Теория Формальных Языков

Доклад

Средства работы с текстовой информацией PostgreSQL

Мамаев А. А.
Пичугин В. Е.
ИУ9-52Б

1. СУБД PostgreSQL

PostgreSQL (произносится «Пост-Грэс-Кью-Эл») — свободная объектно-реляционная система управления базами данных (СУБД).

Существует в реализациях для множества UNIX-подобных платформ, включая AIX, различные BSD-системы, Linux, macOS, Solaris/OpenSolaris, а также для Microsoft Windows.

PostgreSQL создана на основе некоммерческой СУБД Postgres, разработанной как open-source проект в Калифорнийском университете в Беркли. К разработке Postgres, начавшейся в 1986 году, имел непосредственное отношение Майкл Стоунбрейкер, руководитель более раннего проекта Ingres. Название расшифровывалось как «Post Ingres», и при создании Postgres были применены многие ранние наработки.

2. Ограничения
3. Текстовые типы данных
4. Строки ограниченной переменной длины

SQL определяет два основных символьных типа: character varying(n) и character(n), где n — положительное число. Оба эти типа могут хранить текстовые строки длиной до n символов (не байт). Попытка сохранить в столбце такого типа более длинную строку приведёт к ошибке, если только все лишние символы не являются пробелами (тогда они будут усечены до максимально допустимой длины). (Это несколько странное исключение продиктовано стандартом SQL.) Если длина сохраняемой строки оказывается меньше объявленной, значения типа character будут дополнятся пробелами; а тип character varying просто сохранит короткую строку.

При попытке явно привести значение к типу character varying(n) или character(n), часть строки, выходящая за границу в n символов, удаляется, не вызывая ошибки. (Это также продиктовано стандартом SQL.)

Записи varchar(n) и char(n) являются синонимами character varying(n) и character(n), соответственно. Записи character без указания длины соответствует character(1). Если же длина не указывается для character varying, этот тип будет принимать строки любого размера. Это поведение является расширением PostgreSQL.

Для хранения короткой строки (до 126 байт) требуется дополнительный 1 байт плюс размер самой строки, включая дополняющие пробелы для типа character. Для строк длиннее требуется не 1, а 4 дополнительных байта. Система может автоматически сжимать длинные строки, так что физический размер на диске может быть меньше. Очень длинные текстовые строки переносятся в отдельные таблицы, чтобы они не замедляли работу с другими столбцами. В любом случае максимально возможный размер строки составляет около 1 ГБ. (Допустимое значение n в объявлении типа данных меньше этого числа. Это объясняется тем, что в зависимости от кодировки каждый символ может занимать несколько байт. Если вы желаете сохранять строки без определённого предела длины, используйте типы text или character varying без указания длины, а не задавайте какое-либо большое максимальное значение.)

5. Строки фиксированной длины
6. Пример
7. Строки неограниченной длины

Помимо этого, PostgreSQL предлагает тип text, в котором можно хранить строки произвольной длины. Хотя тип text не описан в стандарте SQL, его поддерживают и некоторые другие СУБД SQL. Тип данных citext позволяет исключить вызовы lower в SQL-запросах и позволяет сделать первичный ключ регистронезависимым.

8. Специальные символьные типы

Тип name создан только для хранения идентификаторов во внутренних системных таблицах и не предназначен для обычного применения пользователями. В настоящее время его длина составляет 64 байта (63 ASCII-символа плюс конечный знак), но в исходном коде C она задаётся константой NAMEDATALEN. Эта константа определяется во время компиляции (и её можно менять в особых случаях), а кроме того, максимальная длина по умолчанию может быть увеличена в следующих версиях. Тип "char" (обратите внимание на кавычки) отличается от char(1) тем, что он фактически хранится в одном байте. Он используется во внутренних системных таблицах для простых перечислений.

