Авторы: Егор Рогов, Павел Лузанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Методы доступа (типы индексов)

Классы и семейства операторов

Метод доступа btree и создание класса операторов для него

Идея метода доступа gist и примеры его использования

Методы доступа

Определяют способ получения данных

Табличные методы доступа — организация данных и чтение всей таблицы

Индексные методы доступа (типы индексов) — быстрый доступ к небольшой части данных

- алгоритмы, связанные с поисковой структурой данных
- эффективная одновременная работа
- страничная организация данных
- журналирование операций

Методы доступа определяют способ получения данных. Их можно разделить на табличные и индексные методы.

Табличный метод определяет организацию данных в таблице и работает, когда надо прочитать все данные (Seq Scan). Долгое время в PostgreSQL был единственный жестко определенный метод, но в версии 12 появилась возможность создавать новые табличные методы. Сейчас это скорее экспериментальная возможность, поэтому других готовых методов (таких, например, как колоночное хранилище) пока нет, но они должны появиться в будущем.

https://postgrespro.ru/docs/postgresql/12/tableam

Индексный метод доступа работает, когда надо получить часть данных (обычно небольшую) с помощью индекса.

Индексный метод можно рассматривать как каркас, который реализует основные алгоритмы для работы с индексной структурой данных, а также берет на себя заботу о низкоуровневых деталях, таких как:

- эффективная конкурентная работа с индексной структурой (в том числе стратегия блокирования);

- страничная организация данных индекса;

- журналирование операций над индексом в WAL.

Возможность создания новых индексных методов без изменения ядра системы появилась в версии 9.6.

https://postgrespro.ru/docs/postgresql/12/indexam

Методы доступа

```
=> CREATE DATABASE ext_opclasses;
```

```
CREATE DATABASE
```

```
=> \c ext_opclasses
```

```
You are now connected to database "ext_opclasses" as user "student".
```

В версии 12 имеется единственный встроенный табличный метод доступа:

```
=> SELECT amname FROM pg_am WHERE amtype = 't';
```

```
 amname
--------
 heap
(1 row)
```

Зато много различных индексных методов доступа:

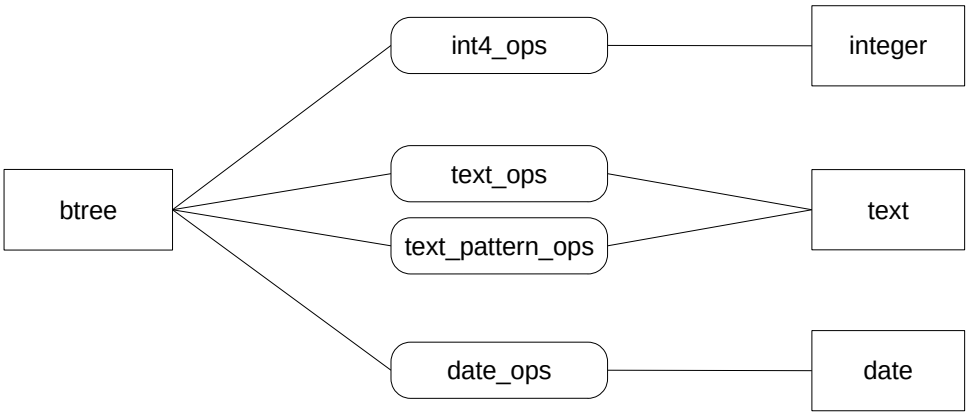
```
=> SELECT amname FROM pg_am WHERE amtype = 'i';
```

```
 amname
--------
 btree
 hash
 gist
 gin
 spgist
 brin
(6 rows)
```

С btree знакомы все — это «обычный» метод доступа на основе B-дерева, который используется по умолчанию и покрывает большинство потребностей. Остальные методы доступа тоже очень полезны, но в специальных ситуациях. Некоторые из них мы рассмотрим позже.

Классы операторов

Набор операторов и вспомогательных функций,
который используется *методом доступа*
для работы с конкретным *типом данных*



Дальше мы будем говорить не о табличном, а об индексном доступе.

Итак, с одной стороны есть индексный метод доступа (например, btree), а с другой — конкретные типы данных (такие, как integer, text и т. п.).

Чтобы связать метод доступа с типами данными, используются **классы операторов**. Класс операторов состоит из набора операторов (и, при необходимости, вспомогательных функций), который реализует API индексного метода доступа.

Важно, что часть логики индексирования находится в методе доступа, а часть — в классе операторов. Поэтому недостаточно понять только устройство метода доступа; надо учитывать и то, какой используется класс операторов.

Для B-деревьев это не так очевидно, поскольку на долю класса операторов приходится совсем немного. Но в других методах доступа разные классы операторов могут радикально менять логику.

Заметим также, что для одного и того же метода доступа и одного и того же типа данных может быть определено несколько классов операторов. В этом случае пользователь может выбирать то поведение, которое требуется.

https://postgrespro.ru/docs/postgresql/12/indexes-opclass

Классы операторов

Посмотрим, какие классы операторов определены для B-дерева и для разных типов данных. Для целых чисел:

```
=> SELECT opcname
FROM pg_opclass
WHERE opcmethod = (SELECT oid FROM pg_am WHERE amname = 'btree')
AND opcintype IN (
    'smallint'::regtype, 'integer'::regtype, 'bigint'::regtype
);
```

```
 opcname
----------
 int2_ops
 int4_ops
 int8_ops
(3 rows)
```

Для удобства такие «похожие по смыслу» классы операторов объединяются в семейства операторов:

```
=> SELECT opcname, opf.opfname
FROM pg_opclass opc, pg_opfamily opf
WHERE opc.opcmethod = (SELECT oid FROM pg_am WHERE amname = 'btree')
AND opc.opcintype IN (
    'smallint'::regtype, 'integer'::regtype, 'bigint'::regtype
)
AND opf.oid = opc.opcfamily;
```

```
 opcname  |   opfname
----------+-------------
 int8_ops | integer_ops
 int4_ops | integer_ops
 int2_ops | integer_ops
(3 rows)
```

Семейства операторов позволяют планировщику работать с выражениями разных (но «похожих») типов, даже если они не приведены к одному общему.

