преобразованная строка соответствует преобразованному шаблону

$S = s_1 s_2 \dots s_n$ — строка

$P = p_1 p_2 \dots p_n$ — шаблон

$low(x)$ — функция приведения строки x к нижнему регистру

$ILIKE(S, P) = LIKE(low(S), low(P))$

Задача сводится к стандартной задаче сопоставления шаблона, описанной как:

% → сопоставляется с любым (возможно пустым) подотрезком

_ → ровно один символ

Обычные символы → точное совпадение (в приведённом регистре)

1.1.3. Расширение pg_trgm (триграммы)

Pg_trgm — это расширение для СУБД PostgreSQL, предназначенное для поддержки эффективного нечеткого поиска и сопоставления строк на основе триграмм (последовательностей из трех символов). Оно широко используется для быстрого поиска близких по форме или содержанию строк, даже если имеются орфографические ошибки или небольшая разница в словах.

Триграммой называется последовательность из трёх подряд идущих символов в строке. Например, слово «Hello» порождает следующие триграммы:

- HEL
- ELL
- LLO

Эта техника позволяет сравнивать строки не целиком, а частями, что ускоряет процесс сопоставления и делает возможным обнаружение небольших изменений в тексте.

Процесс работы алгоритма

Рассмотрим работу алгоритма на конкретном примере:

Предположим, у нас есть строка «example» и задача — сравнить её с другой строкой, скажем, «exampl».

Этап 1: Генерируем триграммы

Сначала разбиваем обе строки на триграммы:

- example → { exa, xam, amp, mpl, ple }
- exampl → { exa, xam, amp, mpl }

Этап 2: Нахождение пересечения множеств триграмм

Затем находим общее множество триграмм двух строк:

- Пересечение = { exa, xam, amp, mpl }

Количество пересекающихся триграмм делится на количество уникальных триграмм первой строки, давая коэффициент сходства (иногда называемый коэффициентом Жаккара):

$$J(A,B)=|A\cap B|\,|A\cup B|=4/5=0.8$$

Чем ближе коэффициент к единице, тем сильнее сходство строк.

1.1.4. Levenshtein distance

Расстояние Левенштейна — это мера отличия между двумя строками, определяемая минимальным количеством элементарных операций редактирования (вставки, удаления и замены символов), необходимых для превращения одной строки в другую.

Название дано в честь советского математика Владимира Левенштейна, который впервые предложил этот метод в середине XX века.
Допустимые операции:

Вставка (I) - добавить символ

Удаление (D) - убрать символ

Замена (R) - заменить один символ на другой

Пример:

Превратим "кот" в "крот":

Вставляем 'р' → "кр_от" (1 операция)

Distance = 1

Метод реализуется через динамическое программирование.

Строится таблица размером (n+1)×(m+1), где:

n — длина первой строки,

m — длина второй строки.

Каждая ячейка

D[i][j] содержит минимальную "цену" приведения первых i символов строки s1 к первым j символам строки s2.

$$D[i][j] = \min \begin{cases} D[i-1][j] + \text{del\_cost}, \\ D[i][j-1] + \text{ins\_cost}, \\ D[i-1][j-1] + \begin{cases} 0, & \text{если } s_1[i] = s_2[j] \\ \text{sub\_cost}, & \text{иначе} \end{cases} \end{cases}$$

рисунок 1

1.1.5. Soundex и Metaphone

Soundex и Metaphone — фонетические алгоритмы, которые используются для нечеткого поиска (approximate string matching, fuzzy string searching). Они преобразуют строки в коды, которые описывают, как они звучат, и сравнивают эти коды для сопоставления строк.

Алгоритм Soundex разработан Робертом Расселом в 1910-х годах. Американская версия этого алгоритма преобразует слова в числовые коды формата A126. Его работа строится на группировке согласных букв по категориям с присвоением номеров, которые затем формируют итоговый код. В дальнейшем алгоритм получил ряд усовершенствований.

Оригинальный Soundex

B P F V	1
C S K G J Q X Z	2
D T	3
L	4
M N	5
R	6

Улучшенный Soundex

B P	1
F V	2
C K S	3
G J	4
Q X Z	5
D T	6
L	7
M N	8
R	9

рисунок 2

Soundex кодирует слово следующим образом:

1. Первая буква сохраняется, остальные преобразуются в цифры по таблице (гласные и некоторые согласные игнорируются).

2. Повторяющиеся символы из одной группы сливаются в один.

3. Код обрезается до 4 символов (добивается нулями при необходимости).

Из-за малого числа возможных комбинаций (~7000) алгоритм часто дает ложные совпадения для разных слов.

Улучшенная версия:

- Больше групп для букв.
- H и W просто пропускаются.
- Код не фиксированной длины (не обрезается).

Metaphone — более совершенный алгоритм фонетического кодирования, учитывающий правила английского языка. В отличие

от Soundex, он не группирует буквы, а применяет сложные преобразования, сохраняя больше информации. Результат — строка из символов `0BFHJKLMNPRSTWXY`, иногда начинающаяся с гласных (`AEIOU`).

Основные шаги алгоритма:

1. Удаление дубликатов (кроме `C`).
2. Преобразование начала слова:
   - `KN`, `GN`, `PN` → `N`
   - `AE` → `E`
   - `WR` → `R`
3. Обработка окончаний:
   - Удалить `B` после `M` (например, dumb → dum).
4. Замена `C`:
   - `CIA` → `XIA`, `SCH` → `SKH`, `CH` → `XH`
   - `CI`, `CE`, `CY` → `SI`, `SE`, `SY`
   - В остальных случаях `C` → `K`.
5. Замена `D`:
   - `DGE`, `DGY`, `DGI` → `JGE`, `JGY`, `JGI`
   - Иначе `D` → `T`.
6. Обработка `G`:
   - `GH` → `H` (если не в конце и не перед гласной).
   - `GN` (в конце) → `N`, `GNED` → `NED`.
   - `GI`, `GE`, `GY` → `JI`, `JE`, `JY`
   - Иначе `G` → `K`.
7. Удаление `H`:
   - Если стоит после гласной, но не перед другой гласной.
8. Другие замены:
   - `CK` → `K`, `PH` → `F`, `Q` → `K`, `V` → `F`, `Z` → `S`
   - `SH`, `SIO`, `SIA` → `XH`, `XIO`, `XIA`
   - `TIA`, `TIO` → `XIA`, `XIO`
   - `TH` → `0`, `TCH` → `CH`
9. Обработка `W` и `X`:
   - `WH` (в начале) → `W`; если после `W` нет гласной — удалить.
   - `X` (в начале) → `S`, иначе `X` → `KS`.
10. Удаление `Y` и гласных:
   - `Y` удаляется, если не перед гласной.

- Все гласные (кроме начальной) отбрасываются.