9. Полнотекстовый поиск

Полнотекстовый поиск (или просто *поиск текста*) — это возможность находить *документы* на естественном языке, соответствующие *запросу*, и, возможно, дополнительно сортировать их по релевантности для этого запроса. Наиболее распространённая задача — найти все документы, содержащие *слова запроса*, и выдать их отсортированными по степени *соответствия* запросу. Понятия запроса и соответствия довольно расплывчаты и зависят от конкретного приложения. В самом простом случае запросом считается набор слов, а соответствие определяется частотой слов в документе.

10. Операторы текстового поиска

Операторы текстового поиска существуют в СУБД уже многие годы. В PostgreSQL для текстовых типов данных есть операторы ~, ~*, LIKE и ILIKE, но им не хватает очень важных вещей, которые требуются сегодня от информационных систем.

11. LIKE

Выражение LIKE возвращает true, если *строка* соответствует заданному *шаблону*. Если *шаблон* не содержит знаков процента и подчёркиваний, тогда шаблон представляет в точности строку и LIKE работает как оператор сравнения. Подчёркивание (_) в *шаблоне* подменяет (вместо него подходит) любой символ; а знак процента (%) подменяет любую (в том числе и пустую) последовательность символов. При проверке по шаблону LIKE всегда рассматривается вся строка.

12. ILIKE

13. SIMILAR TO

Оператор SIMILAR TO возвращает true или false в зависимости от того, соответствует ли данная строка шаблону или нет. Он работает подобно оператору LIKE, только его шаблоны соответствуют определению регулярных выражений в стандарте SQL. Регулярные выражения SQL представляют собой любопытный гибрид синтаксиса LIKE с синтаксисом обычных регулярных выражений (описание которых выходит за рамки доклада).

Как и LIKE, условие SIMILAR TO истинно, только если шаблон соответствует всей строке; это отличается от условий с регулярными выражениями, в которых шаблон может соответствовать любой части строки.

14. Операторы регулярных выражений POSIX

Регулярные выражения POSIX предоставляют более мощные средства поиска по шаблонам, чем операторы LIKE и SIMILAR TO. Регулярное выражение — это последовательность символов, представляющая собой краткое определение набора строк (*регулярное множество*). Строка считается соответствующей регулярному выражению, если она является членом регулярного множества, описываемого регулярным выражением. Как и для LIKE, символы шаблона непосредственно соответствуют символам строки, за исключением специальных символов языка регулярных выражений. При этом спецсимволы регулярных выражений отличается от спецсимволов LIKE. В отличие от шаблонов LIKE, регулярное выражение может совпадать с любой частью строки, если только оно не привязано явно к началу и/или концу строки.

15. Недостатки

Нет поддержки лингвистического функционала, даже для английского языка. Возможности регулярных выражений ограничены — они не рассчитаны на работу со словоформами, например, подходят и подходить. С ними вы можете пропустить документы, которые содержат подходят, но, вероятно, и они представляют интерес при поиске по ключевому слову подходить. Конечно, можно попытаться перечислить в регулярном выражении все варианты слова, но это будет очень трудоёмко и чревато ошибками (некоторые слова могут иметь десятки словоформ).

Они не позволяют упорядочивать результаты поиска (по релевантности), а без этого поиск неэффективен, когда находятся сотни подходящих документов.

Они обычно выполняются медленно из-за отсутствия индексов, так как при каждом поиске приходится просматривать все документы.

16. Полнотекстовая индексация

Для компенсации вышеперечисленных недостатков стандартных операторов текстового поиска в PostgreSQL технология полнотекстового поиска была реализована специальным образом. Первым важным этапом в данной реализации является полнотекстовая индексация.

17. Документ

Документ — это единица обработки в системе полнотекстового поиска; например, журнальная статья или почтовое сообщение. Система поиска текста должна уметь разбирать документы и сохранять связи лексем (ключевых слов) с содержащим их документом.

Впоследствии эти связи могут использоваться для поиска документов с заданными ключевыми словами.

В контексте поиска в PostgreSQL документ — это обычно содержимое текстового поля в строке таблицы или, возможно, сочетание (объединение) таких полей, которые могут храниться в разных таблицах или формироваться динамически.

Документы также можно хранить в обычных текстовых файлах в файловой системе. В этом случае база данных может быть просто хранилищем полнотекстового индекса и исполнителем запросов, а найденные документы будут загружаться из файловой системы по некоторым уникальным идентификаторам.