Вот классы операторов для типа text (если для одного типа есть несколько классов операторов, то один будет помечен для использования по умолчанию):

```
=> SELECT opcname, opcdefault
FROM pg_opclass
WHERE opcmethod = (SELECT oid FROM pg_am WHERE amname = 'btree')
AND opcintype = 'text'::regtype;
```

```
       opcname       | opcdefault
---------------------+------------
 text_ops            | t
 varchar_ops         | f
 text_pattern_ops    | f
 varchar_pattern_ops | f
(4 rows)
```

- Классы операторов pattern_ops отличаются от обычных тем, что сравнивают строки посимвольно, игнорируя правила сортировки (collation).

А так можно посмотреть, какие операторы включены в конкретный класс операторов:

```
=> SELECT amop.amopopr::regoperator
FROM pg_opclass opc
    JOIN pg_opfamily opf ON opf.oid = opc.opcfamily
    JOIN pg_amop amop ON amop.amopfamily = opc.opcfamily
                     AND amop.amoplefttype = opc.opcintype
WHERE opcmethod = (SELECT oid FROM pg_am WHERE amname = 'btree')
AND opc.opcname = 'bool_ops';
```

```
       amopopr
--------------------
 <(boolean,boolean)
 <=(boolean,boolean)
 =(boolean,boolean)
 >=(boolean,boolean)
 >(boolean,boolean)
(5 rows)
```

Метод доступа — это тип индекса, а собственно индекс — это конкретная структура, созданная на основе метода доступа, в которой для каждого столбца используется свой класс операторов.

Пусть имеется какая-нибудь таблица:

```
=> CREATE TABLE t(
    id integer GENERATED ALWAYS AS IDENTITY,
    s text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO t(s) VALUES ('foo'), ('bar'), ('xy'), ('z');
```

```
INSERT 0 4
```

Привычная команда создания индекса выглядит так:

```
CREATE INDEX ON t(id, s);
```

Но это просто сокращение для:

```
=> CREATE INDEX ON t
USING btree -- метод доступа
(
    id int4_ops, -- класс операторов для integer
    s  text_ops  -- класс операторов по умолчанию для text
);
```

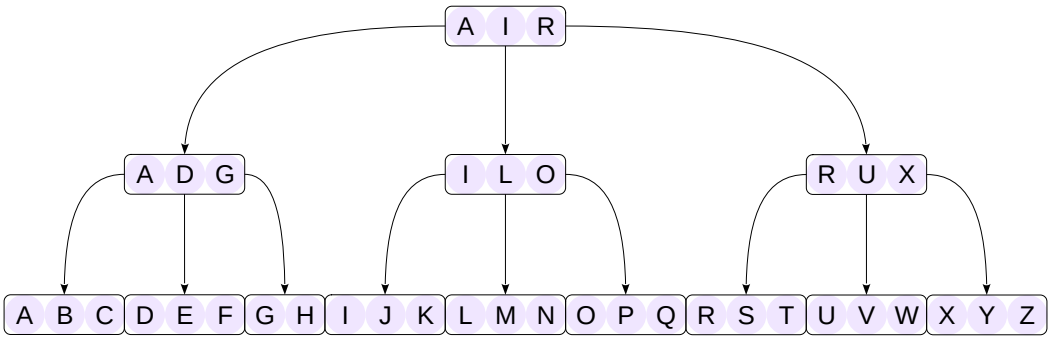
```
CREATE INDEX
```

Метод доступа btree

Сбалансированное дерево, значения упорядочены

метод применим только для сортируемых типов данных

API метода доступа определяет порядок сортировки



Рассмотрим теперь несколько конкретных методов доступа, и начнем с B-деревьев.

Эта структура данных представляет собой дерево (сбалансированное по высоте), которое содержит *упорядоченные* данные. В каждом узле хранится информация о том, какие диапазоны значений расположены в дочерних узлах.

Это позволяет находить искомое значение, спускаясь от вершины дерева к листьям, каждый раз однозначно выбирая подходящий диапазон. Например, для поиска буквы «H» (на рисунке) мы начинаем с корня. Поскольку «A» <= «H» < «I», мы спускаемся в первый дочерний узел, и так далее.

Итак, этот метод доступа применим только для сортируемых данных (к которым относятся большинство «обычных» типов данных). От класса операторов требуется только определить, как именно упорядочиваются значения конкретного типа.

https://postgrespro.ru/docs/postgresql/12/btree

https://postgrespro.ru/docs/postgresql/12/xindex

Класс операторов для B-дерева

Класс операторов для B-дерева определяет, как именно будут сортироваться значения в индексе. Для этого он включает пять операторов сравнения, как мы уже видели на примере bool_ops.

