Bazy danych 2022

Piotr Wieczorek

5 maja 2022

Metody indeksowania

B-tree — indeksowanie wg klucza, dostosowane do dużych danych przechowywanych na dysku; zbalansowane drzewo poszukiwań o bardzo "grubych" wierzchołkach i bardzo dużej arności (w związku z tym płytkie); operatory `<,<=,=, >=,>`

hash — rozrzucanie indeksowanych danych wg funkcji haszującej do "dużych" kubełków; specyficzne metody obsługi przepełnienia kubełków (podwajanie, haszowanie liniowe, haszowanie rozszerzalne); operator `=`

GiST, SP-GiST, GIN, BRIN p. dokumentacja

`https://www.postgresql.org/docs/current/gist-intro.html`
`https://www.postgresql.org/docs/current/spgist.html`
`https://www.postgresql.org/docs/current/gin.html`
`https://www.postgresql.org/docs/current/brin.html`

Indeksowanie za pomocą btree. Rozwijanie postgresql

- Lehman-Yao High-Concurrency btrees

Indeksowanie za pomocą btree. Rozwijanie postgresql

- Lehman-Yao High-Concurrency btrees
- postgres@github
- Developer FAQ
- pgsql-hackers
- Przykładowa dyskusja (inlinowanie podzapytań z WITH aka CTEs)
- todo

Przedmiot/projekt MySQL w następnym semestrze (Bazy danych 2)

- (Oracle) MySQL

Co jest czym?

```
Nested Loop  (cost=4.65..118.62 rows=10 width=488)
                      (actual time=0.128..0.377 rows=10 loops=1)
```

- Estimated start-up cost. e.g., time to do the sorting in a sort node.
- Estimated total cost. (assumed to be run to completion, no LIMIT).
- Estimated number of rows output by this plan node. (assumed to be run to completion).
- Estimated average width of rows output by this plan node (in bytes).
- "actual time" values are in milliseconds of real time, whereas the cost estimates are expressed in arbitrary units;
- since no output rows are delivered to the client, network transmission costs and I/O conversion costs are not included.
- results on a toy-sized table cannot be assumed to apply to large tables (e.g., table on a single disk page).

EXPLAIN ANALYZE actually runs the query, any side-effects will happen

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628 rows=0 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=0.101..0.439 rows
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual time=0.043
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
 Execution time: 14.727 ms

ROLLBACK;
```

Indeksowanie

```sql
SELECT * FROM users WHERE displayname= 'Isaac';

CREATE INDEX i_users_displayname ON users (displayname);
```

Indeksowanie

```
SELECT * FROM users WHERE displayname= 'Isaac';

CREATE INDEX i_users_displayname ON users (displayname);


SELECT * FROM users WHERE lower(displayname)= 'isaac';

CREATE INDEX i_users_displayname ON users (lower(displayname));
```

Metody dostępu

- Sequential scan (czyta wszystko z tabeli)

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli)

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało, random access

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało, random access
- Bitmap Index Scan (najpierw zaznacza sobie strony z pasującymi krotkami i potem je przegląda)

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało, random access
- Bitmap Index Scan (najpierw zaznacza sobie strony z pasującymi krotkami i potem je przegląda) dobry gdy ?, unika random access

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało, random access
- Bitmap Index Scan (najpierw zaznacza sobie strony z pasującymi krotkami i potem je przegląda) dobry gdy ?, unika random access ale robi dwa skany

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało, random access
- Bitmap Index Scan (najpierw zaznacza sobie strony z pasującymi krotkami i potem je przegląda) dobry gdy ?, unika random access ale robi dwa skany
- Łączenie kilku Bitmap Index Scanów - możliwe za pomocą BitmapAnd / BitmapOr.

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało, random access
- Bitmap Index Scan (najpierw zaznacza sobie strony z pasującymi krotkami i potem je przegląda) dobry gdy ?, unika random access ale robi dwa skany
- Łączenie kilku Bitmap Index Scanów - możliwe za pomocą BitmapAnd / BitmapOr.
- Index Only Scan (patrzy tylko do indeksu, nie dotyka tabeli)

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało, random access
- Bitmap Index Scan (najpierw zaznacza sobie strony z pasującymi krotkami i potem je przegląda) dobry gdy ?, unika random access ale robi dwa skany
- Łączenie kilku Bitmap Index Scanów - możliwe za pomocą BitmapAnd / BitmapOr.
- Index Only Scan (patrzy tylko do indeksu, nie dotyka tabeli) - działa gdy wybieramy tylko kolumny z indeksu

Metody dostępu

- Sequential scan (czyta wszystko z tabeli) - dobry gdy pasujących krotek jest dużo
- Index Scan (sprawdza tylko pasujące krotki z tabeli) - dobry gdy pasujących krotek jest mało, random access
- Bitmap Index Scan (najpierw zaznacza sobie strony z pasującymi krotkami i potem je przegląda) dobry gdy ?, unika random access ale robi dwa skany
- Łączenie kilku Bitmap Index Scanów - możliwe za pomocą BitmapAnd / BitmapOr.
- Index Only Scan (patrzy tylko do indeksu, nie dotyka tabeli) - działa gdy wybieramy tylko kolumny z indeksu
- Jeśli indeks nie zawiera wszystkich potrzebnych kolumn to można sztucznie dodać kolumnę ($\rightarrow$ covering index):
  np. dla `SELECT y FROM tab WHERE x = 'key';`
  `CREATE INDEX tab_x_y ON tab(x, y);`
  lub `CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);`

