What's new

PostgreSQL

Oleg Bartunov, CEO Postgres Professional

CTE

- A CTEs (Common Table Expression) is a temporary tables existing for just one query, that can be referenced from a primary query. Useful to break complex query to a readable parts — easy read and maintain.
- Most databases consider CTEs as views and optimize overall query
- Postgres implementation — always materialize CTEs
  - CTE uses work_mem, beware large results of CTEs
  - Optimization fence like <OFFSET 0>

    «*CTEs are also treated as optimization fences; this is not so much an optimizer limitation as to keep the semantics sane when the CTE contains a writable query.», Tom Lane, 2011*

    Logically equivalent queries (subselects and WITH) executed with different plans !

CTE

```
WITH RECURSIVE x(i) — idea by Graeme Job
AS (
    VALUES(0)
UNION ALL
    SELECT i + 1 FROM x WHERE i < 101
),
Z(Ix, Iy, Cx, Cy, X, Y, I)
AS (
    SELECT Ix, Iy, X::float, Y::float, X::float,
Y::float, 0
    FROM (SELECT -2.2 + 0.031 * i, i FROM x) AS
xgen(x,ix)
    CROSS JOIN
        (SELECT -1.5 + 0.031 * i, i FROM x) AS
ygen(y,iy)
    UNION ALL
    SELECT Ix, Iy, Cx, Cy, X*X - Y*Y + Cx AS X,
Y*X*2 + Cy, I + 1
    FROM Z
    WHERE X*X + Y*Y < 16.0  AND I < 27
),
Zt (Ix, Iy, I)
AS (
    SELECT Ix, Iy, MAX(I) AS I
    FROM Z
    GROUP BY Iy, Ix
    ORDER BY Iy, Ix
)
SELECT array_to_string(
    array_agg(
        SUBSTRING(' .,,,,-----++++%%%%@@@@#### ',
                  GREATEST(I,1),1)),''
    )
FROM Zt
GROUP BY Iy
ORDER BY Iy;
```

CTE

- Writable CTEs always executed
- Non-referenced CTEs never executed

```
WITH yy AS
      ( SELECT * FROM cte WHERE y > 1),
not_executed AS
      (SELECT * FROM cte),
always_executed AS
      (INSERT INTO cte VALUES(2,2) RETURNING *)
SELECT FROM yy WHERE x=2;
        QUERY PLAN
----------------------------
 CTE Scan on yy
   Filter: (x = 2)
   CTE yy
     ->  Seq Scan on cte
           Filter: (y > 1)
   CTE always_executed
     ->  Insert on cte cte_1
           ->  Result
(8 rows)
```

CTE is a black box for optimizer

- Break a really complex query to the well readable parts

```
CREATE TABLE cte AS SELECT x, x AS y FROM generate_series(1,10000000) AS x;
CREATE INDEX ON cte(x,y);
               Table "public.cte"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 x      | integer |           |          |
 y      | integer |           |          |
Indexes:
    "cte_x_y_idx" btree (x, y)

-- subselects

SELECT * FROM
    (SELECT * FROM cte WHERE y>1) AS t
WHERE x=2;

— CTE

WITH yy AS (
    SELECT * FROM cte
    WHERE y>1
)
SELECT * FROM  yy
WHERE  x=2;
```

CTE is a black box for optimizer

```
WITH yy AS ( - always materialized and cannot inlined into a parent query
    SELECT * FROM cte
    WHERE y>1
)
SELECT * FROM  yy
WHERE  x=2;

 CTE Scan on yy (actual time=0.099..3672.842 rows=1 loops=1)
   Filter: (x = 2)
   Rows Removed by Filter: 9999998
   CTE yy
     ->  Seq Scan on cte (actual time=0.097..1355.367 rows=9999999 loops=1)
           Filter: (y > 1)
           Rows Removed by Filter: 1
 Planning Time: 0.088 ms
 Execution Time: 3735.986 ms
(9 rows)

SELECT * FROM (SELECT * FROM cte WHERE y>1) as t WHERE X=2;
                                     QUERY PLAN
--------------------------------------------------------------------------------
 Index Only Scan using cte_x_y_idx on cte (actual time=0.013..0.013 rows=1 loop
   Index Cond: ((x = 2) AND (y > 1))
   Heap Fetches: 0
 Planning Time: 0.058 ms
 Execution Time: 0.025 ms
(5 rows)
```

SURPRISE: CTE is 150 000 slower than subselect !

PG12: Controllable CTE materialization

```
WITH cte_name  AS [NOT] MATERIALIZED
```

- Writable WITH query always materialized
- Recursive WITH query always materialized

- No fencing (new default)

```
WITH yy AS (
    SELECT * FROM cte
    WHERE y=2
)
SELECT * FROM  yy
WHERE  x=2;

               QUERY PLAN
------------------------------------------
 Index Only Scan using cte_x_y_idx on cte
   Index Cond: ((x = 2) AND (y = 2))
(2 rows)
```

- Old behavior

```
WITH yy AS MATERIALIZED (
    SELECT * FROM cte
    WHERE y=2
)
SELECT * FROM  yy
WHERE  x=2;

        QUERY PLAN
---------------------------
 CTE Scan on yy
   Filter: (x = 2)
   CTE yy
     ->  Seq Scan on cte
           Filter: (y = 2)
(5 rows)
```

PG12: Controllable CTE materialization

```
WITH cte_name  AS [NOT] MATERIALIZED
```

