

# Расширяемость Классы операторов



## Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

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](mailto: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.

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

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). В PostgreSQL существует единственный табличный метод heap, и других готовых методов (таких, например, как колоночное хранилище) еще нет, но они должны появиться в будущем.

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

**Индексный метод** доступа работает, когда надо быстро 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.

<https://postgrespro.ru/docs/postgresql/16/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 16 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)
```

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

Для получения информации мы делали запросы к таблицам системного каталога, однако в арсенале `psql` есть удобная команда, позволяющая получить информацию о методах доступа, классах и семействах операторов, ей дальше мы и будем пользоваться:

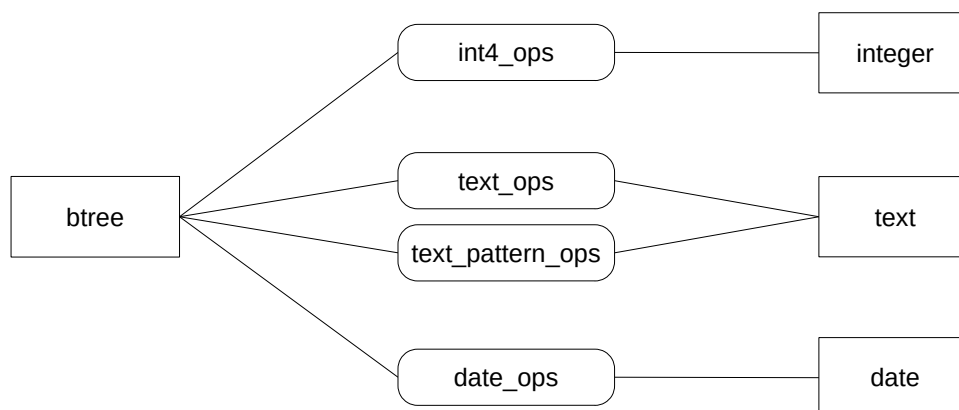
```
=> \dA
```

List of access methods

Name	Type
brin	Index
btree	Index
gin	Index
gist	Index
hash	Index
heap	Table
spgist	Index

(7 rows)

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



5

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 данных, 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/16/indexes-opclass>

## Классы операторов

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

```
=> \dAc btree (smallint|integer|bigint)
```

List of operator classes				
AM	Input type	Storage type	Operator class	Default?
btree	bigint		int8_ops	yes
btree	integer		int4_ops	yes
btree	smallint		int2_ops	yes

(3 rows)

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

```
=> \dAf btree (smallint|integer|bigint)
```

List of operator families		
AM	Operator family	Applicable types
btree	integer_ops	smallint, integer, bigint

(1 row)

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):

```
=> \dAc btree text
```

List of operator classes				
AM	Input type	Storage type	Operator class	Default?
btree	text		text_ops	yes
btree	text		text_pattern_ops	no
btree	text		varchar_ops	no
btree	text		varchar_pattern_ops	no

(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:

```
=> \dAo btree bool_ops
```

List of operators of operator families				
AM	Operator family	Operator	Strategy	Purpose
btree	bool_ops	<(boolean,boolean)	1	search
btree	bool_ops	<=(boolean,boolean)	2	search
btree	bool_ops	=(boolean,boolean)	3	search
btree	bool_ops	>=(boolean,boolean)	4	search
btree	bool_ops	>(boolean,boolean)	5	search

(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
);
```