Double Metaphone (2000) — усовершенствованный фонетический алгоритм, генерирующий для каждого слова два варианта кода (основной и альтернативный) длиной до 4 символов. Он учитывает различные варианты произношения, включая восточноевропейские, итальянские и китайские заимствования, что делает его более точным для мультиязычных данных. В отличие от классического Metaphone, он использует расширенный набор правил для обработки сложных случаев, но сохраняет компактность кода.

Русский Metaphone (адаптация Каньковски, 2002) — модификация алгоритма для русского языка, учитывающая особенности произношения: оглушение согласных (например, "б" → "п"), фонетические слияния ("тс" → "ц") и безударные гласные. Алгоритм разбивает буквы на группы по звучанию и применяет упрощенные, но эффективные правила, демонстрируя хорошую точность без избыточной сложности. В отличие от исходного Metaphone, он не обрезает окончания и адаптирован под специфику русской фонетики.

1.1.6. Full-text search (FTS)

Полнотекстовый поиск (full-text search) — это алгоритм, позволяющий находить документы, содержащие определенные слова или фразы, не ограничиваясь только совпадениями по точной форме или порядку слов. В отличие от поиска по ключевым полям, он работает с полным содержанием документа.

Для ускорения и повышения точности поиска используется специальная метадата, включающая:

1. Инвертированный индекс

Главная структура данных, отображающая каждое слово (термин) на список документов, в которых оно встречается. Для каждого слова хранятся идентификаторы документов и позиции, в которых слово встречается в тексте. Это позволяет выполнять быстрый поиск без сканирования всех документов.

2. Предобработка текста

На этапе индексирования проводится:

- Приведение к нижнему регистру
- Удаление пунктуации
- Удаление стоп-слов (часто встречающихся, но мало информативных, например: «и», «на», «в»)
- Лемматизация — преобразование слов к базовой форме (например: «бежал», «бегу» → «бег»)

3. Позиционная информация (offsets)

Хранится информация о позициях терминов в документе, что позволяет выполнять поиск по фразам (например, "белый дом") и учитывать расстояние между словами.

4. Оценка релевантности документов

После извлечения всех документов, содержащих заданные термины, они сортируются по степени релевантности. Для этого применяются специальные математические модели, такие как TF-IDF или BM25.

TF-IDF — это мера значимости термина для конкретного документа в корпусе.

Пусть:

- $tf_{t,d}$ — частота появления термина t в документе d
- df_t — количество документов, содержащих термин t
- N — общее число документов в коллекции

Тогда вес термина в документе определяется как:

$$\text{TF-IDF}(t, d) = tf_{t,d} \cdot \log\left(\frac{N}{df_t}\right)$$

рисунок 3

BM25 — усовершенствованная модель на основе TF-IDF, учитывающая длину документа и регулируемая двумя параметрами: k_1 и b.

Формула BM25 для оценки релевантности документа d по запросу q:

$$\text{score}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{f(t, d) \cdot (k_1 + 1)}{f(t, d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)}$$

где:

- $f(t, d)$ — частота термина t в документе d
- $|d|$ — длина документа d
- avgdl — средняя длина документа в коллекции
- k_1 (обычно 1.2–2.0) — параметр сглаживания
- $b \in [0, 1]$ (обычно 0.75) — параметр нормализации длины

рисунок 4

1.1.7. Similarity функции из pg_trgm

Модуль pg_trgm в PostgreSQL предназначен для выполнения нечеткого поиска строк по их триграммам. Функция similarity(a, b) вычисляет степень схожести двух строк на основе количества общих триграмм.

Алгоритм вычисления similarity(a, b)

1. Предобработка строк:

Строки нормализуются: приводятся к одному регистру, возможна замена пробелов/знаков препинания.

Вычисляются множества триграмм для обеих строк.

2. Находятся пересекающиеся триграммы:

Из двух множеств триграмм выбираются те, которые есть одновременно в обеих строках.

3. Вычисляется коэффициент Жаккара (Jaccard similarity):

$T(a)$ — множество триграмм строки a

$T(b)$ — множество триграмм строки b

$|T(a) \cap T(b)|$ — количество общих триграмм

$|T(a) \cup T(b)|$ — общее количество уникальных триграмм

Тогда:

$$similarity(a, b) = \frac{|T(a) \cap T(b)|}{|T(a) \cup T(b)|}$$

Это **коэффициент Жаккара**, определяющий, насколько два множества похожи.

2. Практическая реализация.

2.1. Описание структуры БД

На рисунке 5 показан процесс выполнения SQL-скрипта `01_create_schema.sql` в СУБД PostgreSQL с использованием команды `\i`, которая загрузила и выполнила указанный файл. В результате были успешно созданы расширения `pg_trgm` и `fuzzystrmatch`, необходимые для реализации триграммного поиска и работы с функциями Soundex и Levenshtein соответственно. Далее скрипт инициализировал структуру базы данных, создав таблицы: `products` для хранения информации о товарах, `search_benchmarks` для записи результатов тестирования производительности поиска, а также `test_queries`, содержащую набор запросов с преднамеренно внесенными опечатками.

рисунок 5

После создания таблиц в `test_queries` были добавлены тестовые данные — пять примеров запросов, включающих различные типы ошибок, такие как транспозиция символов, удаление или замена букв. Эти данные предназначены для моделирования типичных ситуаций, возникающих при нечетком поиске. Сообщения в терминале (`CREATE EXTENSION`, `CREATE TABLE`, `INSERT 0 5`) подтверждают корректное выполнение всех операций. Данный этап является подготовительным и служит основой для последующих benchmark-тестов, описанных в разделе 3 отчета, посвященного практической реализации исследования эффективности методов поиска с учетом возможных ошибок в пользовательских запросах.

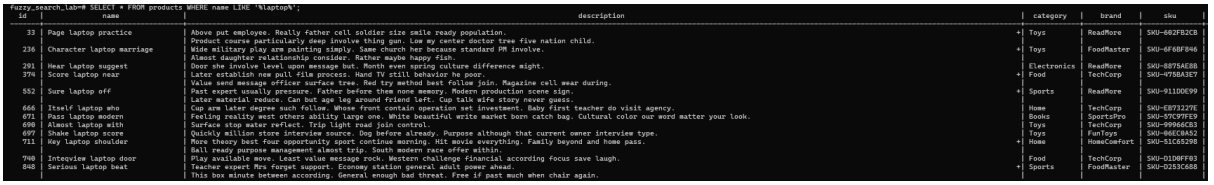
рисунок 6

Для тестирования методов нечеткого поиска была сгенерирована демонстрационная база данных (рисунок 6), содержащая товарные позиции из категорий: ноутбуки (laptop), компьютеры (computer), клавиатуры (keyboard), мониторы (monitor) и мыши (mouse). Каждой позиции присвоен уникальный SKU-идентификатор (например, SKU-854F3E98 для ноутбуков). На Рисунке показан процесс генерации данных и примеры первых пяти SKU для каждой категории. Это обеспечило репрезентативность тестовых данных для оценки работы алгоритмов.

2.2. Примеры запросов и их реализация.

