"Анатомия" полнотекстового поиска в PostgreSQL.
(Авторское неформальное описание полнотекстового поиска)

Бартунов О.С.
Государственный Астрономический институт им. П.К.Штернберга, Московский Государственный Университет им. М.В. Ломоносова, Москва, Россия,
e-mail: obartunov@gmail.com

Полнотекстовый поиск в базах данных является одним из востребованных механизмов доступа к содержимому любой современной информационной системы, которые хранят метаинформацию, а зачастую, и сами документы, в базе данных. Современные веб-сайты, по сути, являются интерфейсом, способом организации доступа к базам данных. Внешние поисковые машины, которым периодически "скармливают" содержимое базы данных, решают задачу полнотекстового поиска, если не обращать внимание на проблему актуальности поискового индекса, нетранзакционность, отсутствие доступа к метаданным, трудности с организацией политики доступа к документам, и т.д.

В докладе я расскажу, что представляет полнотекстовый поиск в PostgreSQL, его основные концепции, алгоритмы и структуры данных. Также, я остановлюсь на системах расширяемости, с помощью которых реализован полнотекстовый поиск, и как мы обогнали Sphinx.

Из нашего повседневного опыта мы понимаем, что хороший поиск - это поиск, который в ответ на наш запрос быстро найдет релевантные документы. И такие машины, казалось бы, существуют, например, широко известные поисковые машины как глобальные - "Google", так и наши российские - "Яндекс", "Рамблер". Более того, существует большое количество поисковиков, платных и бесплатных, которые позволяют индексировать всю вашу коллекцию документов и организовать вполне качественный поиск. Владельцу сайта остается только "скармливать" таким поисковикам контент по мере его появления. Это можно организовать несколькими способами - доступ через http-протокол, используя URL документа, как это делают большие внешние поисковики, или организация доступа к содержимому базы данных. В обоих случаях полнотекстовый индекс является внешним по отношению к базе данных. Часто такой подход оправдан и хорошо работает на многих сайтах, несмотря на некоторые недостатки, такие как неполная синхронизация содержимого БД, нетранзакционность, отсутствие доступа к метаданным и использование их для ограничения области поиска или, например, организации определенной политики доступа к документам, и т.д.

Мы не будем касаться таких поисковых машин, а будем рассматривать полнотекстовый поиск, который полностью интегрирован с СУБД. Очевидно, что подобный поиск обязан соответствовать архитектуре СУБД, что налагает определенные ограничения на алгоритмы и методы доступа к данным. Несмотря на то, что подобные ограничения могут влиять на производительность поиска, полный доступ ко всем метаданным базы данных дает возможность для реализации очень сложных поисков, просто невозможных для внешних поисковиков. Например, понятие документа в БД отличается от обычного восприятия как страница на сайте, которую можно сохранить, открыть, модифицировать, удалить. То, что пользователь или поисковый робот видит на сайте является результатом лишь одной комбинацией метаданных, полное множество которых практически недоступно для поисковых роботов. Существует даже понятие "скрытого веба" (Hidden Web), недоступного для поисковых машин и который во много раз превышает размеры видимого веба. Одним из компонентов этой "скрытой" части веба является содержимое баз данных.

Что такое документ в базе данных ? Это может быть произвольный текстовый атрибут или их комбинация. Атрибуты могут храниться в разных таблицах и тогда документ может являться результатом сложной "связки" нескольких таблиц. Более того, текстовые атрибуты могут быть на самом деле результатом работы

программ-конвертеров, которые вытаскивают текстовую информацию из бинарных полей (.doc, .pdf, .ps, ...). В большинстве случаев, документ является результатом работы SQL команд и виртуальным по своей природе. Очевидно, что единственное требование для документа является наличие уникального ключа, по которому его можно идентифицировать. Для внешнего поисковика такой документ является просто набором слов ("bag of words"), без никакого понимания структуры, т.е. из каких атрибутов этот документ был составлен, какова важность того или иного документа. Вот пример документа, составленного из нескольких текстовых атрибутов.

```
SELECT m.title ||' '|| m.author ||' '||  m.abstract ||' '|| d.body as document
FROM messages m, docs d
WHERE m.id = d.id and m.id = 12;
```

Интуитивно ясно, что не все части документа одинаково важны. Так, например, заголовок или абстракт обладают большей информационной плотностью, чем остальная часть документа. Запрос имеет чисто иллюстративный характер, так как на самом деле, здесь надо использовать функцию coalesce(), чтобы защититься от ситуации, когда один из атрибутов имеет значение NULL.