Определим перечислимый тип для единиц измерения информации:

```
=> CREATE TYPE capacity_units AS ENUM (
    'B', 'kB', 'MB', 'GB', 'TB', 'PB'
);
```

```
CREATE TYPE
```

И объявим составной тип данных для представления объема информации:

```
=> CREATE TYPE capacity AS (
    amount integer,
    unit capacity_units
);
```

```
CREATE TYPE
```

Используем новый тип в таблице, которую заполним случайными значениями.

```
=> CREATE TABLE test (
    cap capacity
);
```

```
CREATE TABLE
```

```
=> INSERT INTO test
    SELECT ( (random()*1023)::integer, u.unit )::capacity
    FROM generate_series(1,100),
         unnest(enum_range(NULL::capacity_units)) AS u(unit);
```

```
INSERT 0 600
```

По умолчанию значения составного типа сортируются в лексикографическом порядке, но этот порядок не совпадает с естественным порядком:

```
=> SELECT * FROM test ORDER BY cap LIMIT 10;
```

```
   cap
---------
 (2,PB)
 (3,MB)
 (4,B)
 (7,PB)
 (9,B)
 (10,B)
 (15,B)
 (15,TB)
 (16,kB)
 (19,B)
(10 rows)
```

Чтобы исправить сортировку, создадим класс операторов. Начнем с функции, которая пересчитает объем в байты.

```
=> CREATE FUNCTION capacity_to_bytes(a capacity) RETURNS numeric
AS $$
SELECT a.amount::numeric *
    1024::numeric ^ ( array_position(enum_range(a.unit), a.unit)-1 );
$$ LANGUAGE sql STRICT IMMUTABLE;
```

```
CREATE FUNCTION
```

Помимо операторов сравнения нам понадобится еще одна вспомогательная функция — тоже для сравнения. Она должна возвращать:

- -1, если первый аргумент меньше второго,
- 0, если аргументы равны,
- 1, если первый аргумент больше второго.

```
=> CREATE FUNCTION capacity_cmp(a capacity, b capacity) RETURNS integer
AS $$
SELECT CASE
    WHEN capacity_to_bytes(a) < capacity_to_bytes(b) THEN -1
    WHEN capacity_to_bytes(a) > capacity_to_bytes(b) THEN 1
    ELSE 0
END;
$$ LANGUAGE sql STRICT IMMUTABLE;

CREATE FUNCTION
```

С помощью этой функции мы определим пять операторов сравнения (и функции для них). Начнем с «меньше»:

```
=> CREATE FUNCTION capacity_lt(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
    RETURN capacity_cmp(a,b) < 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

CREATE FUNCTION

=> CREATE OPERATOR <(
    LEFTARG = capacity,
    RIGHTARG = capacity,
    FUNCTION = capacity_lt
);

CREATE OPERATOR
```

И аналогично остальные четыре.

```
=> CREATE FUNCTION capacity_le(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
    RETURN capacity_cmp(a,b) <= 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

CREATE FUNCTION

=> CREATE OPERATOR <=(
    LEFTARG = capacity,
    RIGHTARG = capacity,
    FUNCTION = capacity_le
);

CREATE OPERATOR

=> CREATE FUNCTION capacity_eq(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
    RETURN capacity_cmp(a,b) = 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

CREATE FUNCTION

=> CREATE OPERATOR =(
    LEFTARG = capacity,
    RIGHTARG = capacity,
    FUNCTION = capacity_eq
);

CREATE OPERATOR

=> CREATE FUNCTION capacity_ge(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
    RETURN capacity_cmp(a,b) >= 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

CREATE FUNCTION

=> CREATE OPERATOR >=(
    LEFTARG = capacity,
    RIGHTARG = capacity,
    FUNCTION = capacity_ge
);

CREATE OPERATOR
```

```
=> CREATE FUNCTION capacity_gt(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
    RETURN capacity_cmp(a,b) > 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

CREATE FUNCTION

=> CREATE OPERATOR >(
    LEFTARG = capacity,
    RIGHTARG = capacity,
    FUNCTION = capacity_gt
);

CREATE OPERATOR
```

Готово. Мы уже можем правильно сравнивать объемы:

```
=> SELECT (1,'MB')::capacity > (512, 'kB')::capacity;

 ?column?
----------
 t
(1 row)
```

Чтобы значения были правильно упорядочены при выборке, нам осталось создать класс операторов. За каждым оператором закреплен собственный номер (в случае btree: 1 — «меньше» и т. д.), поэтому имена операторов могут быть любыми.

```
=> CREATE OPERATOR CLASS capacity_ops
DEFAULT FOR TYPE capacity
USING btree AS
    OPERATOR 1 <,
    OPERATOR 2 <=,
    OPERATOR 3 =,
    OPERATOR 4 >=,
    OPERATOR 5 >,
    FUNCTION 1 capacity_cmp(capacity,capacity);

CREATE OPERATOR CLASS
```

```
=> SELECT * FROM test ORDER BY cap LIMIT 10;

  cap
--------
 (4,B)
 (9,B)
 (10,B)
 (15,B)
 (19,B)
 (22,B)
 (25,B)
 (27,B)
 (33,B)
 (38,B)
(10 rows)
```

Теперь значения отсортированы правильно.

Наш класс операторов будет использоваться по умолчанию при создании индекса:

```
=> CREATE INDEX ON test(cap);

CREATE INDEX
```

Любой индекс в PostgreSQL может использоваться только для выражений вида:

<индексированное-поле> <оператор> <выражение>

Причем оператор должен входить в соответствующий класс операторов.

