Санкт–Петербургский государственный университет

РЯБКОВ Антон Игоревич

Выпускная квалификационная работа

Интроспекция деревьев поиска, основанных на разбиении пространства (SP-GiST) в postgresql

Уровень образования: магистратура
Направление 01.04.02 «Прикладная математика и информатика»
Основная образовательная программа ВМ.5889.2022 "Разработка программного обеспечения и науки о данных"

Научный руководитель:
доцент каф. системного программирования
математико-механического факультета СПбГУ
Булычев Дмитрий Юрьевич

Рецензент:
кандидат физико-математических наук,
старший академический консультант
ООО «МПГ АйТи Солюшнз»
Лозов Петр Алексеевич

Санкт-Петербург
2024 г.

Содержание

Введение

PostgreSQL. База данных — это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе. База данных обычно управляется системой управления базами данных (СУБД). Данные вместе с СУБД, а также приложения, которые с ними связаны, называются системой баз данных, или, для краткости, просто базой данных.

Данные в наиболее распространенных типах современных баз данных обычно хранятся в виде строк и столбцов, формирующих таблицу. Этими данными можно легко управлять, изменять, обновлять, контролировать и упорядочивать. В большинстве баз данных для записи и запросов данных используется язык структурированных запросов (SQL).

PostgreSQL – это свободно распространяемая объектно-реляционная СУБД. На текущий момент PostgreSQL считается наиболее совершенной из свободно распространяемых СУБД и конкурирует с лучшими из коммерческих СУБД таких как Oracle и Microsoft SQL Server. PostgreSQL активно используется в крупнейших государственных и частных организациях по всему миру и популярность ее растет. В числе пользователей PostgreSQL корпорации Sony, Hitachi, Huawei, Yahoo, и многие другие. На основе PostgreSQL были построены такие интернет-гиганты, как Skype и Instagram. Применяется PostgreSQL и в Яндексе и в Mail.Ru.

Индексация. Таблицы в базе данных могут иметь большое количество строк, которые хранятся в произвольном порядке, и их поиск по заданному критерию путём последовательного просмотра таблицы строка за строкой может занимать много времени. Для повышения производительности поиска данных часто создаются индексы – специализированные объекты базы данных. Индекс формируется из значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы, что позволяет искать строки, удовлетворяющие критерию поиска. Ускорение работы с использованием индексов достигается, в первую очередь, за счёт того, что индекс имеет структуру, оптимизированную под поиск, например, сбалансированного дерева.

В PostgreSQL индексы могут создаваться как вручную администраторами СУБД, так и самой СУБД по решению планировщика/оптимизатора при выполнении запроса. На текущий момент поддерживается несколько типов индексов: B-дерево, хеш, GiST, SP-GiST, GIN и BRIN. Для разных типов индексов применяются разные алгоритмы, ориентированные на определённые типы запросов. По умолчанию команда CREATE INDEX создаёт индексы типа B-дерево, эффективные в большинстве случаев.

Индекс типа SP-GiST. Далее в работе речь пойдет про индекс типа SP-GiST. Аббревиатура SP-GiST расшифровывается как «Space-Partitioned GiST», то есть дословно «GiST с разбиением пространства». Индексы типа SP-GiST поддерживают конкурентную работу с деревьями поиска на основе разбиения некоторого пространства, что облегчает разработку широкого спектра различных несбалансированных структур данных, таких как дерево квадрантов, k-мерных и префиксных деревьев. Общей характеристикой этих структур является то, что они последовательно разбивают

пространство поиска на сегменты, которые не обязательно должны быть равного размера. При этом поиск, хорошо соответствующий правилу разбиения, с таким индексом может быть очень быстрым.

Перечисленные выше структуры данных изначально конструировались для работы в оперативной памяти (in-memory). При таком применении они, обычно, представляются в виде набора динамически выделяемых узлов, связываемых указателями. Однако подобную схему нельзя так просто эффективно перенести на диск, так как цепочки указателей могут быть довольно длинными, и поэтому потребуется слишком много обращений к диску. Структуры данных для хранения на диске, наоборот, должны иметь большую разветвлённость для минимизации объёма ввода/вывода. Для решения данной проблемы индекс типа SP-GiST сопоставляет узлы дерева поиска со страницами на диске так, чтобы при поиске требовалось обращаться только к нескольким страницам, даже если при этом нужно просмотреть множество узлов. Более того, индекс обеспечивает конкурентный доступ к дереву с минимальными блокировками (по 1-2 вершине за раз), что позволяет эффективно работать с деревом множеству пользователей.

Важной особенностью индекса типа SP-GiST является возможность разрабатывать и поддерживать нестандартные типы данных с соответствующими методами доступа не специалистами по СУБД, а просто экспертами в соответсвующих предметных областях. Для этого в индексе предусмотрены шаблонные методы, реализовав которые пользователь сможет добиться нужного ему функционала.

Дополнительно поставляемые модули. Вместе с дистрибутивом PostgreSQL поставляются также некоторые дополнительные модули, которые можно найти в папке contrib исходного кода. В их число входят средства портирования, утилиты анализа и подключаемые функции, не включённые в состав основной системы PostgreSQL. В основном, эти модули адресованы ограниченной аудитории или находятся в экспериментальном состоянии, не подходящем для основного дерева кода. Однако это не умаляет их полезность. Одним из таких расширений (о доработке которого далее и пойдет речь) является модуль pageinspect. Этот модуль предоставляет функции, позволяющие исследовать страницы баз данных на низком уровне, что бывает полезно для отладки. Все функции данного модуля могут вызывать только суперпользователи.

1. Постановка задачи и ее актуальность

Несмотря на то, что индексы типа SP-GIST являются расширяемыми, в текущей версии PostgreSQL для них нет встроенных инструментов отладки или интроспекции, нет возможности увидеть итоговую структуру дерева и расположение в памяти его элементов. Информация, которую на данный момент можно получить о дереве внутри SP-GIST, весьма скудна. Это, прежде всего, некоторая статистика касательно количества свободного и занятого мусором места, которую предоставляет поставляемый модуль pgstattuple. Кроме того, используя выше упомянутый модуль pageinspect, можно получить бинарное представление страниц памяти, в которых лежит индекс, но расшифровать эти бинарные данные – задача нетривиальная.

Данная работа своей целью ставит решение проблемы отсутствия встроенных инструментов интроспекции и отладки для индексов типа SP-GiST. Для решения данной задачи был выбран путь расширения дополнительно поставляемого модуля pageinspect поддержкой соответствующих индексов.

2. Возможные подходы и история решения задачи

На данный момент модуль pageinspect является единственным из дополнительно поставляемых модулей, который позволяет исследовать внутреннее устройство индексов. Данный модуль предоставляет возможность в удобном для последующей обработки виде sql-таблиц получить список всех структур данных, хранящихся на странице, вместе с их содержимым. Это позволяет как более тонко анализировать индекс (например, вычислять такие метрики, как сбалансированность или высоту дерева), так и отлаживать пользовательские функции, ошибки в которых возможно таким образом детектировать.

