УДК 004.65 **Информационные технологии**

Виноградова М. В., *кандидат технических наук, доцент; Московский государственный технический университет им. Н.Э. Баумана*

Барашкова Е. С., *магистрант, Московский государственный технический университет им. Н.Э. Баумана*

Березин И. С., *магистрант, Московский государственный технический университет им. Н.Э. Баумана*

Ореликов М. Г., *магистрант, Московский государственный технический университет им. Н.Э. Баумана*

Лузин Д. С., *магистрант, Московский государственный технический университет им. Н.Э. Баумана*

ОБЗОР СИСТЕМЫ ПОЛНОТЕКСТОВОГО ПОИСКА В ПОСТРЕЛЯЦИОННОЙ БАЗЕ ДАННЫХ POSTGRESQL

Аннотация: В статье рассмотрен механизм полнотекстового поиска в реляционной системе управления базами данных (СУБД) PostgreSQL, его отличия от стандартного поиска (поиска по шаблону) и преимущества. Так же приведены результаты экспериментов по поиску слов и словосочетаний, их обоснование и отличие в используемых методах и алгоритмах при поиске. Сделан обоснованный вывод о том, что данный подход при поиске информации в СУБД PostgreSQL можно использовать в качестве основного.

Ключевые слова: База данных, PostgreSQL, полнотекстовый поиск, индекс, информация.

Abstract: The article discusses the full text engine search in the relational database management system (DBMS) PostgreSQL, its differences from the standard search (search by pattern) and its advantages. The results of experiments to search for words and phrases, their justification and the difference in the methods and

algorithms used in the search are also provided. A reasonable conclusion is made that this approach can be used as the main one when searching for information in PostgreSQL.

Keywords: Database, PostgreSQL, full-text search, index, information.

Введение

Полнотекстовый поиск (Full Text Searching, сокр. FTS) – это способ находить *документы* на естественном языке, соответствующие исходному *запросу* пользователя, с возможностью выстраивания найденных документов с наиболее высокой или же с наиболее низкой *степенью соответствия* этих документов параметрам заданного запроса.

Для выполнения полнотекстового поиска в реляционных и объектно-реляционных базах данных используют специальные технологии и механизмы, обычно являющиеся частью СУБД. Эти технологии позволяют сократить время выполнения полнотекстовых запросов, заменив полный просмотр текстовых полей на анализ подготовленных заранее списков лексем, индексированных и нормализованных. Для эффективного использования встроенных механизмов полнотекстового поиска необходимо понимать принципы их работы, а также особенности их использования для решения конкретных задач.

В данной статье рассматриваются возможности и методы полнотекстового поиска, используемые в СУБД PostgreSQL, как широкораспростаненной кроссплатформенной объектно-реляционной СУБД с окрытым исходным кодом. Приведены результаты проведенных экспериментов по сравнению времени выполнения поисковых запросов для различных методов и алгоритмов выполнения поиска.

Полнотекстовый поиск в СУБД PostgreSQL

В системе полнотекстового поиска под документом понимается элемент текстовых данных, являющийся единицей обработки при выполении поискового запроса, например, некоторая статья или сообщение. В контексте СУБД PostgreSQL документ представляет собой содержимое текстового поля в

строке таблицы или объединение текстовых полей. Понятие запроса в самом простом варианте – это набор слов для поиска, а степень соответствия – частота повторения этих слов в документе [1].

PostgreSQL поддерживает два специальных типа данных для выполнения полнотекстового поиска: тип tsvector представляет документ в виде, оптимизированном для полнотекстового поиска, а тип tsquery - запрос на выполнение полнотекстового поиска.

Значение типа tsvector – это отсортированный список неповторяющихся *лексем*, т.е. слов, нормализованных таким образом, что все словоформы сводятся к одной. При переводе строкового значения к типу tsvector автоматически производится сортировка и исключение повторяющихся слов, как показано в примере на рисунке 1.

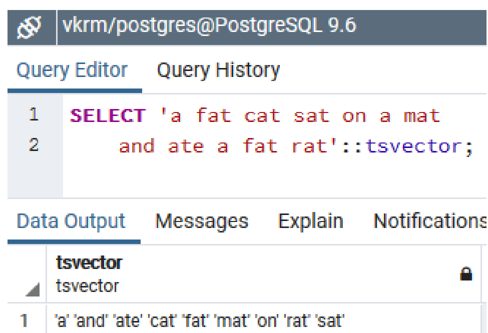

Рисунок 1 – Приведение данных к типу tsvector.

Для лексем также можно указать их целочисленные позиции, чтобы иметь информацию о расположении слов для дальнейшего ее использования при оценке близости (позиция обычно указывает положение исходного слова в документе). Если для лексем заданы позиции, то также можно назначить им и вес, выраженный буквами A, B, C или D (по умолчанию). Назначенным весам можно сопоставить числовые приоритеты в функциях ранжирования результатов.

Важно понимать, что тип tsvector сам по себе не выполняет нормализацию слов, поэтому исходный документ предварительно следует

обработать функцией to_tsvector, нормализующей слова для поиска. Пример использования функции to_tsvector приведен на рисунке 2.

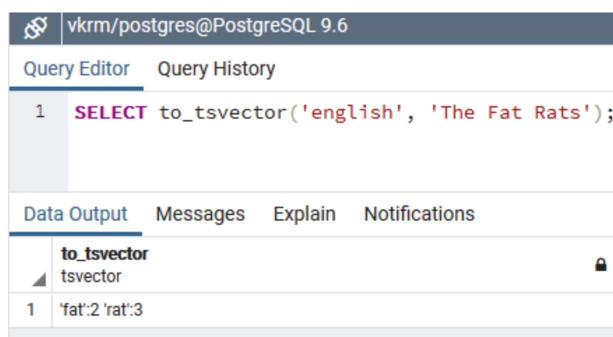

Рисунок 2 – Результат выполнения функции to_tsvector.