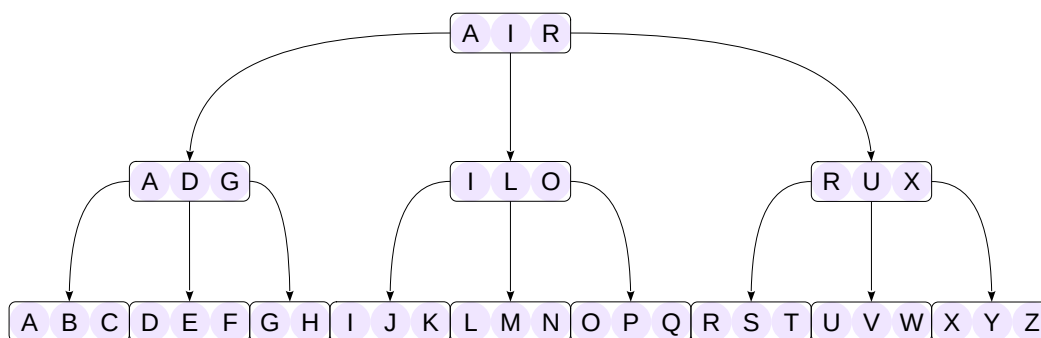
```
CREATE INDEX
```

# Метод доступа btree

Сбалансированное дерево, значения упорядочены

метод применим только для сортируемых типов данных

API метода доступа определяет порядок сортировки



7

Рассмотрим теперь несколько конкретных методов доступа, и начнем с В-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 «Н» (на рисунке) мы начинаем с корня. Поскольку «А» ≤ «Н» < «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/16/btree>

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

## Класс операторов для В-дерева

Класс операторов для В-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  
-----  
(1,kB)  
(2,kB)  
(7,B)  
(11,kB)  
(12,B)  
(15,kB)  
(16,TB)  
(17,MB)  
(18,B)  
(19,PB)  
(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  
LANGUAGE sql STRICT IMMUTABLE  
RETURN a.amount::numeric *  
    1024::numeric ^ ( array_position(enum_range(a.unit), a.unit)-1 );
```

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
LANGUAGE sql STRICT IMMUTABLE
RETURN 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;
```

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
LANGUAGE sql IMMUTABLE STRICT
RETURN capacity_cmp(a,b) < 0;
```

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
LANGUAGE sql IMMUTABLE STRICT
RETURN capacity_cmp(a,b) <= 0;
```

CREATE FUNCTION

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

CREATE OPERATOR

```
=> CREATE FUNCTION capacity_eq(a capacity, b capacity) RETURNS boolean
LANGUAGE sql IMMUTABLE STRICT
RETURN capacity_cmp(a,b) = 0;
```

CREATE FUNCTION

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

CREATE OPERATOR

```
=> CREATE FUNCTION capacity_ge(a capacity, b capacity) RETURNS boolean
LANGUAGE sql IMMUTABLE STRICT
RETURN capacity_cmp(a,b) >= 0;
```

CREATE FUNCTION

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

CREATE OPERATOR

```
=> CREATE FUNCTION capacity_gt(a capacity, b capacity) RETURNS boolean
LANGUAGE sql IMMUTABLE STRICT
RETURN capacity_cmp(a,b) > 0;
```

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
```

```
-----
```

```
(7,B)
```

```
(12,B)
```

```
(18,B)
```

```
(20,B)
```

```
(22,B)
```

```
(24,B)
```

```
(36,B)
```

```
(39,B)
```

```
(42,B)
```

```
(49,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?

```
=> SET enable_seqscan = off; -- временно отключим последовательное сканирование
```

```
SET
```

```
=> EXPLAIN (costs off)
```

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

```
QUERY PLAN
```

```
-----
Bitmap Heap Scan on test
  Filter: (capacity_cmp(cap, '(100,B)::capacity') < 0)
  -> Bitmap Index Scan on test_cap_idx
(3 rows)
```

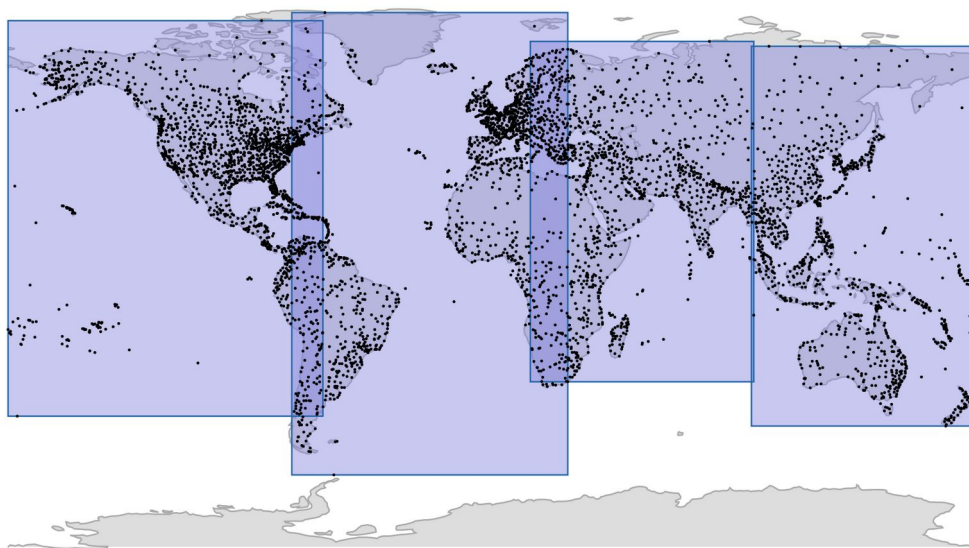
Да, поскольку:

