Bazy danych 2024

Piotr Wieczorek

7 maja 2024

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=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
```

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', domyślnie work_mem = 4MB (dlaczego???)
 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=10
 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
   ->  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
   ->  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

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
- uwaga na użyte algorytmy (external merge sort vs. quicksort) i ustawienia pamięci (SET work_mem)

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

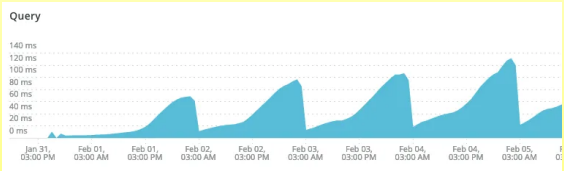
An article on Medium

```
SELECT *
FROM atable t
WHERE t.package_no='GonderiNo' AND
      t.hub_id=42 AND
     (t.date BETWEEN '2021-02-04 21:00:00.000000' AND
                     '2021-02-05 21:00:00.000000');

CREATE INDEX CONCURRENTLY idx_test_date_hub_id_package_no
ON t(date, hub_id, package_no);
```

Indeksowanie punktów za pomocą btree

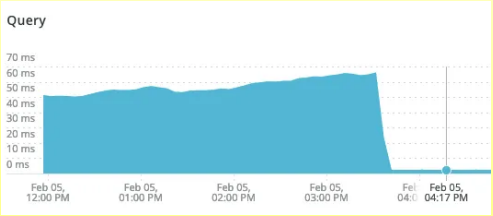
An article on Medium

```sql
SELECT *
FROM atable t
WHERE t.package_no='GonderiNo' AND
      t.hub_id=42 AND
     (t.date BETWEEN '2021-02-04 21:00:00.000000' AND
                     '2021-02-05 21:00:00.000000');

CREATE INDEX CONCURRENTLY idx_test_date_hub_id_package_no
ON t(date, hub_id, package_no);
```



Indeksowanie punktów za pomocą btree

```sql
SELECT *
FROM atable t
WHERE t.package_no='GonderiNo' AND
      t.hub_id=42 AND
     (t.date BETWEEN '2021-02-04 21:00:00.000000' AND
                     '2021-02-05 21:00:00.000000');

CREATE INDEX CONCURRENTLY idx_test_date_hub_id_package_no
  ON t(package_no, hub_id, date);
```



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..
   Index Cond: ((p[0] < '50'::double precision) AND (p[1] < '0.0001'::double precision))
 Execution Time: 33.217 ms
```

Indeksy - powtórka

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ]
       ON [ ONLY ] table_name [ USING method ] ( { column_name | ( expression ) }
       [ COLLATE collation ]
       [ opclass [ ( opclass_parameter = value [, ... ] ) ] ]
       [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
  [ INCLUDE ( column_name [, ...] ) ]
  [ NULLS [ NOT ] DISTINCT ]
  [ WITH ( storage_parameter [= value] [, ... ] ) ]
  [ TABLESPACE tablespace_name ]
  [ WHERE predicate ]
```

Indeksy - powtórka

```
CREATE UNIQUE INDEX title_idx ON films (title);

CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);

CREATE INDEX title_idx ON films (title) WITH (deduplicate_items = off);
-- posting list tuple for duplicate tuples (sorted array)

CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");

CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);

CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);

CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);
-- pending changes list: consistent response time vs. update speed (!!!)

CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
-- more work - two scans - but do not block writes

CREATE INDEX idxgin ON api USING GIN (jdoc);

CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

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`.

Dobry projekt bazy danych

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

OSOBA(ido,tytul,indeks,nazwisko,adres),

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

OSOBA(ido,tytul,indeks,nazwisko,adres),

- Student ma indeks i nie ma tytułu; pracownik — odwrotnie:

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

OSOBA(ido,tytul,indeks,nazwisko,adres),

- Student ma indeks i nie ma tytułu; pracownik — odwrotnie: PR(ido,tytuł,nazwisko,adres), ST(ido,indeks,nazwisko,adres)

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

OSOBA(ido,tytul,indeks,nazwisko,adres),

- Student ma indeks i nie ma tytułu; pracownik — odwrotnie: PR(ido,tytuł,nazwisko,adres), ST(ido,indeks,nazwisko,adres)
- Grupa ma kilka terminów zajęć — kilkakrotne wpisanie grupy do tabeli oznacza redundancję (limit, idk,idp):

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

OSOBA(ido,tytul,indeks,nazwisko,adres), GRUPA(idg,idk,idp,termin,limit,sala)