```
student=> EXPLAIN ANALYZE SELECT * FROM users WHERE lower(displayname)= 'isaac';
-------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on users  (cost=4.31..15.48 rows=3 width=457)
                            (actual time=0.039..0.046 rows=3 loops=1)
   Recheck Cond: (lower(displayname) = 'isaac'::text)
   Heap Blocks: exact=3
   ->  Bitmap Index Scan on users_lower_idx  (cost=0.00..4.31 rows=3 width=0)
                                             (actual time=0.030..0.030 rows=3 loops=1)
         Index Cond: (lower(displayname) = 'isaac'::text)
 Planning Time: 0.157 ms
 Execution Time: 0.090 ms
```

```
student=> EXPLAIN ANALYZE SELECT * FROM users WHERE lower(displayname)= 'isaac';
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on users  (cost=4.31..15.48 rows=3 width=457)
                            (actual time=0.039..0.046 rows=3 loops=1)
   Recheck Cond: (lower(displayname) = 'isaac'::text)
   Heap Blocks: exact=3
   ->  Bitmap Index Scan on users_lower_idx  (cost=0.00..4.31 rows=3 width=0)
                                             (actual time=0.030..0.030 rows=3 loops=1)
         Index Cond: (lower(displayname) = 'isaac'::text)
 Planning Time: 0.157 ms
 Execution Time: 0.090 ms


student=> DROP index users_lower_idx; -- DROP INDEX
student=> EXPLAIN ANALYZE SELECT * FROM users WHERE lower(displayname)= 'isaac';
-------------------------------------------------------------------------------------------------------
 Seq Scan on users  (cost=0.00..447.48 rows=43 width=457)
                    (actual time=0.029..5.877 rows=3 loops=1)
   Filter: (lower(displayname) = 'isaac'::text)
   Rows Removed by Filter: 8629
 Planning Time: 0.137 ms
 Execution Time: 5.921 ms
```

```
student=>  EXPLAIN SELECT * FROM users WHERE id=3;
------------------------------------------------------------------------
Index Scan using users_pkey on users  (cost=0.15..8.17 rows=1 width=36)
Index Cond: (id = 3)
```

```
student=>  EXPLAIN SELECT * FROM users WHERE id=3;
------------------------------------------------------------------------
Index Scan using users_pkey on users  (cost=0.15..8.17 rows=1 width=36)
Index Cond: (id = 3)

student=> EXPLAIN SELECT id FROM users WHERE id=3;
------------------------------------------------------------------------
Index Only Scan using users_pkey on users  (cost=0.15..8.17 rows=1 width=4)
Index Cond: (id = 3)
```

GiST

https://www.postgresql.org/docs/current/gist-builtin-opclasses.html#GIST-BUILTIN-OPCLASSES-TABLE

Table 64.1. Built-in GiST Operator Classes

Name	Indexed Data Type	Indexable Operators	Ordering Operators
box_ops	box	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	
circle_ops	circle	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->

...

Table 9.34. Geometric Operators

Operator	Description	Example
+	Translation	box '((0,0),(1,1))' + point '(2.0,0)'
-	Translation	box '((0,0),(1,1))' - point '(2.0,0)'
*	Scaling/rotation	box '((0,0),(1,1))' * point '(2.0,0)'
/	Scaling/rotation	box '((0,0),(2,2))' / point '(2.0,0)'
#	Point or box of intersection	box '((1,-1),(-1,1))' # box '((1,1),(-2,-2))'
#	Number of points in path or polygon	# path '((1,0),(0,1),(-1,0))'
@-@	Length or circumference	@-@ path '((0,0),(1,0))'
@@	Center	@@ circle '((0,0),10)'
##	Closest point to first operand on second operand	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	Distance between	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	Overlaps? (One point in common makes this true.)	box '((0,0),(1,1))' && box '((0,0),(2,2))'

GiST