Будет ли использоваться созданный индекс в таком запросе?

```
=> EXPLAIN (costs off)
SELECT * FROM test WHERE cap < (100,'B')::capacity;
```

```
                QUERY PLAN
-------------------------------------------
 Index Only Scan using test_cap_idx on test
   Index Cond: (cap < '(100,B)'::capacity)
(2 rows)
```

Да, поскольку:

- поле test.cap проиндексировано с помощью метода доступа btree и класса операторов capacity_ops;
- оператор < входит в класс операторов capacity_ops.

Поэтому и при доступе с помощью индекса значения будут возвращаться в правильном порядке:

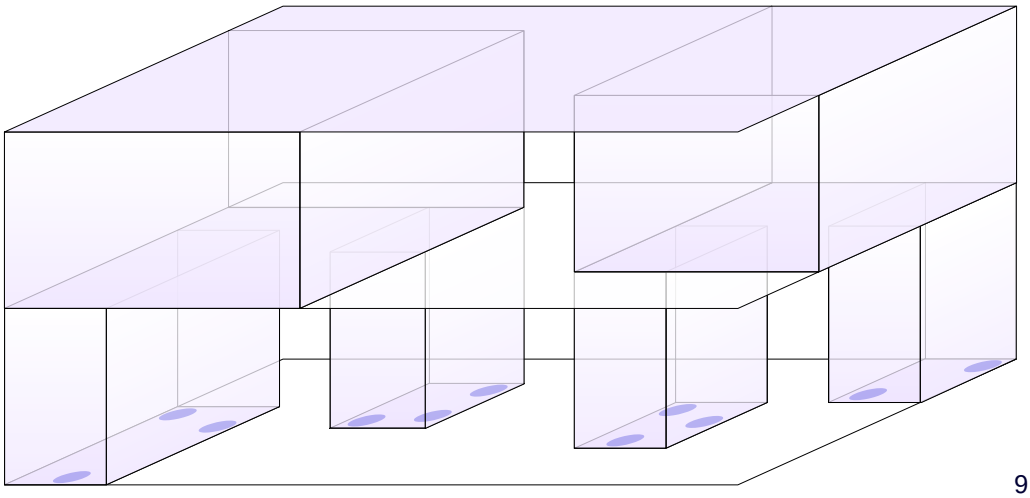
```
=> SELECT * FROM test WHERE cap < (100,'B')::capacity ORDER BY cap;

  cap
--------
 (4,B)
 (9,B)
 (10,B)
 (15,B)
 (19,B)
 (22,B)
 (25,B)
 (27,B)
 (33,B)
 (38,B)
 (39,B)
 (65,B)
 (69,B)
 (71,B)
 (86,B)
 (94,B)
(16 rows)
```

Метод доступа gist

Сбалансированное дерево, общий предикат

API метода доступа определяет, что представляет собой предикат



9

Однако не все типы данных сортируемы. Например, не имеют естественной упорядоченности такие типы, как геометрические фигуры (точки, прямоугольники и т. п.), диапазоны, массивы и т. п. Это не значит, что для таких типов данных нельзя определить операции сравнения, но практической пользы от них будет немного. Важнее уметь использовать индексы, например, для поиска точек, входящих в заданную область, или поиска элементов, входящих в массив, т. е. для операторов, отличных от «больше», «меньше» и т. п.

B-дерево здесь не годится, но идею индексирования с помощью сбалансированного дерева можно обобщить. Пусть для каждого поддерева определено условие (предикат), которому удовлетворяют все данные, попадающие в это поддерево. Тогда предикат можно использовать для определения, в какое поддерево (в общем случае — несколько поддеревьев) нужно спускаться при поиске.

Проще всего представить себе такой GiST-индекс на примере точек. В этом случае предикатом выступает охватывающий прямоугольник, а структура называется R-деревом. На рисунке она изображена схематично. Верхний уровень представлен большим прямоугольником, охватывающем все проиндексированные точки. На втором уровне видим два меньших прямоугольника, на нижнем — еще более мелкие. Суммарно все прямоугольники нижнего уровня охватывают все точки.

Такую же структуру можно построить и для диапазонных типов.

Класс операторов для GiST определяет, как выглядит предикат, и еще многие детали, необходимые для эффективной работы.

https://postgrespro.ru/docs/postgresql/12/gist

Метод доступа GiST

Какие именно операторы поддерживает GiST-индекс, существенно зависит от класса операторов. Информацию можно получить как из документации, так и из системного каталога. Возьмем, например, тип данных point (точки).

Доступный класс операторов:

```
=> SELECT opcname
FROM pg_opclass
WHERE opcmethod = (SELECT oid FROM pg_am WHERE amname = 'gist')
AND opcintype = 'point'::regtype;
```

```
  opcname
-----------
 point_ops
(1 row)
```

Операторы в этом классе:

```
=> SELECT amop.amopopr::regoperator
FROM pg_opclass opc
    JOIN pg_opfamily opf ON opf.oid = opc.opcfamily
    JOIN pg_amop amop ON amop.amopfamily = opc.opcfamily
                     AND amop.amoplefttype = opc.opcintype
WHERE opcmethod = (SELECT oid FROM pg_am WHERE amname = 'gist')
AND opc.opcname = 'point_ops';
```

```
      amopopr
-------------------
 <<(point,point)
 >>(point,point)
 ~=(point,point)
 <^(point,point)
 >^(point,point)
 <->(point,point)
 <@(point,box)
 <@(point,polygon)
 <@(point,circle)
(9 rows)
```

В частности, оператор <@ проверяет, принадлежит ли точка одной из геометрических фигур.

