Введение

Пользователи постоянно что-то ищут в программах и на сайтах, но часто ошибаются: опечатываются, путают раскладку или не знают точного названия. Стандартный поиск, требующий точного совпадения, в таких ситуациях бесполезен — он просто ничего не находит. Чтобы система стала более гибкой и находила нужное даже с ошибками, существуют методы нечеткого поиска (fuzzy search). Это задание посвящено практической проверке таких методов в PostgreSQL. Протестирую их на данных разного объема, замерим скорость работы и проверим, насколько хорошо они справляются с распространенными ошибками. Результатом станут конкретные цифры и советы по применению.

Нечеткий поиск в PostgreSQL

1. Теоретическая часть

1.1. Принципы работы методов нечеткого поиска

1.1.1. Like с wildcard-символами

Текущий алгоритм c wildcard-символами применяется при сравнении текстовых строк, которые в свою очередь могут содержать различные wildcards(подстановочные знаки) такие как %, \_, /.

Знак % означает пропуск как минимум одного символа или может являться пустой строкой.

Знак \_ означает пропуск только одного символа.

Знак \ отвечает за экранирование символов.  
 Наиболее частным местом применения алгоритма может служить обычная командная строка Windows, но также используется в файловых менеджерах, различных интерфейсах поисковых систем и баз данных.

Данные оператор также можно представить в виде автоматов и в виде некого формального языка.

Принцип работы:  
Допустим есть строка apple, записать ее в шаблон LIKE можно, например, как:

1. ‘apple’ LIKE ‘ap%le’
2. ‘apple’ LIKE ‘apple%’
3. ‘apple’ LIKE ‘app%e’
4. ‘apple’ LIKE ‘ap\_le’
5. ‘ap\_le’ LIKE ‘ap\\_le’

Важно заметить, что A!

Но если пробовать перевести обратно можно заметить, что заменить уже можно на какие угодно символы.

Алгоритмическая сложность:

Наиболее большую сложность представляет знак %, например возьмем длину строки в 50(n), а длину шаблона в 10(m) и добавим количество знаков % = 6(k)

Получаем: 50 х 10 = 500 операций

Но присутствие k добавляет ветвление и по итогу получаем количество операций при худшем сценарии.

Применение:  
Примером может служить: номер документа ‘\_-\_\_-\_\_\_’, модель техники Model-N\_\_\_\_, выражение в процентах Exp \_\_\%

Поиск по суффиксу или префиксу: Book%, %List

1.1.2. ILIKE

ILIKE – специализированный оператор SQL для поиска по шаблонам, причем являющегося регистронезависимым.

ILIKE расширяет «возможности» LIKE: теперь есть игнорирование разницы строчных или приписных букв, то есть A=a, а также замена символов буквенными обозначениями в разных языках.

ILIKE также являет «эксклюзивом» для Postgre SQL.

Принцип работы: в основе лежит логика LIKE, но уже регистронезависимая, то есть всё что срабатывало для LIKE в ILIKE тоже будет корректно, но вот что нового позволяет делать эта версия:

1. ‘Apple’ ILIKE ‘apple’
2. ‘APplE’ ILIKE ‘AP%E’

Алгоритмическая сложность:

Допустим возьмем такую же длину строки в 50(n), а длину шаблона в 10(m) и

Получаем: 50 х 10 = 500 операций

Но также имеются дополнительные затраты, в ILIKE используется функция lower() которая при встрече буквы, приводит ее к нижнему регистру.

Также имеют место быть международные символы. Точно оценить нагрузку сложно.

Применение: поиск товаров, например, модели телефона – можно записать исходя из правил и вывод будет всех моделей, независимо от регистров.

1.1.3. Расширение pg\_trgm(триграммы)

Триграммы – мощный инструмент, благодаря которому Postgre SQL превращается в полноценную поисковую систему из-за поддержки нечеткого сопоставления. Можно сказать, что большой процент «человеческого фактора» исключается т.к. опечатка уже слабо влияет на итог.