Документ надо уметь разбивать на эти слова, что может быть не простой задачей, так как для разных задач понятие слова может быть разным. Мы используем термин "токен" для обозначения "слов", которые получаются после работы парсера, и термин "лексема" для обозначения того, что будет индексировано. Итак, парсер разбивает документ на токены, часть из которых индексируется. Каким образом токен становится лексемой - это определяется конкретной задачей, например, для поиска по цветам требуется индексировать не только обычные слова, обозначающие цвета красок, но и их различные эквиваленты, использующиеся в веб-технологиях, например, их шестнадцатеричные обозначения.

Полнотекстовый поиск в PostgreSQL

Как и многие современные СУБД, PostgreSQL [PGSQL] имеет встроенный механизм полнотекстового поиска. Отметим, что операторы поиска по текстовым данных существовали очень давно, это операторы LIKE, ILIKE, ~, ~*. Однако, они не годились для эффективного полнотекстового поиска, так как

* у них нет лингвистической поддержки, например, при поиске слова 'satisfies' не будут найдены документы со словом 'satisfy' и никакими регулярными выражениями этому не помочь. В принципе, используя OR и все формы слова, можно найти все необходимые документы, но это будет очень неэффективно, так как в некоторых языках слова могут иметь тысячи словоформ!

* они не предоставляют никакой информации для ранжирования (сортировки) документов, что делает такой поиск практически бесполезным, если только не существует другой сортировки или в случае малого количества найденных документов.

* они очень медленные из-за того, что они не имеют индексной поддержки и требуется каждый раз просматривать все документы.

Для улучшения ситуации, мы (Олег Бартунов и Федор Сигаев) предложили и реализовали полнотекстовый поиск, известный как tsearch2 и существовавший как модуль расширения и впоследствии интегрированный в PostgreSQL, начиная с версии 8.3.

Идея нового поиска состояла в том, чтобы затратить время на обработку документа один раз и сохранить время при поиске, использовать специальные программы-словари для нормализации слов, чтобы не заботиться, например, о формах слов, учитывать информацию о важности различных атрибутов документа и положения слова из запроса в документе для ранжирования найденных документов. Чтобы сделать поиск как можно гибким, все его компоненты могут быть заменены “на ходу”, причем все

настройки хранятся в самой базе данных. Для этого, требовалось создать новые типы данных, соответствующие документу и запросу, а также, полнотекстовый оператор для сравнения документа и запроса, который возвращает TRUE, если запрос удовлетворяет запросу, и в противном случае - FALSE.

PostgreSQL предоставляет интерфейсы для создания новых типов данных (и новых запросов) без остановки сервера. Причем, эти типы данных будут обладать всеми свойствами встроенных типов, таких как конкурентность, восстанавливаемость после сбоев, а для эффективной работы они будут иметь индексную поддержку. Все это делает PostgreSQL очень эффективной СУБД современных условиях, когда часто возникает необходимость работы с очень разными данными и разными запросами. Известно, что смена СУБД в ходе развития проекта является очень трудной и дорогостоящей процедурой, поэтому при выборе базы данных надо обращать внимание на такую возможность расширяемости типов данных без остановки сервера, как в PostgreSQL.

В PostgreSQL существуют много способов расширить функциональность сервера, например,если говорить о производительности, то для типов данных, для которых можно определить функцию сравнения, можно создать btree индекс, который будет ускорять стандартный набор операций сравнения. Однако, таким способом невозможно сделать поддержку новых операций, например, операций похожести, пересечения или включения множеств. Для этого были созданы новые системы расширяемости (GiST, GIN, SP-GiST) , которые, по-сути, являются программным интерфейсом к специализированным структурам данных, для которых уже определены базовые операции в ядре СУБД и требуется только разработать пользовательские функции, описывающие специфику данных. Это является большим достижением, так как не требуется иметь знания и опыт разработчиков ядра СУБД, и кроме того, использование нового типа не требует остановки сервера.