Создадим таблицу со случайными точками:

```
=> CREATE TABLE points (
    p point
);
```

```
CREATE TABLE
```

```
=> INSERT INTO points(p)
    SELECT point(1 - random()*2, 1 - random()*2)
    FROM generate_series(1,10000);
```

```
INSERT 0 10000
```

Сколько точек расположено в круге радиуса 0.1?

```
=> SELECT count(*) FROM points WHERE p <@ circle '((0,0),0.1)';
```

```
 count
-------
    88
(1 row)
```

Как выполняется такой запрос?

```
=> EXPLAIN (costs off)
SELECT * FROM points WHERE p <@ circle '((0,0),0.1)';
```

```
              QUERY PLAN
---------------------------------------
 Seq Scan on points
   Filter: (p <@ '<(0,0),0.1>'::circle)
(2 rows)
```

Полным перебором всей таблицы.

Создание GiST-индекса позволит ускорить эту операцию. Класс операторов можно не указывать, он один.

```
=> CREATE INDEX ON points USING gist(p);
```

```
CREATE INDEX
```

```
=> EXPLAIN (costs off)
SELECT * FROM points WHERE p <@ circle '((0,0),0.1)';
```

```
                     QUERY PLAN
-----------------------------------------------------
 Bitmap Heap Scan on points
   Recheck Cond: (p <@ '<(0,0),0.1>'::circle)
   ->  Bitmap Index Scan on points_p_idx
         Index Cond: (p <@ '<(0,0),0.1>'::circle)
(4 rows)
```

Еще один интересный оператор <-> вычисляет расстояние от одной точки до другой. Его можно использовать, чтобы найти точки, ближайшие к данной (так называемый поиск ближайших соседей, k-NN search):

```
=> SELECT * FROM points ORDER BY p <-> point '(0,0)' LIMIT 5;
```

```
                        p
-------------------------------------------------
 (0.005469499247332976,-0.004395186344140711)
 (0.008291975916012 7,0.004428304940887529)
 (-0.009259367506636806,-0.012038933464751267)
 (0.01646634749749154,0.013182585991273754)
 (-0.02045396126468546,-0.011083697261398129)
(5 rows)
```

Эта операция (весьма непростая, если реализовывать ее в приложении) также ускоряется индексом:

```
=> EXPLAIN (costs off)
SELECT * FROM points ORDER BY p <-> point '(0,0)' LIMIT 5;
```

```
                      QUERY PLAN
-------------------------------------------------------
 Limit
   ->  Index Only Scan using points_p_idx on points
         Order By: (p <-> '(0,0)'::point)
(3 rows)
```

GiST-индекс можно построить и для столбца диапазонного типа:

```
=> SELECT amop.amopopr::regoperator
FROM pg_opclass opc
    JOIN pg_opfamily opf ON opf.oid = opc.opcfamily
    JOIN pg_amop amop ON amop.amopfamily = opc.opcfamily
                     AND amop.amoplefttype = opc.opcintype
WHERE opcmethod = (SELECT oid FROM pg_am WHERE amname = 'gist')
AND opc.opcname = 'range_ops';
```

```
         amopopr
-------------------------
 @>(anyrange,anyelement)
 <<(anyrange,anyrange)
 &<(anyrange,anyrange)
 &&(anyrange,anyrange)
 &>(anyrange,anyrange)
 >>(anyrange,anyrange)
 -|-(anyrange,anyrange)
 @>(anyrange,anyrange)
 <@(anyrange,anyrange)
 =(anyrange,anyrange)
(10 rows)
```

Здесь мы видим другой набор операторов.

Одно из применений GiST-индекса — поддержка ограничений целостности типа EXCLUDE (ограничения исключения).

Возьмем классический пример бронирования аудиторий:

```
=> CREATE TABLE booking (
    during tstzrange NOT NULL
);
```

```
CREATE TABLE
```

Ограничение целостности можно сформулировать так: нельзя, чтобы одновременно существовали два пересекающихся (оператор &&) интервала. Такое ограничение можно задать декларативно на уровне базы данных:

```
=> ALTER TABLE booking ADD CONSTRAINT no_intersect
    EXCLUDE USING gist(during WITH &&);
```

```
ALTER TABLE
```

Проверим:

```
=> INSERT INTO booking(during)
    VALUES ('[today 12:00,today 14:00)'::tstzrange);
```

```
INSERT 0 1
```

```
=> INSERT INTO booking(during)
    VALUES ('[today 13:00,today 16:00)'::tstzrange);
```

```
ERROR:  conflicting key value violates exclusion constraint "no_intersect"
DETAIL:  Key (during)=(["2020-11-06 13:00:00+03","2020-11-06 16:00:00+03")) conflicts with existing key (during)=(["2020-11-06 12:00:00+03","2020-11-06 14:00:00+03")).
```

Частая ситуация — наличие дополнительного условия в таком ограничении целостности. Добавим номер переговорной:

```
=> ALTER TABLE booking ADD room integer NOT NULL DEFAULT 1;
```

```
ALTER TABLE
```

Но мы не сможем добавить этот столбец в ограничение целостности, поскольку класс операторов для метода gist и типа integer не определен.

```
=> ALTER TABLE booking DROP CONSTRAINT no_intersect;
```

```
ALTER TABLE
```

```
=> ALTER TABLE booking ADD CONSTRAINT no_intersect
    EXCLUDE USING gist(during WITH &&, room WITH =);
```

```
ERROR:  data type integer has no default operator class for access method "gist"
HINT:  You must specify an operator class for the index or define a default operator class for the data type.
```