18. Разбор документов на фрагменты

При этом полезно выделить различные классы фрагментов, например, числа, слова, словосочетания, почтовые адреса и т. д., которые будут обрабатываться по-разному. В принципе классы фрагментов могут зависеть от приложения, но для большинства применений вполне подойдёт предопределённый набор классов. Эту операцию в PostgreSQL выполняет *анализатор* (parser). Вы можете использовать как стандартный анализатор, так и создавать свои, узкоспециализированные.

19. Преобразование фрагментов в лексемы.

Лексема — это нормализованный фрагмент, в котором разные словоформы приведены к одной. Например, при нормализации буквы верхнего регистра приводятся к нижнему, а из слов обычно убираются окончания (в частности, s или es в английском). Благодаря этому можно находить разные формы одного слова, не вводя вручную все возможные варианты.

Кроме того, на данном шаге обычно исключаются стоп-слова, то есть слова, настолько распространённые, что искать их нет смысла. (Другими словами, фрагменты представляют собой просто подстроки текста документа, а лексемы — это слова, имеющие ценность для индексации и поиска.) Для выполнения этого шага в PostgreSQL используются словари. Набор существующих стандартных словарей при необходимости можно расширять, создавая свои собственные.

20. Словари

21. Хранение документов

Каждый документ может быть представлен в виде отсортированного массива нормализованных лексем. Помимо лексем часто желательно хранить информацию об их положении для *ранжирования по близости*, чтобы документ, в котором слова запроса расположены «плотнее», получал более высокий ранг, чем документ с разбросанными словами.

Для хранения подготовленных документов в PostgreSQL предназначен тип данных 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, тогда как этот тип, конечно, содержит только компактное представление всего документа.

22. Хранение поисковых запросов

Для представления обработанных запросов в PostgreSQL существует тип данных tsquery.

Значение tsquery содержит искомые лексемы, объединяемые логическими операторами & (И), | (ИЛИ) и ! (НЕ), а также оператором поиска фраз <-> (ПРЕДШЕСТВУЕТ). Также допускается вариация оператора ПРЕДШЕСТВУЕТ вида <*N*>, где *N* — целочисленная константа, задающая расстояние между двумя искомыми лексемами. Запись оператора <-> равнозначна <1>.

Для группировки операторов могут использоваться скобки. Без скобок эти операторы имеют разные приоритеты, в порядке убывания: ! (НЕ), <-> (ПРЕДШЕСТВУЕТ), & (И) и | (ИЛИ).

23. Пример

24. Оператор соответствия @@

Полнотекстовый поиск в PostgreSQL реализован на базе оператора соответствия @@, который возвращает true, если tsvector (документ) соответствует tsquery (запросу). Для этого оператора не важно, какой тип записан первым

25. Варианты использования

Оператор `@@` также может принимать типы `text`, позволяя опустить явные преобразования текстовых строк в типы `tsvector` и `tsquery` в простых случаях. Всего есть четыре варианта этого оператора:
tsvector @@ tsquery
tsquery @@ tsvector
text @@ tsquery
text @@ text
Первые два мы уже видели раньше. Форма `text@@tsquery` равнозначна выражению `to_tsvector(x) @@ y`, а форма `text@@text` — выражению `to_tsvector(x) @@ plainto_tsquery(y)`.

26. Поиск в таблице

27. Разбор документов

Для преобразования документа в тип tsvector PostgreSQL предоставляет функцию to_tsvector.

```
to_tsvector([конфигурация regconfig,] документ text) returns tsvector
```

to_tsvector разбирает текстовый документ на фрагменты, сводит фрагменты к лексемам и возвращает значение tsvector, в котором перечисляются лексемы и их позиции в документе. При обработке документа используется указанная конфигурация текстового поиска или конфигурация по умолчанию. Простой пример:

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

В этом примере мы видим, что результирующий tsvector не содержит слова a, on и it, слово rats превратилось rat, а знак препинания «-» был проигнорирован.