```
CREATE TABLE points(p POINT);

INSERT INTO points(p) VALUES
  (point '(1,1)'), (point '(1,4)'), (point '(4,1)'),
  (point '(4,4)'), (point '(2,2)');

INSERT INTO points(p)
 SELECT point(n*random()/10000, n*random()/10000)
 FROM generate_series(1,10000) AS n;

CREATE INDEX ON points USING GIST (p)

SELECT p FROM points WHERE p <@ box '(3,3),(7,7)'

SELECT * FROM points ORDER BY p <-> point '(0,0)' LIMIT 10;
```

GiST

```
CREATE TABLE lectures(during tsrange);
INSERT INTO lectures(during) VALUES
 ('["2021-04-22 10:15","2021-04-22 12:00")');

CREATE index ON lectures USING GIST(during);

SELECT * FROM lectures where during && '[2021-04-22 11:00, 2021-04-22 11:00]';
SELECT * FROM lectures where during && '[2021-04-22 11:00, 2021-04-22 11:15)';
SELECT * FROM lectures where during <@ '(2021-04-22 11:00, 2021-04-22 11:15]'; -- no
SELECT * FROM lectures where during @> '[2021-04-22 11:00, 2021-04-22 11:15]';
```

ranges

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

ranges

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

- `int4range, int8range` — Range of integer/bigint
- `numrange` — Range of numeric
- `tsrange, tstzrange` — Range of timestamp without/with time zone
- `daterange` — Range of date

ranges

```
-- Containment
SELECT int4range(10, 20) @> 3;

-- Overlaps
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound
SELECT upper(int8range(15, 25));

-- Compute the intersection, i.e.,  [15,20)
SELECT int4range(10, 20) * int4range(15, 25);

-- Is the range empty?
SELECT isempty(numrange(1, 5));
SELECT isempty(numrange('empty'));
```

GIN

Table 66.1. Built-in GIN Operator Classes

Name	Indexed Data Type	Indexable Operators
array_ops	anyarray	&& <@ = @>
jsonb_ops	jsonb	? ?& ?| @> @? @@
jsonb_path_ops	jsonb	@> @? @@
tsvector_ops	tsvector	@@ @@@

```
-- contains
  ARRAY[1,4,3] @> ARRAY[3,1,3]
-- is contained by
  ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6]
-- overlap (have elements in common)
  ARRAY[1,4,3] && ARRAY[2,1]
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);

INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);

INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;

CREATE INDEX ON points USING GIST (p);
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);

INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;

CREATE INDEX ON points USING GIST (p);

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);

INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;

CREATE INDEX ON points USING GIST (p);

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;

-- selektywne (tzn. wybiera kilkaset krotek)
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);

INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;

CREATE INDEX ON points USING GIST (p);

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;

-- selektywne (tzn. wybiera kilkaset krotek)

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);

INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;

CREATE INDEX ON points USING GIST (p);

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;

-- selektywne (tzn. wybiera kilkaset krotek)

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';

-- mało selektywne (wszystkie krotki - 1 mln)
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);

INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;

CREATE INDEX ON points USING GIST (p);

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;

-- selektywne (tzn. wybiera kilkaset krotek)

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';

-- mało selektywne (wszystkie krotki - 1 mln)

EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' LIMIT 200;
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);

INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;

CREATE INDEX ON points USING GIST (p);

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;

-- selektywne (tzn. wybiera kilkaset krotek)

EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';

-- mało selektywne (wszystkie krotki - 1 mln)

EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' LIMIT 200;

-- tylko 200 krotek
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);
INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;
CREATE INDEX ON points USING GIST (p);
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;
-- selektywne (tzn. wybiera kilkaset krotek)
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';
-- mało selektywne (wszystkie krotki - 1 mln)
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' LIMIT 200;
-- tylko 200 krotek
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)'
```

Kiedy indeksowanie się przydaje?