Помимо поставляемых с PostgreSQL модулей, есть также отдельное расширение Gevel [1], которое содержит различные функции для анализа индексов типа GiST, GIN и SP-GiST. На данный момент для индексов типа SP-GiST в нем реализованы две функции: spgist_stat и spgist_print. Функция spgist_stat выдает набор метрик дерева (см. листинг 1), функция же spgist_print выводит список (живых) вершин дерева с некоторыми их параметрами (см. листинг 2). Однако, хотя так и может показаться на первый взгляд, данное расширение не является полноценным решением поставленной задачи.

```
SELECT spgist_stat('spgist_idx');
          spgist_stat
----------------------------------
 totalPages:      21          +
 deletedPages:    0           +
 innerPages:      3           +
 leafPages:       18          +
 emptyPages:      1           +
 usedSpace:       121.27 kbytes+
 freeSpace:       46.07 kbytes +
 fillRatio:       72.47%      +
 leafTuples:      3669        +
 innerTuples:     20          +
 innerAllTheSame: 0           +
 leafPlaceholders: 569        +
 innerPlaceholders: 0         +
 leafRedirects:   0           +
 innerRedirects:  0
```

Листинг 1. Пример работы функции spgist_stat из расширения Gavel.

```
FUNCTION spgist_print(IN idx_name text,
   OUT tid tid,
   OUT allthesame bool,
   OUT node_n int,
   OUT level int,
   OUT tid_pointer tid,
   OUT prefix point,
   OUT node_label int,
   OUT leaf_value point)
```

Листинг 2. Примерная сигнатура функции spgist_print из расширения Gavel.

Проблема в том, что данное расширение не предоставляет всю информацию об индексе. В отличие от подхода, используемого pageinspect, в Gevel обрабатывается все дерево целиком и информация о физическом расположении данных в дереве не собирается. Это очень серьезное ограничение. Таким образом, например, невозможно с помощью расширения Gevel обнаружить причину большого количества мусора в индексе, так как оно может быть вызвано как плохой утилизацией ресурсов (что требует настройки vacuum), так и неверно выбранной функцией разделения узла дерева. Кроме того, такой подход ведет к ошибкам анализа. Например, сильная разбалансированность дерева, которая часто ведет к проблемам с производительностью в in-memory деревьях, здесь может не являться проблемой ввиду расположения вершин дерева в памяти всего на нескольких страницах, время на загрузку которых мажорирует время обработки страницы. Через Gevel данная информация, увы, не доступна.

Вышеуказанная проблема была также замечена и сообществом разработчиков PostgreSQL. В июле 2017 года автор расширения Gevel подготовил патч [2], который переносит функционал расширения в модуль pageinspect. Ввиду вышеуказанного недостатка, после долгого обсуждения, коммит отклонили [3]. Кроме того, дизайн данного расширения, увы, оказался несовместим с новыми требованиями [4].

Последнее на данный момент изменение в Gevel было в июне 2020 года. Поддерживается ли оно в настоящее время – сказать трудно, однако видимой работы по редизайну дополнения не ведется. Иных инструментов анализа индексов типа SP-GiST найдено не было. Таким образом, для поддержки полноценной интроспекции индексов типа SP-GiST можно выделить два пути: через расширение функционала модуля pageinspect и через реализацию собственного дополнения в папке contrib.

Подход, на котором основан модуль pageinspect никак не противоречит поставленной задаче. Кроме того, как мы далее увидим, расширение уже существующего модуля влечет определенные плюсы с точки зрения разработки, а именно: уже настроенную систему тестирования, версионирования и интеграции с остальным репозиторием PostgreSQL, а также уже реализованную блокировку

страниц памяти и нижележащих буфферов, что обеспечивает консистентность полученных данных. По этим причинам решено было именно расширять уже имеющийся модуль, а не писать новый.

3. Проделанная работа

Результатом данной работы, как и было заявлено, стало обновление для модуля pageinspect. В данном обновлении было добавлено четыре новых функции, которые позволяют на низком уровне изучать страничное устройство дерева типа SP-GiST. Все функции прошли тщательное тестирование, по результатам которого можно утверждать, что реализованные функции работают корректно и не ломают уже имеющийся в PostgreSQL функционал.

Сразу стоит упомянуть о специфике работы с PostgreSQL. Данный проект разрабатывался с 1986 года [5] и на данный момент обладает кодовой базой с более чем 775 000 строк кода. Сделать подробную архитектурную документацию к настолько большому проекту крайне трудно, поэтому основное время в процессе разработки уходит не на написание нового кода, а на просмотр уже имеющегося. Так, например, в официальной документации PostgreSQL [6] индексу типа SP-GiST посвящен целый раздел 69, который, тем не менее, содержит лишь инструкцию по расширению индекса с помощью реализации шаблонных методов и список уже поддержаных типов данных с перечнем разрешенных операторов. Таких же сведений как, например, описание реализованного алгоритма или используемых структур данных там нет. Описание алгоритма, лежащего в основе индекса типа SP-GiST, есть лишь в файле README.md в папке /src/backend/access/spgist/. Структуры данных же удалось найти лишь в исходном коде СУБД. Таким образом, первая и самая обширная часть проделанной работы заключалась в изучении уже написанного кода и поиске уже реализованных инструментов для работы с ними, что позволило в итоге написать короткий и читаемый код, максимально переиспользуя уже написанный.

Вторая часть работы заключалась в проектировании и реализации самих функций. На примере Gevel мы уже видели, как важен правильный дизайн, поэтому все реализованные функции проектировались так, чтобы все хранящиеся на странице данные были показаны пользователю, а так же чтобы сами возвращаемые таблицы были удобны для работы (в частности, нормализованы).

Третья, и последняя, часть работы заключалась в тестировании полученных функций. В данной части задача была продемонстрировать и проверить корректность работы на уже поддержанных типах данных со всеми стандартными типами индексов типа SP-GiST, таких как k-d дерево, дерево квадрантов и префиксное дерево.

Далее приведем результаты каждого из перечисленных этапов по отдельности.

4. Изучение кода

В данной главе речь пойдет об основных структурах данных и особенностях архитектуры PostgreSQL, которые понадобились для решения поставленной задачи. Данные знания критически

важны для решений, принятых на этапе дизайна новых функций модуля pageinspect. Для удобства чтения знания разделены по подглавам.