Функция to_tsvector внутри вызывает анализатор, который разбивает текст документа на фрагменты и классифицирует их. Для каждого фрагмента она проверяет список словарей (Раздел 12.6), определяемый типом фрагмента. Первый же словарь, распознавший фрагмент, выдаёт одну или несколько представляющих его лексем. Например, rats превращается в rat, так как один из словарей понимает, что слово rats — это слово rat во множественном числе. Некоторое слова распознаются как стоп-слова (Подраздел 12.6.1) и

игнорируются как слова, фигурирующие в тексте настолько часто, что искать их бессмысленно. В нашем примере это a, on и it. Если фрагмент не воспринимается ни одним словарём из списка, он так же игнорируется. В данном примере это происходит со знаком препинания -, так как с таким типом фрагмента (символы-разделители) не связан никакой словарь и значит такие фрагменты никогда не будут индексироваться. Выбор анализатора, словарей и индексируемых типов фрагментов определяется конфигурацией текстового поиска (Раздел 12.7). В одной базе данных можно использовать разные конфигурации, в том числе, предопределённые конфигурации для разных языков. В нашем примере мы использовали конфигурацию по умолчанию для английского языка — english.

28. Разбор запросов
29. Разбор запросов
30. Разбор запросов

PostgreSQL предоставляет функции to_tsquery, plainto_tsquery, phraseto_tsquery и websearch_to_tsquery для приведения запроса к типу tsquery. Функция to_tsquery даёт больше возможностей, чем plainto_tsquery или phraseto_tsquery, но более строга к входным данным. Функция websearch_to_tsquery представляет собой упрощённую версию to_tsquery с альтернативным синтаксисом, подобным тому, что принят в поисковых системах в Интернете.

to_tsquery([конфигурация regconfig,] текст_запроса text) returns tsquery

to_tsquery создаёт значение tsquery из текста_запроса, который может состоять из простых фрагментов, разделённых логическими операторами tsquery: & (И), | (ИЛИ), ! (НЕ) и <-> (ПРЕДШЕСТВУЕТ), возможно, сгруппированных скобками. Другими словами, входное значение для to_tsquery должно уже соответствовать общим правилам для значений tsquery, описанным в Подразделе 8.11.2. Различие их состоит в том, что во вводимом в tsquery значении фрагменты воспринимаются буквально, тогда как to_tsquery нормализует фрагменты, приводя их к лексемам, используя явно указанную или подразумеваемую конфигурацию, и отбрасывая стоп-слова. Например:

Как и при вводе значения tsquery, для каждой лексемы можно задать вес(а), чтобы при поиске можно было выбрать из tsvector только лексемы с заданными весами. Например:

К лексеме также можно добавить *, определив таким образом условие поиска по префиксу:

Такая лексема будет соответствовать любому слову в tsvector, начинающемуся с данной подстроки.

to_tsquery может также принимать фразы в апострофах. Это полезно в основном когда конфигурация включает тезаурус, который может обрабатывать такие фразы. В показанном ниже примере предполагается, что тезаурус содержит правило supernovae stars : sn:

Если убрать эти апострофы, to_tsquery не примет фрагменты, не разделённые операторами И, ИЛИ и ПРЕДШЕСТВУЕТ, и выдаст синтаксическую ошибку.

plainto_tsquery([конфигурация regconfig,] текст_запроса text) returns tsquery

plainto_tsquery преобразует неформатированный текст_запроса в значение tsquery. Текст разбирается и нормализуется подобно тому, как это делает to_tsvector, а затем между оставшимися словами вставляются операторы & (И) типа tsquery.

Заметьте, что plainto_tsquery не распознает во входной строке операторы tsquery, метки весов или обозначения префиксов:

В данном случае все знаки пунктуации были отброшены как символы-разделители.

phraseto_tsquery([конфигурация regconfig,] текст_запроса text) returns tsquery

phraseto_tsquery ведёт себя подобно plainto_tsquery, за исключением того, что она вставляет между оставшимися словами оператор <-> (ПРЕДШЕСТВУЕТ) вместо оператора & (И). Кроме того, стоп-слова не просто отбрасываются, а подсчитываются, и вместо операторов <-> используются операторы <N> с подсчитанным числом. Эта функция полезна при поиске точных последовательностей лексем, так как операторы ПРЕДШЕСТВУЕТ проверяют не только наличие всех лексем, но и их порядок.