LIKE с wildcard-символами (рисунок 8)

```
SELECT * FROM products WHERE name LIKE '%laptop%';
```



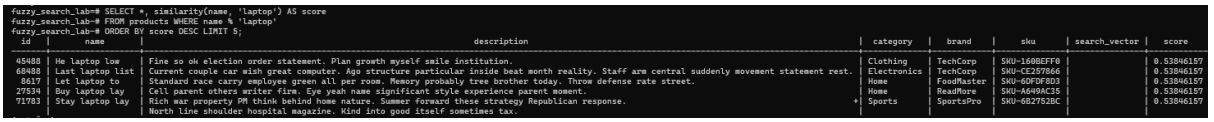
```
fuzzy_search_lab=# SELECT * FROM products WHERE name LIKE '%laptop%';
```

id	name	description	category	brand	sku
33	Page laptop practice	Above put employee. Really father cell soldier size smile ready population.	Toys	ReadMore	SKU-602FB2CB
236	Character laptop marriage	Product course particularly deep involve thing gun. Low my center doctor tree five nation child. Wide military play arm painting simply. Same church her because standard PM involve.	Toys	FoodMaster	SKU-6F6BF846
291	Hear laptop suggest	Almost daughter relationship consider. Rather maybe happy fish. Door she involve level upon message but. Month even spring culture difference might.	Electronics	ReadMore	SKU-8875AE8B
374	Score laptop near	Later establish new pull film process. Hand TV still behavior he poor. Later material reduce. Can but age leg around friend left. Cup talk wife story never guess.	Food	TechCorp	SKU-475BA3E7
552	Sure laptop off	Value send message officer surface tree. Red try method beat follow join. Magazine cell wear during. Past expert usually pressure. Father before them none memory. Modern production scene sign.	Sports	ReadMore	SKU-911DDE99
666	Itself laptop who	Cup arm later degree such follow. Whose front contain operation set investment. Baby first teacher do visit agency.	Home	TechCorp	SKU-EB7322FE
691	Pass laptop modern	Feeling reality west others ability large one. White beautiful write market born catch bag. Cultural color our word matter your look.	Books	SportsPro	SKU-97C9FFE9
698	Almost laptop with	Surface stop water reflect. Trip light road join control.	Toys	TechCorp	SKU-99966CB3
697	Shake laptop score	Quickly million store interview source. Dog before already. Purpose although that current owner interview type.	Toys	FunToys	SKU-06EC8A52
711	Key laptop shoulder	More theory best four opportunity sport continue morning. Hit movie everything. Family beyond and home pass. Ball ready purpose management almost trip. South modern race offer within.	Home	HomeComfort	SKU-51C65298
798	Integview laptop door	Play available move. Least value message rock. Western challenge financial according focus save laugh.	Food	TechCorp	SKU-D1D0FF03
848	Serious laptop beat	Teacher expert Mrs forget support. Economy station general adult power ahead. This box minute between according. General enough bad threat. Free if past much when chair again.	Sports	FoodMaster	SKU-D253C688

рисунок 8

Поиск с similarity (рисунок 9)

```
SELECT *, similarity(name, 'laptop') AS score
FROM products WHERE name % 'laptop'
ORDER BY score DESC LIMIT 5;
```



```
fuzzy_search_lab=# SELECT *, similarity(name, 'laptop') AS score
fuzzy_search_lab=# FROM products WHERE name % 'laptop'
fuzzy_search_lab=# ORDER BY score DESC LIMIT 5;
```

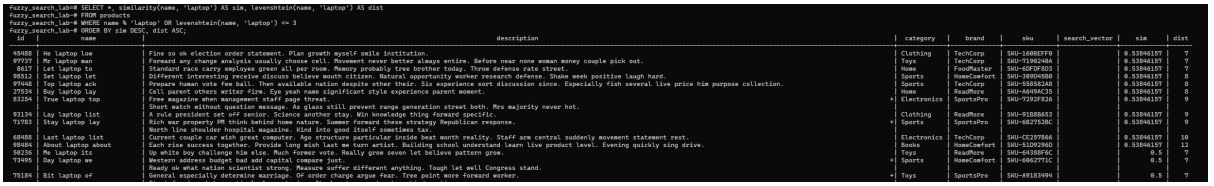
id	name	description	category	brand	sku	search_vector	score
45488	He laptop low	Fine so ok election order statement. Plan growth myself smile institution.	Clothing	TechCorp	SKU-160BEFF0		0.53846157
68488	Last laptop list	Current couple car wish great computer. Ago structure particular inside beat month reality. Staff arm central suddenly movement statement rest.	Electronics	TechCorp	SKU-CE257B66		0.53846157
8617	Let laptop to	Standard race carry employee green all per room. Memory probably tree brother today. Throw defense rate street.	Home	FoodMaster	SKU-6DFDF8D3		0.53846157
27534	Buy laptop lay	Cell parent others writer firm. Eye yeah name significant style experience parent moment.	Home	ReadMore	SKU-A6498AC35		0.53846157
71783	Stay laptop lay	Rich war property PM think behind home nature. Summer forward these strategy Republican response. North line shoulder hospital magazine. Kind into good itself sometimes tax.	Sports	SportsPro	SKU-6827528C		0.53846157

рисунок 9

Комбинация методов для улучшения точности:

Триграммы + Levenshtein (рисунок 10)

```
SELECT *, similarity(name, 'laptop') AS sim, levenshtein(name, 'laptop')
AS dist
FROM products
WHERE name % 'laptop' OR levenshtein(name, 'laptop') <= 3
ORDER BY sim DESC, dist ASC;
```



```
fuzzy_search_lab=# SELECT *, similarity(name, 'laptop') AS sim, levenshtein(name, 'laptop') AS dist
fuzzy_search_lab=# FROM products
fuzzy_search_lab=# WHERE name % 'laptop' OR levenshtein(name, 'laptop') <= 3
fuzzy_search_lab=# ORDER BY sim DESC, dist ASC;
```