4.1. **Кеш индекса.** PostgreSQL хранит метаданные объектов базы данных, таких как таблицы, атрибуты, функции, операторы и т. д., в таблицах, называемых системными каталогами, которые очень похожи на таблицы, содержащие пользовательские данные. Пользователи могут запрашивать их с помощью SQL так же, как и свои собственные таблицы. Кроме того, хотя технически их можно изменить с помощью INSERT, UPDATE, DELETE, делать это не рекомендуется при нормальной работе базы данных; единственные изменения, которые они когда-либо получают, это те, которые автоматически выполняются системой при обработке операций DDL над объектами базы данных. В PostgreSQL существует несколько типов системных каталогов. Тот, знания о котором понадобятся нам далее, это relation cache (кеш индекса). Данный кеш хранит метаинформацию об индексе и полное его определение можно посмотреть в файле /src/include/utils/rel.h исходного кода PostgreSQL. Из более 60 полей данного класса нам понадобится одно – rd_amcache. Это данные, которые каждый тип индекса определяет себе самостоятельно. Индекс типа SP-GiST определяет структуру своего поля rd_amcache в файле /src/include/access/spgist_private.h следующим образом:

```
1   /*
2   * This struct is what we actually keep in index->rd_amcache. It includes
3   * static configuration information as well as the lastUsedPages cache.
4   */
5   typedef struct SpGistCache
6   {
7      spgConfigOut config;  /* filled in by opclass config method */
8
9      SpGistTypeDesc attType;  /* type of values to be indexed/restored */
10     SpGistTypeDesc attLeafType; /* type of leaf-tuple values */
11     SpGistTypeDesc attPrefixType; /* type of inner-tuple prefix values */
12     SpGistTypeDesc attLabelType; /* type of node label values */
13
14     SpGistLUPCache lastUsedPages; /* local storage of last-used info */
15  } SpGistCache;
```

Листинг 3. Кеш индекса типа SP-GiST.

Инициализация поля rd_amcache производится каждым индексом по-своему. В частности, в индексе типа SP-GiST данное поле в теории может оказаться неинициализированным. Однако вручную инициализировать его не придется. В файле /src/backend/access/spgist/spgutils.c исходного кода PostgreSQL уже определена функция spgGetCache, которая принимает на вход идентификатор индекса, инициализирует поле rd_amcache, если оно не было определено ранее, и возвращает ссылку на уже точно инициализированную структуру SpGistCache. Данный кеш понадобится нам

для корректной декодировки бинарных данных, так как де факто является единственным местом, где хранятся типы некоторых данных, о которых далее пойдет речь.

4.2. **О типах tid и datum.** Далее нам встретятся некоторые специфичные типы данных, о которых стоит сказать заранее. Тип ItemPointerData, часто называемый tid или ctid в документации, определен в файле /src/include/storage/itemptr.h исходного кода PostgreSQL и хранит в себе номер страницы и смещение в ней. Однако хранит он эти данные в сжатом виде (используя битовые оптимизации). Важно всегда проверять валидность ссылки специальной функцией ItemPointerIsValid, определенной в том же файле исходного кода. Как мы увидим далее, в структурах, отвечающих некоторым типам вершин в дереве, указатель специально делается невалидным.

Также нам часто будет встречаться тип данных Datum. Это абстракция над бинарными данными:

```
1   /*
2   * A Datum contains either a value of a pass-by-value type or a pointer to a
3   * value of a pass-by-reference type. Therefore, we require:
4   *
5   * sizeof(Datum) == sizeof(void *) == 4 or 8
6   *
7   * The functions below and the analogous functions for other types should be used
        to
8   * convert between a Datum and the appropriate C type.
9   */
10
11  typedef uintptr_t Datum;
```

Листинг 4. Определение Datum в коде PostgreSQL.

Однако бинарный вид данных пользователю возвращать, конечно, нельзя. С помощью хеша индекса можно узнать тип хранимых данных и далее перевести их в строковое представление с помощью функции OidOutputFunctionCall, определенной в файле /src/backend/utils/fmgr/fmgr.c исходного кода PostgreSQL.

4.3. **Страничное устройство БД.** Каждая таблица и индекс в PostgreSQL хранятся в виде массива страниц фиксированного размера (обычно 8 КБ, хотя при компиляции сервера можно выбрать другой размер страницы). В отличие от таблиц, где все страницы одинаковые, в индексах первая страница обычно зарезервирована как метастраница, содержащая управляющую информацию. Также в индексе могут быть разные типы страниц, в зависимости от метода доступа к индексу.

Первые 24 байта каждой страницы состоят из заголовка страницы (PageHeaderData). После заголовка страницы идут идентификаторы элементов (ItemIdData), каждый из которых занимает четыре байта. Идентификатор элемента содержит смещение в байтах от начала элемента, его длину в байтах и несколько битов атрибутов, которые влияют на его интерпретацию. Идентификаторы новых

элементов выделяются по мере необходимости с начала нераспределенного пространства. Количество присутствующих идентификаторов элементов можно определить, посмотрев на поле pd_lower в заголовке сраницы, которое увеличивается для выделения нового идентификатора. Поскольку идентификатор элемента никогда не перемещается до тех пор, пока он не будет освобожден, его индекс можно использовать в долгосрочной перспективе для ссылки на элемент, даже если сам элемент перемещается по странице для сжатия свободного пространства. Фактически, каждый указатель на элемент, созданный СУБД, состоит из номера страницы и индекса идентификатора элемента.

Сами элементы хранятся в пространстве, выделенном назад от конца нераспределенного пространства. Точная структура варьируется в зависимости от того, что должна содержать таблица.

Последний раздел — это «специальный раздел», который может содержать все, что желает сохранить индекс. Например, индексы b-дерева хранят ссылки на левые и правые одноуровневые страницы страницы, а также некоторые другие данные, относящиеся к структуре индекса. Обычные таблицы вообще не используют специальный раздел (такие страницы обозначаются установкой поля pd_special равным размеру страницы).

```
/*
* +----------------+---------------------------------+
* | PageHeaderData | linp1 linp2 linp3 ...      |
* +-----------+----+---------------------------------+
* | ... linpN |                         |
* +-----------+-----------------------------------+
* |       ^ pd_lower                    |
* |                                |
* |        v pd_upper                   |
* +-------------+-----------------------------------+
* |        | tupleN ...                        |
* +-------------+------------------+----------------+
* |     ... tuple3 tuple2 tuple1 | "special space" |
* +--------------------------------+----------------+
*                      ^ pd_special
*/
```

Листинг 5. Структура страницы памяти в PostgreSQL.

Более подробно об этом написано в главе 73.6 документации PostgreSQL. Кроме того, самая актуальная информация содержится в файле src/include/storage/bufpage.h исходного кода PostgreSQL.