Как и plainto_tsquery, функция phraseto_tsquery не распознает во входной строке операторы типа tsquery, метки весов или обозначения префиксов:

31. Оператор конкатенации векторов
32. Задание веса вектору
33. Ранжирование результатов поиска

Ранжирование документов можно представить как попытку оценить, насколько они релевантны заданному запросу и отсортировать их так, чтобы наиболее релевантные выводились первыми. В PostgreSQL встроены две функции ранжирования, принимающие во внимание лексическую, позиционную и структурную информацию; то есть, они учитывают, насколько часто и насколько близко встречаются в документе ключевые слова и какова важность содержащей их части документа. Однако само понятие релевантности довольно размытое и во многом определяется приложением. Приложения могут использовать для ранжирования и другую информацию, например, время изменения документа. Встроенные функции ранжирования можно рассматривать лишь как примеры реализации. Для своих конкретных задач вы можете разработать собственные функции ранжирования и/или учесть при обработке их результатов дополнительные факторы.

Ниже описаны две встроенные функции ранжирования:

ts_rank([*веса* float4[],] *вектор* tsvector, *запрос* tsquery [, *нормализация* integer]) returns float4

Ранжирует векторы по частоте найденных лексем.

ts_rank_cd([*веса* float4[],] *вектор* tsvector, *запрос* tsquery [, *нормализация* integer]) returns float4

Плотность покрытия вычисляется подобно рангу ts_rank, но в расчёт берётся ещё и близость соответствующих лексем друг к другу. Для вычисления результата этой функции требуется информация о позиции лексем. Поэтому она игнорируют «очищенные» от этой информации лексемы в tsvector. Если во входных данных нет неочищенных лексем, результат будет равен нулю.

Для обеих этих функций аргумент *веса* позволяет придать больший или меньший вес словам, в зависимости от их меток. В передаваемом массиве весов определяется, насколько весома каждая категория слов, в следующем порядке:

```
{вес D, вес C, вес B, вес A}
```

Если этот аргумент опускается, подразумеваются следующие значения:

```
{0.1, 0.2, 0.4, 1.0}
```

Обычно весами выделяются слова из особых областей документа, например из заголовка или краткого введения, с тем, чтобы эти слова считались более и менее значимыми, чем слова в основном тексте документа.

Ранжирование может быть довольно дорогостоящей операцией, так как для вычисления ранга необходимо прочитать tsvector каждого подходящего документа и это займёт значительное время, если придётся обращаться к диску. К сожалению, избежать этого вряд ли возможно, так как на практике по многим запросам выдаётся большое количество результатов.

34. Пример

35. Конфигурации

Как было упомянуто ранее, весь функционал текстового поиска позволяет пропускать определённые слова (стоп-слова), обрабатывать синонимы и выполнять сложный анализ слов, например, выделять фрагменты не только по пробелам. Все эти функции управляются *конфигурациями текстового поиска*. В PostgreSQL есть набор предопределённых конфигураций для многих языков, но можно создавать и собственные конфигурации.

У каждой функции текстового поиска, зависящей от конфигурации, есть необязательный аргумент regconfig, в котором можно явно указать конфигурацию для данной функции.

Для упрощения создания конфигураций текстового поиска они строятся из более простых объектов. В PostgreSQL есть четыре типа таких объектов:

Анализаторы текстового поиска разделяют документ на фрагменты и классифицируют их (например, как слова или числа).

Словари текстового поиска приводят фрагменты к нормализованной форме и отбрасывают стоп-слова.

Шаблоны текстового поиска предоставляют функции, образующие реализацию словарей. (При создании словаря просто задаётся шаблон и набор параметров для него.)

Конфигурации текстового поиска выбирают анализатор и набор словарей, который будет использоваться для нормализации фрагментов, выданных анализатором.