```
-- btree index on id
CREATE TABLE points(id SERIAL PRIMARY KEY, p point, t text, n numeric, ts timestamp);
INSERT INTO points(p, t, n, ts)
 SELECT point(n*random()/10000, n*random()/10000),
  MD5(random()::text) || ' Does this example work at all? ',
 random()*1234567::numeric, now()
 FROM generate_series(1,1000000) AS n;
CREATE INDEX ON points USING GIST (p);
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;
-- selektywne (tzn. wybiera kilkaset krotek)
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';
-- mało selektywne (wszystkie krotki - 1 mln)
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' LIMIT 200;
-- tylko 200 krotek
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)'
-- wszystkie krotki - 1 mln
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;
-- P1, bez indeksu
 Gather  (cost=1000.00..33336.33 rows=1000 width=16)
                      (actual time=0.384..58.582 rows=319 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on points  (cost=0.00..32236.33 rows=417 width=16)
                      (actual time=0.446..51.965 rows=106 loops=3)
         Filter: (p <@ '(0.1,0.1),(0.05,0.05)'::box)
         Rows Removed by Filter: 333227
 Execution Time: 58.630 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0.05,0.05),(0.1,0.1)' ;
-- P1, bez indeksu
 Gather  (cost=1000.00..33336.33 rows=1000 width=16)
                       (actual time=0.384..58.582 rows=319 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on points  (cost=0.00..32236.33 rows=417 width=16)
                       (actual time=0.446..51.965 rows=106 loops=3)
         Filter: (p <@ '(0.1,0.1),(0.05,0.05)'::box)
         Rows Removed by Filter: 333227
 Execution Time: 58.630 ms
-- P2, z indeksem
 Index Only Scan using points_p_idx on points  (cost=0.29..57.78 rows=1000 width=16)
                           (actual time=0.028..0.132 rows=316 loops=1)
   Index Cond: (p <@ '(0.1,0.1),(0.05,0.05)'::box)
   Heap Fetches: 0
 Execution Time: 0.165 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';

-- P3, bez indeksu
 Gather  (cost=1000.00..11714.33 rows=1000 width=16)
                                (actual time=0.681..136.516 rows=1000000 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on points  (cost=0.00..10614.33 rows=417 width=16)
 (actual time=0.017..48.662 rows=333333 loops=3)
         Filter: (p <@ '(1444,1444),(0,0)'::box)
 Execution Time: 179.058 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(1444,1444)';
-- P3, bez indeksu
 Gather  (cost=1000.00..11714.33 rows=1000 width=16)
                                (actual time=0.681..136.516 rows=1000000 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on points  (cost=0.00..10614.33 rows=417 width=16)
 (actual time=0.017..48.662 rows=333333 loops=3)
         Filter: (p <@ '(1444,1444),(0,0)'::box)
 Execution Time: 179.058 ms
-- P4, z indeksem

 Index Only Scan using points_p_idx on points  (cost=0.29..57.78 rows=1000 width=16)
                               (actual time=1.141..219.205 rows=1000000 loops=1)
   Index Cond: (p <@ '(1444,1444),(0,0)'::box)
   Heap Fetches: 0
 Execution Time: 261.925 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' LIMIT 200;
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' LIMIT 200;

-- P5, bez indeksu
 Limit  (cost=51244.41..51267.74 rows=200 width=24) (actual time=126.050..126.119 rows=200 loops=1)
   ->  Gather Merge  (cost=51244.41..148473.49 rows=833334 width=24) (actual time=126.048..128.323 rows=2
         Workers Planned: 2
         Workers Launched: 2
         ->  Sort  (cost=50244.38..51286.05 rows=416667 width=24) (actual time=120.592..120.613 rows=200
               Sort Key: ((p <-> '(0,0)'::point))
               Sort Method: top-N heapsort  Memory: 44kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 51kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 50kB
               ->  Parallel Seq Scan on points  (cost=0.00..32236.33 rows=416667 width=24) (actual time=0
 Execution Time: 128.397 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' LIMIT 200;