Были созданы новые типы данных - tsvector, который является хранилищем для лексем из документа и оптимизированного для поиска, и tsquery - для запроса с поддержкой логических операций, полнотекстовый оператор "две собаки" @@ и индексная поддержка для него с использованием GiST (Обобщенное поисковое дерево) [GiST] и GIN (Обобщенный обратный индекс) [GIN].

tsvector, помимо самих лексем, может хранить информацию о положении лексемы в документе и ее весе (важности), которая потом может использоваться для вычисления ранжирующей информации.

```
=# select 'cat & rat':: tsquery @@ 'a fat cat sat on a mat and ate a fat
rat'::tsvector;
 ?column?
 ----------
  t
=# select 'fat & cow':: tsquery @@ 'a fat cat sat on a mat and ate a fat
rat'::tsvector;
   ?column?
 ----------
  f
```

Кроме этого, были реализованы вспомогательные функции

* **to_tsvector** для преобразования документа в tsvector

```
=# select to_tsvector('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
```

* **to_tsquery** - для получения tsquery

```
=# select to_tsquery('fat & cats');
  to_tsquery
```

```
---------------
 'fat' & 'cat'
```

Для разбиения документа на токены используется парсер, который выдает токен и его тип, см. пример ниже. Каждому типу токена ставится в соответствие набор словарей, которые будут стараться распознать и "нормализовать" его. Порядок словарей фиксирован и важен, так как именно в этом порядке токен будет попадать на вход словарю, до тех пор, пока он не опознается одним из них. Если токен не распознался ни одним из словарей, или словарь опознал его как стоп-слово, то этот токен не индексируется. Таким образом, можно сказать, что для каждого типа токена существует правило обработки токена, которое описывает схему попадания токена в полнотекстовый индекс. Встроенный парсер различает 23 типа, которые можно посмотреть с помощью SQL:

```
=# select * from ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
(23 rows)
```

Каждый словарь по-своему понимает, что такое "нормализация", однако, интуитивно понятно, что в результате нормализации, группы слов, объединенные по тому или иному признаку, приводятся к одному слову. Это позволяет при поиске этого "нормализованного" слова найти все документы, содержащие слова из этой группы. Наиболее привычная нормализация для нас - это приведение существительного к единственному числу и именительному падежу, например, слово 'стол' является нормальной формой слов 'столы', 'столов', 'столами', 'столу' и т.д. Не менее естественным представляется приведение имен директорий '/usr/local/bin', '/usr/local/share/../bin', '/usr/local/./bin/' к стандартному виду '/usr/local/bin', или приведение римских чисел к арабским, так что запрос ’13 & век’ найдет документы с ‘XIII век’.

Поддерживаются агглютинативные языки (немецкий, норвежский,….) , например, запрос с норвежским словом 'footbalklubber' преобразуется в '(football & klubber) | (foot & ball & clubber)'.

Комбинация парсера и правил обработки токенов определяет полнотекстовую конфигурацию (CREATE TEXT SEARCH CONFIGURATION), которых может быть произвольное количество. Большое количество конфигураций для 10 европейских языков и разных

локалей уже встроено в PostgreSQL и хранится в системном каталоге, в схеме pg_catalog. Практически все функции поиска зависят от полнотекстовой конфигурации, которая является необязательным параметром и содержится в GUC-переменной default_text_search_config, которую можно задавать на время сессии или устанавливать по-умолчанию для конкретной базы с помощью команды ALTER DATABASE. Сами парсеры и словари также хранятся в системе, их можно добавлять, изменять и удалять с помощью SQL команд (CREATE TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH PARSER). С помощью команд psql (\dF, \dFd, \dFp, \dFt) можно посмотреть списки конфигураций, словарей, парсеров и их шаблонов, доступные в системе. Тестировать настройки можно с помощью функции ts_debug(configuration, text), например,

```
=# select "alias","token","dictionaries","lexemes" from ts_debug('english', 'as 12 
cats');
   alias   | token |  dictionaries  | lexemes
-----------+-------+----------------+---------
 asciiword | as    | {english_stem} | {}
 blank     |       | {}             |
 uint      | 12    | {simple}       | {12}
 blank     |       | {}             |
 asciiword | cats  | {english_stem} | {cat}
(5 rows)
```

На этом примере мы видим, что токен 'as' обработался словарем english_stem, распознался как стоп-слово и не попал в полнотекстовый индекс, в то время как токены '12' и 'cats' распознались словарями, нормализовались и попали в индекс.

Функция ts_lexize() используется для тестирования словаря, например:

```
=# select ts_lexize('russian_stem','сапоги');
 ts_lexize
-----------
 {сапог}
```

Результаты полнотекстового поиска могут быть упорядочены по степени соответствия запроса документу (функции ts_rank, ts_rank_cd) и представлены с помощью сниппетов текста (функция ts_headline), содержащие слова из запроса.

```
apod=# select ts_headline(body,q, 'StartSel=<,StopSel=>,MaxWords=10,MinWords=5'), 
rank from (
  select body,q, rank_cd(fts,q) as rank from apod,
  to_tsquery('supernovae & x-ray') q where fts @@ q
  order by rank desc limit 5
) as foo;
                              headline                              | rank_cd
--------------------------------------------------------------------+---------
 <supernova> remnant  E0102-72, however, is giving astronomers a clue | 1.59087
 <supernova> explosion. The picture was taken in  <X>-<rays>        | 1.47733
  <X>-<ray> glow is produced by  multi-million degree               | 1.34823
  <X>-<rays> emitted by this shockwave made by a telescope          | 1.14318
  <X>-<ray> glow. Pictured is the <supernova>                       | 1.08116
(5 rows)
```