Функция to_tsvector вызывает анализатор, который разбивает текст документа на фрагменты и классифицирует их. Для каждого фрагмента она проверяет список словарей, определяемый типом фрагмента. Первый же словарь, распознавший фрагмент, выдаёт одну или несколько представляющих его лексем. Выбор анализатора, словарей и индексируемых типов фрагментов определяется конфигурацией текстового поиска. В одной базе данных можно использовать разные конфигурации. В рассматриваемых далее примерах использована конфигурация по умолчанию для английского языка – «english».

Первый параметр функции to_tsvector - это название конфигурации, а второй - текстовая строка для обработки.

При выполнении запроса, представленного на рисунке 2, произошло преобразование исходных слов к их исходным словоформам, например, слово «rats» преобразовалось в «rat». А также был исключен предлог «the», поскольку он не несет никакой информационной ценности. Такие слова называют стоп-словами, и функция to_tsvector исключает их и не вносит в возвращаемое ей значение.

Значение типа tsquery содержит лексемы, объединенные логическими операторами «&» (И), «|» (ИЛИ), «!» (НЕ), а также оператором поиска фраз «<->» (предшествует). Скобки, как и в математике, могут использоваться для

группировки операторов. Без них эти операторы имеют разные приоритеты, в порядке убывания: «НЕ», «ПРЕДШЕСТВУЕТ», «И», «ИЛИ».

В значениях типа tsquery для лексем можно дополнительно определить буквы весов, при этом они будут сопоставляться только тем лексемам в tsvector, которые имеют какой-либо из этих весов. Предварительная нормализация слов необходима и должна выполняться до приведения значения к типу tsquery. Для такой нормализации удобно использовать функцию to_tsquery, пример использования которой представлен на рисунке 3.

```
2  SELECT to_tsquery('Fat:ab & Cats');
```

Data Output | Messages | Explain | Notifications

	to_tsquery tsquery
1	'fat':AB & 'cat'

Рисунок 3 – Результат выполнения функции to_tsquery.

Документы можно хранить в обычных текстовых файлах в файловой системе. В этом случае база данных будет просто хранилищем полнотекстового индекса и исполнителем запросов, а найденные документы будут загружаться из файловой системы по некоторым уникальным идентификаторам. Однако, для загрузки внешних файлов требуются права суперпользователя или поддержка специальных функций, так что это менее удобно, чем хранить все данные внутри базы данных. Кроме того, когда всё хранится в базе данных, то упрощается доступ к метаданным документов при индексации и выводе результатов [1].

Для нужд полнотекстового поиска каждый документ должен быть приведён к специальному формату tsvector. Поиск и ранжирование выполняется исключительно с этим представлением документа — исходный текст потребуется извлечь, только когда документ будет отобран для вывода пользователю. Поэтому мы часто подразумеваем под tsvector документ, хотя в

действительности это тип, содержащий компактное представление всего документа.

Подготовка к выполнению экспериментов

Для исследования и анализа особенностей полнотекстового поиска создадим в СУБД PostgreSQL базу данных «vkrm» с помощью команды CREATE TABLE [2]. В ней схему с тем же именем; в схеме создадим таблицу «articles», в которой изначально будут присутствовать 3 столбца (колонки): id (номер статьи, первичный ключ, целочисленный тип данных), title (заголовок статьи, тип данных varchar – текст с ограничением по длине, в нашем случае в 128 символов) и content (текст статьи, текстовый тип данных без ограничений на длину). На рисунке 4 представлен запрос создания описанной таблицы.

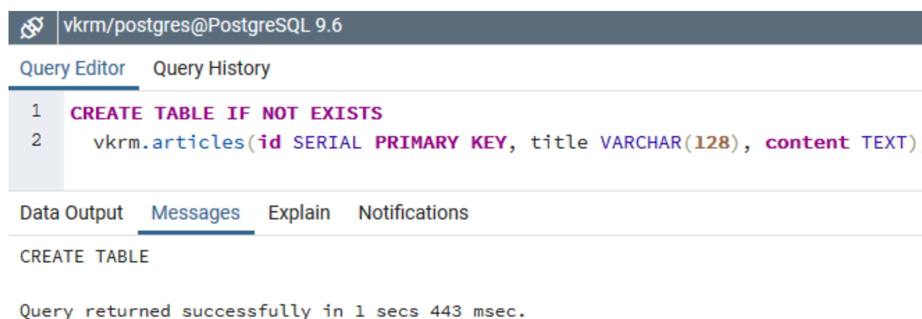

Рисунок 4 - Запрос создания таблицы «articles».

Для эксперимента попробуем сделать полнотекстовый поиск по статьям из Википедии. Для этого выгрузим дамп Википедии в созданную ранее таблицу. При написании этой статьи использовалась таблица, содержащая 3 238 504 статей, что примерно составляет 10% от дампа Википедии от июля 2019 года.

Чтобы избежать дублирования кода, создадим хранимую процедуру на диалекте PL/pgSQL. На рисунке 5 представлено создание хранимой процедуры.

Рисунок 5 – Запрос создания хранимой процедуры.

Данная хранимая процедура принимает на вход два аргумента: заголовок и основной текст статьи, а возвращает соответствующий ей тип данных tsvector. Функция setweight используется, чтобы придать заголовкам статьи больший вес, чем ее содержимому. Веса потребуются для ранжирования статей при поиске. Данная процедура была объявлена как immutable, чтобы ее можно было использовать при построении индекса в дальнейшем.

Поиск по шаблону

Операторы для поиска текста существуют в СУБД много лет, с помощью них можно производить поиск текста по шаблону тремя различными способами [1]:

1) *Стандартный оператор LIKE.*
2) *Оператор SIMILAR TO.*
3) *Регулярные выражения в стиле POSIX.*

Рассмотрим эти способы подробнее.

Стандартный оператор LIKE

Оператор LIKE используется в предложении WHERE для поиска заданного шаблона в поле; возвращает true (истину), если строка соответствует заданному шаблону поиска.