-- P5, bez indeksu
 Limit  (cost=51244.41..51267.74 rows=200 width=24) (actual time=126.050..126.119 rows=200 loops=1)
   ->  Gather Merge  (cost=51244.41..148473.49 rows=833334 width=24) (actual time=126.048..128.323 rows=20
         Workers Planned: 2
         Workers Launched: 2
         ->  Sort  (cost=50244.38..51286.05 rows=416667 width=24) (actual time=120.592..120.613 rows=200 l
               Sort Key: ((p <-> '(0,0)'::point))
               Sort Method: top-N heapsort  Memory: 44kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 51kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 50kB
               ->  Parallel Seq Scan on points  (cost=0.00..32236.33 rows=416667 width=24) (actual time=0
 Execution Time: 128.397 ms
-- P6, z indeksem
 Limit  (cost=0.29..33.20 rows=200 width=24) (actual time=0.204..0.760 rows=200 loops=1)
   ->  Index Only Scan using points_p_idx on points  (cost=0.29..164596.29 rows=1000000 width=24) (actual
         Order By: (p <-> '(0,0)'::point)
         Heap Fetches: 200
 Execution Time: 0.816 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' ;
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' ;

-- P7, bez indeksu
 Sort  (cost=309163.09..311616.46 rows=981348 width=24) (actual time=1036.562..1197.327 rows=1000000 loops
   Sort Key: ((p <-> '(0,0)'::point))
   Sort Method: external merge  Disk: 33296kB
   ->  Seq Scan on points  (cost=0.00..191368.85 rows=981348 width=24) (actual time=52.899..559.299 rows=
 Execution Time: 1266.082 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)' ;

-- P7, bez indeksu
 Sort  (cost=309163.09..311616.46 rows=981348 width=24) (actual time=1036.562..1197.327 rows=1000000 loops
   Sort Key: ((p <-> '(0,0)'::point))
   Sort Method: external merge  Disk: 33296kB
   ->  Seq Scan on points  (cost=0.00..191368.85 rows=981348 width=24) (actual time=52.899..559.299 rows=
 Execution Time: 1266.082 ms
-- P7A, bez indeksu ale po SET work_mem = '128MB'
 Sort  (cost=136548.84..139048.84 rows=1000000 width=24) (actual time=624.471..782.473 rows=1000000 loops
   Sort Key: ((p <-> '(0,0)'::point))
   Sort Method: quicksort  Memory: 102702kB
   ->  Seq Scan on points  (cost=0.00..36891.00 rows=1000000 width=24) (actual time=0.032..235.571 rows=1
 Execution Time: 890.924 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)';
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)';
-- P8, z indeksem, niedługo po operacji INSERT Limit  (cost=0.29..153828.29 rows=1000000 width=24) (actual
   ->  Index Only Scan using points_p_idx on points  (cost=0.29..153828.29 rows=1000000 width=24) (actual
         Order By: (p <-> '(0,0)'::point)
         Heap Fetches: 1000000
 Execution Time: 1687.223 ms
```

Kiedy indeksowanie się przydaje?