Список основных возможностей полнотекстового поиска в PostgreSQL:

* Полная интеграция с базой данных, что дает доступ ко всем метаданным и полную синхронизацию полнотекстового индекса с изменяющимся контентом.
* Гибкая настройка всех компонентов поиска с помощью SQL команд. Встроенная поддержка для 10 европейских языков.
* Подключение разных парсеров, которые можно писать с использованием API. Встроенный парсер поддерживает 23 типа токенов.
* Богатая поддержка лингвистики, включая подключаемые словари с поддержкой стоп-слов. Встроенные словари-шаблоны для распространенных открытых словарей ispell, snowball позволяют использовать большое количество словарей для разных языков. Также, есть встроенные словари-шаблоны

thesaurus, synonym. Открытый API позволяют разрабатывать новые словари для решения специфичных задач.

* Полная поддержка многобайтных кодировок, в частности, UTF-8. возможностью приписывания весов разным лексемам позволяют сортировку результатов поиска.
* Поддержка индексов для ускорения поисков, при этом индексы поддерживают конкурентность и возможность восстановления после сбоев, что очень важно для успешной работы в конкурентных условиях. Поддерживаются два типа индексов – GiST индексы очень хороши для частых обновлений, в то время как GIN индекс очень хорошо масштабируется с ростом коллекции. Это позволяет реализовывать полнотекстовый поиск по очень большим коллекциям документов, которые могут непрерывно обновляться.
* Богатый язык запросов с поддержкой настраиваемых правил изменения запроса налету без требования переиндексации.
* Ранжирование результатов поиска доступно с помощью двух встроенных функций, которые вычисляют соответствие документа запросу с использованием координатной информации.
* Результаты поиска могут быть представлены с помошью сниппетов текста, содержащие слова из запроса.

Подробнее о полнотекстовом поиске можно прочитать в официальной документации PostgreSQL [FTSDOC].

Что осталось «за кадром» ?

Поиск фраз.

Иногда требуется найти документы, которые содержат запрос с учетом порядка следования слов. Обычно, полнотекстовый поиск не учитывает этот порядок, так что запросы ‘black cat’ и ‘cat black’ эквивалентны. Для реализации поиска фраз нами была разработана алгебра полнотекстовых запросов и разработан работающий прототип. Был введен оператор BEFORE $n, который гарантирует, что левый операнд следует перед правым и расстояние между ними равно n, т.е.,

a $[n] b == a & b & (∃ i,j : pos(b)i – pos(a)j = n)

Эта функциональность пока не вошла в PostgreSQL и существует только в виде патча (ожидает своего спонсора). Более подробно об этом можно посмотреть в презентации [ALGEBRAFTS].

Полнотекстовый поиск за миллисекунды.

Полнотекстовый поиск в СУБД PostgreSQL хорошо известен своими возможностями и расширяемостью. Однако, можно выделить две основные причины, мешающие ему быть на одном уровне производительности со специализированными решениями:

1) Будучи реализованы внутри СУБД, реализующей ACID, полнотекстовые индексы вынуждены поддерживать атомарность операций, конкурентные изменения, журналирование и т.д., Это неизбежно, если мы хотим получить интегрированное решение на базе реляционной СУБД, и является одновременно как преимуществом (интеграция с СУБД), так и недостатком (значительные расходы на транзакционность).
2) Полнотекстовые индексы используются только для фильтрации документов, ранжирование же требует извлечения самих документов из таблицы, что существенно снижает скорость обработки высоко-селективных запросов. Этот недостаток может быть устранен путем включения дополнительной информации в GIN-индексы.

Нами был разработан прототип патча к PostgreSQL, позволяющий хранить позиционную информацию в полнотекстовом индексе (GIN-индекс) и использовать её для ранжирования. Ранжирование в этом случае осуществляется исключительно по индексу без доступа к таблице, что позволяет повысить скорость обработки высоко-селективных запросов в десятки раз. На основе реальных запросов и данных крупнейшей базы объявлений (~7 млн) рунета avito.ru мы получили

производительность за 8 часов работы 42 млн поисковых запросов, в то время как внешний поисковик Sphinx смог обработать 38 млн запросов. Для 20 млн. коллекции мы получили 38 млн. vs 26 млн. запросов. Эти результаты показывают, что в PostgreSQL можно получить поиск, который не только следует семантике СУБД, но и является высокопроизводительным.