СУБД PostgreSQL расширяет стандартный оператор LIKE, описанный в стандарте SQL, дополнительными операторами – ILIKE и двумя инверсными операторами NOT LIKE и NOT ILIKE от выражений LIKE и ILIKE

соответственно. Выражение ILIKE используется для поиска текста с учетом регистр-независимости текущей языковой среды.

Вышеперечисленные стандартные команды текстового поиска работают как оператор сравнения в случае, когда шаблон поиска не содержит знаков процента (%) и подчеркивания (_), тогда шаблон представляет собой в точности строку. Для того, чтобы подменить какой-либо символ, необходимо использовать подчеркивание, если же есть необходимость подменить последовательность символов (в том числе пустую), то следует использовать знак процента. Когда необходимо найти последовательность символов где-то в середине строки, в шаблоне поиска должны присутствовать знаки процента (в начале и в конце), поскольку при проверке по шаблону LIKE всегда рассматривается вся строка.

Продемонстрируем выполнение операторов LIKE. На рисунках 6 и 7 представлены запросы с оператором LIKE, которые ищут по объединению полей «title» и «content» таблицы «articles» слова, содержащие последовательность символов «friend», или же само это слово, а также последовательность символов «aegilops», или же само это слово, означающее в переводе на русский название рода травы. Одно из этих слов довольно часто употребляется и хорошо всем знакомо (слово «friend» - друг), другое же слово, очевидно, употребляется лишь в кругах специалистов по ботанике. Такой разброс с употреблением данных слов выбран не случайно, будем проводить эксперименты, чтобы увидеть, влияет ли частота повтора слова в тексте различного рода статей на время работы алгоритмов поиска в базе данных.

```
1  SELECT title FROM vkrm.articles
2      WHERE (title || ' ' || content) LIKE '%friend%';
```

Data Output | Messages | Explain | Notifications

	title character varying (128)
1	Aristotle Onassis
2	Springfield, Missouri
3	Richmond Park
4	Luis Barahona de Soto
5	Christian II of Denmark

Рисунок 6 – Запрос поиска слова «friend» с помощью оператора LIKE.

```
1  SELECT title FROM vkrm.articles
2      WHERE (title || ' ' || content) LIKE '%aegilops%';
```

Data Output | Messages | Explain | Notifications

	title character varying (128)
1	Quercus cerris
2	Almyros
3	Quercus trojana
4	Quercus pubescens
5	Aegilops

Рисунок 7 – Запрос поиска слова «aegilops» с помощью оператора LIKE.

На рисунке 8 представлен план выполнения запроса по поиску слова «friend», построенный встроенным планировщиком СУБД PostgreSQL EXPLAIN. Планировщик выполнет преобразование команды запроса в выражение реляционной алгебры, а затем выполняет оптимизацию этого выражения [3].

```
1  EXPLAIN SELECT title FROM vkrm.articles
2      WHERE (title || ' ' || content) LIKE '%friend%';
```

Data Output | Messages | Explain | Notifications

	QUERY PLAN text
1	Seq Scan on articles (cost=0.00..183239.52 rows=1036 width=21)
2	Filter: ((((title)::text || ' '::text) || content) ~~ '%friend%'::text)

Рисунок 8 – План выполнения запроса с оператором LIKE для поиска слова «friend».

Как видно из плана запроса, алгоритм поиска содержит операцию Seq Scan (при поиске слова «aegilops» будет использоваться она же), которая представляет собой последовательный перебор всех строк таблицы в поиске интересующего значения - последовательности символов «friend».

На рисунках 9 и 10 представлена статистика выполнения запросов: количество статей, в которых встречаются данные слова, и время их поиска.

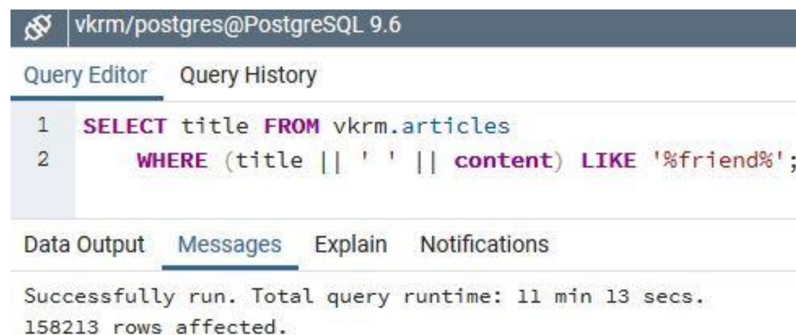

Рисунок 9 – Статистика выполнения запроса с оператором LIKE для поиска слова «friend».

Рисунок 10 – Статистика выполнения запроса с оператором LIKE для поиска слова «aegilops».

Как видно из полученных данных, для слова «friend» было найдено 158213 соответствий, а время поиска заняло 11 минут и 13 секунд; для слова «aegilops» было найдено всего 12 соответствий, а время поиска составило 11 минут и 49 секунд.

Из вышесказанного можно сделать вывод, что при использовании оператора LIKE для поиска слов неважно, как часто встречаются слова, которые мы ищем – время поиска практически одинаково, поскольку мы просматриваем каждую строку таблицы и затем выполняем поиск по шаблону.

Как видно, запросы выполнялись почти 12 минут, что неприемлемо для современных поисковых систем. Помимо слишком большого времени отклика существуют еще и другие ограничения данных операторов текстового поиска, о которых будет сказано далее.

Оператор SIMILAR TO

Оператор SIMILAR TO возвращает «истину» или «ложь» в зависимости от соответствия строки шаблону поиска. Условие SIMILAR TO истинно в случае, когда шаблон поиска соответствует всей строке.

Он аналогичен оператору LIKE, но отличается от условий с регулярными выражениями, в которых шаблон может соответствовать любой части строки. Подобно оператору LIKE, этот оператор воспринимает символ процента и знак подчеркивания. Отметим, что в регулярных выражениях POSIX им аналогичны символ точки (.) и точки со звездочкой (.*).

Оператор SIMILAR TO также поддерживает метасимволы, которые были унаследованы от регулярных выражений POSIX.