- Student ma indeks i nie ma tytułu; pracownik — odwrotnie: PR(ido,tytuł,nazwisko,adres), ST(ido,indeks,nazwisko,adres)
- Grupa ma kilka terminów zajęć — kilkakrotne wpisanie grupy do tabeli oznacza redundancję (limit, idk,idp): GR(idg,idk,idp,limit), TS(idg,termin,sala)

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

OSOBA(ido,tytul,indeks,nazwisko,adres), GRUPA(idg,idk,idp,termin,limit,sala)

- Student ma indeks i nie ma tytułu; pracownik — odwrotnie: PR(ido,tytuł,nazwisko,adres), ST(ido,indeks,nazwisko,adres)
- Grupa ma kilka terminów zajęć — kilkakrotne wpisanie grupy do tabeli oznacza redundancję (limit, idk,idp): GR(idg,idk,idp,limit), TS(idg,termin,sala)
- Łatwo sprawdzić:
  - grupa ma jednego prowadzącego
  - studenci mają unikalne indeksy

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

OSOBA(ido,tytul,indeks,nazwisko,adres), GRUPA(idg,idk,idp,termin,limit,sala)

- Student ma indeks i nie ma tytułu; pracownik — odwrotnie: PR(ido,tytuł,nazwisko,adres), ST(ido,indeks,nazwisko,adres)
- Grupa ma kilka terminów zajęć — kilkakrotne wpisanie grupy do tabeli oznacza redundancję (limit, idk,idp): GR(idg,idk,idp,limit), TS(idg,termin,sala)
- Łatwo sprawdzić:
  - grupa ma jednego prowadzącego
  - studenci mają unikalne indeksy
- Trudno sprawdzić, że identyfikatory osób są unikalne.

Dobry projekt bazy danych

NULL-e — krotki powinny pasować do schematu tabeli,

Redundancja — informacja nie powinna być zapisywana wielokrotnie,

Kontrola więzów — sprawdzanie własności klucza, unikalności i innych więzów powinno być łatwe,

Obliczanie złączeń — jest trudne, więc nie należy rozdrabniać zbytnio bazy.

OSOBA(ido,tytul,indeks,nazwisko,adres), GRUPA(idg,idk,idp,termin,limit,sala)

- Student ma indeks i nie ma tytułu; pracownik — odwrotnie: PR(ido,tytuł,nazwisko,adres), ST(ido,indeks,nazwisko,adres)
- Grupa ma kilka terminów zajęć — kilkakrotne wpisanie grupy do tabeli oznacza redundancję (limit, idk,idp): GR(idg,idk,idp,limit), TS(idg,termin,sala)
- Łatwo sprawdzić:
  - grupa ma jednego prowadzącego
  - studenci mają unikalne indeksy
- Trudno sprawdzić, że identyfikatory osób są unikalne.
- Wyznaczenie terminu i sali zajęć grupy wymaga złączenia.

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)
- Pole nie ma wartości (student nie ma promotora (jeszcze))

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)
- Pole nie ma wartości (student nie ma promotora (jeszcze))
- Nie wiadomo, która z powyższych (może ma jakiegoś promotora, a może nie)

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)
- Pole nie ma wartości (student nie ma promotora (jeszcze))
- Nie wiadomo, która z powyższych (może ma jakiegoś promotora, a może nie)
- tablice Codda (każdy NULL to potencjalnie inna wartość)

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)
- Pole nie ma wartości (student nie ma promotora (jeszcze))
- Nie wiadomo, która z powyższych (może ma jakiegoś promotora, a może nie)
- tablice Codda (każdy NULL to potencjalnie inna wartość)
- tablice naiwne (NULL to zmienna)

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)
- Pole nie ma wartości (student nie ma promotora (jeszcze))
- Nie wiadomo, która z powyższych (może ma jakiegoś promotora, a może nie)
- tablice Codda (każdy NULL to potencjalnie inna wartość)
- tablice naiwne (NULL to zmienna)
- tablice warunkowe (kombinacje boolowskie w krotkach!)

Czym są NULLe?

```
IF(OLD.text!=NEW.text) THEN         -- OLD.text<>NEW.text
          NEW.lasteditdate:=now();
          INSERT INTO commenthistory(commentid, creationdate, text)
             VALUES(OLD.id, OLD.lasteditdate, OLD.text);
```

Czym są NULLe?

```
IF(OLD.text!=NEW.text) THEN         -- OLD.text<>NEW.text
          NEW.lasteditdate:=now();
          INSERT INTO commenthistory(commentid, creationdate, text)
             VALUES(OLD.id, OLD.lasteditdate, OLD.text);
IF(OLD.text IS DISTINCT FROM NEW.text) THEN
          NEW.lasteditdate:=now();
          INSERT INTO commenthistory(commentid, creationdate, text)
             VALUES(OLD.id, OLD.lasteditdate, OLD.text);
```

Czym są NULLe?