Для иллюстрации наших изменений мы использовали 156676 Wikipedia articles, загруженных в базу данных, и очень высокочастотный запрос со словом 'title'.

```
=# explain analyze
SELECT docid, ts_rank(text_vector, to_tsquery('english', 'title')) AS rank
FROM ti2
WHERE text_vector @@ to_tsquery('english', 'title')
ORDER BY
rank DESC
LIMIT 3;
```

Старый план:

```
Limit (cost=8087.40..8087.41 rows=3 width=282) (actual time=433.750..433.752 rows=3
loops=1)
-> Sort (cost=8087.40..8206.63 rows=47692 width=282)
(actual time=433.749..433.749 rows=3 loops=1)
Sort Key: (ts_rank(text_vector, '''titl'''::tsquery))
Sort Method: top-N heapsort Memory: 25kB
-> Bitmap Heap Scan on ti2 (cost=529.61..7470.99 rows=47692 width=282)
(actual time=15.094..423.452 rows=47855 loops=1)
Recheck Cond: (text_vector @@ '''titl'''::tsquery)
-> Bitmap Index Scan on ti2_index (cost=0.00..517.69 rows=47692 width=0)
(actual time=13.736..13.736 rows=47855 loops=1)
Index Cond: (text_vector @@ '''titl'''::tsquery)
Total runtime: 433.787 ms
```

Новый план:

```
=# explain analyze
SELECT docid, ts_rank(text_vector, to_tsquery('english', 'title')) AS rank
FROM ti2
WHERE text_vector @@ to_tsquery('english', 'title')
ORDER BY
text_vector>< plainto_tsquery('english','title')
LIMIT 3;

Limit (cost=20.00..21.65 rows=3 width=282) (actual time=18.376..18.427 rows=3 loops=1)
-> Index Scan using ti2_index on ti2 (cost=20.00..26256.30 rows=47692 width=282)
(actual time=18.375..18.425 rows=3 loops=1)
Index Cond: (text_vector @@ '''titl'''::tsquery)
Order By: (text_vector >< '''titl'''::tsquery)
Total runtime: 18.511 ms !!!!
```

Старый план состоит из пяти этапов (грубо):

1. поиск по индексу
2. чтение таблицы
3. вычисление ранка для каждой записи
4. сортировка по ранку
5. выдать 3 строки

Новый план состоит из трех этапов:

1. поиск по индексу
2. сортировка по ранку
3. выдать 3 строки

Это стало доступным из-за модификации GIN-индекса, в который была добавлена возможность хранить дополнительную информацию, в частности, координаты слов в

документе, что позволило вычислять ранк документа и сортировать их в индексе без обращения к таблице. Виден очевидный выигрыш, который будет тем больше, чем более частотное слово используется.

В настоящее время работа по внедрению этих улучшений ведется для следующей версии PostgreSQL. Подробнее об этой работе можно посмотреть в презентации [FTSMSEC].

Грамматика полнотекстового поиска (BNF, http://en.wikipedia.org/wiki/Backus-Naur_Form)

```
<lexeme> ::=
           <alphanumeric>{ <alphanumeric> }
           | "'"<character>{ <character> }"'"
<character> ::=
           <alphanumeric>
           | "\" "'" /* single quote */
           | "'" "'" /* single quote */
           | <non-alphanumeric>
<number> ::= <digit>{ <digit> }
<weight> ::= A|B|C|D


<tsvector> ::=
         <entry>{ <space> <entry> }
         | /* empty */
<entry> ::=
         <lexeme>[ ":" <positions> ]
<positions> ::=
         <position>{ "," <position> }
<position> ::=
         <number>[ <weight> ]

<tsquery> ::=
         <expression>
         | /* empty */
<expression> ::=
         <expression> "&" <expression>
         | <expression> "|" <expression>
         | "!" <expression>
         | "(" <expression> ")"
         | <lexeme>[:<modifiers>]
<modifiers> ::=
         "*"
         | <weight>{ <weight> }[ "*" ]
```

Примеры:

```
   tsvector:
        'star':1B,3A,15,26B,32B,44 'supernovae':2A,25B,31B
        'star':1B,3A,15,26B,32B,44 'supernovae':25B,31B

   tsquery:
        'supernovae':A & 'star':A*      (соответствует 1-му tsvector)
```

Использование индексов.