- 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
-----
(7,B)
(12,B)
(18,B)
(20,B)
(22,B)
(24,B)
(36,B)
(39,B)
(42,B)
(49,B)
(58,B)
(83,B)
(86,B)
(92,B)
(92,B)
(15 rows)
```



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

В-дерево здесь не годится, но идею индексирования с помощью сбалансированного дерева можно обобщить. При поиске по дереву в каждом узле принимается решение, в каком поддереве (в общем случае — в нескольких поддеревьях) продолжить поиск. Для этого каждому поддереву ставится в соответствие условие (предикат), которому удовлетворяют все находящиеся в нем данные.

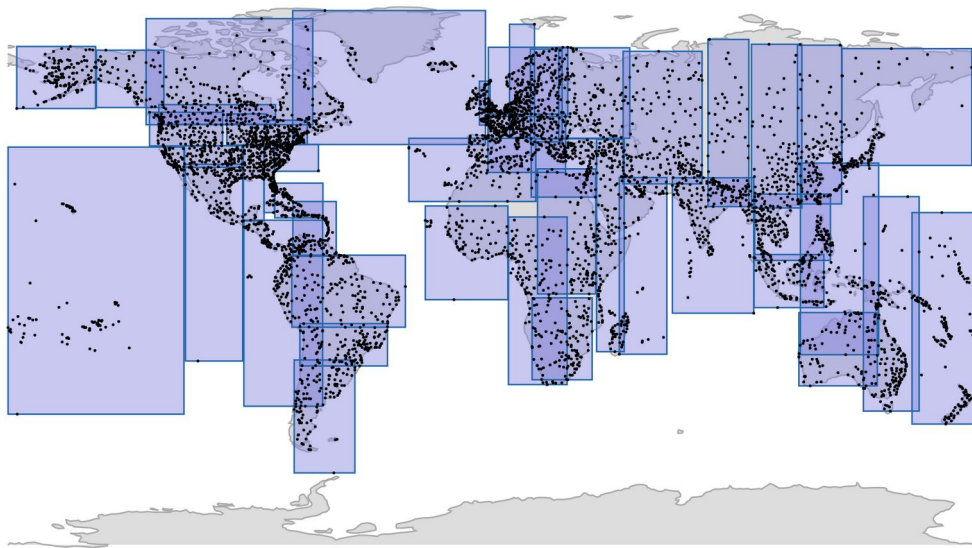
Класс операторов для GiST (generalized search tree — обобщенное дерево поиска) определяет, как выглядит предикат, и еще многие детали, необходимые для эффективной работы.

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

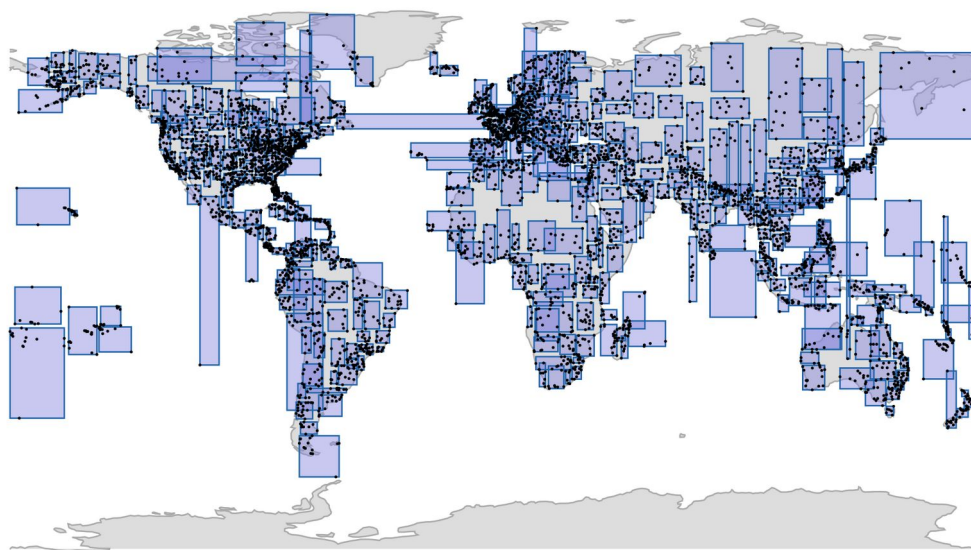
Плоскость разбивается на несколько прямоугольников, которые в сумме покрывают все индексируемые точки. Эти прямоугольники составляют верхний уровень дерева.

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

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



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



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 поддерева. 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.

## 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:

```
=> \dAc gist point
```

```

               List of operator classes
  AM  | Input type | Storage type | Operator class | Default?
-----+-----+-----+-----+-----
gist | point      | box          | point_ops      | yes
(1 row)
```

Операторы в этом классе:

```
=> \dAo gist point_ops
```

```

               List of operators of operator families
  AM  | Operator family | Operator          | Strategy | Purpose
-----+-----+-----+-----+-----
gist | point_ops      | <<(point,point)   | 1        | search
gist | point_ops      | >>(point,point)   | 5        | search
gist | point_ops      | ~=(point,point)   | 6        | search
gist | point_ops      | <<|(point,point)  | 10       | search
gist | point_ops      | |>>(point,point)  | 11       | search
gist | point_ops      | <->(point,point)  | 15       | ordering
gist | point_ops      | <^(point,point)   | 29       | search
gist | point_ops      | >^(point,point)   | 30       | search
gist | point_ops      | <@(point,box)     | 28       | search
gist | point_ops      | <@(point,circle)  | 68       | search
gist | point_ops      | <@(point,polygon) | 48       | search
(11 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(1,10_000);
```

```
INSERT 0 10000
```

Сколько точек расположено в круге радиуса 0.1?

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

```

 count
-----
      87
(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, ближайšie к данной (так называемый поиск ближайших соседей, k-NN search):

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

      p
-----
(-0.002459011726443805,-0.00634023829618835)
(0.015307941481957243,0.008562940357308424)
(-0.018211947588224753,0.0028783970911323564)
(-0.009564770729331151,-0.017729788910614275)
(-0.01743846352769962,0.014206376524721342)
(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:

```
=> \dAo gist range_ops
```

List of operators of operator families				
AM	Operator family	Operator	Strategy	Purpose
gist	range_ops	<<(anyrange,anyrange)	1	search
gist	range_ops	&<(anyrange,anyrange)	2	search
gist	range_ops	&&(anyrange,anyrange)	3	search
gist	range_ops	&>(anyrange,anyrange)	4	search
gist	range_ops	>>(anyrange,anyrange)	5	search
gist	range_ops	-  (anyrange,anyrange)	6	search
gist	range_ops	@>(anyrange,anyrange)	7	search
gist	range_ops	<@>(anyrange,anyrange)	8	search
gist	range_ops	=(anyrange,anyrange)	18	search
gist	range_ops	@>(anyrange,anyelement)	16	search
gist	range_ops	<<(anyrange,anymultirange)	1	search
gist	range_ops	&<(anyrange,anymultirange)	2	search
gist	range_ops	&&(anyrange,anymultirange)	3	search
gist	range_ops	&>(anyrange,anymultirange)	4	search
gist	range_ops	>>(anyrange,anymultirange)	5	search
gist	range_ops	-  (anyrange,anymultirange)	6	search
gist	range_ops	@>(anyrange,anymultirange)	7	search
gist	range_ops	<@>(anyrange,anymultirange)	8	search

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 в разных строках таблицы были два пересекающихся (оператор &&) диапазона. 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)=(["2025-06-24 13:00:00+03","2025-06-24 16:00:00+03"]) conflicts  
with existing key (during)=(["2025-06-24 12:00:00+03","2025-06-24 14:00:00+03"]).

