

Доступ к данным Типы индексов



Авторские права

© Postgres Professional, 2019–2024

Авторы: Егор Рогов, Павел Лузанов, Павел Толмачев, Илья Баштанов

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Хеш-индекс

GiST

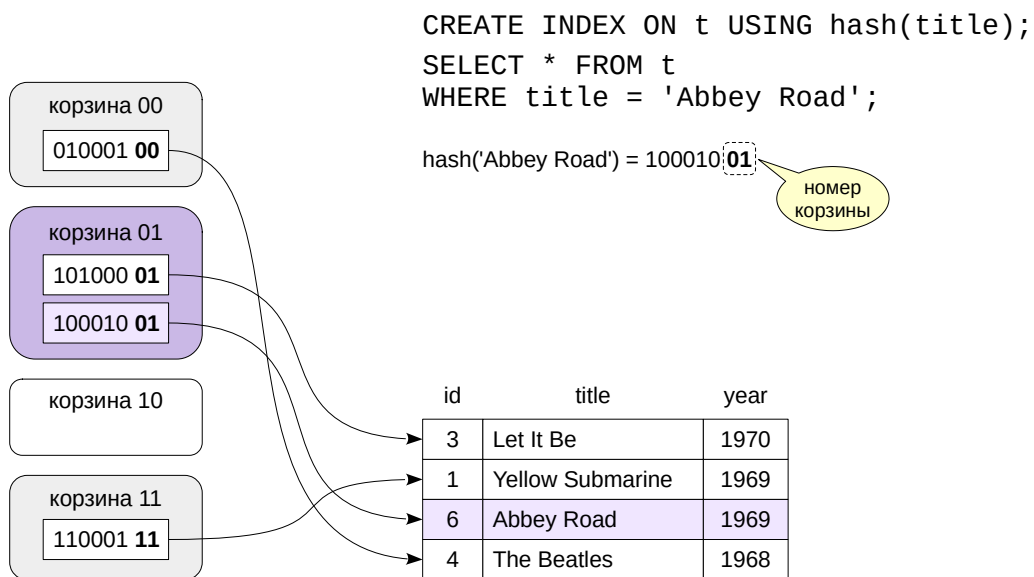
Класс операторов

SP-GiST

GIN

BRIN

Идея хеширования



3

Идея хеширования состоит в том, что значения любого типа данных *равномерно* распределяются по ограниченному количеству корзин хеш-таблицы с помощью функции хеширования. Если хеш-таблица имеет достаточный размер, чтобы в одну корзину в среднем попадало одно значение (хеш-код), поиск значения в хеш-таблице выполняется за константное время. Для этого:

- 1) вычисляется хеш-функция от заданного значения;
- 2) по нескольким битам полученного хеш-кода определяется номер корзины;
- 3) корзина просматривается в поисках хеш-кода.

Если данные распределены неравномерно, в одну корзину могут попасть много значений. В этом случае эффективность поиска будет страдать.

По сути, хеш-индекс представляет собой хеш-таблицу, которая хранится на диске.

<https://postgrespro.ru/docs/postgresql/16/hash-intro>

Хранит только хеш-коды, но не исходные значения

размер не зависит от ключа индексирования
невозможно сканирование только индекса

Индекс увеличивается динамически

скачкообразный рост

Поиск только по условию равенства

Хеш-индекс хранит только значения хеш-функции и ссылки на версии строк; само индексируемое значение не сохраняется. Поэтому размер хеш-индекса не зависит от размера ключа индексирования, но теряется возможности сканирования только индекса — значение можно прочитать только из таблицы.

Размер хеш-индекса увеличивается динамически при добавлении новых значений. При увеличении количество корзин удваивается, поэтому рост размера происходит скачкообразно.

В отличие от В-дерева, хеш-индексы имеют много ограничений, в частности:

- единственная поддерживаемая операция — поиск по условию равенства, поскольку хеш-функция не сохраняет порядок следования значений;
- не поддерживается ограничение уникальности;
- нельзя создать многоколоночный индекс и добавить к индексу дополнительные include-столбцы.

Поэтому хеш-индексы не получили широкого распространения. Однако за счет меньшего размера и фиксированного времени поиска хеш-индекс может в ряде случаев работать быстрее, чем индекс на основе В-дерева.

Хеш-индекс

Посмотрим на план запроса, получающего список кодов самолетов, в которых есть место с определенным номером:

```
=> EXPLAIN (costs off)
SELECT * FROM seats WHERE seat_no = '31D';
```

QUERY PLAN

```
-----
Seq Scan on seats
  Filter: ((seat_no)::text = '31D'::text)
(2 rows)
```

Поскольку подходящего индекса нет, используется последовательное сканирование. Создадим хеш-индекс по полю seat_no и повторим запрос:

```
=> CREATE INDEX ON seats USING hash(seat_no);
```

CREATE INDEX

```
=> EXPLAIN (costs off)
SELECT * FROM seats WHERE seat_no = '31D';
```

QUERY PLAN

```
-----
Bitmap Heap Scan on seats
  Recheck Cond: ((seat_no)::text = '31D'::text)
  -> Bitmap Index Scan on seats_seat_no_idx
       Index Cond: ((seat_no)::text = '31D'::text)
(4 rows)
```