```
EXPLAIN ANALYZE SELECT p FROM points ORDER BY p <-> point '(0,0)';
-- P8, z indeksem, niedługo po operacji INSERT Limit  (cost=0.29..153828.29 rows=1000000 width=24) (actual
   ->  Index Only Scan using points_p_idx on points  (cost=0.29..153828.29 rows=1000000 width=24) (actual
         Order By: (p <-> '(0,0)'::point)
         Heap Fetches: 1000000
 Execution Time: 1687.223 ms
-- P9, z indeksem, po wydaniu polecenia VACUUM
 Index Only Scan using points_p_idx on points  (cost=0.29..56716.19 rows=1020195 width=24) (actual time=0
   Order By: (p <-> '(0,0)'::point)
   Heap Fetches: 0
 Execution Time: 765.871 ms
```

Wnioski:

- Zapytanie selektywne (P2) lub z LIMIT (P6) dużo szybsze z indeksem (P1, P5) ponieważ indeks pomaga wybrać krotki wynikowe,

Wnioski:

- Zapytanie selektywne (P2) lub z LIMIT (P6) dużo szybsze z indeksem (P1, P5) ponieważ indeks pomaga wybrać krotki wynikowe,
- Dla zapytań mało selektywnych (P3 vs P4 oraz P7 vs P8 vs P9) - index-only scan nie jest wybierany albo działa (mniej więcej) podobnie jak wersja z seqscan.

Wnioski:

- Zapytanie selektywne (P2) lub z LIMIT (P6) dużo szybsze z indeksem (P1, P5) ponieważ indeks pomaga wybrać krotki wynikowe,
- Dla zapytań mało selektywnych (P3 vs P4 oraz P7 vs P8 vs P9) - index-only scan nie jest wybierany albo działa (mniej więcej) podobnie jak wersja z seqscan.
- Nie ma się co dziwić - tak trzeba wszystkie krotki przejrzeć i indeks nie pomaga ograniczyć tej liczby.

Wnioski:

- Zapytanie selektywne (P2) lub z LIMIT (P6) dużo szybsze z indeksem (P1, P5) ponieważ indeks pomaga wybrać krotki wynikowe,
- Dla zapytań mało selektywnych (P3 vs P4 oraz P7 vs P8 vs P9) - index-only scan nie jest wybierany albo działa (mniej więcej) podobnie jak wersja z seqscan.
- Nie ma się co dziwić - tak trzeba wszystkie krotki przejrzeć i indeks nie pomaga ograniczyć tej liczby.
- Ale czasami (zwłaszcza dla tabel z wieloma kolumnami) plan index-only może być szybszy nawet jak zwraca dużo krotek (patrz P7 vs P8 vs P9)

Wnioski:

- Zapytanie selektywne (P2) lub z LIMIT (P6) dużo szybsze z indeksem (P1, P5) ponieważ indeks pomaga wybrać krotki wynikowe,
- Dla zapytań mało selektywnych (P3 vs P4 oraz P7 vs P8 vs P9) - index-only scan nie jest wybierany albo działa (mniej więcej) podobnie jak wersja z seqscan.
- Nie ma się co dziwić - tak trzeba wszystkie krotki przejrzeć i indeks nie pomaga ograniczyć tej liczby.
- Ale czasami (zwłaszcza dla tabel z wieloma kolumnami) plan index-only może być szybszy nawet jak zwraca dużo krotek (patrz P7 vs P8 vs P9)
- ew. przyspieszenie zależy od niuansów: VACUUM ANALYZE & visibility map, uwaga na Heap Fetches

Indeksowanie punktów za pomocą btree?

```
CREATE INDEX points_btree_id ON points USING btree ((p <-> point (0,0)));
CREATE INDEX points_btree_desc_id ON points USING btree ((p <-> point (0,0)) DESC);
```

Indeksowanie punktów za pomocą btree?

```
CREATE INDEX points_btree_id ON points USING btree ((p <-> point (0,0)));
CREATE INDEX points_btree_desc_id ON points USING btree ((p <-> point (0,0)) DESC);
-- multicolumn
(...) ORDER BY x ASC, y DESC
```

Indeksowanie punktów za pomocą btree

```
CREATE INDEX points_btree_id ON points USING btree (p[0], p[1]);
```

Indeksowanie punktów za pomocą btree

```
CREATE INDEX points_btree_id ON points USING btree (p[0], p[1]);
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1;
 (...)
   ->  Bitmap Index Scan on points_btree_id  (cost=0.00..184.09 rows=7955 width=0)
                         (actual time=0.691..0.691 rows=7909 loops=1)
         Index Cond: (p[0] < '0.1'::double precision)
 Execution Time: 4.986 ms
```

Indeksowanie punktów za pomocą btree

```
CREATE INDEX points_btree_id ON points USING btree (p[0], p[1]);
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1;
 (...)
   ->  Bitmap Index Scan on points_btree_id  (cost=0.00..184.09 rows=7955 width=0)
                         (actual time=0.691..0.691 rows=7909 loops=1)
         Index Cond: (p[0] < '0.1'::double precision)
 Execution Time: 4.986 ms
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1 and p[1]<0.1;
```

Indeksowanie punktów za pomocą btree

```
CREATE INDEX points_btree_id ON points USING btree (p[0], p[1]);
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1;
 (...)
   ->  Bitmap Index Scan on points_btree_id  (cost=0.00..184.09 rows=7955 width=0)
                         (actual time=0.691..0.691 rows=7909 loops=1)
         Index Cond: (p[0] < '0.1'::double precision)
 Execution Time: 4.986 ms
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1 and p[1]<0.1;
 Index Scan using points_btree_id on points  (cost=0.42..392.66 rows=62 width=16) (actual time=0.023..2.0
   Index Cond: ((p[0] < '0.1'::double precision) AND (p[1] < '0.1'::double precision))
 Execution Time: 2.195 ms
```

Indeksowanie punktów za pomocą btree