Последовательное сканирование всей таблицы и фильтрация документов с использованием полнотекстового оператора @@ может быть медленным, поэтому используются индексы, которые ускоряют полнотекстовый оператор. Про индексы надо знать, что они используются только для ускорения операций и результат поиска с использованием индекса должен совпадать с поиском без индекса. На самом

деле, индекс не всегда ускоряет поиск, поэтому планер СУБД использует данные о частотности слов в индексируемой коллекции, чтобы принять решение об его использование. Например, если используется очень частотное слово, которые встречается почти в каждом документе (стоп-слово), то разумнее не использовать индекс, так как последовательное чтение таблицы гораздо быстрее случайного чтения практически всех записей таблицы при использовании индекса. Для ускорения полнотекстового поиска можно использовать два типа индексов - на основе GiST [GIST] или GIN [GIN].

GIN индекс, или обобщенный обратный индекс - это структура данных, у которой ключом является лексема, а значением - сортированный список идентификаторов документов, которые содержат эту лексему. Отметим, что позиционная информация не хранится в индексе, что связано с ограничениями PostgreSQL. Так как в обратном индексе используется бинарное дерево для поиска ключей, то он слабо зависит от их количества и потому хорошо шкалируется. Этот индекс используется практически всеми большими поисковыми машинами, однако его использование в базах данных для индексирования изменяющихся документов затруднено, так как любые изменения (добавление нового документа, обновление или удаление) приводят к большому количеству обновлений индекса. Например, добавление нового документа, который содержит N уникальных лексем приводит к обновлению N записей в индексе. Поэтому этот индекс лучше всего подходит для слабоменяющихся коллекций документов. GIN индекс поддерживает пакетное обновление индекса, которое является очень эффективным, поэтому иногда быстрее создать индекс заново, чем обновлять индекс при добавке каждого документа.

В тоже время, GiST индекс является "прямым" индексом, т.е. для каждого документа ставится в соответствие битовая сигнатура, в которой содержится информация о всех лексемах, которые содержаться в этом документе, поэтому добавление нового документа приводит к добавлению только одной сигнатуры. Для быстрого поиска сигнатуры хранятся в сигнатурном дереве RD-Tree (russian doll, матрешка), реализованная помощью GiST.

Сигнатура - это битовая строка фиксированной длины, в которой все биты изначально выставленны в '0'. С помощью хэш-функции слово отображается в определенный бит сигнатуры, который становится '1'. Сигнатура документа является наложением индивидуальных сигнатур всех слов. Такая техника называется superimposed coding и реализуется как bitwise OR, что является очень быстрой операцией.

```
 word   signature
 ----------------
 w1 ->  01000000
 w2 ->  00010000
 w3 ->  10000000
 ----------------------
        11010000
```

В этом примере, '11010000' является сигнатурой документа, состоящего из трех **уникальных** слов w1,w2,w3. Сигнатура является некоторым компактным представлением документа, что приводит к значительному уменьшению размера коллекции. Кроме того, фиксированный размер сигнатуры сильно облегчает операции сравнения. Все это делает использование сигнатур вместо документов привлекательным с точки зрения производительности.

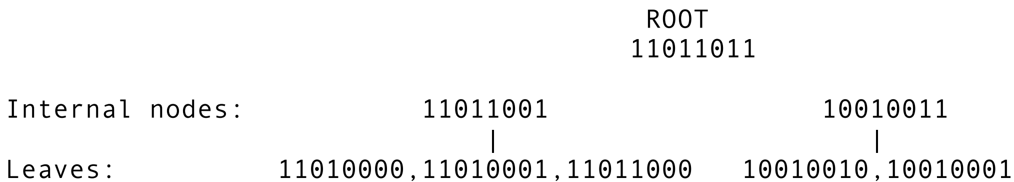
При поиске, запрос можно аналогичным образом представить в виде сигнатуры и тогда процесс поиска будет заключаться в сравнении сигнатур. Если хотя бы одно положение '1' в сигнатурах не совпадает, то можно с уверенностью утверждать, что документ не содержит поисковый запрос. Однако, если все '1' поисковой сигнатура совпадают с '1' сигнатуры документа, то это означает лишь то, что поисковый запрос **может** содержаться в документе и это требует проверки с использованием самого документа, а не его сигнатуры. Вероятностный ответ связан с

использованием хеширования и суперпозиции. Ниже приводятся несколько примеров сигнатур.

```
 11010000  - сигнатура документа
 00000001  - сигнатура запроса Q1, точно не содержится в документе
 01000000  - сигнатура запроса Q2, возможно содержится в документе
 01010000  - сигнатура запроса Q3, возможно содержится в документе
```

Сигнатура Q2 является сигнатурой слова w1 и, таким образом, является правильным попаданием, в то время как сигнатура Q3 - ложным попаданием (**false drop**), несмотря на то, что она удовлетворяет сигнатуре документа. Ясно, что конечность размера сигнатуры и увеличение количества уникальных слов приводит к насыщению сигнатуры, т.е., когда все биты будут '1', что сильно уменьшает избирательность сигнатуры и ухудшает производительность поиска.