Теперь планировщик использует хеш-индекс и строит битовую карту. Поменяем условие равенства на «больше»:

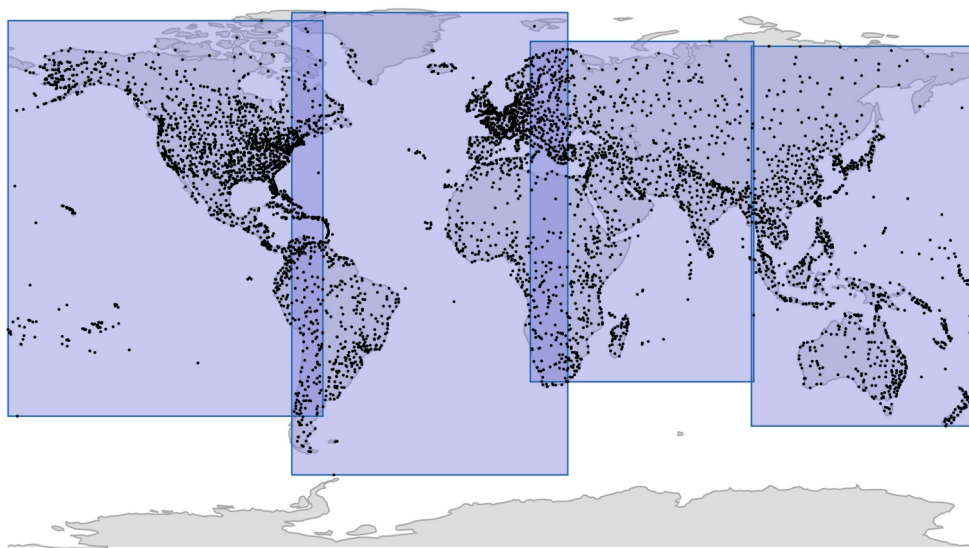
```
=> EXPLAIN (costs off)
SELECT * FROM seats WHERE seat_no > '31D';
```

QUERY PLAN

```
-----
Seq Scan on seats
  Filter: ((seat_no)::text > '31D'::text)
(2 rows)
```

С неравенствами хеш-индекс использоваться не может.

Пример индекса GiST



6

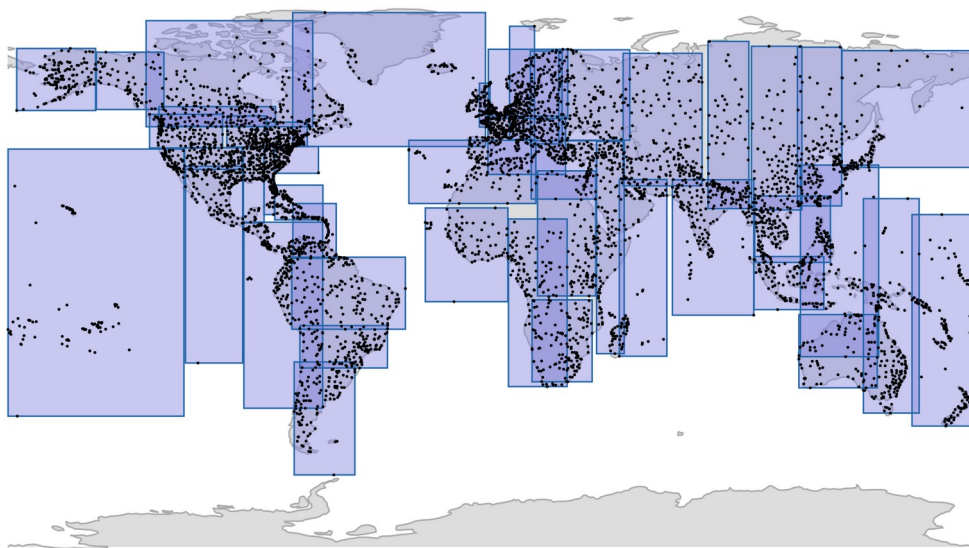
GiST расшифровывается как *generalized search tree* — обобщенное дерево поиска.

Идею работы GiST-индексов рассмотрим на примере точек на плоскости.

Плоскость разбивается на несколько прямоугольников, которые в сумме покрывают все индексируемые точки. Эти прямоугольники составляют верхний уровень дерева.

Как видно на рисунке, прямоугольники могут пересекаться (хотя это и уменьшает эффективность поиска).