Регулярные выражения в стиле POSIX

Регулярное выражение — это совокупность символов, представляющая собой некоторое определение шаблона для выбора набора строк (регулярное множество). Как и в случае с оператором LIKE, символы шаблона поиска соответствуют символам строки, не учитывая специальных символов языка регулярных выражений. Они могут совпадать с любой частью строки, если не привязаны явно к началу и/или концу строки (в отличии от оператора LIKE) [1].

Поиск по регулярным выражениям зачастую бывает очень быстрым и эффективным. Однако, они бывают настолько сложными, что их обработка может занять большое количество времени и объёма памяти. Следовательно, лучше не использовать шаблоны регулярных выражений, поступающих из непроверенных источников. Поиск по шаблонам SIMILAR TO несёт те же риски безопасности, так как конструкция SIMILAR TO предоставляет во многом те же возможности, что и регулярные выражения в стиле POSIX.

Рассмотренным выше операторам шаблонного текстового поиска не хватает очень важных вещей, которые требуются сегодня от информационных систем, в том числе:

1. Отсутствует поддержка лингвистического функционала; возможности регулярных выражений ограничены – они не рассчитаны на работу со словоформами. Из-за этого имеется большая вероятность пропустить документы, которые содержат различные формы одного и того же слова.
2. Они не позволяют упорядочивать результаты поиска, например, по релевантности, а без этого поиск неэффективен, когда находятся сотни подходящих документов.
3. Обычно они выполняются медленно из-за отсутствия индексов, так как при каждом поиске приходится просматривать все документы.

Использование средств полнотекстового поиска

Для снятия ограничений шаблонного поиска в СУБД PostgreSQL был разработан механизм полнотекстового поиска, реализованнный на базе оператора соответствия «@@», который возвращает true, если документ (значение типа tsvector) соответствует поисковому запросу (значению типа tsquery) [1].

С применением созданной ранее функции (рисунок 5) будем производить полнотекстовый поиск по тем же словам, что и в разделе, посвященном оператору LIKE.

Следующий простой запрос, представленный на рисунке 11, выводит заголовок (title) каждой строки, содержащей слово «friend» в объединении полей «title» и «content». Выражение, которое записано между предикатом WHERE и оператором соответствия (@@), равносильно выражению, написанному ранее: «(title || ‘ ‘ || content)» – конкатенация строк.

```
1  SELECT title FROM vkrm.articles
2      WHERE vkrm.make_tsvector(title, content) @@ to_tsquery('english', 'friend')
```

Data Output | Messages | Explain | Notifications

	title character varying (128)
1	Anarchism
2	Autism
3	Alexander the Great
4	Adrian Lamo
5	Astrobiology

Рисунок 11 – Запрос для поиска слова «friend».

На рисунке 12 представлен план выполнения этого запроса.

vkrm/postgres@PostgreSQL 9.6

Query Editor | Query History

```
1  EXPLAIN SELECT title FROM vkrm.articles
2      WHERE vkrm.make_tsvector(title, content) @@ to_tsquery('english', 'friend')
3
```

Data Output | Messages | Explain | Notifications

	QUERY PLAN text
1	Seq Scan on articles (cost=0.00..976710.73 rows=16193 width=21)
2	Filter: (vkrm.make_tsvector((title)::text, content) @@ "friend"::tsquery)

Рисунок 12 – План выполнения запроса для поиска слова «friend».

Как видно из плана выполнения запроса, снова используется операция Seq Scan. На рисунке 13 представлена статистика выполнения запроса: время и количество найденых документов.

vkrm/postgres@PostgreSQL 9.6

Query Editor | Query History

```
1  SELECT title FROM vkrm.articles
2      WHERE vkrm.make_tsvector(title, content) @@ to_tsquery('english', 'friend')
```

Data Output | Messages | Explain | Notifications

```
Successfully run. Total query runtime: 1 hr 59 min.
161495 rows affected.
```

Рисунок 13 – Статистика выполнения запроса для поиска слова «friend».

Итак, нашлось соответствующих строк в количестве 161495. Напомним, что оператор LIKE вернул нам 158213 совпадений. Время выполнения запроса составило практически 2 часа, а оператор LIKE нашел их за 11 минут. Это крайне неприемлемое время выполнения поискового запроса.

Рассмотрим статистику выполнения запроса по поиску слова «aegilops». На рисунке 14 представлен сам запрос, план выполнения которого показал, что опять был проведен полный перебор записей таблицы.

Рисунок 14 – Запрос для поиска слова «aegilops».

На рисунке 15 представлена статистика выполнения запроса: время и число найденных документов.

Рисунок 15 – Статистика выполнения запроса для поиска слова «aegilops».

Были найдены 74 соответствующих строки, а оператор LIKE ранее вернул нам 12 совпадений. Время выполнения запроса составило практически 1 час 9 минут, в то время как оператор LIKE отработал за 11 минут.

Первое впечатление от инструмента полнотекстового поиска весьма негативно. Он кажется неэффективным, неприменимым в реальных проектах методом поиска информации. С использованием полнотекстового поиска мы добились большего количества совпадений в статьях, но проиграли по времени в 10 раз. Увеличение количества совпадений связано с тем, что после приведения каждого слова в тексте полей, по которым производился поиск, к одной словоформе, были обнаружены новые совпадения. Проигрыш во времени обусловлен тем, что перед непосредственным поиском было выполенено приведение слов к их словоформам.

Индексы для полнотекстового поиска

Механизм полнотекстового поиска предоставляет возможность использовать при поиске индексы, которые позволяют существенно повысить производительность работы с базой данных [2].

Создадим GIN индекс как более предпочтительный для текстового поиска. Будучи инвертированным индексом, он содержит записи для всех отдельных слов с компактным списком мест их вхождений. При поиске нескольких слов можно найти первое, а затем воспользоваться индексом и исключить строки, в которых дополнительные слова отсутствуют [1].

В нашем примере индекс создается по объединению полей «title» и «content» с помощью функции типа immutable, созданной ранее (рисунок 5), которая возвращает нормализованные формы слов с соответствующим им весами. При этом больший приоритет отдается словам в заголовках статей. Команда создания индекса приведена на рисунке 16.

