Основы полнотекстового поиска в PostgreSQL

eax.me/postgresql-full-text-search/

22 мая 2017

Полнотекстовый поиск (Full-Text Search, FTS) это когда вы ищите какие-то документы, скажем, товары в интернет-магазине или статьи в блоге, по текстовому запросу, как в Google. Немногие знают, что в PostgreSQL из коробки есть полнотекстовый поиск, притом, в отличие от некоторых других РСУБД, очень даже неплохой. Далее в этой заметке будет рассказано, как им пользоваться.

Введение

Использовать встроенный в PostgreSQL полнотекстовый поиск вместо связки из PostgreSQL и специализированного ПО для FTS, такого, как Sphinx, ElasticSearch или Solr, интересно по следующим причинам. Во-первых, данные не приходится хранить в двух экземплярах. То есть, если у вас 500 Гб данных, не приходится использовать 1 Тб места на диске, что есть удобно. Во-вторых, данные всегда консистентны. Скажем, если у вас связка из PostgreSQL и ElasticSearch, и синхронизация документов работает с запаздыванием или ломается, в поиске вы можете увидеть ерунду. Например, уже удаленные документы. Наконец, в-третьих, не приходится устанавливать и поддерживать какого-либо дополнительного ПО — обновлять его, бэкапить, мониторить, писать какие-то скрипты синхронизации, и так далее. Если у вас уже есть PostgreSQL, все просто работает.

Из преимуществ Sphinx, Solr и так далее, следует отметить как минимум то, что будучи специализированными средствами для полнотекстового поиска, они могут работать быстрее. Хотя бы по той причине, что им не приходится беспокоиться о транзакциях, проверках целостности данных, и так далее. Если на каком-то этапе развития проекта вы видите, что скорость полнотекстового поиска в PostgreSQL вас не устраивает, и вы не видите простого способа это исправить (вертикальное масштабирование, добавить реплик, тюнинг параметров), имеет смысл обратить внимание на альтернативы. Поскольку делать бенчмарки правильно очень сложно, вопрос сравнения производительности решений для FTS я вынужден оставить за рамками сего поста.

Все описанное ниже было проверено на PostgreSQL 10, который на момент написания статьи находился в бете. Отличия от других версий, как более ранних, так и более поздних, по идее должны быть минимальными.

Немного теории

Для полнотекстового поиска в PostgreSQL предусмотрено несколько специальных типов.

Тип tsvector представляет собой что-то вроде нормализованной строки, по которой будет производиться поиск. Под нормализацией понимается выкидывание стоп-слов, таких, как предлоги, обрезание окончаний слов, и так далее. Преобразование обычной стоки в tsvector осуществляется при помощи процедуры to_tsvector:

```
# select to_tsvector('My name is Alex and I''m a software developer.');
```

```
                    to_tsvector
-----------------------------------------------------
 'alex':4 'develop':10 'm':7 'name':2 'softwar':9
(1 row)
# select to_tsvector('russian', 'Меня зовут Саша и я программист.');
            to_tsvector
-----------------------------------
 'зовут':2 'программист':6 'саш':3
(1 row)
```

Тип tsvector можно конкатенировать:

```
# select to_tsvector('Hello,') || to_tsvector('world!');
      ?column?
--------------------
 'hello':1 'world':2
(1 row)
```

Тип tsquery используется для представления запросов. Для преобразования строки с запросом в tsquery используется процедура to_tsquery:

```
# select to_tsquery('Hello | world');
    to_tsquery
-------------------
 'hello' | 'world'
(1 row)
# select to_tsquery('russian', 'Привет & мир');
    to_tsquery
------------------
 'привет' & 'мир'
(1 row)
```

Но обычные пользователи, как правило, не используют в поисковых запросах логические операторы и подобные вещи. Для преобразования в tsquery запросов, написанных типичными пользователями, предусмотрены процедуры plainto_tsquery и phraseto_tsquery:

```
# select plainto_tsquery('Hello world');
  plainto_tsquery
-------------------
 'hello' & 'world'
(1 row)
# select phraseto_tsquery('russian', 'Привет мир');
  phraseto_tsquery
--------------------
 'привет' <-> 'мир'
(1 row)
```