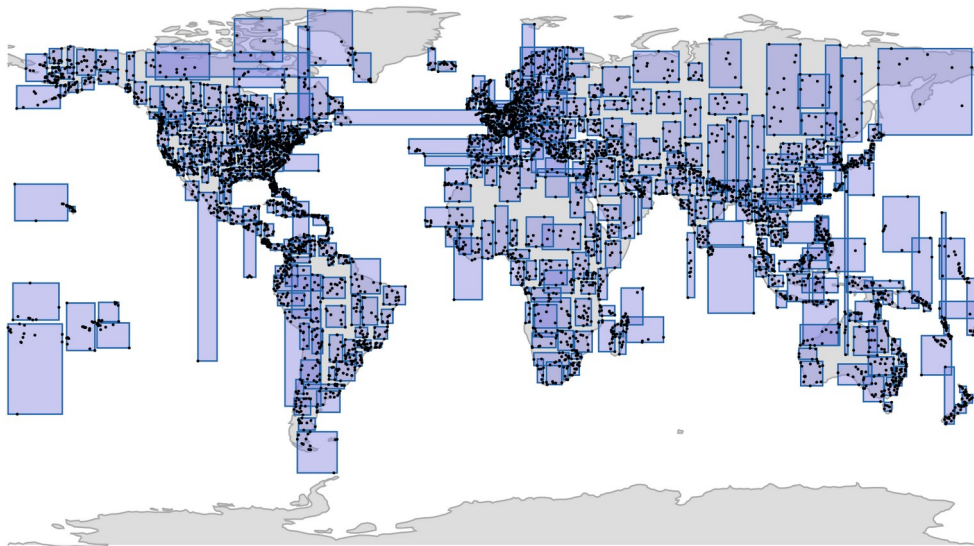
Пример индекса GiST



7

На следующем уровне дерева каждый из больших прямоугольников распадается на прямоугольники меньшего размера.

Пример индекса GiST



8

На последнем уровне дерева каждый ограничивающий прямоугольник будет содержать столько точек, сколько помещается на одну индексную страницу.

Общее условие разбиения таково: прямоугольник родительской вершины охватывает все прямоугольники соответствующего поддеревя. Это позволяет, например, быстро находить точки, лежащие внутри определенной области:

- 1) находим прямоугольники, пересекающиеся с заданной областью, на верхнем уровне индекса;
- 2) спускаемся в выбранные поддеревья и повторяем поиск в них.

Такой алгоритм индексирования называется R-деревом.

Сбалансированное дерево поиска

- рассчитано на произвольные типы данных
- не требуется упорядоченность

Типичные поддерживаемые операции

- вхождение в область
- нахождение слева, справа, сверху, снизу от области

Поиск ближайших соседей

- первые k значений, ближайших к заданному

В-дерево представляет собой сбалансированное дерево, значения в котором упорядочены в соответствии с операциями «больше» и «меньше». Индекс GiST также образует сбалансированное дерево, но значения в нем распределяются по другому принципу, например, на основе взаимного расположения точек на плоскости.

Поэтому GiST можно применять к тем типам данных, для которых не имеют смысла операции «больше» и «меньше», и ускорять другие, более важные для этих типов, операции. Например, для геометрических фигур на плоскости индекс GiST может ускорять поиск вхождения значения в определенную область или поиск значений, находящихся в определенной стороне от заданной области.

Другая важная особенность GiST-индекса — поддержка поиска ближайших соседей. Индекс позволяет быстро получить несколько значений, ближайших к заданному.

<https://postgrespro.ru/docs/postgresql/16/gist-intro>

Индекс GiST

Для демонстрации работы GiST-индекса обратимся к таблице `airports_data` (на этой таблице построено представление `airports`). В таблице есть поле `coordinates` типа `point`, по нему и будем строить GiST-индекс.

Но сначала выполним следующий запрос — найдем все аэропорты, находящиеся недалеко от Москвы:

```
=> EXPLAIN (costs off)
SELECT airport_code
FROM airports_data
WHERE coordinates <@ '<(37.622513,55.753220),1.0>'::circle;

QUERY PLAN
-----
Seq Scan on airports_data
  Filter: (coordinates <@ '<(37.622513,55.753220),1>'::circle)
(2 rows)
```

Без индекса просматривается вся таблица. Создадим GiST-индекс:

```
=> CREATE INDEX airports_gist_idx ON airports_data
USING gist(coordinates);
```

CREATE INDEX

Таблица `airports_data` невелика, поэтому планировщик все равно будет использовать последовательное сканирование. Временно отключим этот метод доступа:

```
=> SET enable_seqscan = off;
```

SET

Повторим запрос:

```
=> EXPLAIN (costs off)
SELECT airport_code
FROM airports_data
WHERE coordinates <@ '<(37.622513,55.753220),1.0>'::circle;

QUERY PLAN
-----
Index Scan using airports_gist_idx on airports_data
  Index Cond: (coordinates <@ '<(37.622513,55.753220),1>'::circle)
(2 rows)
```

Теперь планировщик получает нужные строки, обращаясь к индексу `airports_gist_idx`.