В этом случае поможет расширение btree_gist, которое добавляет классы операторов для типов данных, которые обычно индексируются с помощью B-деревьев:

```
=> CREATE EXTENSION btree_gist;
```

```
CREATE EXTENSION
```

```
=> ALTER TABLE booking ADD CONSTRAINT no_intersect
    EXCLUDE USING gist(during WITH &&, room WITH =);
```

```
ALTER TABLE
```

Теперь разные переговорные можно бронировать на одно время:

```
=> INSERT INTO booking(room, during)
    VALUES (2, '[today 13:00,today 16:00)'::tstzrange);
```

```
INSERT 0 1
```

Но одну и ту же — нельзя:

```
=> INSERT INTO booking(room, during)
    VALUES (1, '[today 13:00,today 16:00)'::tstzrange);
```

```
ERROR:  conflicting key value violates exclusion constraint "no_intersect"
DETAIL:  Key (during, room)=(["2020-11-06 13:00:00+03","2020-11-06 16:00:00+03"), 1) conflicts with existing key (during, room)=(["2020-11-06 12:00:00+03","2020-11-06 14:00:00+03"), 1)
```

Итоги

PostgreSQL предоставляет разнообразную индексную поддержку любых типов данных, включая пользовательские

Методы доступа — каркас индексирования, реализующий основной алгоритм и низкоуровневые детали

btree, hash, gist, sp-gist, gin, brin...

Классы операторов связывают метод с типами данных

В этой теме рассматриваются лишь самые основные понятия, связанные с индексной поддержкой. Заинтересованным в подробностях рекомендуем внимательно ознакомиться с разделами документации, ссылки на которые приведены в комментариях к слайдам, а также с серией статей:

https://habr.com/ru/company/postgrespro/blog/326096/

Практика

1. Добавьте в таблицу retail_prices ограничение целостности, не позволяющее создать пересекающиеся интервалы действия цен для одной книги.
   Проверьте план выполнения запроса для поиска актуальной цены (get_retail_price). Использует ли он индекс, созданный для поддержки ограничения целостности?
2. Сортировка книг в приложении по формату издания работает неверно, поскольку значения типа для формата издания упорядочиваются лексикографически. Сделайте так, чтобы значения упорядочивались по площади книжной страницы и проверьте, что сортировка в приложении исправилась.

1. Пример похожего ограничения целостности приводился в демонстрации.

Если индекс не будет использоваться запросом из функции get_retail_price, дело может быть в том, что в таблице retail_prices слишком мало строк и полное сканирование оказывается выгоднее. Чтобы проверить применимость индекса, вы можете либо добавить больше строк в таблицу, либо временно запретить использование полного сканирования:

```
SET enable_seqscan = off;
```

2. Создайте класс операторов для метода доступа btree и типа данных book_format, как было показано в демонстрации.

```
student$ psql bookstore2
```

1. Ограничение целостности для retail_prices

Добавим ограничение целостности в таблицу:

```
=> CREATE EXTENSION btree_gist;

CREATE EXTENSION

=> ALTER TABLE retail_prices ADD
    EXCLUDE USING gist(book_id WITH =, effective WITH &&);

ALTER TABLE
```

Такое ограничение гарантирует, что данные не будет повреждены, даже если в коде функции set_retail_price будет допущена ошибка.

В функции get_retail_price используется запрос следующего вида:

```
=> EXPLAIN (costs off)
SELECT rp.price
FROM retail_prices rp
WHERE rp.book_id = 1
  AND rp.effective @> current_timestamp;

                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on retail_prices rp
   Filter: ((book_id = 1) AND (effective @> CURRENT_TIMESTAMP))
(2 rows)
```

Как видно из плана, таблица по-прежнему перебирается полностью. Но планировщик будет использовать созданный индекс, как только это окажется выгодным:

```
=> BEGIN;

BEGIN

=> WITH s AS (
    SELECT empapi.set_retail_price(
               b.book_id, 100.00, current_timestamp
           ),
           empapi.set_retail_price(
               b.book_id, 200.00, current_timestamp + interval '1 day'
           ),
           empapi.set_retail_price(
               b.book_id, 300.00, current_timestamp + interval '2 days'
           )
    FROM books b
)
SELECT count(*) FROM s;

 count
-------
    96
(1 row)


=> EXPLAIN (costs off)
SELECT rp.price
FROM retail_prices rp
WHERE rp.book_id = 1
  AND rp.effective @> current_timestamp;

                          QUERY PLAN
---------------------------------------------------------------
 Bitmap Heap Scan on retail_prices rp
   Recheck Cond: (effective @> CURRENT_TIMESTAMP)
   Filter: (book_id = 1)
   ->  Bitmap Index Scan on retail_prices_book_id_effective_excl
         Index Cond: (effective @> CURRENT_TIMESTAMP)
(5 rows)


=> ROLLBACK;

ROLLBACK
```

2. Упорядочивание для форматов издания

Создадим класс операторов, как было показано в демонстрации.

```
=> CREATE FUNCTION book_format_area(f book_format)
RETURNS numeric
AS $$
SELECT f.width::numeric * f.height::numeric / f.parts::numeric;
$$ LANGUAGE sql STRICT IMMUTABLE;
```

```
CREATE FUNCTION
```

```
=> CREATE FUNCTION book_format_cmp(a book_format, b book_format)
RETURNS integer
AS $$
SELECT CASE
    WHEN book_format_area(a) < book_format_area(b) THEN -1
    WHEN book_format_area(a) > book_format_area(b) THEN 1
    ELSE 0
END;
$$ LANGUAGE sql STRICT IMMUTABLE;
```

```
CREATE FUNCTION
```

```
=> CREATE FUNCTION book_format_lt(a book_format, b book_format)
RETURNS boolean AS $$
BEGIN
    RETURN book_format_cmp(a,b) < 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