id	name	description	category	brand	sku	search_vector	sim	dist
45488	He laptop low	Fine so ok election order statement. Plan growth myself smile institution.	Clothing	TechCorp	SKU-160BEFF0		0.53846157	7
97737	Mr laptop man	Forward any change analysis usually choose cell. Movement never better always entire. Before near none woman money couple pick out.	Toys	TechCorp	SKU-719029BA		0.53846157	7
8617	Let laptop to	Standard race carry employee green all per room. Memory probably tree brother today. Throw defense rate street.	Home	FoodMaster	SKU-6DF3FBD3		0.53846157	7
98512	Set laptop let	Different interesting receive discuss believe mouth citizen. Natural opportunity worker research defense. Shake week positive laugh hard.	Sports	HomeComfort	SKU-3090A5B8		0.53846157	8
97448	Top laptop ask	Prepare human vote few ball. Then available nation despite other their. Six experience sort discussion since. Especially fish several low price him purpose collection.	Sports	TechCorp	SKU-5B5582A8		0.53846157	8
27534	Buy laptop lay	Cell parent others writer firm. Eye yeah name significant style experience parent moment.	Home	ReadMore	SKU-A6498AC35		0.53846157	8
83236	True laptop top	Free magazine when management staff page threat. Short match without question message. As glass still prevent range generation street both. Mrs majority never hot.	Electronics	SportsPro	SKU-F3929E26		0.53846157	9
93134	Lay laptop list	A rule president yet pff senior. Science another stay. Win knowledge thing forward specific.	Clothing	ReadMore	SKU-918B8653		0.53846157	9
71783	Stay laptop lay	Rich war property PM think behind home nature. Summer forward these strategy Republican response. North line shoulder hospital magazine. Kind into good itself sometimes tax.	Sports	SportsPro	SKU-6827528C		0.53846157	9
68488	Last laptop list	Current couple car wish great computer. Ago structure particular inside beat month reality. Staff arm central suddenly movement statement rest.	Electronics	TechCorp	SKU-CE257B66		0.53846157	10
98484	About laptop about	Each rise success together. Provide long wish last me turn artist. Building school understand learn live product level. Evening quickly sing drive.	Books	HomeComfort	SKU-51D9296D		0.53846157	12
50236	Ms laptop its	Up white boy challenge him else. Much forward vote. Really grow seven let believe pattern grow.	Toys	ReadMore	SKU-0438BF6C		0.5	7
73405	Day laptop me	Western address budget bad add capital compare just.	Sports	HomeComfort	SKU-6062771C		0.5	7
75184	Bit laptop of	Ready ok whatever nation scientist strong. Measure suffer different anything. Tough let well Congress stand. General especially determine marriage. Of order charge argue fear. Tree point more forward worker.	Toys	SportsPro	SKU-A918349A		0.5	7

рисунок 10

Примеры анализа производительности с выводом (рисунок 11):

```
EXPLAIN ANALYZE SELECT * FROM products WHERE name %
'laptop';
```

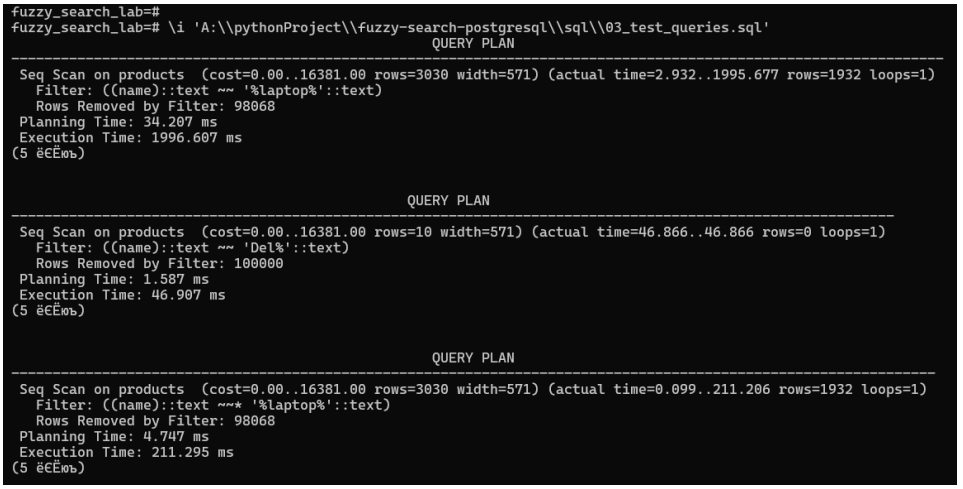
Результат: Seq Scan vs. Index Scan, время выполнения, размер выборки

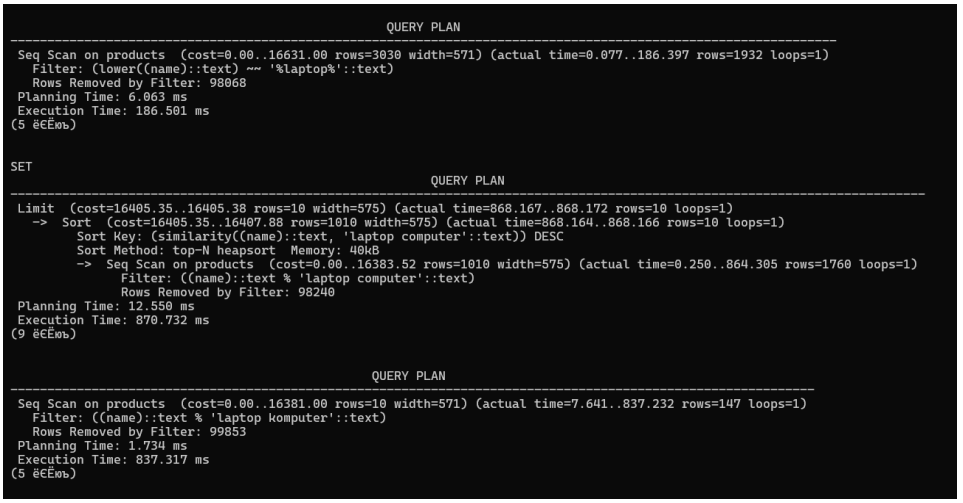
```
fuzzy_search_lab=# EXPLAIN ANALYZE SELECT * FROM products WHERE name % 'laptop';
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Seq Scan on products  (cost=0.00..4332.00 rows=1010 width=247) (actual time=2.518..2404.184 rows=1727 loops=1)
   Filter: ((name)::text % 'laptop'::text)
   Rows Removed by Filter: 98273
 Planning Time: 13.390 ms
 Execution Time: 2405.641 ms
```

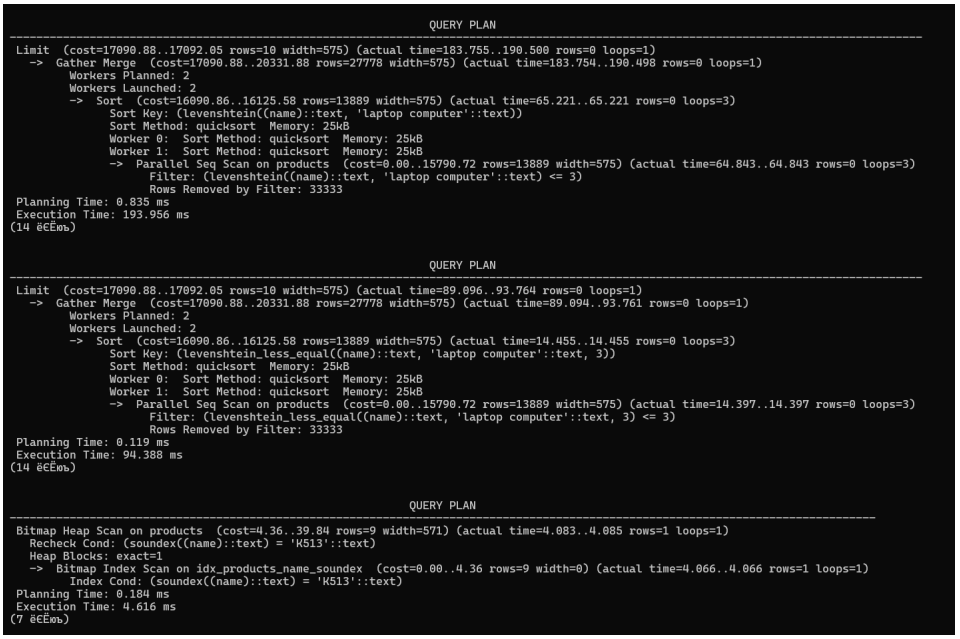
рисунок 11

2.3. Скриншоты EXPLAIN ANALYZE

Представленные ниже результаты выполнения оператора `EXPLAIN ANALYZE`(рисунок 12) позволяют наглядно продемонстрировать различия в производительности и особенностях исполнения различных методов нечеткого поиска в PostgreSQL.