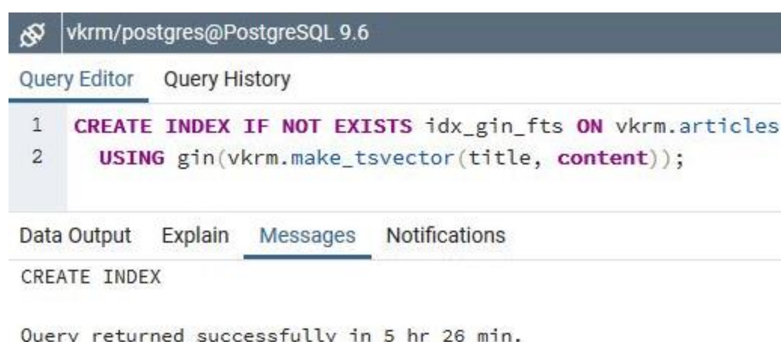

Рисунок 16 – Создание индекса GIN по двум полям.

На создание данного индекса ушло почти пять с половиной часов, а занимаемое им место на диске оказалось равным примерно 5 ГБ.

На рисунке 17 показан план выполенения запроса по поиску слова «friend» с помощью механизма полнотекстового поиска и индекса GIN.

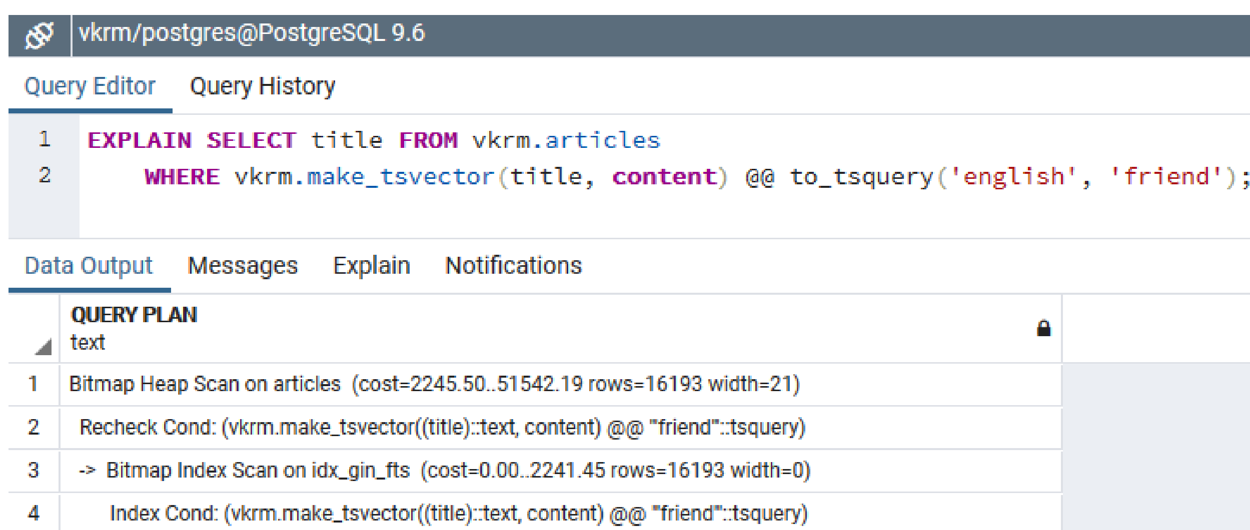

Рисунок 17 – План выполнения запроса по поиску слова «friend» с помощью индекса.

Посмотрев на план выполнения запроса, можно увидеть, что при поиске используется совокупность операций Bitmap Heap Scan, Bitmap Index Scan и Recheck Cond. Это сканирование, подразумевающее под собой использование битовых карт, когда строки выбираются одновременно все для каждого удовлетворяющего попадания, а не последовательно одна за одной в отличие от простого индексного сканирования. Изначально поиск ведется с применением индекса, а уже потом выполняется многочисленное чтение определенных данных по найденным в индексе указателям из таблицы. Помимо этого, указатели сперва ранжируются в порядке, в котором требуемые данные физически находятся в базе данных для ускорения их чтения. С этой целью и применяется битовая карта, требующая дополнительных ресурсов для своего поддержания, но компенсирующая это повышенной скоростью чтения данных[2].

Операция Recheck Cond подразумевает под собой повторную проверку условий запроса, и используется в том случае, когда размер данных в таблице очень большой, с целью сохранять актуальное состояние битовой карты. При этом будут сохранены уже не прямые указатели на строки, а указатели на страницы, в которых эти строки хранятся. Именно поэтому после первого прохода с использованием индекса необходима дополнительная проверка выбираемых данных на соответствие запросу. Приведенное сканирование — это, по сути, совокупность методов, описанных выше - последовательное выполнение операции ввода/вывода и выборка по индексу [3].

На рисунке 18 представлена статистика выполнения запроса по поиску слова «friend» с помощью механизма полнотекстового поиска и индекса GIN.

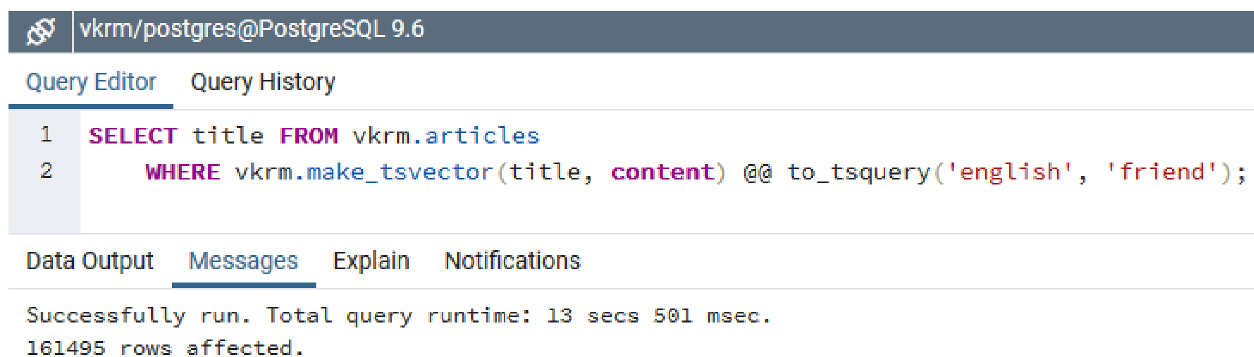

Рисунок 18 – Статистика выполнения запроса по поиску слова «friend» с помощью индекса.