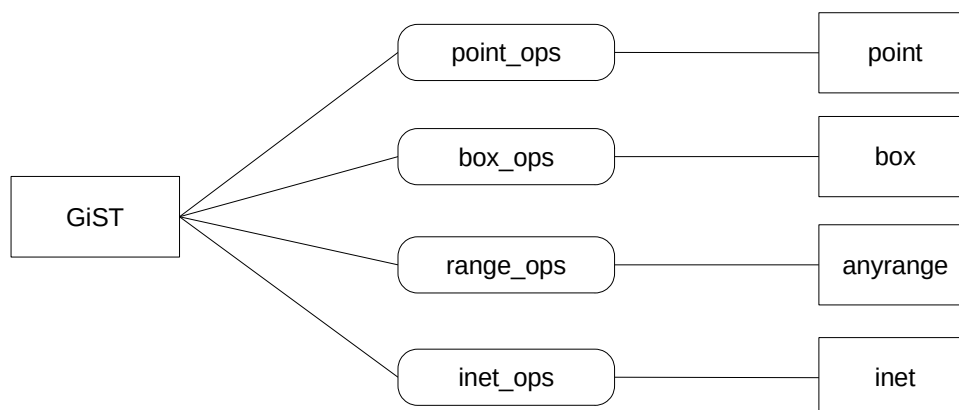
Удалим созданный индекс:

```
=> DROP INDEX airports_gist_idx;
```

DROP INDEX

Класс операторов

Посредник между индексным методом и типом данных
Может включать значительную часть логики индексирования



11

Чтобы индексные методы доступа могли работать с разными типами данных (которые в PostgreSQL могут подключаться на лету), между индексами и типами данных есть посредник — класс операторов, в который входят необходимые операторы и вспомогательные функции.

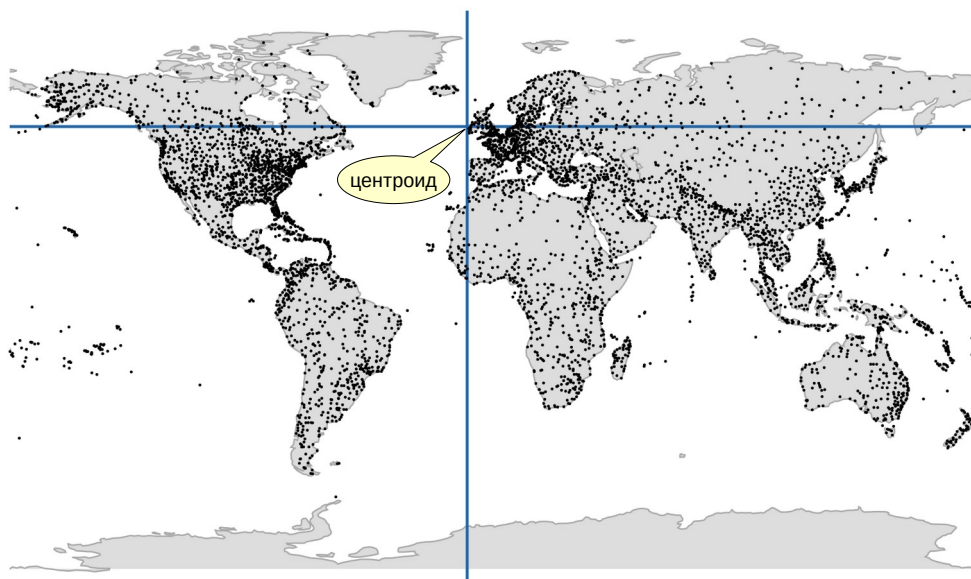
Такие индексы, как B-деревья и хеш-индексы, тоже используют классы операторов, но они довольно простые: содержат такие операторы, как «равно», «больше» и «меньше».

Классы операторов для GiST-индексов включают в себя значительную часть логики индексирования и определяют правила, по которым значения добавляются в индекс и затем ищутся в нем. Поэтому для разных типов данных GiST может ускорять разные операции.

Например, с помощью GiST можно индексировать значения диапазонных типов (таких, как `int4range` или `tstzrange`). Такой индекс позволяет находить диапазоны, включенные в заданный диапазон, пересекающиеся с заданным диапазоном, примыкающие к заданному диапазону и т. п.

GiST можно считать каркасом, на основе которого строятся произвольные схемы индексации (не только R-дерево), реализуя нужным образом класс операторов. Это гораздо проще, чем создать с нуля новый тип индекса, что весьма трудоемко и требует очень высокой квалификации разработчика.