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

```
=> 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 аудитории 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)=(["2025-06-24 13:00:00+03","2025-06-24 16:00:00+03"], 1)  
conflicts with existing key (during, room)=(["2025-06-24 12:00:00+03","2025-06-24  
14:00:00+03"], 1).

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 главами 4 и 5 книги <https://postgrespro.ru/education/books/internals>



1. Добавьте в таблицу `retail_prices` ограничение целостности, не позволяющее создать пересекающиеся диапазоны 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.

## 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 будут 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
```

```
-----
```

```
100
```

```
(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
LANGUAGE sql STRICT IMMUTABLE
RETURN f.width::numeric * f.height::numeric / f.parts::numeric;

CREATE FUNCTION

=> CREATE FUNCTION book_format_cmp(a book_format, b book_format) RETURNS integer
LANGUAGE sql STRICT IMMUTABLE
RETURN
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;

CREATE FUNCTION

=> CREATE FUNCTION book_format_lt(a book_format, b book_format) RETURNS boolean
LANGUAGE sql IMMUTABLE STRICT
RETURN book_format_cmp(a,b) < 0;

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
LANGUAGE sql IMMUTABLE STRICT
RETURN book_format_cmp(a,b) <= 0;

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
LANGUAGE sql IMMUTABLE STRICT
RETURN book_format_cmp(a,b) = 0;

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
LANGUAGE sql IMMUTABLE STRICT
RETURN book_format_cmp(a,b) >= 0;

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
LANGUAGE sql IMMUTABLE STRICT
RETURN book_format_cmp(a,b) > 0;

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)  
(84,108,16)  
(60,90,8)  
(11 rows)
```

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?

15

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/16/pgtrgm>

Для проверки можно воспользоваться таблицей с данными почтовой рассылки `pgsql-hackers`. Чтобы восстановить эту таблицу в базу данных `ext_opclasses`, выполните команду:

```
student$ zcat ~/mail_messages.sql.gz | psql -d 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 timestampz  
);
```

```
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(1,10_000);
```

```
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
```

Ускорить необходимые нам запросы поможет GiST-индекс:

```
=> 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
2022-06-24 01:57:35.035681+03	03:31:10.292112
2022-06-24 10:08:43.937497+03	04:39:58.609704
2022-06-23 23:32:20.934654+03	05:56:24.393139
2022-06-24 20:36:55.191085+03	15:08:09.863292
2022-06-25 00:54:16.888671+03	19:25:31.560878
2022-06-23 09:09:05.245305+03	20:19:40.082488
2022-06-23 05:10:26.405952+03	1 day 00:18:18.921841
2022-06-25 07:02:58.769333+03	1 day 01:34:13.44154
2022-06-23 01:14:30.973903+03	1 day 04:14:14.35389
2022-06-22 23:13:32.75483+03	1 day 06:15:12.572963

(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
ALTER TABLE
DROP TABLE
DROP SEQUENCE
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: 409,858 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: 27,345 ms
```

```
=> \timing off
```

```
Timing is off.
```

---

Операторы для класса операторов gist\_trgm\_ops:

```
=> \dAo gist gist_trgm_ops
```

List of operators of operator families				
AM	Operator family	Operator	Strategy	Purpose
gist	gist_trgm_ops	%(text,text)	1	search
gist	gist_trgm_ops	<->(text,text)	2	ordering
gist	gist_trgm_ops	~~(text,text)	3	search
gist	gist_trgm_ops	~~*(text,text)	4	search
gist	gist_trgm_ops	~(text,text)	5	search
gist	gist_trgm_ops	~*(text,text)	6	search
gist	gist_trgm_ops	%>(text,text)	7	search
gist	gist_trgm_ops	<->>(text,text)	8	ordering
gist	gist_trgm_ops	%>>(text,text)	9	search
gist	gist_trgm_ops	<->>>(text,text)	10	ordering
gist	gist_trgm_ops	=(text,text)	11	search

(11 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.