Принцип работы:  
Триграмма разбивает слово на некое множество из трех символов. Но перед этим следует поставить 2 пробела перед словом и 1 после, для учета начало и окончания слова.

Пример со словом ‘apple’: Запрос(слово) разбивает на (‘a’, ‘ap’, ‘app’, ‘ppl’, ‘ple’, ‘le’). И допустим человек ищет ‘app’ – также разобьет на символы (‘a’, ‘ap’, ‘app’, ‘pp’). Ищем общие, их будет 3.

Далее по формуле Жаккара: совпадающие триграммы/общее количество разбиений исключая совпадающие, в данном случае получим .

Алгоритмическая сложность:

O(n) где n это длина строки, как можно заметить количество триграмм всегда равно n+1, в слове apple n=5, количество триграмм 6.

O(1) является упрощением т.к. поиск по индексу зависит от количества триграмм в запросе и размера инвертированного списка для каждой триграммы. То есть каждая триграмма создается за «счёт» O(1).

GIN это инвертированный индекс. Для каждой триграммы хранится список строк, в которых эта триграмма встречается. Считается что для каждой триграммы в запросе доступ к инвертированному списку происходит за O(1)

Если использовать битовые карты, то объединение для некого количества(x) триграмм может быть O(x), но x ограничено длиной запроса и не зависит от размера таблицы. Именно поэтому O(x)=O(1) относительно n.

Случаи применения:

Случай опечаток: apple(‘a’, ‘ap’, ‘app’, ‘ppl’, ‘ple’, ‘le’) и aple (‘a’, ‘ap’ , ‘apl’ , ‘ple’ , ‘le’) теряется только 2 триграммы и получаем совпадение в

Также могут быть случаи: перестановки символов, добавление лишнего символа, замены символа.

В жизни таким случаем может быть например поиск телевизора Samsung, но запрос будет выглядеть как Smsung.

1.1.4. Levenshtein distance

Расстояние Левенштейна — это алгоритм для измерения разницы двух строк, которая показывает минимальное количество действий необходимых для преобразования одной строки в другую. Эти операции включают в себя: добавление символа, удаление символа, замена символа.

Принцип работы:

Допустим слово ‘яблоко’ нужно преобразовать в ‘локация’. Первое это удаление ‘я’ и ‘б’ (2 операции), получаем ‘локо’. Далее идёт замена второго ‘о’ на ‘а’ (1 операция). В конце добавляем ‘ц’, ‘и’, ‘я’ (3 операции). Получаем что слово ‘яблоко’ преобразовано в слово ‘локация’ за 6 операций.

Алгоритмическая сложность:

Если переходить на программирование в начале создается матрица (n+1)x(m+1) ячеек (n, m длина первой и второй строки соответственно) за эти следует уже логика описанная ранее.

d[i][j] = min(

d[i-1][j] + 1, // удаление

d[i][j-1] + 1, // вставка

d[i-1][j-1] + cost // замена (cost=0 если символы равны, иначе 1)) , где d[i][0] = i (удаление i символов), аd[0][j] = j (вставка j символов)

Сложность в целом заключается в размере слов и в их различии(что и так понятно), время преобразования вычисляется как произведение количества символов первого слова на количество символов второго слова.

Случаи применения: примером может служить автозамена или исправление ошибок в программа, то есть при написании слова происходит его сравнение со словарем и предложение наиболее подходящих слов к написанному. Яблоко – можно увидеть яблочный сок и т.п. Автодополнение слов.

1.1.5. Soundex и Metaphone

Данный метод больше походит на кодирование, в котором каждой букве есть аналог в цифровом виде.

Принцип работы:

Первая буква слова всегда остается как основа для дальнейшей кодировки. Все остальные буквы преобразуются цифры согласно таблице:

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

При этом все гласные, также H и W игнорируются.

Всё дело в диалекте и влиянии на соседние звуки, они могут искажать смысл.

Также следует заметить, что последовательные цифры объединяются и код имеет вид: Первая буква + 3 цифры, но, если цифр менее трех добавляются нули.  
Пример: ‘Heavy’ = H100

Алгоритмическая сложность:

Программирование происходит по алгоритму: Словоего инициализациязапись первого символапосимвольное кодирование пока слово не окончитсяпроверка на количество символов(помним что буква+3 цифры) при не хватке добавление нулей.