<https://postgrespro.ru/docs/postgresql/16/indexes-opclass>

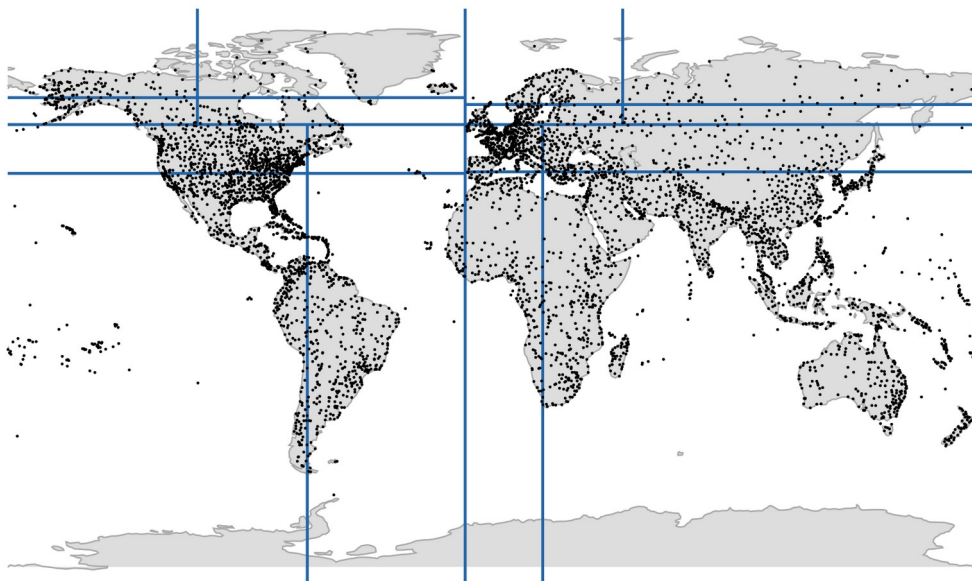


SP-GiST расшифровывается как space partitioning GiST. Это тоже обобщенное дерево поиска, но оно строится путем разбиения пространства поиска на непересекающиеся области.

Рассмотрим пример индекса SP-GiST для точек на плоскости.

Один из вариантов — дерево квадрантов. Корневой узел разбивает плоскость на четыре части (квадранта) по отношению к выбранной точке-центроиду.

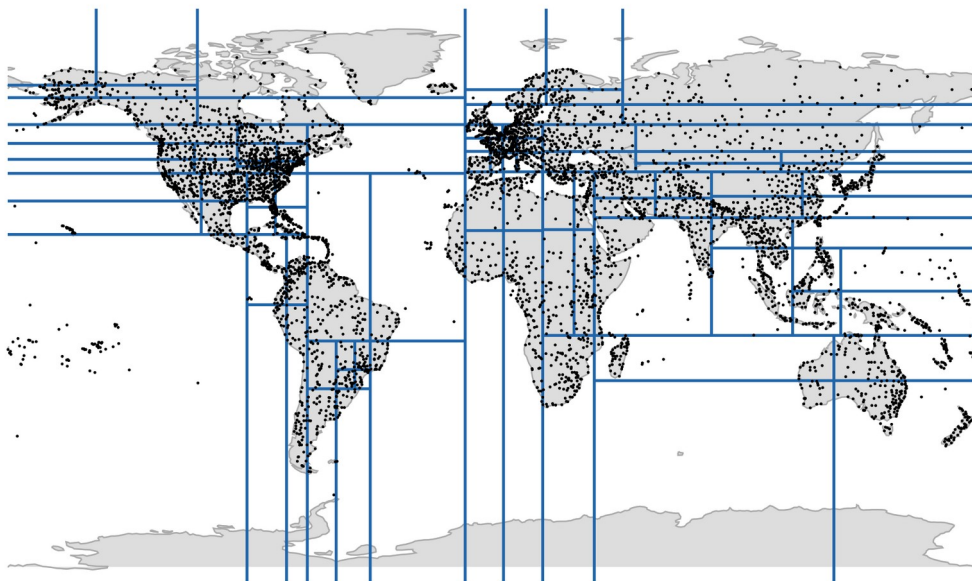
Пример индекса SP-GiST



13

Далее каждый из четырех квадрантов делится на собственные квадранты.

Пример индекса SP-GiST



14

Разбиение продолжится, пока все точки в квадранте не станут помещаться на одну индексную страницу.

Несбалансированное дерево поиска

слабо ветвящееся дерево с большой глубиной
рассчитано на произвольные типы данных

Операции аналогичны GiST

вхождение в область
нахождение слева, справа, сверху, снизу от области
поиск ближайших соседей

SP-GiST, как и GiST, является каркасом, на основе которого, реализуя классы операторов, можно строить произвольные схемы индексации. Например, дерево квадрантов для точек реализуется классом операторов `point_ops`. Другой способ деления плоскости — на две, а не на четыре части. Такой способ называется k-мерным деревом и реализуется другим классом операторов — `kd_point_ops`.

Разбиение плоскости на непересекающиеся области порождает несбалансированные деревья, которые обычно слабо ветвятся и имеют большую глубину.

Как правило, индексы SP-GiST предоставляют поддержку тех же типов данных и операторов, что и GiST. Но из-за другой структуры индекса они могут оказаться как более, так и менее эффективными, чем GiST.

<https://postgrespro.ru/docs/postgresql/16/spgist>

Индекс SP-GiST

Создадим индекс SP-GiST по полю `coordinates` в таблице `airports_data`. Для точек есть два класса операторов. По умолчанию используется `point_ops` (дерево квадрантов), а мы для примера укажем `kd_point_ops` (k-мерное дерево):

```
=> CREATE INDEX airports_spgist_idx ON airports_data
    USING spgist(coordinates kd_point_ops);
```

```
CREATE INDEX
```

Попробуем найти все аэропорты, расположенные выше (севернее) Надыма:

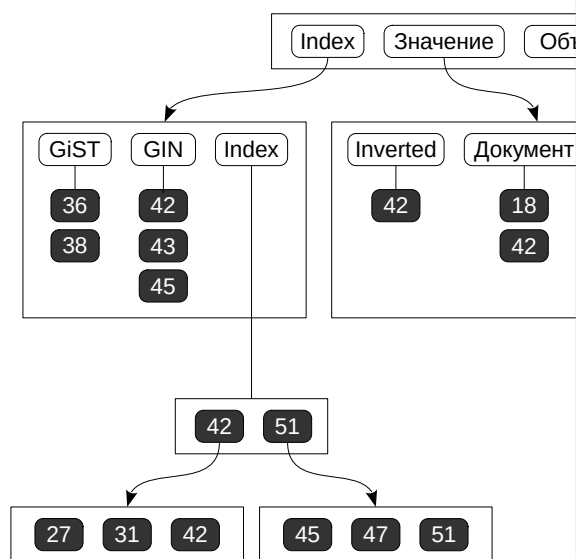
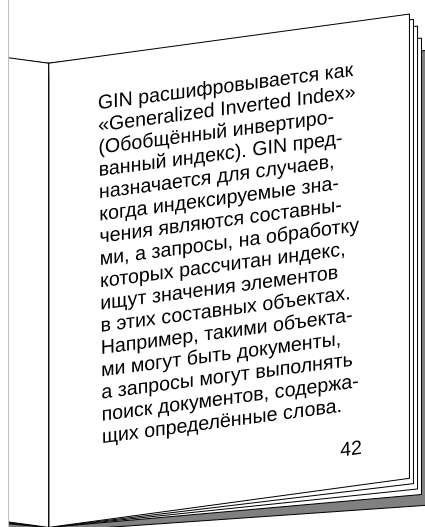
```
=> EXPLAIN (costs off)
SELECT airport_code
FROM airports_data
WHERE coordinates >^ '(72.69889831542969,65.48090362548828)::point;
```

QUERY PLAN

```
-----
Bitmap Heap Scan on airports_data
  Recheck Cond: (coordinates >^ '(72.69889831542969,65.48090362548828)::point)
    -> Bitmap Index Scan on airports_spgist_idx
          Index Cond: (coordinates >^ '(72.69889831542969,65.48090362548828)::point)
(4 rows)
```

Теперь таблица сканируется по битовой карте, построенной на основе SP-GiST-индекса `airports_spgist_idx`.

Идея GIN-индекса



17

GIN — generalized inverted index, обобщенный инвертированный индекс.

Идею этого индексного метода проще всего понять на примере предметного указателя в обычной книге. На страницах книги встречаются термины, а в предметном указателе перечислены в алфавитном порядке все термины с указанием номеров страниц, на которых они присутствуют.

Индекс GIN в первую очередь используется для индексации документов с целью ускорения полнотекстового поиска. По сути, это обычное B-дерево, но в него помещены не сами документы, а составляющие их слова. GIN-индекс оптимизирован с учетом того, что каждое слово может встречаться во многих документах. Если «список страниц» очень велик, он может храниться не в самой индексной странице, а в отдельном B-дереве.

<https://postgrespro.ru/docs/postgresql/16/gin>

Инвертированный список

для типов данных, значения которых (документы) состоят из элементов
индексируются элементы, а не документы

Типичная поддерживаемая операция

проверка соответствия документа поисковому запросу
проверка вхождения элемента в массив
поиск документов JSON по ключам или значениям

GIN работает с типами данных, значения которых состоят из элементов, а не являются атомарными. При этом индексируются не сами значения, а их элементы.

Как и GiST и SP-GiST, метод GIN является каркасом, который можно настроить не только на работу с текстом (состоящим из слов), но и с другими типами данных: с массивами (состоящими из элементов), с документами JSON (состоящими из ключей и значений). Для этого создается класс операторов, реализующий разбиение документа на элементы и проверку соответствия документа поисковому запросу.

Индекс GIN

В столбце `days_of_week` представления `routes` хранится массив номеров дней недели, по которым выполняется рейс:

```
=> SELECT flight_no, days_of_week FROM routes LIMIT 5;
```

```
flight_no | days_of_week
-----+-----
PG0001    | {6}
PG0002    | {7}
PG0003    | {2,6}
PG0004    | {3,7}
PG0005    | {2,5,7}
(5 rows)
```

Для представления нельзя построить индекс, поэтому сохраним его строки в отдельной таблице:

```
=> CREATE TABLE routes_tbl
AS SELECT * FROM routes;
```

```
SELECT 710
```

Теперь создадим GIN-индекс:

```
=> CREATE INDEX routestbl_gin_idx ON routes_tbl USING gin(days_of_week);
```

```
CREATE INDEX
```

С помощью GIN-индекса можно, например, отобрать рейсы, отправляющиеся только по средам и субботам:

```
=> EXPLAIN (costs off)
SELECT flight_no, departure_airport_name AS departure,
       arrival_airport_name AS arrival, days_of_week
FROM routes_tbl
WHERE days_of_week = ARRAY[3,6];
```

```
QUERY PLAN
-----
Bitmap Heap Scan on routes_tbl
  Recheck Cond: (days_of_week = '{3,6}'::integer[])
    -> Bitmap Index Scan on routestbl_gin_idx
          Index Cond: (days_of_week = '{3,6}'::integer[])
(4 rows)
```

Созданный GIN-индекс содержит всего семь элементов: целые числа от 1 до 7, представляющие дни недели. Для каждого из них в индексе хранятся ссылки на рейсы, выполняющиеся в этот день.