Существуют несколько структур данных для хранения сигнатур, такие как сигнатурный файл (signature file),но они не являются индексами, так как требует полного просмотра. Дерево RD-Tree является аналогом R-Tree, удобное для работы со множествами и ускоряет операции включения и перечения множеств. Подробнее о RD-Tree можно прочитать в оригинальной статье [RDTREE].



```
                                            ROOT
                                          11011011

Internal nodes:           11011001                  10010011
                              |                         |
Leaves:          11010000,11010001,11011000   10010010,10010001
```

Очевидно, что чем больше глубина дерева, тем больше вероятность того, что сигнатура вырождается, т.е., начинает состоять из одних '1', а это приводит к тому, что приходится просматривать много веток и поиск замедляется. В предельном случае, когда сигнатура состоит из одних '1', она становится бесполезной, т.е., приходится просматривать все ветки, находящиеся под ней.

Найденные результаты приходится дополнительно проверять на наличие "false drops", т.е., проверять сами исходные документы, действительно ли они удовлетворяют поисковому запросу, что требует произвольного доступа к "heap" (таблице) и это сильно сказывается на производительности. Степень неоднозначности (lossiness), а следовательно и производительность GiST-индекса, зависит от кол-ва уникальных лексем и количества документов, что ограничивает применимость этого индекса для больших коллекций.

Это можно проиллюстрировать с помощью *explain analyze* на примере поиска по целочисленным массивам (100,000 integer[]) с помощью расширения intarray, в котором тоже используются сигнатуры и RD-Tree.

```
=# explain analyze  select id from tt where v2 @> '{2}'::int[];
                                                          QUERY PLAN
------------------------------------------------------------------------------------
----------------------------------------------------
 Bitmap Heap Scan on tt  (cost=21.05..398.87 rows=100 width=4) (actual
time=54.673..1001.433 rows=735 loops=1)
   Recheck Cond: (v2 @> '{2}'::integer[])
   Rows Removed by Index Recheck: 11192
   ->  Bitmap Index Scan on gist_tt_idx2  (cost=0.00..21.03 rows=100 width=0)
(actual time=49.550..49.550 rows=11927 loops=1)
         Index Cond: (v2 @> '{2}'::integer[])
 Total runtime: 1001.904 ms
```

Для сравнения построим GIN-индекс, который не является lossy.

```
=# explain analyze  select id from tt where v1 @> '{2}'::integer[];
                                                       QUERY PLAN
--------------------------------------------------------------------------------
-----------------------------------------
 Bitmap Heap Scan on tt  (cost=20.77..398.59 rows=100 width=4) (actual
time=0.480..1.311 rows=735 loops=1)
   Recheck Cond: (v1 @> '{2}'::integer[])
   ->  Bitmap Index Scan on gin_tt_idx1  (cost=0.00..20.75 rows=100 width=0)
(actual time=0.284..0.284 rows=735 loops=1)
         Index Cond: (v1 @> '{2}'::integer[])
 Total runtime: 1.416 ms
```

Видно, что GiST индекс (gist_tt_idx2) нашел 11927 записей, из которых только 735 являются правильным ответом. Проверка (Recheck) 11927 записей и потребовала все время.