Случаи применения:

Поиск похожих по звучанию слов или имен/фамилий.  
Алёна и Алена в документах.

Мюнхен в различных языках звучит по-разному из-за гласных.

1.1.6. Full-text search (FTS)

Полнотекстовый поиск — это достаточно сложный процесс, состоящий из большого количества шагов и правил.

Принцип работы:

Текст разбивается на слова(токены) и все слова приводятся к нижнему регистру, при это союзы и предлоги игнорируются(удаляются). Пример: A dog runs in the fielddog run field

Также происходит редукция слов к основе: Пример: runsrun

И так называемая лемматизация: micemouse

Изначальный текст: A dog runs in the field подвергается нумерации его составляющих, A(1) dog(2) и т.п. После всех преобразований оставшиеся слова записываются в алфавитном порядке: dogfieldrun и уже подставляют его изначальный номер: dog(2) field(6) run(3).

Алгоритмическая сложность:

O(log n) с GIN индексом: Каждой лексеме сопоставлен список позиций, все лексемы хранятся в дереве и поиск необходимой лексемы: O(log m), где m-число уникальных лексем.

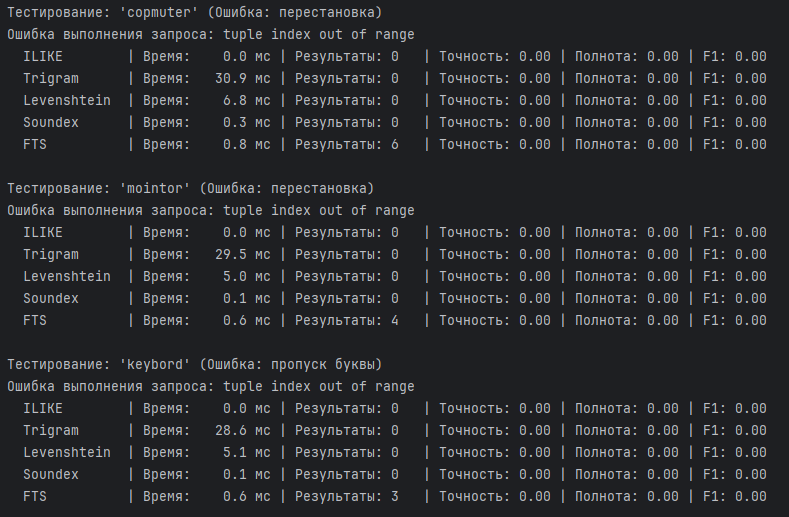
Случаи использования: Поиск информации по огромным базам данных компаний, поиск информации в исторических документах. Много положительных сторон одни из которых возможность поиска по синонимам и по весу документа.