4.4. **Типы страниц в индексах типа SP-GiST.** Вся информация о страницах хранится в файле src/include/access/spgist_private.h исходного кода PostgreSQL. В частности, специальный раздел для всех страниц индекса типа SP-GiST реализован структурой SpGistPageOpaqueData:

Здесь же зарезервированы доступные в структуре флаги:

```
/*
* Contents of page special space on SPGiST index pages
*/
typedef struct SpGistPageOpaqueData
{
   uint16     flags;       /* see bit definitions below */
   uint16     nRedirection; /* number of redirection tuples on page */
   uint16     nPlaceholder; /* number of placeholder tuples on page */
   /* note there's no count of either LIVE or DEAD tuples ... */
   uint16     spgist_page_id; /* for identification of SP-GiST indexes */
} SpGistPageOpaqueData;

typedef SpGistPageOpaqueData *SpGistPageOpaque;
```

Листинг 6. Специальный раздел страниц индекса типа SP-GiST.

```
/* Flag bits in page special space */
#define SPGIST_META     (1<<0)
#define SPGIST_DELETED  (1<<1)  /* never set, but keep for backwards
                         * compatibility */
#define SPGIST_LEAF     (1<<2)
#define SPGIST_NULLS  (1<<3)
```

Листинг 7. Флаги страниц индекса типа SP-GiST.

Исходя из описания алгоритма SP-GiST в файле /src/backend/access/spgist/README.md, кроме метастраницы (которая, не трудно догадаться, помечается флагом SPGIST_META), также есть еще два типа страниц: те, которые содержат только внутренние вершины дерева, и те, которые содержат только листовые вершины. Внутренние и внешние вершины хранятся в разных структурах, что позволяет экономить память. При этом страницы, предназначенные для хранения листовых вершин, помечаются флагом SPGIST_LEAF (и флагом SPGIST_NULLS, если страница в данный момент пуста), тогда как страницы со внутренними вершинами дерева не помечаются никакими флагами вообще.

4.5. **Структуры в памяти, отвечающие индексу типа SP-GiST.** Так как нам важно не упустить никакую важную для конечного пользователя информацию, кратко перечислим все структуры, которые хранятся на страницах индекса типа SP-GiST. Кроме уже упомянутого заголовка страницы и специального раздела, можно выделить три группы сущностей в зависимости от типа страницы на которых они встречаются.

Первая группа – это структуры, хранящиеся на метастранице индекса. На данной странице хранится лишь одна структура (см. листинг 8), которая содержит магическое число – уникальное число

для каждого индекса, которое позволяет их верифицировать, и содержит кеш недавно загружавшихся страниц:

```
1   * Each backend keeps a cache of last-used page info in its index->rd_amcache
2   * area. This is initialized from, and occasionally written back to,
3   * shared storage in the index metapage.
4   */
5   typedef struct SpGistLastUsedPage
6   {
7      BlockNumber blkno;      /* block number, or InvalidBlockNumber */
8      int         freeSpace;  /* page's free space (could be obsolete!) */
9   } SpGistLastUsedPage;
10
11  #define SPGIST_CACHED_PAGES 8
12
13  typedef struct SpGistLUPCache
14  {
15     SpGistLastUsedPage cachedPage[SPGIST_CACHED_PAGES];
16  } SpGistLUPCache;
17
18  /*
19  * metapage
20  */
21  typedef struct SpGistMetaPageData
22  {
23     uint32     magicNumber; /* for identity cross-check */
24     SpGistLUPCache lastUsedPages; /* shared storage of last-used info */
25  } SpGistMetaPageData;
26
27  #define SPGIST_MAGIC_NUMBER (0xBA0BABEE)
```

Листинг 8. Информация на метастранице индекса типа SP-GiST.

Отметим, что на метастранице не хранится никакой специфичной для дерева информации. Таким образом, нам нет смысла перегружать наше решение ненужным функционалом: вся эта информация нужна лишь для быстродействия хранилища и может быть получена анализом остальных страниц. Теперь расмотрим вторую группу структур – хранящиеся на страницах с внутренними вершинами.

Сразу стоит отметить разницу в терминологии. Во всем проекте структура, описывающая вершину дерева, обозначается термином tuple, то есть дословно “кортеж”, а структура, описывающая ребра в дереве, по неизвестным мне причинам, обозначается термином node, то есть, дословно,

как раз “вершина”. Далее по тексту под “вершинами” и “ребрами” будут подразумеваться именно вершины и ребра в смысле классической теории графов, то есть то, что в англоязычной литературе обозначается терминами vertices и edges.

Итак, каждый верхнеуровневый элемент страницы (тот, на который указывает один из идентификаторов на странице) здесь имеет тип SpGistInnerTuple и отвечает внутренней вершине:

```
1    typedef struct SpGistInnerTupleData
2    {
3       unsigned int tupstate:2, /* LIVE/REDIRECT/DEAD/PLACEHOLDER */
4                allTheSame:1, /* all nodes in tuple are equivalent */
5                nNodes:13,    /* number of nodes within inner tuple */
6                prefixSize:16; /* size of prefix, or 0 if none */
7       uint16     size;      /* total size of inner tuple */
8       /* On most machines there will be a couple of wasted bytes here */
9       /* prefix datum follows, then nodes */
10   } SpGistInnerTupleData;
11
12   typedef SpGistInnerTupleData *SpGistInnerTuple;
```

Листинг 9. Структура для хранения внутренней вершины дерева SP-GiST.

Каждая вершина дерева имеет одно из четырех зарезервированных состояний (см. листинг 10), число ребер, размер префикса и суммарный свой размер. Данные в префиксе зависят от реализации шаблонных методов индекса SP-GiST и доступны только через кеш индекса, о котором мы говорили ранее.

```
1    /* values of tupstate (see README for more info) */
2    #define SPGIST_LIVE     0  /* normal live tuple (either inner or leaf) */
3    #define SPGIST_REDIRECT  1 /* temporary redirection placeholder */
4    #define SPGIST_DEAD     2  /* dead, cannot be removed because of links */
5    #define SPGIST_PLACEHOLDER 3 /* placeholder, used to preserve offsets */
```

Листинг 10. Доступные состояния вершин дерева SP-GiST.

Не трудно заметить, что доступ к ребрам дерева и префиксу производится по смещениям. Однако руками перебирать смещения не нужно. Разработчики уже написали для этого удобные макросы в том же файле:

Ребра дерева индекс типа SP-GiST использует стандартные, объявленные в файле /src/include/access/itup.h (см. листинг 12), и использует на данный момент в них лишь поле t_tid, что, судя по комментариям в коде, может быть изменено в будущем.