```
CREATE FUNCTION
```

```
=> CREATE OPERATOR <(
    LEFTARG = book_format,
    RIGHTARG = book_format,
    FUNCTION = book_format_lt
);
```

```
CREATE OPERATOR
```

```
=> CREATE FUNCTION book_format_le(a book_format, b book_format)
RETURNS boolean AS $$
BEGIN
    RETURN book_format_cmp(a,b) <= 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

```
CREATE FUNCTION
```

```
=> CREATE OPERATOR <=(
    LEFTARG = book_format,
    RIGHTARG = book_format,
    FUNCTION = book_format_le
);
```

```
CREATE OPERATOR
```

```
=> CREATE FUNCTION book_format_eq(a book_format, b book_format)
RETURNS boolean AS $$
BEGIN
    RETURN book_format_cmp(a,b) = 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

```
CREATE FUNCTION
```

```
=> CREATE OPERATOR =(
    LEFTARG = book_format,
    RIGHTARG = book_format,
    FUNCTION = book_format_eq
);
```

```
CREATE OPERATOR
```

```
=> CREATE FUNCTION book_format_ge(a book_format, b book_format)
RETURNS boolean AS $$
BEGIN
    RETURN book_format_cmp(a,b) >= 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

```
CREATE FUNCTION
```

```
=> CREATE OPERATOR >=(
    LEFTARG = book_format,
    RIGHTARG = book_format,
    FUNCTION = book_format_ge
);
```

```
CREATE OPERATOR
```

```
=> CREATE FUNCTION book_format_gt(a book_format, b book_format)
RETURNS boolean AS $$
BEGIN
    RETURN book_format_cmp(a,b) > 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

```
CREATE FUNCTION
```

```
=> CREATE OPERATOR >(
    LEFTARG = book_format,
    RIGHTARG = book_format,
    FUNCTION = book_format_gt
);
```

```
CREATE OPERATOR
```

```
=> CREATE OPERATOR CLASS book_format_ops
DEFAULT FOR TYPE book_format
USING btree AS
    OPERATOR 1 <,
    OPERATOR 2 <=,
    OPERATOR 3 =,
    OPERATOR 4 >=,
    OPERATOR 5 >,
    FUNCTION 1 book_format_cmp(book_format,book_format);
```

```
CREATE OPERATOR CLASS
```

Все готово. Проверим:

```
=> SELECT format FROM books GROUP BY format ORDER BY format;
```

```
   format
-------------
 (76,100,32)
 (84,108,32)
 (60,84,16)
 (60,88,16)
 (60,90,16)
 (66,90,16)
 (60,100,16)
 (70,90,16)
 (70,100,16)
 (76,100,16)
 (84,108,16)
 (60,90,8)
(12 rows)
```

Практика

1. В таблице хранятся даты событий. Какой индекс позволит ускорить запросы вида «найти определенное количество событий, наиболее близких по времени к указанной дате»? Проверьте.
2. Расширение pg_trgm добавляет функции и операторы для работы с *триграммами*, а также индексную поддержку. В частности, GiST-индекс ускоряет поиск по условиям вида

   `столбец LIKE '%что-то%'`

   Проверьте работу индекса. Найдите в системном каталоге все доступные операторы для класса gist_trgm_ops. Что в случае триграмм может служить общим предикатом, которому удовлетворяют все данные поддерева?

2. Триграммы — это последовательности из трех символов, на которые разбивается строка. Например, из «что-то» получаются следующие триграммы: « ч», « чт», «что», «то-», «о-т», «-то», «то », «о ».

Триграммы позволяют определять «похожесть» строк: если и в одной, и в другой много одинаковых триграмм, то и сами строки похожи.

Поиск по LIKE, где в начале шаблона находятся обычные символы (`'что-то%'`) ускоряется и простым индексом на основе B-дерева. Но такой индекс бесполезен, если шаблон начинается с %.

https://postgrespro.ru/docs/postgresql/12/pgtrgm

Для проверки можно воспользоваться таблицей с данными почтовой рассылки pgsql-hackers. Выполните команду

```
student$ zcat ~/mail_messages.sql.gz | psql -d ext_opclasses
```

чтобы восстановить эту таблицу в базу данных ext_opclasses.

В качестве запроса можно использовать следующий:

```
SELECT count(*)
FROM mail_messages
WHERE subject ILIKE '%magic%';
```

1. Индекс для событий

```
=> CREATE DATABASE ext_opclasses;
```

```
CREATE DATABASE
```

```
=> \c ext_opclasses
```

```
You are now connected to database "ext_opclasses" as user "student".
```

Создадим таблицу и заполним ее случайными данными:

```
=> CREATE TABLE events(
    date_happen timestamptz
);
```

```
CREATE TABLE
```

```
=> INSERT INTO events(date_happen)
    -- события за 10 лет
    SELECT now() - 10*365*24*60*60 * random() * interval '1 sec'
    FROM generate_series(1,10000);
```

```
INSERT 0 10000
```

Поиск наиболее близких дат — задача поиска ближайших соседей. Метод доступа btree не поддерживает (пока) поиск ближайших соседей, поэтому придется воспользоваться методом gist (и расширением btree_gist, добавляющим, в том числе, класс операторов для дат).