- Operacje arytmetyka, porównania na NULLach - wynikiem NULL (UNKNOWN)

Czym są NULLe?

- Operacje arytmetyka, porównania na NULLach - wynikiem NULL (UNKNOWN)
- `IS [ NOT ] NULL`

Czym są NULLe?

- Operacje arytmetyka, porównania na NULLach - wynikiem NULL (UNKNOWN)
- `IS [ NOT ] NULL`
- `a IS [ NOT ] DISTINCT FROM b`

Czym są NULLe?

- Operacje arytmetyka, porównania na NULLach - wynikiem NULL (UNKNOWN)
- `IS [ NOT ] NULL`
- `a IS [ NOT ] DISTINCT FROM b`

a	*b*	*a* AND *b*	*a* OR *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Czym są NULLe?

- `COUNT(*)` zlicza NULLe

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi
- `SUM(kol)` ignoruje NULLe, sumuje resztę

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi
- `SUM(kol)` ignoruje NULLe, sumuje resztę
- `SUM(kol)` zwraca NULL dla pustego zbioru krotek

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi
- `SUM(kol)` ignoruje NULLe, sumuje resztę
- `SUM(kol)` zwraca NULL dla pustego zbioru krotek
- podobnie inne funkcje agregujące (za wyjątkiem `COUNT(*)` i `COUNT(kol)`, one zwracają 0)

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi
- `SUM(kol)` ignoruje NULLe, sumuje resztę
- `SUM(kol)` zwraca NULL dla pustego zbioru krotek
- podobnie inne funkcje agregujące (za wyjątkiem `COUNT(*)` i `COUNT(kol)`, one zwracają 0)
- `COALESCE(SUM(kol),0)` - przydatne, zwłaszcza przy złączeniach zewnętrznych

Czym są NULLe?

ORDERS

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

Czym są NULLe?

ORDERS

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

Czym są NULLe?

ORDERS

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
     ( SELECT * FROM Orders O, Payments P
       WHERE  C.cust_id = P.cust_id
         AND  P.order_id = O.order_id )
```

Czym są NULLe?

ORDERS

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
     ( SELECT * FROM Orders O, Payments P
       WHERE  C.cust_id = P.cust_id
         AND  P.order_id = O.order_id )
```

- Unpaid orders -> `Ord3`, Customers with no order -> EMPTY

Czym są NULLe?

ORDERS

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
     ( SELECT * FROM Orders O, Payments P
       WHERE  C.cust_id = P.cust_id
         AND  P.order_id = O.order_id )
```

- Unpaid orders -> `Ord3`, Customers with no order -> EMPTY
- Co gdy w `Payments` wartość `Ord2` stanie się NULLem?

Czym są NULLe?

Orders

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

Payments

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

Customers

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
     ( SELECT * FROM Orders O, Payments P
       WHERE  C.cust_id = P.cust_id
         AND  P.order_id = O.order_id )
```

- Unpaid orders -> `Ord3`, Customers with no order -> EMPTY
- Co gdy w `Payments` wartość `Ord2` stanie się NULLem?
- Unpaid orders -> EMPTY, Customers with no order -> `Cust2`

Czym są NULLe?

Orders

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

Payments

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

Customers

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
     ( SELECT * FROM Orders O, Payments P
       WHERE  C.cust_id = P.cust_id
         AND  P.order_id = O.order_id )
```

- Unpaid orders -> `Ord3`, Customers with no order -> EMPTY
- Co gdy w `Payments` wartość `Ord2` stanie się NULLem?
- Unpaid orders -> EMPTY, Customers with no order -> `Cust2`
- Więcej: P.Guagliardo, L. Libkin. *Correctness of SQL queries on databases with nulls.* SIGMOD Record, 46(3): 5-16 (2017).

Czym są NULLe?

Zamówienia, dla których mamy pewność, że nie zostały zapłacone (jeden NULL i pusty wynik):

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

Czym są NULLe?

Zamówienia, dla których mamy pewność, że nie zostały zapłacone (jeden NULL i pusty wynik):

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

Zamówienia, dla których nie mamy pewności, że zostały zapłacone (czyli wszystkie potencjalnie problematyczne):

```
SELECT O.order_id
FROM Orders O LEFT JOIN
     Payments P ON O.order_id = P.order_id
WHERE
     P.order_id IS NULL;
```