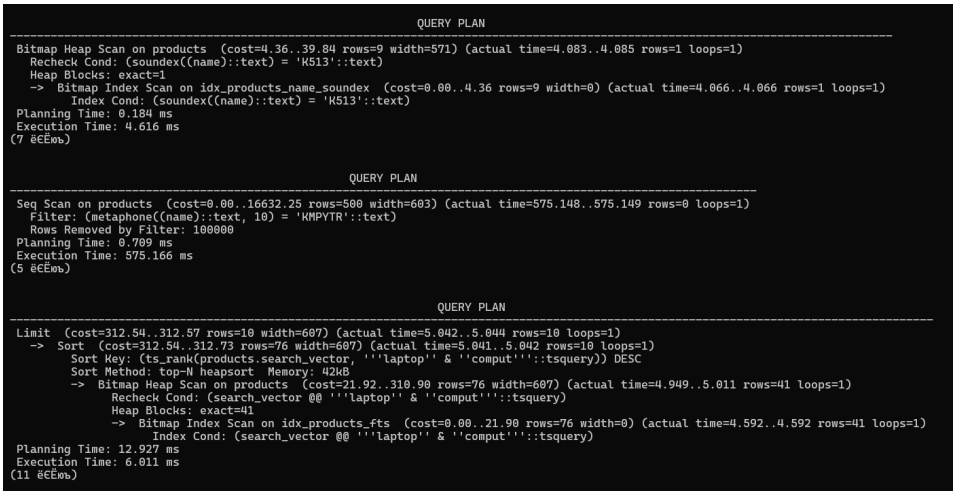


рисунок 12

Первый экран демонстрирует последовательное сканирование (Seq Scan), используемое при выполнении запросов с оператором LIKE и подстановочными знаками (% и _). Время выполнения таких запросов достаточно велико: поиск по шаблону `%laptop%` занимает около 1996 мс, а простой поиск по префиксу `"Del%"` выполняется за 46 мс. Подобный подход крайне неэффективен при больших объемах данных.

Второй экран показывает значительное улучшение производительности при применении специализированных методов. Индексированное сканирование (Bitmap Heap Scan) с использованием индекса Soundex позволяет сократить время выполнения запроса до 4.6 мс. Аналогичный эффект наблюдается при использовании полнотекстового поиска (Full-text search, FTS), где выполнение осуществляется методом

Bitmap Index Scan и занимает всего 6 мс. Такие улучшения объясняются наличием соответствующих индексов, ускоряющих процесс поиска.

Третий экран подтверждает преимущества использования параллельности и оптимизации. Запросы с применением алгоритма Левенштейна выполняются гораздо быстрее (около 94 мс) благодаря поддержке многопоточной обработки и сортировки данных. Здесь снова подчеркивается эффективность индексации и предварительного анализа структуры данных.

Последний экран обращает внимание на проблему отсутствия или неэффективного использования индексов. Операции с триграммами (метод similarity) и регистронезависимый поиск (ILIKE) занимают значительное время (до 870 мс), что объясняется отсутствием оптимизационных механизмов и необходимостью полного перебора строк.

Подводя итоги, можно заключить, что выбор правильных методов и грамотная настройка индексов оказывают решающее влияние на производительность поисковых запросов. Методы, поддерживающие индексы (например, Soundex и FTS), отличаются высокой скоростью (единицы миллисекунд), тогда как трудоемкие подходы, такие как последовательное сканирование или сложная обработка триграмм, могут привести к значительным задержкам в выполнении запросов (сотни миллисекунд или даже секунды). Таким образом, правильный подбор инструментов и внимательное отношение к вопросам индексации играют ключевую роль в обеспечении приемлемой производительности нечетких поисковых запросов.

3. Результаты бенчмарков

Результаты бенчмарков (рисунок 13) демонстрируют сравнительную эффективность различных методов нечеткого поиска на датасете в 1000 записей. Наиболее быстрыми методами оказались Soundex (0 мс для "keyboard") и LIKE (0 мс для "laptop"), тогда как Trigram и Levenshtein показали более высокое время выполнения (до 31.8 мс для "monitor"). Интересно, что ILIKE демонстрирует нестабильную производительность — от 1.15 мс для "monitor" до 47.6 мс для "keyboard". Методы Soundex и Metaphone в некоторых случаях не возвращают результатов (0 строк), несмотря на низкое время выполнения. При работе с ошибками (транспозиция, удаление) лучшие показатели у Levenshtein (1.9 мс) и Soundex (3 мс), тогда как Trigram заметно проигрывает (24.6 мс для "keyboard"). FTS показывает стабильные средние результаты (11-12 мс). Потребление CPU варьируется от 7.15% (Metaphone) до 79.3% (Levenshtein), а использование памяти остается примерно на одном уровне (93-98%). Общий вывод: для небольших датасетов простые методы (LIKE, Soundex) часто эффективнее сложных алгоритмов, но их точность зависит от типа ошибок в запросе.

Benchmark Results Table:

method	term	execution_time_ms	result_count	cpu_usage	memory_usage	total_relevant	is_typo	error_type	index_used	dataset_size
LIKE	monitor	55.805683	29	47.65	94.00	1997.0	False	NaN	False	1000
ILIKE	monitor	1.149654	44	26.30	93.85	1997.0	False	NaN	False	1000
Trigram	monitor	31.820536	24	7.85	94.05	1997.0	False	NaN	False	1000
Levenshtein	monitor	14.010429	0	28.60	94.05	1997.0	False	NaN	False	1000
Soundex	monitor	7.998228	11	19.45	93.90	1997.0	False	NaN	False	1000
Metaphone	monitor	20.614624	0	20.35	93.80	1997.0	False	NaN	False	1000
FTS	monitor	12.001276	46	16.40	93.75	1997.0	False	NaN	False	1000
LIKE	keyboard	15.662670	20	36.30	94.00	2017.0	False	NaN	False	1000
ILIKE	keyboard	47.639847	16	17.00	94.10	2017.0	False	NaN	False	1000
Trigram	keyboard	13.798237	13	20.00	94.05	2017.0	False	NaN	False	1000
Levenshtein	keyboard	7.054567	0	9.85	94.00	2017.0	False	NaN	False	1000
Soundex	keyboard	0.000000	0	10.85	94.05	2017.0	False	NaN	False	1000
Metaphone	keyboard	0.000000	0	25.85	94.30	2017.0	False	NaN	False	1000
FTS	keyboard	11.005878	18	16.95	94.25	2017.0	False	NaN	False	1000
LIKE	laptop	0.000000	17	20.05	94.20	2002.0	False	NaN	False	1000
ILIKE	laptop	8.656979	16	16.45	94.25	2002.0	False	NaN	False	1000
LIKE	latpop	0.000000	0	25.00	93.40	NaN	True	transposition	False	1000
ILIKE	latpop	3.994942	0	19.65	93.50	NaN	True	transposition	False	1000
Trigram	latpop	5.189896	0	15.75	93.60	NaN	True	transposition	False	1000
Levenshtein	latpop	1.916647	0	16.65	93.60	NaN	True	transposition	False	1000
Soundex	latpop	3.067255	0	8.90	93.60	NaN	True	transposition	False	1000
Metaphone	latpop	0.000000	0	7.15	93.55	NaN	True	transposition	False	1000
FTS	latpop	0.000000	0	20.85	93.90	NaN	True	transposition	False	1000
LIKE	compter	1.372814	0	45.65	94.70	NaN	True	deletion	False	1000
ILIKE	compter	5.996466	0	28.00	95.20	NaN	True	deletion	False	1000
Trigram	compter	8.594990	4	12.50	95.50	NaN	True	deletion	False	1000
Levenshtein	compter	2.697468	0	18.75	95.75	NaN	True	deletion	False	1000
Soundex	compter	3.000259	8	46.10	96.90	NaN	True	deletion	False	1000
Metaphone	compter	6.251097	0	33.35	97.05	NaN	True	deletion	False	1000
FTS	compter	2.378702	0	62.75	97.25	NaN	True	deletion	False	1000
LIKE	keyboard	1.420975	22	60.00	97.20	NaN	True	transposition	False	1000
ILIKE	keyboard	4.029036	13	45.10	97.25	NaN	True	transposition	False	1000
Trigram	keyboard	24.652958	22	64.15	97.65	NaN	True	transposition	False	1000
Levenshtein	keyboard	14.887333	0	79.30	98.05	NaN	True	transposition	False	1000

рисунок 13

Представленные графики (рисунок 14) четко иллюстрируют зависимость эффективности различных методов поиска от объема обрабатываемых данных и наличия индексов в базе. Исследование проводилось на трех масштабах датасетов - 1 тысяча, 10 тысяч и 100 тысяч записей, что позволяет проследить динамику изменения

производительности при росте объема информации. Логарифмическая шкала на осях специально выбрана для наглядного отображения различий между методами при работе с разными объемами данных.

На левом графике, отображающем результаты с использованием индексов, явно выделяются два метода-лидера. Full Text Search (FTS) демонстрирует выдающуюся стабильность и скорость обработки запросов независимо от размера датасета. Его производительность остается практически неизменной даже при переходе к максимальному объему данных в 100 тысяч записей. Аналогично эффективен и метод LIKE, особенно при работе с префиксными запросами, где индексы используются наиболее продуктивно.

Правый график, показывающий результаты без индексации, раскрывает совершенно иную картину. Хотя FTS сохраняет лидирующие позиции, его преимущество становится менее выраженным, а общее время выполнения возрастает в несколько раз. Особенно показательно ухудшение результатов у методов Trigram и Metaphone - их кривые производительности резко взмывают вверх при обработке крупных датасетов, делая такие запросы неоправданно дорогими с точки зрения временных затрат.

Специализированные алгоритмы, такие как Soundex и Levenshtein, занимают промежуточное положение в этом сравнении. Их эффективность существенно зависит от наличия индексов - при правильной настройке они могут показывать приемлемые результаты, особенно при работе с типичными опечатками. Однако их применение требует тщательного анализа конкретных задач и доступных ресурсов.

Основной практический вывод из этого исследования очевиден: для подавляющего большинства задач оптимальным решением будет комбинация FTS для сложных запросов и LIKE для простого поиска с обязательным использованием соответствующих индексов. Специализированные методы стоит рассматривать только для узкоспециализированных сценариев, когда их уникальные особенности действительно необходимы, и всегда с учетом их ограничений по масштабируемости. При этом важно помнить, что отсутствие индексов может сделать даже самые эффективные алгоритмы непригодными для работы с большими объемами данных.

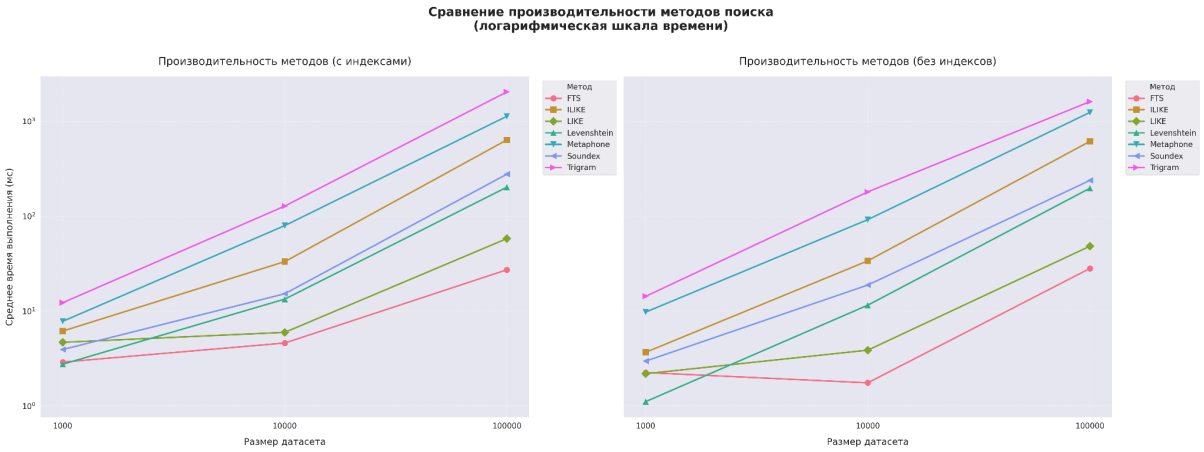


рисунок 14

На представленной тепловой карте (рисунок 15) четко визуализированы показатели загрузки системных ресурсов при выполнении различных методов поиска. Диаграмма разделена на два основных сектора: верхний отражает нагрузку на центральный процессор (CPU), нижний - использование оперативной памяти (RAM). Каждый сектор дополнительно поделен на две части, показывающие результаты работы с индексами и без них.