На рисунке 19 представлена статистика выполнения запроса по поиску слова «aegilops» с помощью механизма полнотекстового поиска и индекса GIN. Алгоритм выполенения запроса тот же, что и при поиске по слову «friend».

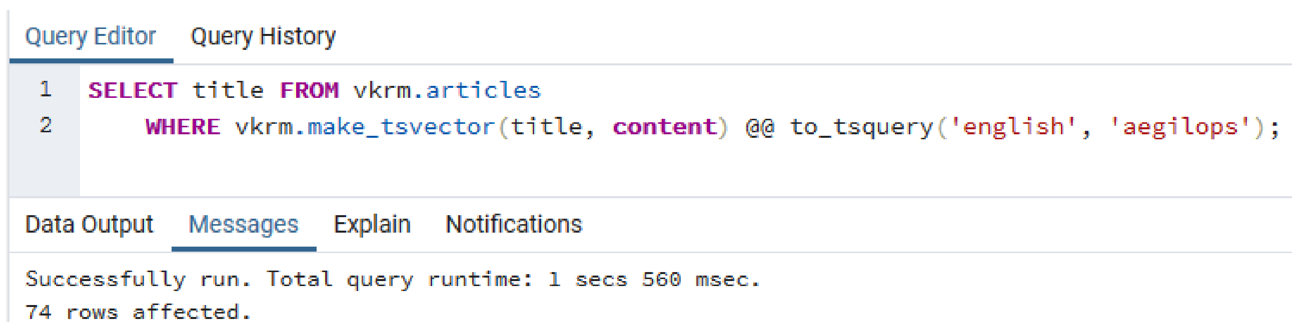

Рисунок 19 – Статистика выполнения запроса по поиску слова «aegilops» с помощью индекса.

По итогам выполнения запросов было найдено столько же совпадений, сколько и без использования индекса ранее (рисунки 13 и 15), однако время выполнения запроса заняло всего лишь 13 с половиной секунд для слова «friend» и приблизительно полторы секунды для слова «aegilops». Это несравнимо с теми результатами, что мы получали ранее. В результате можно сделать вывод, что при рассмотренном методе поиска частота нахождения слов при поиске играет весьма значимую роль.

Полнотекстовый поиск по полю типа tsvector

СУБД PostgreSQL позволяет создавать в таблицах поля типа tsvector для хранения данных, заранее подготовленных к полнотекстовому поиску. Проведем эксперименты, аналогичные рассмотренным выше, для исследования особенностей поиска по полям типа tsvector.

Пусть в таблице с документами для полнотекстового поиска имеется поле «fts_index_col» типа tsvector, хранящее в себе результат применения функции to_tsvector к объединению колонок «title» и «content». Построим индекс относительно этого поля. На рисунке 20 представлен план выполнения запроса по поиску слова «friend» по полю «fts_index_col» типа tsvector.

Рисунок 20 – План выполнения запроса по поиску слова «friend» по полю типа tsvector.

На плане видно, что применяется операция полного перебора всех строк (Seq Scan). Заметим, что при поиске слова «aegilops» она также будет применена.

На рисунках 21 и 22 представлены запросы по поиску слов «friend» и «aegilops» по полю «fts_index_col» типа tsvector соответственно.

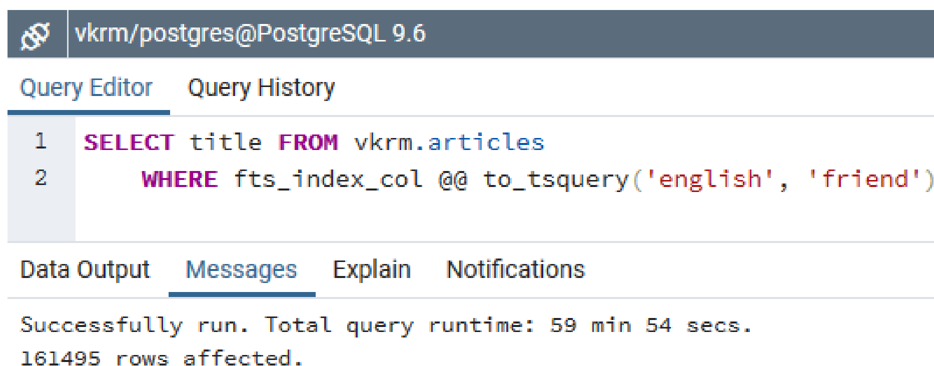

Рисунок 21 – Статистика выполнения запроса поиска слова «friend» по полю типа tsvector.

Рисунок 22 – Статистика выполнения запроса поиска слова «aegilops» по полю типа tsvector.

В результате выполнения запросов по полю типа tsvector было найдено столько же совпадений, сколько и ранее в запросах с индексом, однако, время выполнения запроса заняло целый час для слова «friend» и приблизительно 20 минут для слова «aegilops». Из этого можно сделать вывод, что в данном случае частота обнаружения искомых слов при поиске также имеет значение.

Запросы по полю типа tsvector аналогичны запросам без применения индексов, которые были представлены на рисунках 13 и 15. Время выполнения

последних из упомянутых оказалось в два раза хуже, нежели по отдельному полю, хранящему словоформы слов.

Далее рассмотрим выполнение запросов поиска по полю типа tsvector с применением индекса GIN, создав его на поле «fts_index_col». Запрос создания индекса показан на рисунке 23.