После основных полей структуры IndexTupleData идут данные в бинарном виде – метка на соответствующем ребре дерева (label). Так, например, в реализации префиксного дерева на ребрах

```
#define SGITDATAPTR(x)  ((x)->prefixSize ? _SGITDATA(x) : NULL)
#define SGITDATUM(x, s)  ((x)->prefixSize ? \
                     ((s)->attPrefixType.attbyval ? \
                        *(Datum *) _SGITDATA(x) : \
                        PointerGetDatum(_SGITDATA(x))) \
                     : (Datum) 0)

/* Macro for iterating through the nodes of an inner tuple */
#define SGITITERATE(x, i, nt) \
   for ((i) = 0, (nt) = SGITNODEPTR(x); \
      (i) < (x)->nNodes; \
      (i)++, (nt) = (SpGistNodeTuple) (((char *) (nt)) + IndexTupleSize(nt)))
```

Листинг 11. Макросы для работы с внутренними вершинами дерева SP-GiST.

```
typedef IndexTupleData SpGistNodeTupleData;
typedef SpGistNodeTupleData *SpGistNodeTuple;
```

```
typedef struct IndexTupleData
{
   ItemPointerData t_tid;  /* reference TID to heap tuple */

   /* ---------------
    * t_info is laid out in the following fashion:
    *
    * 15th (high) bit: has nulls
    * 14th bit: has var-width attributes
    * 13th bit: AM-defined meaning
    * 12-0 bit: size of tuple
    * ---------------
    */

   unsigned short t_info;  /* various info about tuple */

} IndexTupleData;       /* MORE DATA FOLLOWS AT END OF STRUCT */
```

Листинг 12. Определение ребер дерева SP-GiST.

хранится код символа, по которому осуществляется переход. Тип метки также можно узнать только из кеша индекса.

Последняя группа структур – хранящиеся на страницах с листовыми вершинами. Здесь верхоуровневые элементы бывают двух типов: живые и мертвые (см. листинги 13 и 14 соответственно). Данное разделение введено лишь для унификации доступа к данным. На деле же, каждая мертвая

вершина располагается в той же памяти, где раньше была живая вершина, наследуя при этом большинство полей своего предка. Разница лишь в том, что после основных полей в живой вершине идет бинарное представление индексированных данных, а в мертвой вершине лишь идентификатор транзакции, которая вставила эту вершину. Кроме того, указатель в живой вершине указывает на данные в таблице, по которой построен индекс, тогда как в мертвой вершине, если указатель валиден, то он указывает на элемент в самом индексе.

```
1   typedef struct SpGistLeafTupleData
2   {
3      unsigned int tupstate:2, /* LIVE/REDIRECT/DEAD/PLACEHOLDER */
4               size:30;   /* large enough for any palloc'able value */
5      uint16     t_info;       /* nextOffset, which links to the next tuple
6                           * in chain, plus two flag bits */
7      ItemPointerData heapPtr; /* TID of represented heap tuple */
8      /* nulls bitmap follows if the flag bit for it is set */
9      /* leaf datum, then any included datums, follows on a MAXALIGN boundary */
10  } SpGistLeafTupleData;
```

Листинг 13. Структура для хранения живой листовой вершины дерева SP-GiST.

```
1   typedef struct SpGistDeadTupleData
2   {
3      unsigned int tupstate:2, /* LIVE/REDIRECT/DEAD/PLACEHOLDER */
4               size:30;
5      uint16     t_info;       /* not used in dead tuples */
6      ItemPointerData pointer; /* redirection inside index */
7      TransactionId xid;       /* ID of xact that inserted this tuple */
8   } SpGistDeadTupleData;
9
10  typedef SpGistDeadTupleData *SpGistDeadTuple;
```

Листинг 14. Структура для хранения мертвой листовой вершины дерева SP-GiST.

Зачем индексу хранить идентификатор транзакции? Листовые вершины могут быть мертвыми по двум причинам: кто-то удалил данную вершину из таблицы или же данная вершина была перенесена в другое место для оптимизации поиска самим индексом. Во втором случае, так как структура допускает конкурентный доступ с минимальными блокировками, то, для сохранения консистентности, перед тем, как окончательно удалить вершину, нужно сначала заменить ее ссылкой на свою копию, а потом уже удалять. При такой процедуре приходится писать и читать из нескольких страниц, что работает долго, поэтому на деле ликвидацией ссылок занимается сборщик мусора (vacuum), а не сама процедура переноса, которая лишь создает копию вершины и оставляет на нее ссылку. Сборщику мусора и нужен id транзакции, чтобы не удалить как мусор вершины еще

не завершенной транзакции. В целом, информация, которая отличается у мертвых вершин – это внутренняя информация СУБД, не имеющая отношения к данным в индексе. Выдавать такую информацию пользователю не нужно.

Отметим также, что как и в случае с внутренними вершинами, нам не нужно самим извлекать Datum (который может содержать значения нескольких колонок), для этого уже реализована функция spgDeformLeafTuple, находящаяся в файле /src/backend/access/spgist/spgutils.c и возвращающая список из Datum и nulls для каждой хранящейся колонки. Здесь nulls – это булево значение, означающее содержит ли Datum данные или же искомое значение – это null.

Таким образом, мы описали все данные, которые хранятся на страницах индекса, и способы работы с ними. Опишем теперь сами реализованные функциях и то API, которое они предоставляют пользователю.

5. Разработанные функции

Начнем с описания дизайна дополнительного поставляемого модуля pageinspect. В целях обратной совместимости все функции данного модуля и изменения в них содержатся в отдельных sql-патчах. На данный момент реализовано 12 патчей. Каждый патч содержит список объявлений функций через метод CREATE FUNCTION как в листинге 15. При этом в методе передается название функции, которая используется для обработки данного запроса.

```
1   --
2   -- get_raw_page()
3   --
4   CREATE FUNCTION get_raw_page(text, int4)
5   RETURNS bytea
6   AS 'MODULE_PATHNAME', 'get_raw_page'
7   LANGUAGE C STRICT PARALLEL SAFE;
8
9   CREATE FUNCTION get_raw_page(text, text, int4)
10  RETURNS bytea
11  AS 'MODULE_PATHNAME', 'get_raw_page_fork'
12  LANGUAGE C STRICT PARALLEL SAFE;
```

Листинг 15. Функция загрузки бинарной страницы памяти из модуля pageinspect.

Чтобы функция-обработчик нашлась, нужно чтобы для нее существовала так называемая info function, которая ассоциирована с данной функцией-обработчиком. Чтобы создать info function, достаточно в коде вызвать макрос PG_FUNCTION_INFO_V1, объявленный в файле /src/include/fmgr.h исходного кода PostgreSQL, передав в него название функции-обработчика.

Модуль pageinspect активно переиспользует уже написанный код. Так, например, в функции get_raw_page, приведенной на листинге (см. листинг 15), уже реализованы все необходимые блокировки для доступа к страницам индекса. Таким образом, для выполнения поставленной задачи нет

необходимости писать блокировку самостоятельно. Наши функции могут принимать на вход уже скопированную страницу памяти. Более того, такой дизайн является идеоматичным для данного модуля и им пользуются функции, реализующие интроспекцию для индексов, например, типа GiST и GIN.

По аналогии с функциями для близкого по API индекса типа GiST было реализовано 4 фукнции. Далее разберем каждую из них по отдельности.