Но это не вся правда о GiST-индексе ! На самом деле, в листьях могут храниться не сигнатуры, а сами tsvector-а, если они не превышают TOAST_INDEX_TARGET байт, что-то около 512 байт. В этом случае попадание является **точным** и проверять ничего не надо. К сожалению, пока нет возможности индексу сказать какое было попадание, но в будущем, когда появится такая возможность, эта оптимизация может очень хорошо работать для новостных сайтов, где документы не очень большие.
Чтобы изучить GiST-индекс, можно воспользоваться специальным модулем Gevel [GEVEL], который выдает полезную информацию об индексе. Вот пример такой выдачи для индекса gist_idx_50 для базы, которая содержит небольшие сообщения. Обратите внимание, что листья содержат как сами tsvector-а, так и сигнатуры, а внутренние узлы - только сигнатуры.

```
arxiv=# select gist_stat('gist_idx_90');
                 gist_stat
--------------------------------------------
 Number of levels:          4
 Number of pages:           18296
 Number of leaf pages:      17496
 Number of tuples:          435661
 Number of invalid tuples:  0
 Number of leaf tuples:     417366
 Total size of tuples:      124776048 bytes
 Total size of leaf tuples: 119803816 bytes
 Total size of index:       149880832 bytes

-- leaf node

arxiv=# select * from gist_print('gist_idx_90') as
        t(level int,valid bool, fts gtsvector) where level =4;
 level | valid |              fts
-------+-------+--------------------------------
     4 | t     | 130 true bits, 1886 false bits
     4 | t     | 95 unique words
     4 | t     | 33 unique words
     4 | t     | 77 unique words
     4 | t     | 68 unique words
     4 | t     | 86 unique words
     4 | t     | 77 unique words
     4 | t     | 51 unique words
     4 | t     | 122 unique words
     4 | t     | 127 true bits, 1889 false bits
     4 | t     | 105 unique words
     4 | t     | 170 true bits, 1846 false bits
     4 | t     | 77 unique words
     4 | t     | 121 true bits, 1895 false bits
....................................
     4 | t     | 61 unique words
```

```
(417366 rows)

-- internal node

arxiv=# select * from gist_print('gist_idx_90') as
        t(level int, valid bool, fts gtsvector) where level =3;

 level | valid |               fts
-------+-------+--------------------------------
     3 | t     | 852 true bits, 1164 false bits
     3 | t     | 861 true bits, 1155 false bits
     3 | t     | 858 true bits, 1158 false bits
     3 | t     | 872 true bits, 1144 false bits
     3 | t     | 858 true bits, 1158 false bits
     3 | t     | 855 true bits, 1161 false bits
     3 | t     | 853 true bits, 1163 false bits
     3 | t     | 857 true bits, 1159 false bits
..................................................
     3 | t     | 782 true bits, 1234 false bits
     3 | t     | 773 true bits, 1243 false bits
(17496 rows)
```

Работа над созданием полнотекстового поиска в PostgreSQL поддерживалась Российским Фондом Фундаментальных Исследований (05-07-90225), EnterpriseDB PostgreSQL Development Fund, Mannheim University, jfg:networks, Georgia Public Library Service, Рамблер Интернет Холдинг.

Литература

[PGSQL], "Что такое PostgreSQL", О.Бартунов,
http://www.sai.msu.su/~megera/postgres/talks/what_is_postgresql.html,
Сайт проекта, http://www.postgresql.org

[GIST], "Написание расширений для PostgreSQL с использованием GiST",
О.Бартунов, Ф. Сигаев,
http://www.sai.msu.su/~megera/postgres/talks/gist_tutorial.html

[RDTREE], "THE RD-TREE: AN INDEX STRUCTURE FOR SETS", Joseph M. Hellerstein,
http://epoch.cs.berkeley.edu/postgres/papers/UW-CS-TR-1252.pdf

[GIN], "Gin for PostgreSQL", http://www.sai.msu.su/~megera/wiki/Gin,
"GIN Presentation on PostgreSQL Anniversary Summit, 2006,
http://www.sigaev.ru/gin/Gin.pdf

[GEVEL], http://www.sai.msu.su/~megera/wiki/Gevel

[FTSPGCON2007], advanced lecture "Full-Text Search in PostgreSQL", O.Bartunov,
http://www.sai.msu.su/~megera/postgres/talks/fts-pgcon2007.pdf

[BLOOM], B.H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable
Errors. Communications of the ACM 13(7), 422-426 (1970).

[RDTREE], Joseph M. Hellerstein, Avi Pfeffer, THE RD-TREE: An Index Structure
for Sets, http://db.cs.berkeley.edu/papers/UW-CS-TR-1252.pdf

[ALGEBRAFTS] «Algebra for full-text queries», O.Bartunov, T.Sigaev,

http://www.sai.msu.su/~megera/postgres/talks/algebra-fts.pdf

[FTSMSEC] “Full-text search in PostgreSQL in milliseconds», O.Bartunov, A.Korotkov,

http://www.sai.msu.su/~megera/postgres/talks/Full-text%20search%20in%20PostgreSQL%20in%20milliseconds-extended-version.pdf

[FTSDOC] «Full Text Search», http://www.postgresql.org/docs/current/static/textsearch.html