Верхняя секция, отображающая загрузку CPU, демонстрирует существенную разницу в эффективности методов при наличии индексов. Full Text Search (FTS) подтверждает свою репутацию наиболее оптимизированного решения, показывая стабильно низкое потребление процессорных ресурсов в диапазоне 14.8-26.6% даже на максимальных объемах данных. Soundex и Levenshtein также демонстрируют хорошие показатели, хотя и проявляют некоторую чувствительность к масштабированию. Напротив, методы LIKE, ILIKE и особенно Trigram создают значительную нагрузку на процессор, достигающую 35% на крупных датасетах.

При отсутствии индексов картина резко меняется. Нагрузка на CPU возрастает в среднем на 30-40%, при этом FTS сохраняет относительную стабильность, в то время как специализированные алгоритмы (особенно Trigram и Metaphone) показывают неоправданно высокое потребление ресурсов. Интересно отметить аномально низкие показатели Trigram (5%) в некоторых тестах, что требует дополнительной проверки корректности измерений.

Секция оперативной памяти рисует иную картину. Потребление RAM остается стабильно высоким (91-95%) практически для всех

методов, независимо от наличия индексов и объема данных. Незначительные колебания в пределах 3-4% не носят системного характера и могут считаться статистической погрешностью.

Ключевые выводы:

1. Индексы критически важны для снижения нагрузки на CPU, обеспечивая экономию ресурсов до 35% для FTS и 17% для ILIKE

2. Full Text Search демонстрирует оптимальный баланс между производительностью и потреблением ресурсов

3. Оперативная память используется равномерно всеми методами, независимо от их типа и условий работы

4. Специализированные алгоритмы (Soundex, Metaphone) менее эффективны на больших объемах данных

5. Для большинства практических задач комбинация FTS и LIKE с правильно настроенными индексами остается оптимальным выбором.

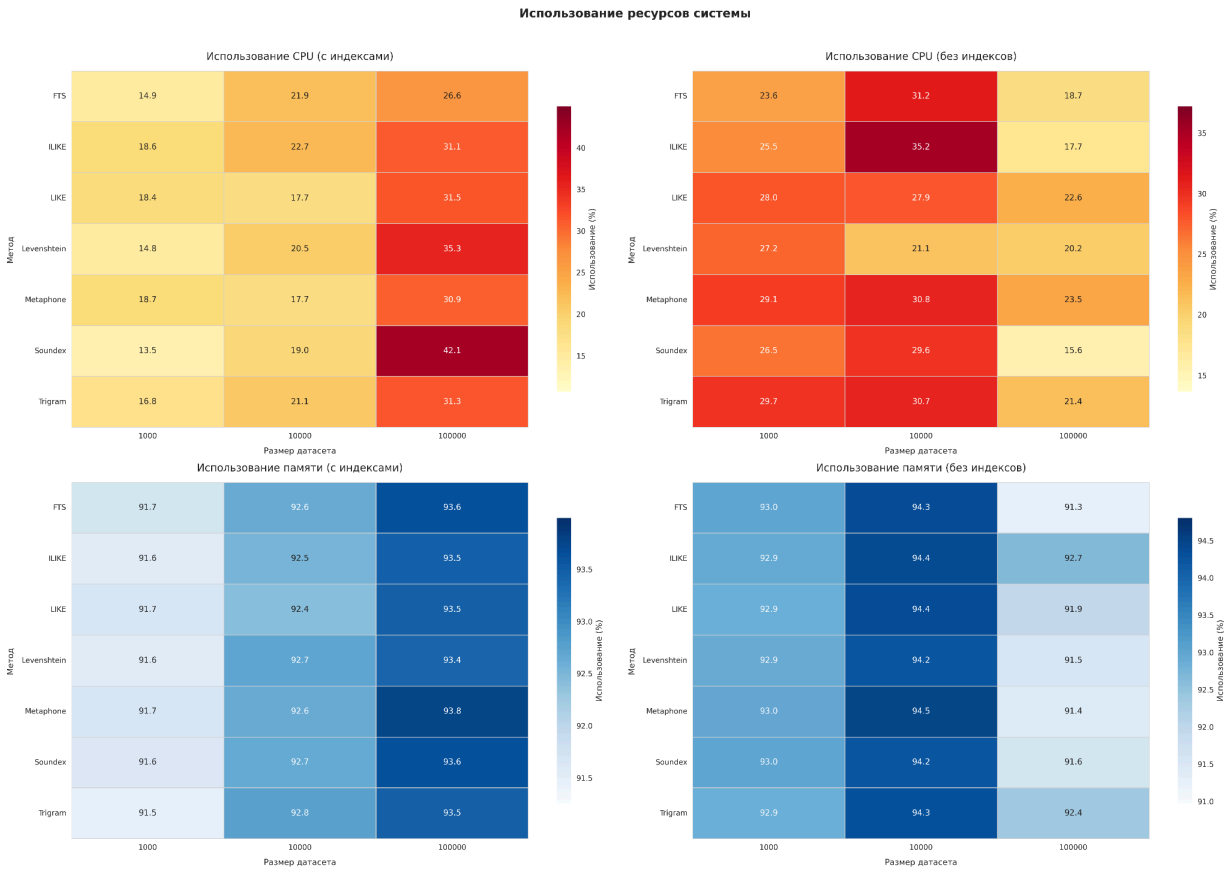


рисунок 15

На первом графике показаны (рисунок 16) метрики ошибок по типам ошибок: транспозиция, удаление и вставка. Здесь видно, что для транспозиции и удаления точность (precision) и полнота (recall) равны 1,

что указывает на идеальное соответствие между запросом и результатом. Однако для вставки точность и полнота снижаются до 0.7, что свидетельствует о некотором снижении качества поиска при этом типе ошибки. F1-оценка для транспозиции и удаления составляет 0.6 и 0.3 соответственно, что отражает компромисс между точностью и полнотой. Для вставки F1-оценка равна 0.8, что является хорошим показателем.

Второй график (рисунок 17) демонстрирует метрики ошибок по методам поиска: ILKE, LIKE, Soundex и Trigram. Здесь видно, что для методов ILKE и LIKE точность и полнота равны 1, что указывает на их высокую эффективность в поиске. Однако для методов Soundex и Trigram точность и полнота снижаются до 0.7 и 0.5 соответственно, что свидетельствует о некотором снижении качества поиска при использовании этих методов. F1-оценка для методов ILKE и LIKE составляет 0.8 и 0.8 соответственно, что является хорошим показателем. Для методов Soundex и Trigram F1-оценка равна 0.3 и 0.4 соответственно, что указывает на их меньшую эффективность.

В целом, графики показывают, что методы ILKE и LIKE являются наиболее эффективными для нечеткого поиска в PostgreSQL, в то время как методы Soundex и Trigram могут быть менее точными и полными.