5.1. **Специальная информация.** Первая функция, по аналогии с соответствующей функцией gist_page_opaque_info для индекса типа GiST, принимает на вход лишь копию страницы памяти и возвращает пользователю информацию о типе страницы:

```
1   --
2   -- spgist_page_opaque_info()
3   --
4   CREATE FUNCTION spgist_page_opaque_info(IN page bytea,
5      OUT lsn pg_lsn,
6      OUT nDirection smallint,
7      OUT nPlaceholder smallint,
8      OUT flags text[])
9   AS 'MODULE_PATHNAME', 'spgist_page_opaque_info'
10  LANGUAGE C STRICT PARALLEL RESTRICTED;
```

Листинг 16. Функция интроспекции специальной информации страницы индекса SP-GiST.

Исходя из приведенного в прошлой части описания структур данных, это вся доступная информация касательно типа страницы. Результат представляет собой таблицу с одной строкой-результатом и четыремя столбцами. В первом столбце хранится номер последнего изменения страницы, число вершин типа SPGIST_REDIRECT, число вершин типа SPGIST_PLACEHOLDER и флаги, по которым можно определить тип страницы.

Данная функция, как и все разобранные далее, доступна к вызову только суперпользователем (администратором СУБД) и верифицирует страницу, то есть, если передать страницу, которая не пренадлежит индексу типа SP-GiST, то вернется ошибка.

5.2. **Внутренние вершины.** Вторая реализованная функция (см. листинг 17) принимает на вход не только копию страницы памяти, но и название индекса. Это необходимо для того, чтобы обратиться к кешу индекса и узнать какие типы данных хранятся в бинарном виде в поле prefix. Данное решение не нарушает дизайн модуля pageinspect, так как по тем же причинам, например, для индекса типа GiST функция gist_page_items также требует названия индекса.

Функция spgist_inner_tuples возвращает список всех вершин на странице. Для каждой вершины, при этом, в одной строке выводится ее смещение на странице, состояние, флаг all_the_same

```
--
-- spgist_inner_tuples()
--
CREATE FUNCTION spgist_inner_tuples(IN page bytea,
   IN index_oid regclass,
   OUT tuple_offset smallint,
   OUT tuple_state text,
   OUT all_the_same boolean,
   OUT node_number int,
   OUT prefix_size int,
   OUT total_size int,
   OUT pref text)
RETURNS SETOF record
AS 'MODULE_PATHNAME', 'spgist_inner_tuples'
LANGUAGE C STRICT PARALLEL RESTRICTED;
```

Листинг 17. Функция интроспекции страницы с внутренними вершинами SP-GiST.

(который выставляется, если у вершины слишком много детей, и вершины пришлось разбить на несколько объединенных одной виртуальной), число ребер, размер поля prefix, размер всей вершины и текстовое представление префикса. Таким образом, перечисляются все поля соответствующей структуры, кроме ребер.

Ребра было решено вынести в отдельную функцию. Сделано это по нескольким причинам. Прежде всего, это, конечно, нормализация таблицы. Каждая вершина может иметь множество ребер, так что генерировать по отдельной строчке на каждое ребро – не разумно. Рассматривался вариант склейки описания всех ребер в одну строчку, как это сделано в соответствующей функции для анализа индексов типа GiST. Причиной отказа от такого решения (которое и было изначально реализовано) служит число ребер. На этапе тестирования выяснилось, что, например, у префиксного дерева у каждой внутренней вершины может быть вплоть до 50 ребер и больше. Склейка информации воедино давала очень длинную строку, которую сложно как парсить так и читать. По этой причине для работы со страницами, хранящими внутренние вершины, было реализовано две функции.

5.3. **Ребра внутренних вершин.** Вторая функция по API аналогична первой (см. листинг 17), но возвращает лишь смещение вершины, номер блока вершины, на которую ссылается ребро, смещение этой вершины в блоке и текстовое представление метки (label) на ребре. Метка на странице хранится в бинарном виде, поэтому для текстового представления в данной функции нам также нужно название индекса.

5.4. **Листовые вершины.** Четвертая реализованная функция (см. листинг 19), как и две предыдущих, требует на вход не только копию страницы памяти, но и название индекса. Возвращает она список всех листовых вершин на странице.

```
--
-- spgist_inner_tuples_nodes()
--
CREATE FUNCTION spgist_inner_tuples_nodes(IN page bytea,
    IN index_oid regclass,
    OUT tuple_offset smallint,
    OUT node_block_num int,
    OUT node_offset smallint,
    OUT node_label text)
RETURNS SETOF record
AS 'MODULE_PATHNAME', 'spgist_inner_tuples_nodes'
LANGUAGE C STRICT PARALLEL RESTRICTED;
```

Листинг 18. Функция интроспекции ребер внутренних вершин на странице SP-GiST.

```
   --
   -- spgist_page_items()
   --
   CREATE FUNCTION spgist_leaf_tuples(IN page bytea,
      IN index_oid regclass,
      OUT item_offset smallint,
      OUT item_state text,
      OUT item_size int,
      OUT item_info smallint,
      OUT leaf_key text,
      OUT pointer_block_num int,
      OUT pointer_offset smallint)
   RETURNS SETOF record
   AS 'MODULE_PATHNAME', 'spgist_leaf_tuples'
   LANGUAGE C STRICT PARALLEL SAFE;
```

Листинг 19. Функция интроспекции страницы с листовыми вершинами SP-GiST.

Каждая строка полученной таблицы содержит всю информацию об одной листовой вершине, а именно ее смещение на странице, состояние, размер, поле t_info, ключ таблицы, соответсвующий данной вершине, а также на какой блок с каким смещением вершина ссылается.

Как упоминалось в предыдущей главе, на страницах данного типа есть два типа структур, различия между которыми для пользователя лишь в наличии ключа. Однако, хотя структуры и не различаются, можно заметить, что на их содержимое вляет тип хранимой вершины. Так, в живых вершинах (SPGIST_LIVE) есть ключ, а в остальных вершинах он будет null. В вершинах-ссылках (SPGIST_REDIRECT) указатель ссылается на данные индекса, а в живых вершинах на данные индексируемой таблицы. С точки зрения нормализации эту таблицу тоже нужно разделять, однако было принято решение этого не делать. Сделано это было ввиду небольшого размера самой выходной таблицы. В 8kb стандартной страницы PostgreSQL помещается всего 200-300 вершин.

Написать простой фильтр по типу вершин пользователю куда проще, чем объединять множество запросов.

Таким образом, была поддержана интроспекция индексов типа SP-GiST. Каждая функция была протестирована на разных реализациях индекса. Об этом и пойдет речь в следующей главе.

6. Тестирование

Система тестов, встроенная в PostgreSQL, довольно примитивна. В папке sql/ внутри модуля pageinspect для каждого файла вида <name>.c исходного кода модуля ищется файл вида <name>.sql. Этот файл исполняется на временном сервере PostgreSQL, и результаты выполнения запроса сравниваются с “ожидаемым” файлом с названием вида <name>.out, лежащим в папке expected/ модуля pageinspect. Если файлы посимвольно совпадают, то тест пройден, иначе система выдает ошибку.