Пример индекса BRIN

```
SELECT * FROM t WHERE temperature BETWEEN 20 AND 30;
```

группа
последовательно
расположенных
страниц

зона	страницы	сводная информация
1	1 .. 128	{ -15 .. -3 }
2	129 .. 256	{ -7 .. -10 }
3	257 .. 384	{ 6 .. 19 }
4	385 .. 512	{ 15 .. 25 }
5	513 .. 640	{ 21 .. 32 }
6	641 .. 768	{ 13 .. 19 }
7	769 .. 896	{ 3 .. 15 }
8	897 .. 1024	{ -5 .. 6 }
9	1025 .. 1152	{ -18 .. -4 }

20

BRIN — block range index, «индекс зон блоков». Таблица разбивается на зоны определенной (настраиваемой) длины, каждая из которых охватывает набор последовательно расположенных страниц. Например, на слайде показан индекс с зоной размером 128 страниц. По каждой зоне собирается сводная информация, например, минимум и максимум значений индексируемого столбца.

При выполнении запроса можно пропустить все зоны, значения в которых гарантированно не попадают под условие. В примере на слайде только две зоны могут содержать значения температуры, удовлетворяющие условию.

Индекс BRIN не хранит идентификаторы версий строк, как все другие индексы, поэтому в выбранных зонах необходимо просмотреть все версии строк. В некотором смысле BRIN можно рассматривать как ускоритель последовательного сканирования.

<https://postgrespro.ru/docs/postgresql/16/brin>

Список зон со сводной информацией

зона охватывает группу последовательно расположенных страниц

сводная информация: минимум, максимум и т. п.

требуется корреляция с физическим расположением строк

Не хранит ссылки на версии строк

только сканирование по битовой карте

Предназначен для очень больших таблиц

небольшой размер

настраиваемое соотношение размера и точности

Используя классы операторов, можно выбирать сводную информацию, хранимую в индексе для каждой зоны. Это может быть как просто минимум и максимум, так и несколько диапазонов значений, а для геометрических типов данных можно хранить охватывающий прямоугольник (как в GiST).

В любом случае для эффективной работы BRIN необходима корреляция между значениями столбца и физическим расположением строк, чтобы в одну зону попадали значения со сходной сводной информацией. Обновления данных нарушают корреляцию и могут сказаться на эффективности индекса.

Поскольку BRIN не хранит ссылки на версии строк, он возвращает перечень страниц зоны в виде неточной битовой карты. Обычное индексное сканирование (и сканирование только индекса) невозможны.

Зато BRIN-индекс имеет очень небольшой размер, который, к тому же, может настраиваться за счет указания размера зоны. Чем больше зона, тем меньше индекс, но меньше и точность. Благодаря этому BRIN идеально подходит для очень больших таблиц, характерных для хранилищ данных.

Индекс BRIN

Для примера построим индекс BRIN по самой большой таблице:

```
=> CREATE INDEX tflights_brin_idx ON ticket_flights USING brin(flight_id);
```

CREATE INDEX

Индексы BRIN дают ощутимый эффект только для очень больших таблиц. Тем не менее, планировщик использует построенный индекс, поскольку по сводной информации (минимум и максимум) можно отсеять зоны, не содержащие требуемых значений:

```
=> EXPLAIN (analyze, costs off, timing off)
```

```
SELECT *
```

```
FROM ticket_flights
```

```
WHERE flight_id BETWEEN 3000 AND 4000;
```

QUERY PLAN

```
-----
Gather (actual rows=46357 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Bitmap Heap Scan on ticket_flights (actual rows=15452 loops=3)
        Recheck Cond: ((flight_id >= 3000) AND (flight_id <= 4000))
        Rows Removed by Index Recheck: 2377352
        Heap Blocks: lossy=19927
        -> Bitmap Index Scan on tflights_brin_idx (actual rows=598480 loops=1)
              Index Cond: ((flight_id >= 3000) AND (flight_id <= 4000))
Planning Time: 0.907 ms
Execution Time: 12242.497 ms
(11 rows)
```

Поскольку BRIN не хранит ссылки на версии строк, единственный возможный способ доступа — сканирование по неточной (lossy) битовой карте.

Помимо B-дерева, существуют и другие, более специфичные типы индексов

Хеш-индекс для поиска по равенству

GiST и SP-GiST для несортируемых типов данных

GIN для документов

BRIN для очень больших таблиц

1. Сравните размеры и время построения хеш-индекса по полям разного размера (`book_ref` и `contact_data`) таблицы билетов `tickets`.
Повторите то же для индекса на основе В-дерева.
2. Воспользуйтесь GIN-индексом и расширением `pg_trgm` для поиска пассажиров, номер телефона которых содержит последовательность цифр 1234.
Можно ли ускорить такой запрос с помощью В-дерева?

1. Для оценки размера индекса можно использовать функцию `pg_total_relation_size('имя-индекса')`.