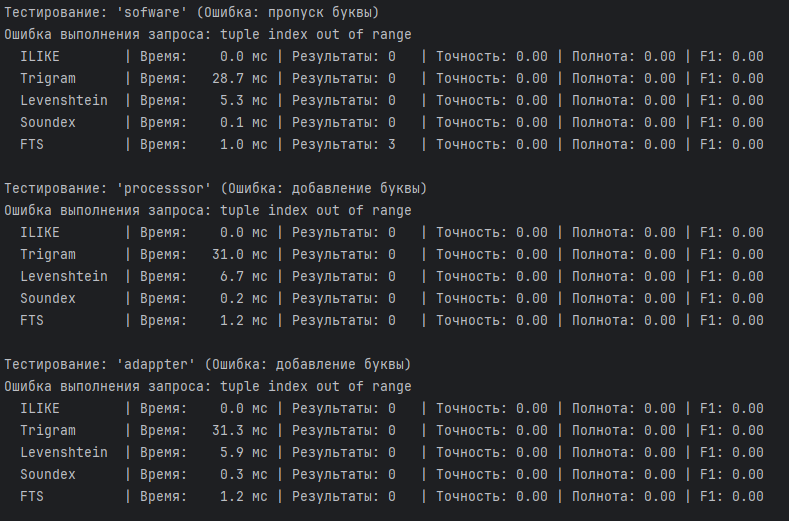
1.1.7. Similarity функции из pg\_trgm

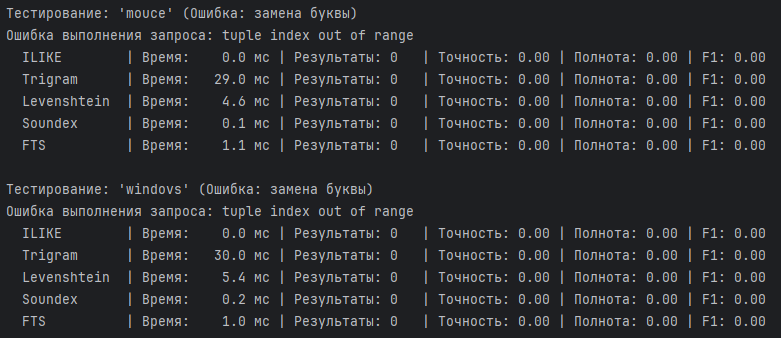
Возможно, я что-то не понял или не нашел, но основная часть этого расширения уже была разобрана ранее. Могу только добавить некоторые функции, которые дополнительно нашел. Например, иная запись алгоритма.

Функция коэффициента Жакара: , где A и B – множества триграмм для двух строк, пересечение множеств(совпадающие общие триграммы), – объединение множеств( все триграммы в двух строках), а модуль показывает мощность множества(количество элементов).

Эффективность намного выше чем в алгоритме Левенштейна.

Итоги работы программы:  






К сожалению, в конце работы появилась ошибка, которую не смог вовремя исправить.

Возможно это связано с малым количество генерации и мне не повезло что опечаток не было почти, но вряд ли.  
  
Сравнительная характеристика методов текстового поиска

| Метод | Сложность с индексом | Тип индекса | Преимущества | Ограничения |
| --- | --- | --- | --- | --- |
| LIKE / ILIKE | O(log N) | B-Tree | Быстрое выполнение префиксных запросов, интуитивный синтаксис. | Низкая эффективность при поиске подстрок. |
| Расстояние Левенштейна | Не поддерживается | — | Точный расчет редакторской дистанции между строками. | Высокие вычислительные затраты, непригодность для больших объемов данных. |
| Soundex / Metaphone | O(log N) | B-Tree | Успешное распознавание фонетически близких слов (имена, топонимы). | Сниженная точность для неанглоязычных данных. |
| pg\_trgm | ≈ O(1) | GIN | Устойчивость к распространенным опечаткам, точное ранжирование по схожести. | Слабая обработка лексем короче 3 символов. |
| Полнотекстовый поиск (FTS) | O(log N) | GIN | Лингвистический анализ с учетом морфологии языка. | Не предназначен для коррекции орфографических ошибок. |

Рекомендации по выбору метода

* Автодополнение / Фильтрация по началу строки: Оптимален ILIKE с шаблоном 'префикс%'.
* Универсальный поиск с коррекцией опечаток: Используйте pg\_trgm с GIN-индексом и финальным ранжированием через similarity().
* Поиск по фонетическому сходству (имена, названия): Примените алгоритм Metaphone.
* Семантический поиск в текстовых массивах (документы, статьи): Выбирайте Full-Text Search.
* Точное сопоставление среди похожих вариантов: Комбинируйте pg\_trgm (предварительный отбор) и levenshtein() (финальное ранжирование).

Стратегии оптимизации

1. Использование специализированных индексов: Применяйте B-Tree для префиксов и фонетики, GIN — для триграмм и FTS.
2. Комбинированный подход: Фильтруйте данные быстрыми методами (например, pg\_trgm), затем уточняйте результаты на малой выборке ресурсоемкими алгоритмами (например, Левенштейном).
3. Предобработка данных: Нормализуйте строки (приведение к нижнему регистру, удаление лишних символов) перед индексацией и поиском.