На данный момент у индекса типа SP-GiST существует несколько стандартных реализаций шаблонных методов. Есть три реализации в файлах spgkdtreeproc.c, spgquadtreeproc.c и spgtextproc.c папки /src/backend/access/spgist/ исходного кода PostgreSQL, которые реализуют, соответственно, k-d дерево и дерево квадрантов для типа пары, а также префиксное дерево для строк. Кроме того, есть еще несколько реализаций для более сложных типов данных, вроде областей или ip-адресов, в файлах rangetypes_spgist.c, geo_spgist.c и network_spgist.c папки /src/backend/utils/adt/ исходного кода PostgreSQL. На данный момент это все стандартные реализации. Полный список реализаций шаблонных методов можно найти в файле /src/include/catalog/pg_amproc.dat, где хранятся данные о таких реализациях вообще для всех типов индексов.

Выбор типов данных, хранящихся на страницах индекса в бинарном формате, а также выбор используемого алгоритма определяется каждой реализацией шаблонного метода самостоятельно. Однако для тестирования нашего модуля нам не нужно тестировать каждую из этих реализаций, так как наш код никаким образом не опирается на выбранные типы хранимых данных (генерация текстового представления данных делегируется внешней системе) и алгоритмы (мы работаем лишь со данными, которые всегда уложены одинаково). Важно, чтобы были протестированы все крайние случаи, когда, например, данные могут отсутствовать, ссылки могут быть не валидными и так далее. Для того, чтобы покрыть все эти случаи, раскрытые в главе с описанием структур данных, хватило всего пары тестов. Для наглядности, по итогу, было реализовано четыре теста: на дерево квадрантов, префиксное дерево, k-d дерево и дерево квадрантов на основе областей.

Опишем структуру каждого теста на примере дерева квадрантов. Для типа данных point стандартным алгоритмом является реализация дерева квадрантов, описанная в файле /src/backend/access/spgist/spgquadtreeproc.c исходного кода PostgreSQL. В данном дереве на ребрах нет пометок, а префикс имеет тип point.

Инициализация теста происходит стандартным образом как показано на листинге 20.

После инициализации мы можем воспользоваться функцией spgist_page_opaque_info для получения информации об одной из страниц. Чтобы получить данные по всем страницам, в одной таблице

```
--
-- Create a test table for future quad index
--
CREATE TABLE test_spgist (
    pt point,
    t varchar(256)
);
--
--
-- Create quad index to test
--
CREATE INDEX test_spgist_idx ON test_spgist USING spgist (pt);
--
--
-- Add data to test table
--
INSERT INTO test_spgist
    SELECT point(i+500,i), i::text
    FROM generate_series(1,500) i;
--
INSERT INTO test_spgist
    SELECT point(i,i+500), (i+500)::text
    FROM generate_series(1,500) i;
--
```

Листинг 20. Инициализация теста.

в тестах использовался метод UNION (см. листинг 21). Как и ожидалось, в дереве первая страница содержит метаинформацию. Кроме того, страницы 2, 5 и 6 содержат лишь внутренние вершины, а остальные 12 страниц содержат листовые вершины.

Несложным sql-запросом, использующим функции spgist_inner_tuples и spgist_inner_tuples_nodes, выведем все вершины на шестой странице (см. листинг 22). Поле label, ожидаемо, пусто, prefix же отобразился корректно и имеет корректный размер.

Теперь воспользуемся функцией spgist_leaf_tuples, чтобы увидеть, какие вершины хранятся на седьмой странице тестового индекса (см. листинг 23). Как мы видим, ключи в таблице отображаются корректно, вершины-ссылки ожидаемо не содержат ключей, но также ссылаются на другие вершины. Кроме того, можно заметить, что 114-я вершина, которая была листовой, ссылается на внутреннюю вершину 2, информацию о которой мы уже видели (см. листинг 22). Это совершенно нормальное, ожидаемое поведение: квадрант, соответствующий данной вершине, получил в ходе заполнения индекса еще одну точку, после чего вершина из внешней стала внутренней. Также можно заметить, что живые вершины указывают на страницу с номером 0. Это не метастраница индекса, а первая страница реальной таблицы test_spgist, то есть косвенно подтверждается корректность считанных из памяти данных.

Таким образом, мы проверили, что все данные отображаются корректно. Однако, мы не встретили одного типа вершин – мертвых. Действительно, в колонке item_state мы ни разу не встретили состояния dead. Дело тут в том, что тест отработал быстро, и сборщик мусора просто не успел еще отработать. Это можно исправить вручную вызвав команду

```
--
-- cannot actually test lsn because the value can vary, so check everything else
-- Page 0 is the root, the rest are leaf pages:
--
SELECT 0 pageNum, nDirection, nPlaceholder, flags FROM spgist_page_opaque_info(get_raw_page('test_spgist_idx',
    0)) UNION
SELECT 1, nDirection, nPlaceholder, flags FROM spgist_page_opaque_info(get_raw_page('test_spgist_idx', 1))
    UNION
...
...
SELECT 15, nDirection, nPlaceholder, flags FROM spgist_page_opaque_info(get_raw_page('test_spgist_idx', 15))
ORDER BY pageNum;
 pagenum | ndirection | nplaceholder | flags
---------+------------+--------------+-------------
       0 |          0 |            0 | {meta}
       1 |          0 |            0 | {}
       2 |          0 |            0 | {leaf,nulls}
       3 |          1 |          112 | {leaf}
       4 |          0 |            0 | {}
       5 |          0 |            0 | {}
       6 |          1 |          168 | {leaf}
       7 |          1 |          140 | {leaf}
       8 |          1 |          154 | {leaf}
       9 |          1 |           51 | {leaf}
      10 |          1 |          225 | {leaf}
      11 |          1 |          112 | {leaf}
      12 |          1 |          168 | {leaf}
      13 |          1 |          140 | {leaf}
      14 |          1 |          154 | {leaf}
      15 |          0 |            0 | {leaf}
(16 rows)
```

Листинг 21. Просмотр типов страниц индекса типа SP-GiST.

```
VACUUM (INDEX_CLEANUP ON) test_spgist;
```

Вышеуказанные запросы были, конечно, выполнены и для всех остальных страниц индекса. В конце теста база данных удаляется и начинается следующий тест (см. листинг 24).

Остальные 3 теста также следуют описанной стратегии, однако используют реальные датасеты, такие как bounding boxes стран мира или координаты городов. Также, при добавлении этих датасетов в таблицу используется “приближенная к жизни” стратегия, когда после, например, восьмидесяти добавленных строчек двадцать рандомных удаляются, и такая процедура повторяется 4 раза. Данные тесты были призваны потенциально выявить какие-либо скрытые проблемы/баги написанного кода, но в результате выяснилось лишь, что таблица с внутренними вершинами нуждается в нормализации, что было исправлено. Также, некоторые sql-запросы имели более сложную структуру как, например, на листинге 22. Это делалось исключительно для улучшения читаемости таблиц или демонстрации, почему отсутствие нормализации таблицы для внутренних вершин влечет полную нечитабельность результата.