2. Создайте GIN-индекс по выражению `contact_data->>'phone'`, используя класс операторов `gin_trgm_ops` из расширения `pg_trgm`.

Страница документации: <https://postgrespro.ru/docs/postgresql/16/pgtrgm>

1. Сравнение размера и времени создания индексов

Посмотрим структуру таблицы tickets:

```
=> \d tickets
```

Table "bookings.tickets"				
Column	Type	Collation	Nullable	Default
ticket_no	character(13)		not null	
book_ref	character(6)		not null	
passenger_id	character varying(20)		not null	
passenger_name	text		not null	
contact_data	jsonb			

Indexes:

"tickets_pkey" PRIMARY KEY, btree (ticket_no)

Foreign-key constraints:

"tickets_book_ref_fkey" FOREIGN KEY (book_ref) REFERENCES bookings(book_ref)

Referenced by:

TABLE "ticket_flights" CONSTRAINT "ticket_flights_ticket_no_fkey" FOREIGN KEY (ticket_no) REFERENCES tickets(ticket_no)

Поле book_ref имеет фиксированный размер, а поле contact_data имеет тип jsonb.

Включим подсчет времени выполнения запросов:

```
=> \timing on
```

Timing is on.

Создадим hash-индексы...

```
=> CREATE INDEX tickets_hash_br ON tickets USING hash(book_ref);
```

CREATE INDEX

Time: 10132,746 ms (00:10,133)

```
=> CREATE INDEX tickets_hash_cd ON tickets USING hash(contact_data);
```

CREATE INDEX

Time: 12881,499 ms (00:12,881)

...и индексы B-tree:

```
=> CREATE INDEX tickets_btree_br ON tickets(book_ref);
```

CREATE INDEX

Time: 14531,115 ms (00:14,531)

```
=> CREATE INDEX tickets_btree_cd ON tickets(contact_data);
```

CREATE INDEX

Time: 45936,032 ms (00:45,936)

Время создания хеш-индексов примерно одинаково, а время создания btree-индексов зависит от размера индексируемого поля.

```
=> \timing off
```

Timing is off.

Теперь проверим размеры полученных индексов:

```
=> SELECT pg_size_pretty(pg_total_relation_size('tickets_hash_br')) "hash book_ref",
       pg_size_pretty(pg_total_relation_size('tickets_hash_cd')) "hash contact_data",
       pg_size_pretty(pg_total_relation_size('tickets_btree_br')) "btree book_ref",
       pg_size_pretty(pg_total_relation_size('tickets_btree_cd')) "btree contact_data" \gx
```

```
-[ RECORD 1 ]-----+-----
hash book_ref      | 81 MB
hash contact_data  | 80 MB
btree book_ref     | 59 MB
btree contact_data | 226 MB
```

Размер hash-индексов тоже примерно одинаковый. А размер индексов B-tree (как и время их построения) зависит от размера индексируемого поля, поскольку такие индексы хранят индексируемые значения.

2. Расширение pg_trgm

Выполним запрос:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM tickets
WHERE contact_data->>'phone' LIKE '%1234%';
```

QUERY PLAN

```
-----
Gather (actual rows=1801 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  Buffers: shared hit=128 read=49287
  -> Parallel Seq Scan on tickets (actual rows=600 loops=3)
        Filter: ((contact_data ->> 'phone'::text) ~ '%1234%'::text)
        Rows Removed by Filter: 982685
        Buffers: shared hit=128 read=49287
Planning:
  Buffers: shared hit=23 read=10 dirtied=2
Planning Time: 11.971 ms
Execution Time: 1461.388 ms
(12 rows)
```

Обратите внимание на количество прочитанных страниц (Buffers) — их почти пятьдесят тысяч.

Добавим расширение pg_trgm:

```
=> CREATE EXTENSION pg_trgm;
```

```
CREATE EXTENSION
```

Создадим GIN-индекс с классом операторов gin_trgm_ops:

```
=> CREATE INDEX tickets_gin
ON tickets USING GIN ((contact_data->>'phone') gin_trgm_ops);
```

```
CREATE INDEX
```

Такой класс операторов ускоряет поиск по шаблону, который начинается на знак процента, и даже по регулярным выражениям. Индекс на основе B-дерева этого не позволяет.

Повторим запрос:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM tickets
WHERE contact_data->>'phone' LIKE '%1234%';
```

QUERY PLAN

```
-----
Bitmap Heap Scan on tickets (actual rows=1801 loops=1)
  Recheck Cond: ((contact_data ->> 'phone'::text) ~ '%1234%'::text)
  Rows Removed by Index Recheck: 46
  Heap Blocks: exact=1820
  Buffers: shared hit=23 read=1818
  -> Bitmap Index Scan on tickets_gin (actual rows=1847 loops=1)
        Index Cond: ((contact_data ->> 'phone'::text) ~ '%1234%'::text)
        Buffers: shared hit=21
Planning:
  Buffers: shared hit=26 read=1 dirtied=1
Planning Time: 26.563 ms
Execution Time: 435.976 ms
(12 rows)
```

Количество прочитанных страниц сократилось более чем на порядок, время выполнения существенно уменьшилось.