Итак, теперь у нас есть tsvector и tsquery. Для сопоставления первого со вторым используется оператор `@@`:

```
# select to_tsvector('Hello, world!') @@
```

```
  plainto_tsquery('hello world') as match;
 match
-------
 t
(1 row)
# select to_tsvector('Hello, world!') @@
  plainto_tsquery('goodbye world') as match;
 match
-------
 f
```

Все это, конечно, здорово. Вот только доставать из базы мы хотим не константы, а конкретные поля, и для этого, наверное, нужно использовать какие-то индексы. Для полнотекстового поиска PostgreSQL предлагает два индекса на выбор:

- GIN — быстро ищет, но не слишком быстро обновляется. Отлично работает, если вы сравнительно редко меняете данные, по которым ищите;
- GiST — ищет медленнее GIN, зато очень быстро обновляется. Может лучше подходить для поиска по очень часто обновляемым данным;

Если сомневаетесь, берите GIN. Обычно все используют именно его. GiST же имеет смысл использовать при довольно экзотической нагрузке. Узнать чуть больше о кишочках работы полнотекстового поиска поверх GIN и GiST можно здесь.

Fun fact! Заметьте также, что если у вас небольшой объем данных, который полностью помещается в память, может оказаться, что по нему быстрее искать вообще без индексов.

Это, конечно, далеко не вся теория, но для начала нам хватит. С остальными моментами мы разберемся уже по ходу.

Практика

Для эксперимента попробуем сделать полнотекстовый поиск по статьям из Википедии. Ссылку на скачивание дампа Википедии, а также скрипт на Python для импорта этого дампа в PostgreSQL, вы найдете в этом репозитории на GitHub.

Здесь же я приведу только схему таблицы:

```
CREATE TABLE IF NOT EXISTS
  articles(id SERIAL PRIMARY KEY, title VARCHAR(128), content TEXT);
```

Само собой разумеется, импортировать всю Википедию целиком не требуется. При написании этой заметки мною использовалась таблица, содержащая всего лишь 25 117 статей.

Чтобы избежать дублирования кода, нам понадобится следующая хранимая процедура на PL/pgSQL:

```
CREATE OR REPLACE FUNCTION make_tsvector(title TEXT, content TEXT)
   RETURNS tsvector AS $$
BEGIN
  RETURN (setweight(to_tsvector('english', title),'A') ||
```

```
    setweight(to_tsvector('english', content), 'B'));
END
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

Данная процедура принимает заголовок и содержимое статьи и возвращает соответствующий ей tsvector. Здесь используется новая для нас процедура setweight, с тем, чтобы придать заголовкам статьи больший вес, чем содержимому. Веса потребуются нам чуть позже для ранжирования статей. Заметьте, что представленная процедура была объявлена, как immutable, чтобы ее можно было использовать при построении индекса.

Итак, теперь построим индекс и попробуем поискать с его помощью статьи:

```
CREATE INDEX IF NOT EXISTS idx_fts_articles ON articles
  USING gin(make_tsvector(title, content));
SELECT id, title FROM articles WHERE
  make_tsvector(title, content) @@ to_tsquery('bjarne <-> stroustrup');
```

Используемый в to_tsquery оператор `<->` означает, что слова должны идти последовательно друг за другом. В моем случае результат получился таким:

```
  id   |                        title
-------+-----------------------------------------------------------
  2470 | Binary search algorithm
  2129 | Bell Labs
  2130 | Bjarne Stroustrup
  3665 | C (programming language)
...
(17 rows)
Time: 136.357 ms
```

Обратите внимание на время выполнения запроса. Довольно неплохо для поиска по 25 тысячам статей, особенно для моего не такого уж и нового ноутбука.

Давайте теперь попробуем как-то выделить Бьерна в результатах поиска, чтобы его было лучше видно:

```
SELECT id, ts_headline(title, q) FROM articles,
    to_tsquery('bjarne <-> stroustrup') AS q
  WHERE make_tsvector(title, content) @@ q;
```

Обратите внимание на использование новой для нас процедуры ts_headline, а также на переменную `q`, благодаря которой tsquery не приходится писать два раза. Результат:

```
  id   |                        ts_headline
-------+-----------------------------------------------------------
  2470 | Binary search algorithm
  2129 | Bell Labs
  2130 | <b>Bjarne</b> <b>Stroustrup</b>
  3665 | C (programming language)
...
(17 rows)
```

Способ выделения при желании можно и изменить:

```
SELECT id, ts_headline(title, q, 'StartSel=<em>, StopSel=</em>')
  FROM articles, to_tsquery('bjarne <-> stroustrup') AS q
  WHERE make_tsvector(title, content) @@ q;
```

Результат:

```
  id   |                    ts_headline
-------+-----------------------------------------------------
  2470 | Binary search algorithm
  2129 | Bell Labs
  2130 | <em>Bjarne</em> <em>Stroustrup</em>
  3665 | C (programming language)
...
(17 rows)
```

Наконец, можно отсортировать статьи по их релевантности:

```
SELECT id, ts_headline(title, q, 'StartSel=<em>, StopSel=</em>')
  FROM articles, to_tsquery('bjarne <-> stroustrup') AS q
  WHERE make_tsvector(title, content) @@ q
  ORDER BY ts_rank(make_tsvector(title, content), q) DESC;
```

Обратите внимание на использование новой для нас процедуры ts_rank.

Результат:

```
  id   |                    ts_headline
-------+-----------------------------------------------------
  2130 | *Bjarne Stroustrup*
  3665 | C (programming language)
  6266 | Edsger W. Dijkstra
 11825 | Legacy system
  2470 | Binary search algorithm
 20432 | Template (C++)
  2129 | Bell Labs
...
(17 rows)
Time: 240.304 ms
```

Само собой разумеется, время выполнения запроса увеличилось. Заметьте, что при использовании GIN и GiST сортировка документов по релевантности происходит в памяти. Это может плохо работать при плохой селективности запросов. Если это как раз ваш случай, советую ознакомиться с новым индексом для полнотекстового поиска под названием RUM, решающим описанную проблему. Используется он практически так же, как и GIN, поэтому в рамках данного поста более подробно на RUM мы останавливаться не будем.

Заключение

Приведенных выше сведений вам будет более чем достаточно в 95% случаев. Дополнительную информацию можно найти по следующим ссылкам:

- Официальная документация PostgreSQL по полнотекстовому поиску является неисчерпаемым кладезем знаний;
- Также рекомендую ознакомиться с документацией по GIN. Например, из нее можно узнать про параметр `fastupdate = off`. Он увеличивает время обновления индекса, но зато делает его куда более предсказуемым, что в некоторых случаях может быть полезным.
- Модуль pg_trgm предназначен для поиска по триграммам. В частности, с его помощью можно находить что-то даже по запросам, содержащим опечатки;
- Начиная с PostgreSQL 10 полнотекстовый поиск также превосходно работает по JSON'у прямо из коробки;
- На HabraHabr есть прекрасная серия статей за авторством Егора Рогова, посвященная индексам в PostgreSQL. В частности, подробно рассматривается устройство индексов GIN, GiST и RUM;

Как видите, полнотекстовый поиск в PostgreSQL работает достаточно хорошо. Особенно, если учесть, что от разработчика приложения для этого не требуется ничего, кроме создания дополнительного индекса. Как уже отмечалось, в специализированных решениях полнотекстовый поиск вполне может работать быстрее, чем в PostgreSQL. Хотя, возможно, что это и не так (например, потому что Java) — я не проверял. Так или иначе, есть множество приложений, которым полнотекстового поиска PostgreSQL будет более, чем достаточно.

А что вы в это время суток используете для полнотекстового поиска, и каковы ваши впечатления от используемого решения?

Дополнение: Также вас могут заинтересовать статьи Нечеткий поиск по тексту в PostgreSQL с помощью pg_trgm и Поиск по географическим данным при помощи PostGIS.

Метки: PostgreSQL, СУБД.