Также стоит отметить, что реализованные функции никак не меняли уже имеющийся в СУБД код (вне модуля pageinspect) и потому не могли сломать уже имеющийся функционал. Тем не

```
WITH nodes as (
    SELECT
        tuple_offset tuple_offset,
        STRING_AGG('(BN=' || node_block_num || ', Offset=' || node_offset || ', Label=' || node_label || ')',
            ', ') edges
    FROM spgist_inner_tuples_nodes(get_raw_page('test_spgist_idx', 5), 'test_spgist_idx')
    GROUP BY tuple_offset
) SELECT
    tuples.tuple_offset AS offset,
    tuples.tuple_state AS state,
    tuples.all_the_same AS same,
    tuples.node_number,
    tuples.prefix_size,
    tuples.total_size,
    tuples.pref,
    nodes.edges
FROM spgist_inner_tuples(get_raw_page('test_spgist_idx', 5), 'test_spgist_idx') AS tuples
JOIN nodes
ON tuples.tuple_offset = nodes.tuple_offset;
 offset | state | same | node_number | prefix_size | total_size | pref | edges
--------+-------+------+-------------+-------------+------------+------+--------
1 | live | f | 4 | 16 | 56 | (670.5,170.5) | (BN=5, Offset=2, Label=), (BN=6, Offset=113, Label=)
2 | live | f | 4 | 16 | 56 | (755.5,255.5) | (BN=5, Offset=3, Label=), (BN=7, Offset=169, Label=)
3 | live | f | 4 | 16 | 56 | (826.5,326.5) | (BN=5, Offset=4, Label=), (BN=8, Offset=128, Label=)
4 | live | f | 4 | 16 | 56 | (904.5,404.5) | (BN=9, Offset=156, Label=), (BN=9, Offset=155, Label=)
5 | live | f | 4 | 16 | 56 | (114,614)     | (BN=5, Offset=6, Label=), (BN=11, Offset=226, Label=)
6 | live | f | 4 | 16 | 56 | (170.5,670.5) | (BN=5, Offset=7, Label=), (BN=12, Offset=57, Label=)
7 | live | f | 4 | 16 | 56 | (255.5,755.5) | (BN=5, Offset=8, Label=), (BN=13, Offset=169, Label=)
8 | live | f | 4 | 16 | 56 | (326.5,826.5) | (BN=5, Offset=9, Label=), (BN=14, Offset=128, Label=)
9 | live | f | 4 | 16 | 56 | (404.5,904.5) | (BN=15, Offset=156, Label=), (BN=15, Offset=155, Label=)
(9 rows)
```

Листинг 22. Просмотр внутренних вершин с ребрами на странице индекса типа SP-GiST.

менее по окончанию работы над модулем были запущены все тесты, имеющиеся в PostgreSQL. Тесты отработали в штатном режиме, что позволяет утверждать, что модуль не нарушает работу остальных составляющих СУБД.

```
--
-- Page 6
--
SELECT * FROM spgist_leaf_tuples(get_raw_page('test_spgist_idx', 6), 'test_spgist_idx');
 item_offset | item_state | item_size | item_info | leaf_key     | pointer_block_num | pointer_offset
-------------+------------+-----------+-----------+--------------+-------------------+----------------
           1 | placeholder |      16 |        0 |                    |                   |
           2 | placeholder |      16 |        0 |                    |                   |
           3 | placeholder |      16 |        0 |                    |                   |
           4 | placeholder |      16 |        0 |                    |                   |
           5 | placeholder |      16 |        0 |                    |                   |
...
...
         110 | live        |      32 |      109 | (pt)=("(617,117)") |                 0 |           117
         111 | live        |      32 |      110 | (pt)=("(616,116)") |                 0 |           116
         112 | live        |      32 |      111 | (pt)=("(615,115)") |                 0 |           115
         113 | live        |      32 |      112 | (pt)=("(614,114)") |                 0 |           114
         114 | redirect    |      16 |        0 |                    |                 5 |             2
         115 | placeholder |      16 |        0 |                    |                   |
         116 | placeholder |      16 |        0 |                    |                   |
         117 | placeholder |      16 |        0 |                    |                   |
         118 | placeholder |      16 |        0 |                    |                   |
...
...
         224 | placeholder |      16 |        0 |                    |                   |
         225 | placeholder |      16 |        0 |                    |                   |
         226 | placeholder |      16 |        0 |                    |                   |
(226 rows)
```

Листинг 23. Просмотр листовых вершин на странице индекса типа SP-GiST.

```
--
-- Page 16
--
SELECT * FROM spgist_leaf_tuples(get_raw_page('test_spgist_idx', 16), 'test_spgist_idx');
ERROR: block number 16 is out of range for relation "test_spgist_idx"
--
--
-- Drop a test table for quad tree
--
DROP TABLE test_spgist;
```

Листинг 24. Завершение теста.

Заключение

В ходе проделанной работы была изучена часть исходного кода PostgreSQL, ответственная за работу индексов. По результатам исследования были реализованы и тщательно протестированы 4 новые функции для модуля pageinspect, являющегося дополнительно поставляемым модулем PostgreSQL. Данные функции добавляют возможность исследовать внутреннее устройство индекса типа SP-GiST и отлаживать кастомные реализации его шаблонных методов.

При реализации данных функций был учтен неудачный опыт расширения Gevel, патч на добавление функций которого в модуль pageinspect был отклонен в 2017 году. Также, написанный код максимально переиспользует возможности, предоставляемые API компонент PostgreSQL, что делает решение более легковесным и устойчивым к будущим изменениям в СУБД.

Патч с реализованными функциями планируется отправить на добавление в официальный репозиторий PostgreSQL. Есть надежда, что наличие данного функционала в новых версиях дополнительно поставляемого модуля pageinspect облегчит работу многим администраторам и инженерам, работающим с PostgreSQL.

Список литературы

[1] Link: http://www.sai.msu.su/~megera/wiki/Gevel

[2] Link: https://www.postgresql.org/message-id/accae316-5e4d-8963-0c3d-277ef13c396c%40postgrespro.ru

[3] Link: https://www.postgresql.org/message-id/CAB7nPqRkxoaXceBTMj_5ob7vTK-ikG537uYZfu-JMJEOY4toig%40mail.gmail.com

[4] Link: https://www.postgresql.org/message-id/CAPpHfdu1ZoQXvRp9PGnO5rarwPardiS1vC0iO6pakwyc395eFQ%40mail.gmail.com

[5] Link: https://en.wikipedia.org/wiki/PostgreSQL

[6] Link: https://www.postgresql.org/docs/current/internals.html