```
CREATE INDEX points_btree_id ON points USING btree (p[0], p[1]);
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1;
 (...)
   ->  Bitmap Index Scan on points_btree_id  (cost=0.00..184.09 rows=7955 width=0)
                         (actual time=0.691..0.691 rows=7909 loops=1)
         Index Cond: (p[0] < '0.1'::double precision)
 Execution Time: 4.986 ms
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1 and p[1]<0.1;
 Index Scan using points_btree_id on points  (cost=0.42..392.66 rows=62 width=16) (actual time=0.023..2.0
   Index Cond: ((p[0] < '0.1'::double precision) AND (p[1] < '0.1'::double precision))
 Execution Time: 2.195 ms
EXPLAIN ANALYZE SELECT p FROM points WHERE p[1]<0.1;
```

Indeksowanie punktów za pomocą btree

```
CREATE INDEX points_btree_id ON points USING btree (p[0], p[1]);
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1;
 (...)
   ->  Bitmap Index Scan on points_btree_id  (cost=0.00..184.09 rows=7955 width=0)
                         (actual time=0.691..0.691 rows=7909 loops=1)
         Index Cond: (p[0] < '0.1'::double precision)
 Execution Time: 4.986 ms
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<0.1 and p[1]<0.1;
 Index Scan using points_btree_id on points  (cost=0.42..392.66 rows=62 width=16) (actual time=0.023..2.0
   Index Cond: ((p[0] < '0.1'::double precision) AND (p[1] < '0.1'::double precision))
 Execution Time: 2.195 ms
EXPLAIN ANALYZE SELECT p FROM points WHERE p[1]<0.1;
   (...)
   ->  Parallel Seq Scan on points  (cost=0.00..22450.33 rows=3240 width=16)
         (actual time=0.011..48.619 rows=2635 loops=3)
         Filter: (p[1] < '0.1'::double precision)
         Rows Removed by Filter: 330699
 Execution Time: 55.287 ms
```

Indeksowanie punktów za pomocą btree

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(50,0.0001)' ;
```

Indeksowanie punktów za pomocą btree

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(50,0.0001)' ;
 Index Only Scan using points_p_idx1 on points  (cost=0.29..57.78 rows=1000 width=16) (actual time=0.080.
   Index Cond: (p <@ '(50,0.0001),(0,0)'::box)
   Heap Fetches: 0
 Execution Time: 0.275 ms
```

Indeksowanie punktów za pomocą btree

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(50,0.0001)' ;
 Index Only Scan using points_p_idx1 on points  (cost=0.29..57.78 rows=1000 width=16) (actual time=0.080.
   Index Cond: (p <@ '(50,0.0001),(0,0)'::box)
   Heap Fetches: 0
 Execution Time: 0.275 ms
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<50 and p[1]<0.0001;
```

Indeksowanie punktów za pomocą btree

```
EXPLAIN ANALYZE SELECT p FROM points WHERE p <@ box '(0,0),(50,0.0001)' ;
 Index Only Scan using points_p_idx1 on points  (cost=0.29..57.78 rows=1000 width=16) (actual time=0.080.
   Index Cond: (p <@ '(50,0.0001),(0,0)'::box)
   Heap Fetches: 0
 Execution Time: 0.275 ms
EXPLAIN ANALYZE SELECT p FROM points WHERE p[0]<50 and p[1]<0.0001;
 Index Scan using points_btree_id on points  (cost=0.42..21750.80 rows=85 width=16) (actual time=0.046..33
   Index Cond: ((p[0] < '50'::double precision) AND (p[1] < '0.0001'::double precision))
 Execution Time: 33.217 ms
```

Tuning (strojenie) bazy

- Always run ANALYZE first (and VACUUM).

Tuning (strojenie) bazy

- Always run ANALYZE first (and VACUUM).
- Use real data for experimentation.

Tuning (strojenie) bazy

- Always run ANALYZE first (and VACUUM).
- Use real data for experimentation.
- When indexes are not used, it can be useful for testing to force their use.
  `SET enable_seqscan=off, SET enable_nestloop=OFF`.