```
=> CREATE EXTENSION btree_gist;
```

```
CREATE EXTENSION
```

```
=> CREATE INDEX ON events USING gist(date_happen);
```

```
CREATE INDEX
```

Вот как может выглядеть запрос:

```
=> SELECT date_happen, date_happen <-> now() - interval '3 years' delta
FROM events
ORDER BY date_happen <-> now() - interval '3 years'
LIMIT 10;
```

```
          date_happen          |         delta
-------------------------------+-----------------------
 2017-11-06 11:02:10.648824+03 | 06:02:19.42393
 2017-11-07 17:54:09.454028+03 | 1 day 00:49:39.381274
 2017-11-07 22:59:13.361399+03 | 1 day 05:54:43.288645
 2017-11-07 23:37:12.406775+03 | 1 day 06:32:42.334021
 2017-11-08 00:25:57.690603+03 | 1 day 07:21:27.617849
 2017-11-08 03:28:19.97448+03  | 1 day 10:23:49.901726
 2017-11-05 04:02:51.638094+03 | 1 day 13:01:38.43466
 2017-11-08 06:31:12.541614+03 | 1 day 13:26:42.46886
 2017-11-05 00:24:53.728749+03 | 1 day 16:39:36.344005
 2017-11-08 11:45:15.153764+03 | 1 day 18:40:45.08101
(10 rows)
```

Его план выполнения использует созданный индекс:

```
=> EXPLAIN (costs off)
SELECT *
FROM events
ORDER BY date_happen <-> now() - interval '3 years'
LIMIT 10;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Limit
   ->  Index Only Scan using events_date_happen_idx on events
         Order By: (date_happen <-> (now() - '3 years'::interval))
(3 rows)
```

2. Расширение pg_trgm

Установим расширение:

```
=> CREATE EXTENSION pg_trgm;

CREATE EXTENSION
```

Посмотрим, как с его помощью ускоряется поиск по условию LIKE, на примере таблицы с архивом почтовой рассылки.

```
student$ zcat ~/mail_messages.sql.gz | psql -d ext_opclasses

SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
SET
ERROR:  relation "public.mail_messages" does not exist
ERROR:  table "mail_messages" does not exist
ERROR:  sequence "mail_messages_id_seq" does not exist
CREATE SEQUENCE
SET
SET
CREATE TABLE
COPY 356125
 setval
---------
 2317563
(1 row)

ALTER TABLE

=> \timing on

Timing is on.

=> SELECT count(*) FROM mail_messages WHERE subject ILIKE '%magic%';

 count
-------
   103
(1 row)

Time: 746,865 ms

=> \timing off

Timing is off.
```

Создадим индекс:

```
=> CREATE INDEX ON mail_messages USING gist(subject gist_trgm_ops);

CREATE INDEX

=> EXPLAIN (costs off)
SELECT count(*) FROM mail_messages WHERE subject ILIKE '%magic%';

                        QUERY PLAN
-----------------------------------------------------------
 Aggregate
   ->  Bitmap Heap Scan on mail_messages
         Recheck Cond: (subject ~~* '%magic%'::text)
         ->  Bitmap Index Scan on mail_messages_subject_idx
               Index Cond: (subject ~~* '%magic%'::text)
(5 rows)


=> \timing on

Timing is on.

=> SELECT count(*) FROM mail_messages WHERE subject ILIKE '%magic%';
```

```
 count
-------
   103
(1 row)

Time: 54,965 ms

=> \timing off

Timing is off.
```

Операторы для класса операторов gist_trgm_ops:

```
=> SELECT amop.amopopr::regoperator
FROM pg_opclass opc
    JOIN pg_opfamily opf ON opf.oid = opc.opcfamily
    JOIN pg_amop amop ON amop.amopfamily = opc.opcfamily
                     AND amop.amoplefttype = opc.opcintype
WHERE opcmethod = (SELECT oid FROM pg_am WHERE amname = 'gist')
AND opc.opcname = 'gist_trgm_ops';

     amopopr
------------------
 %(text,text)
 <->(text,text)
 ~~(text,text)
 ~~*(text,text)
 ~(text,text)
 ~*(text,text)
 %>(text,text)
 <->>(text,text)
 %>>(text,text)
 <->>>(text,text)
(10 rows)
```

- Операторы ~~ и ~~* эквивалентны LIKE и ILIKE;
- Операторы ~ и ~* проверяют соответствие регулярному выражению (с учетом и без учета регистра).

Значение остальных операторов описано на странице документации расширения pg_trgm.

Класс операторов для триграмм использует так называемое сигнатурное дерево. Триграммы заменяются на сигнатуры, то есть двоичные числа, в которых все биты нулевые, и лишь один определенный бит установлен в единицу. И вся строка заменяется на сигнатуру, которая вычисляется как побитовое ИЛИ сигнатур составляющих ее триграмм. Таким образом, вместо строки в индексе сохраняется одно число.

Общим предикатом в сигнатурном дереве является побитовое ИЛИ всех сигнатур, которые находятся ниже в поддереве.

Замена индексируемых элементов битовой сигнатурой позволяет индексировать с помощью GiST вообще все, что угодно (например, изображения). Но, к сожалению, это не слишком эффективный метод. Во-первых, число бит в сигнатуре ограничено, и поэтому некоторые разные элементы будут иметь одинаковую сигнатуру. Во-вторых, чем выше узел стоит в дереве, тем больше в его сигнатуре будет единиц. Это означает, что при поиске придется спускаться в лишние поддеревья и необходимо все время перепроверять полученные результаты.

В теме «Слабоструктурированные данные» будет показан другой, более эффективный (но менее универсальный) метод доступа — GIN.