Анализаторы и шаблоны текстового поиска строятся из низкоуровневых функций на языке C; чтобы создать их, нужно программировать на C, а подключить их к базе данных может только суперпользователь.

36. Индексы

Полнотекстовый поиск можно выполнить, не применяя индекс. Однако при сложных запросах для большинства приложений скорость будет неприемлемой; этот подход рекомендуется только для нерегулярного поиска и динамического содержимого. Для практического применения полнотекстового поиска обычно создаются индексы.

37. Типы индексов

Для ускорения полнотекстового поиска можно использовать индексы двух видов: GIN и GiST.

Более предпочтительными для текстового поиска являются индексы GIN. Они содержат записи для всех отдельных слов (лексем) с компактным списком мест их вхождений. Индексы GIN хранят только слова (лексемы) из значений tsvector, и теряют информацию об их весах.

Индекс GiST допускает *неточности*, то есть он допускает ложные попадания и поэтому их нужно исключать дополнительно, сверяя результат с фактическими данными таблицы. (PostgreSQL делает это автоматически.) Индексы GiST являются неточными, так как все документы в них представляются сигнатурой фиксированной длины. Эта сигнатура создаётся в результате представления присутствия каждого слова как одного бита в строке из n-бит, а затем логического объединения этих битовых строк. Если двум словам будет соответствовать одна битовая позиция, попадание оказывается ложным. Если для всех слов оказались установлены соответствующие биты (в случае фактического или ложного попадания), для проверки правильности предположения о совпадении слов необходимо прочитать строку таблицы.

Неточность индекса приводит к снижению производительности из-за дополнительных обращений к записям таблицы, для которых предположение о совпадении оказывается ложным.

38. Создание индексов

Используется функция to_tsvector с двумя аргументами. В выражениях, определяющих индексы, можно использовать только функции, в которых явно задаётся имя конфигурации текстового поиска. Это объясняется тем, что содержимое индекса не должно зависеть от значения параметра default_text_search_config. В противном случае содержимое индекса может быть неактуальным, если разные его элементы tsvector будут создаваться с разными конфигурациями текстового поиска и нельзя будет понять, какую именно использовать. Выгрузить и восстановить такой индекс будет невозможно.

Так как при создании индекса использовалась версия to_tsvector с двумя аргументами, этот индекс будет использоваться только в запросах, где to_tsvector вызывается с двумя аргументами и во втором передаётся имя той же конфигурации.

Индекс можно создать более сложным образом, определив для него имя конфигурации в другом столбце таблицы. Так можно сохранить имя конфигурации, связанной с элементом индекса, и, таким образом, иметь в одном индексе элементы с разными конфигурациями. Это может быть полезно, например, когда в коллекции документов хранятся документы на разных языках.

Индексы могут создаваться даже по объединению столбцов:

39. Оптимизация индексов

Ещё один вариант — создать отдельный столбец tsvector, в котором сохранить результат to_tsvector. Чтобы этот столбец автоматически синхронизировался с исходными данными, он создаётся как сохранённый генерируемый столбец.

Хранение вычисленного выражения индекса в отдельном столбце даёт ряд преимуществ. Во-первых, для использования индекса в запросах не нужно явно указывать имя конфигурации текстового поиска. Как показано в вышеприведённом примере, в этом случае запрос может зависеть от default_text_search_config. Во-вторых, поиск выполняется быстрее, так как для проверки соответствия данных индексу не нужно повторно выполнять to_tsvector.

40. Ограничения

Текущая реализация текстового поиска в PostgreSQL имеет следующие ограничения:

Длина лексемы не может превышать 2 килобайта

Длина значения tsvector (лексемы и их позиции) не может превышать 1 мегабайт

Число лексем должно быть меньше 264

Значения позиций в tsvector должны быть от 0 до 16383

Расстояние в операторе <*N*> (ПРЕДШЕСТВУЕТ) типа tsquery не может быть больше 16384

Не больше 256 позиций для одной лексемы

Число узлов (лексемы + операторы) в значении tsquery должно быть меньше 32768