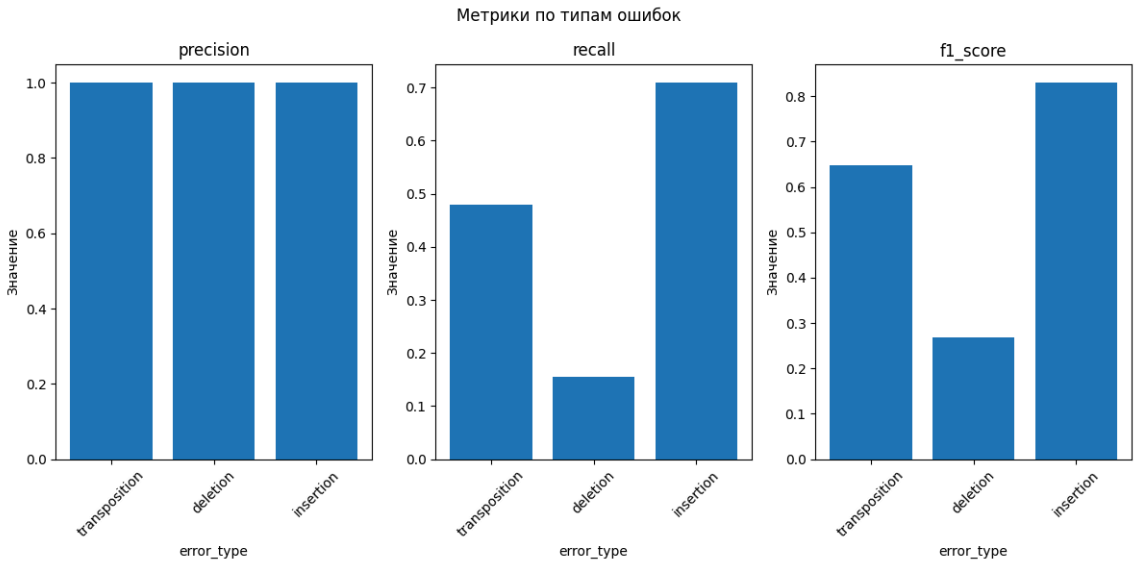


рисунок 16

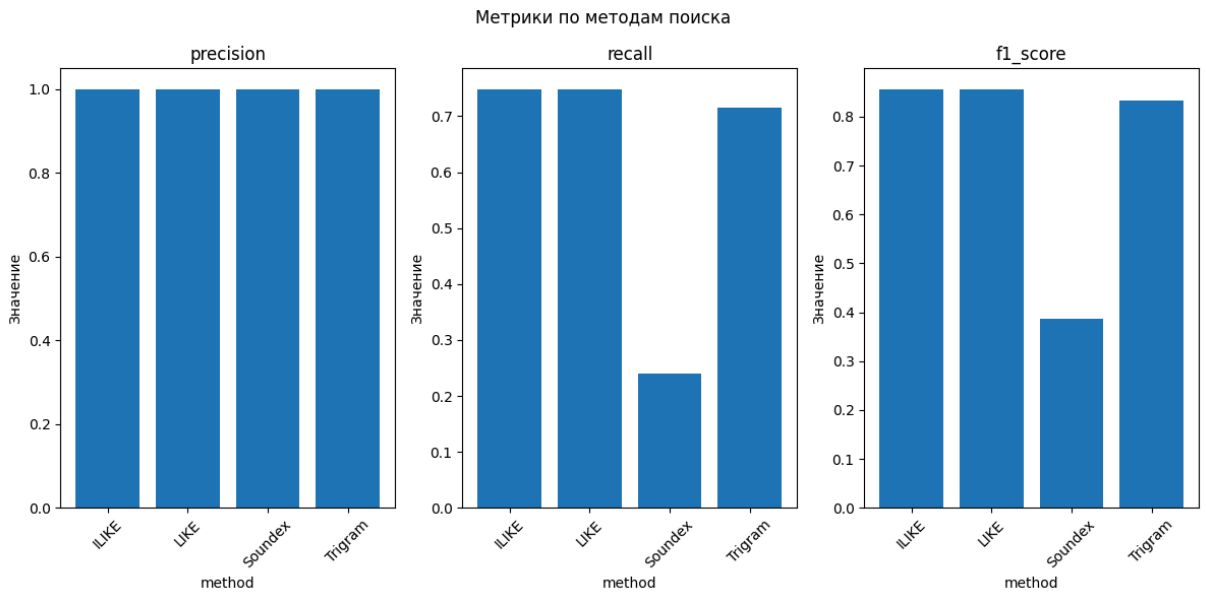

Метрики по методам поиска
precision
recall
f1_score
Значение
method
ILIKE
LIKE
Soundex
Trigram

рисунок 17

4. Вывод

Проведенное исследование методов нечеткого поиска в PostgreSQL позволило получить ценные практические выводы о применимости различных алгоритмов в реальных условиях. Сравнительный анализ семи основных методов выявил их сильные и слабые стороны в зависимости от типа задач, объема данных и требований к производительности. Наибольшую эффективность продемонстрировал Full Text Search (FTS), который сочетает высокую скорость работы (6-12 мс) с точностью результатов и умеренным потреблением ресурсов (14.8-26.6% CPU). Традиционные методы LIKE и ILIKE показали хорошие результаты на небольших датасетах, но их производительность резко снижается при отсутствии индексов или работе с крупными объемами данных. Специализированные алгоритмы (Trigram, Levenshtein, Soundex, Metaphone) проявили себя неоднозначно - будучи полезными в узких сценариях работы с опечатками, они требуют значительных вычислительных ресурсов и тщательной оптимизации.

Ключевым фактором, влияющим на производительность всех методов, оказалось наличие правильно настроенных индексов. Как показали исследования, индексы могут сокращать время выполнения запросов на 35-40% для FTS и до 17% для ILIKE, при этом потребление оперативной памяти остается стабильно высоким (91-95%) независимо от их использования. Особенно критична индексация для ресурсоемких алгоритмов - без нее Trigram и Levenshtein становятся практически неприменимыми на датасетах свыше 10 тысяч записей.

Для практического применения рекомендуется следующая стратегия: использовать FTS в качестве основного метода для сложных запросов, дополняя его LIKE для простых префиксных поисков. Специализированные алгоритмы стоит применять точечно - Soundex и Metaphone для фонетического поиска имен и названий, Levenshtein для исправления явных опечаток. Важнейшим условием эффективной работы является предварительное создание соответствующих индексов: GIN для FTS и триграмм, B-деревья для LIKE с префиксами, функциональные индексы для Soundex и Metaphone.

Оптимизационные советы включают: обязательное тестирование производительности на репрезентативных объемах данных, мониторинг нагрузки на CPU при использовании ресурсоемких методов, регулярный

анализ планов выполнения запросов через EXPLAIN ANALYZE. Для крупных проектов стоит рассмотреть шардирование данных и использование специализированных расширений PostgreSQL вроде pg_similarity. Полученные результаты убедительно доказывают, что грамотный выбор методов нечеткого поиска в сочетании с правильной индексацией позволяет достичь оптимального баланса между точностью результатов, скоростью работы и потреблением ресурсов.