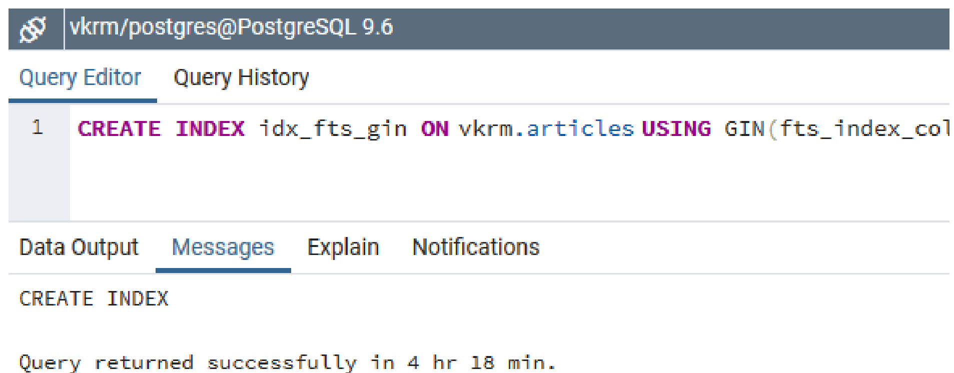

Рисунок 23 – Создание индекса GIN по полю «fts_index_col».

На построение индекса GIN по полю «fts_index_col» ушло 4 часа 18 минут, что несколько меньше, чем на построение аналогичного индекса на объединении полей.

На рисунке 24 показан план выполнения запроса по поиску слова «friend» с помощью механизма полнотекстового поиска по полю «fts_index_col» и индекса GIN. Аналогичный алгоритм используется при поиске слова «aegilops».

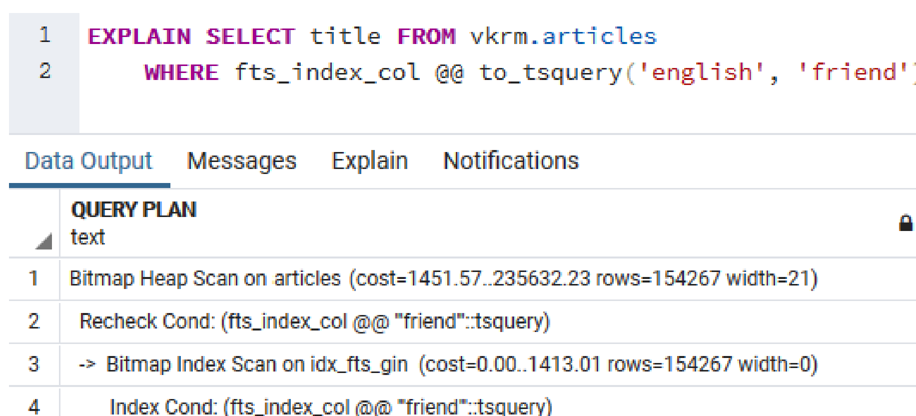

Рисунок 24 – План выполнения запроса по поиску слова «friend» по полю типа tsvector с использованием индекса GIN.

На рисунках 25 и 26 показана статистика выполнения запросов по поиску ранее приведенных слов с помощью механизма полнотекстового поиска по колонке «fts_index_col» и индекса GIN.

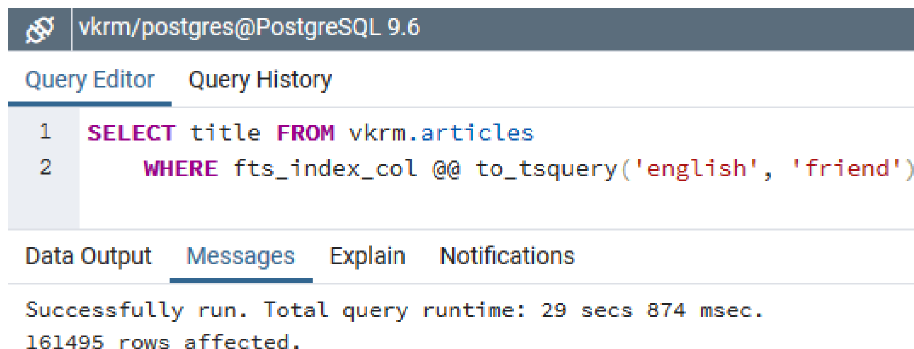

Рисунок 25 – Статистика выполнения запроса по поиску слова «friend» по полю типа tsvector с использованием индекса GIN.

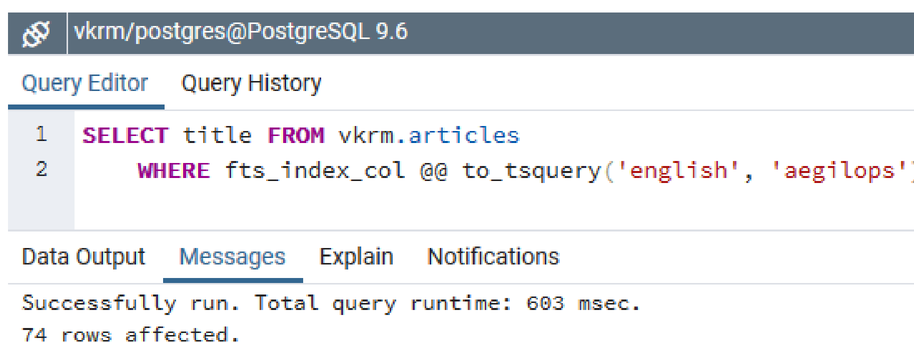

Рисунок 26 – Статистика выполнения запроса по поиску слова «aegilops» по полю типа tsvector с использованием индекса GIN.

В результате выполнения запросов было найдено столько же совпадений, сколько и ранее в запросах без шаблонного поиска, однако, время выполнения запроса заняло почти 30 секунд для слова «friend» и приблизительно половину секунды для слова «aegilops». Из этого можно сделать вывод, что в данном случае частота нахождения слов при поиске также имеет значение.

В итоге делаем вывод, что последний из рассмотренных методов поиска приводит к дополнительному расходу дискового простанства на хранение

подготовленных данных, но позволяет быстрее прозводить поиск при отсутствии индексов.

Дополнительные возможности полнотекстового поиска

Система полнотекстового поиска в СУБД PostgreSQL предоставляет возможность задания сложных условий с помощью логических операторов «И», «ИЛИ»; оценки релевантности и ранжирования полученных результатов поиска; выделения полученных результатов поиска [1].

Далее рассмотрим запрос, который демонстрирует дополнительные возможности механизма полнотекстового поиска в СУБД PostgreSQL. В запросе, представленном на рисунке 27, выполняется поиск заголовков статей для строк, содержащих в полях «title» или «content» текст в соответствии с заданным условием, содержащим логические операции сравнения. Выполняем поиск слова «neutrino» или (|) словосочетания «dark и (&) matter», отсортированных по релевантности и с выделением результатов поиска с помощью функции ts_headline.

```
1 SELECT ts_headline(title, query),
2   ts_rank_cd(vkrm.make_tsvector(title, content), query) AS rank
3     FROM vkrm.articles, to_tsquery('neutrino|(dark & matter)') query
4       WHERE query @@ vkrm.make_tsvector(title, content)
5         ORDER BY rank DESC;
```

Data Output | Messages | Explain | Notifications

	ts_headline text	rank real
1	Neutrino	109.681
2	Dark matter	97.8164
3	Neutrino oscillation	80.2
4	Sterile neutrino	50.48
5	Neutrino detector	45.4
6	Weakly interacting massive particles	44.5137
7	IceCube Neutrino Observatory	44.2618
8	Super-Kamiokande	38.8
9	Dark matter in fiction	36.7233
10	Sudbury Neutrino Observatory	27

Рисунок 27 – Дополнительные возможности системы полнотекстового поиска.

Релевантность в приведенном выше запросе оценивает функция ts_rank_cd, вычисляющая плотность покрытия подобно функции ts_rank, и

учитывающая близость соответствующих лексем друг к другу. Функция ts_headline по умолчанию выделяет искомые результаты поиска в тег «b», но тег можно изменить по своему усмотрению.

Помимо того, что с помощью средств полнотекстового поиска можно производить быстрый поиск информации в полном ее объеме благодаря использованию индексов; выделять результаты поиска для удобства представления; сортировать результаты поиска по релевантности, необходимо отметить еще одну важную особенность - использование словаря синонимов, с помощью которого можно находить еще больший объем информации за счет заложенных в этот словарь синонимов к каждому слову. Словари позволяют управлять нормализацией фрагментов с большой гибкостью: указывать стоп-слова, которые не будут индексироваться, сопоставлять синонимы с одним словом при поиске.

Также можно использовать тезаурусы, с помощью которых возможно заменять исключаемые слова или словосочетания предпочитаемыми пользователем, сопоставлять словосочетания с одним словом или различными склонениями слова, а также хранить исходные слова для индексации [1].

Сравнение методов полнотекстового поиска

В таблице 1 приведены сводные результаты экспериментов по рассмотренным в статье методам полнотекстового и шаблонного поиска, где под двумя буквами «П» идущих подряд понимается – «полнотекстовый поиск». Для каждого метода указано количество найденных строк для поисковых запросов по часто (friends) и редко (aegilops) встречающимся в тексте словам.

Таблица 1 – Количество найденных записей

Поиск по слову	Поиск по шаблону (LIKE)	ПП без индекса (Seq Scan)	ПП с индексом (битовая карта)	ПП без индекса по полю типа tsvector (Seq Scan)	ПП с индексом по полю типа tsvector (битовая карта)
friend	158213	161495	161495	161495	161495
aegilops	12	74	74	74	74

В таблице 2 приведены сводные результаты экспериментов по времени выполнения запросов при поиске слов различными методами поиска.

Таблица 2 – Время исполнения поисковых запросов

Поиск по слову	Поиск по шаблону (LIKE)	ПП без индекса (Seq Scan)	ПП с индексом (битовая карта)	ПП без индекса по полю типа tsvector (Seq Scan)	ПП с индексом по полю типа tsvector (битовая карта)
friend	11 минут 13 секунд	1 час 59 минут	13,5 секунд	59 минут 54 секунд	29,9 секунд
aegilops	11 минут 49 секунд	1 час 9 минут	1,5 секунд	20 минут 11 секунд	0,6 секунд

На основании проведенных экспериментов были сделаны следующие выводы по применению полнотекстового поиска:

- применение полнотекстового поиска дает возможность находить более полный список документов, содержащих в себе слова, по которым проводился поиск, чем применение поиска по шаблону;
- применение полнотекстового поиска без построения индексов не эффективно;
- индексы значительно сокращают время поиска, но занимают дополнительное пространство на диске и на их построение или обновление уходит значительное время.

Выбор конкретного метода полнотекстового поиска должен выбираться исходя из объема базы данных, частоты ее обновления, требований к времени отклика и типовых поисковых запросов.

Заключение

В статье дан обзор основных возможностей полнотекстового поиска в СУБД PostgreSQL, приведено описание основных механизмов его работы и рассмотрены методы по его практическому применению. Проведены эксперименты, позволяющие оценить результаты поиска и время выполнения поисковых запросов для различных методов полнотекстового поиска и поиска по шаблону с применением индексов и без них.

Сделаны выводы об эффективности применения индексов, а также предварительно подготовленных данных при полнотекстовом поиске.

Библиографический список:

1. Официальная русскоязычная документация по PostgreSQL [Электронный ресурс]: – Режим доступа: https://postgrespro.ru/docs/postgresql/9.6/textsearch (дата обращения 20.04.2020).

2. Моргунов Е.П. под ред. Рогова Е.В., Лузанова П.В. PostgreSQL. Основы языка SQL. - Спб.: Изд-во «БХВ-Петербург», 2018. – 336 с.

3. Григорьев Ю.А., Плутенко А.Д., Плужникова О.Ю. Реляционные базы данных и системы NoSQL: учебное пособие. - Благовещенск: Амурский гос. ун-т, 2018. – 424 с.