- If a WITH query is referred to multiple times, CTE “materialize” its result to prevent double execution, use EXPLICIT NOT MATERIALIZED

```
WITH yy AS  ( SELECT * FROM cte WHERE y > 1) SELECT (SELECT count(*) FROM
yy WHERE x=2), (SELECT count(*) FROM yy WHERE x=2);
                                  QUERY PLAN
-----------------------------------------------------------------------------
 Result (actual time=3922.274..3922.275 rows=1 loops=1)
   CTE yy
     ->  Seq Scan on cte (actual time=0.023..1295.262 rows=9999999 loops=1)
           Filter: (y > 1)
           Rows Removed by Filter: 1
   InitPlan 2 (returns $1)
     ->  Aggregate (actual time=3109.687..3109.687 rows=1 loops=1)
           ->  CTE Scan on yy (actual time=0.027..3109.682 rows=1 loops=1)
                 Filter: (x = 2)
                 Rows Removed by Filter: 9999998
   InitPlan 3 (returns $2)
     ->  Aggregate (actual time=812.580..812.580 rows=1 loops=1)
           ->  CTE Scan on yy yy_1 (actual time=0.016..812.575 rows=1
loops=1)
                 Filter: (x = 2)
                 Rows Removed by Filter: 9999998
 Planning Time: 0.136 ms
 Execution Time: 3939.848 ms
(17 rows)
```

PG12: Controllable CTE materialization

```
WITH cte_name  AS [NOT] MATERIALIZED
```

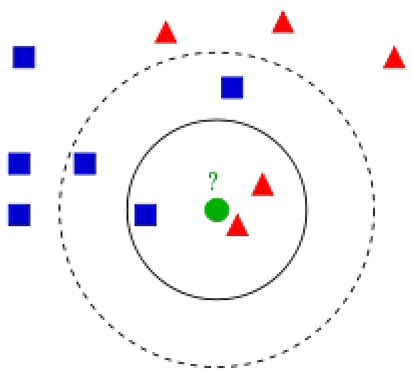
- If a WITH query is referred to multiple times, CTE “materialize” its result to prevent double execution, use EXPLICIT NOT MATERIALIZED

```
 WITH yy AS NOT MATERIALIZED  ( SELECT * FROM cte WHERE y > 1) SELECT
(SELECT count(*) FROM yy WHERE x=2), (SELECT count(*) FROM yy WHERE x=2);
                                               QUERY PLAN
---------------------------------------------------------------------------
 Result (actual time=0.035..0.035 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Aggregate (actual time=0.024..0.024 rows=1 loops=1)
           ->  Index Only Scan using cte_x_y_idx on cte (actual
time=0.019..0.020 rows=1 loops=1)
                 Index Cond: ((x = 2) AND (y > 1))
                 Heap Fetches: 1
   InitPlan 2 (returns $1)
     ->  Aggregate (actual time=0.006..0.006 rows=1 loops=1)
           ->  Index Only Scan using cte_x_y_idx on cte cte_1 (actual
time=0.004..0.005 rows=1 loops=1)
                 Index Cond: ((x = 2) AND (y > 1))
                 Heap Fetches: 1
 Planning Time: 0.253 ms
 Execution Time: 0.075 ms
(13 rows)
```

Efficient K-nearest neighbour search in PostgreSQL

Knn-search: The problem

- What are the closest restaurants near HaUmanim 12, Tel Aviv ?
- What happens in the world near the launch of Sputnik ?
- Reverse image search, search by image
- ....
- GIS, Science (high-dimensional data)



K-nearest neighbour search

- 10 closest events to the launch of Sputnik ?

```
SELECT id, date, event FROM events
ORDER ABS(date - '1957-10-04'::date) ASC LIMIT 10;
   id    |    date    |                                event
--------+------------+-----------------------------------------------------------------
  58136 | 1957-10-04 | "Leave It to Beaver," debuts on CBS
  58137 | 1957-10-04 | U.S.S.R. launches Sputnik I, 1st artificial Earth satellite
 117062 | 1957-10-04 | Gregory T Linteris, Demarest, New Jersey, astronaut, sk: STS 83
 117061 | 1957-10-04 | Christina Smith, born in Miami, Florida, playmate, Mar, 1978
 102671 | 1957-10-05 | Lee "Kix" Thompson, saxophonist, Madness-Baggy Trousers
 102670 | 1957-10-05 | Larry Saumell, jockey
  58292 | 1957-10-05 | Yugoslav dissident Milovan Djilos sentenced to 7 years
  58290 | 1957-10-05 | 11th NHL All-Star Game: All-Stars beat Montreal 5-3 at Montreal
  31456 | 1957-10-03 | Willy Brandt elected mayor of West Berlin
  58291 | 1957-10-05 | 12th Ryder Cup: Britain-Ireland, 7 -4 at Lindrick GC, England
(10 rows)
```

- Slow: Index is useless, full heap scan, sort, limit

```
 Limit (actual time=54.481..54.485 rows=10 loops=1)
   Buffers: shared hit=1824
   ->  Sort (actual time=54.479..54.481 rows=10 loops=1)
         Sort Key: (abs((date - '1957-10-04'::date)))
         Sort Method: top-N heapsort  Memory: 26kB
         Buffers: shared hit=1824
         ->  Seq Scan on events (actual time=0.020..25.896 rows=151643 loops=1)
               Buffers: shared hit=1824
 Planning Time: 0.091 ms
 Execution Time: 54.513 ms
(10 rows)
```

Knn-search: Existing solutions

- Traditional way to speedup query
  - Indexes are very inefficient (no predicate)
  - Constrain data space (range search)
    - Incremental search → to many queries
    - Need to know in advance size of neighbourhood, how ?
      1Km is ok for Paris, but too small for Siberia
    - Maintain 'density map' ?

Knn-search: What do we want !

- We want to avoid full table scan – read only <right> tuples
  - So, we need index
- We want to avoid sorting – read <right> tuples in <right> order
  - So, we need special strategy to traverse index
- We want to support tuples visibility
  - So, we should be able to resume index traverse

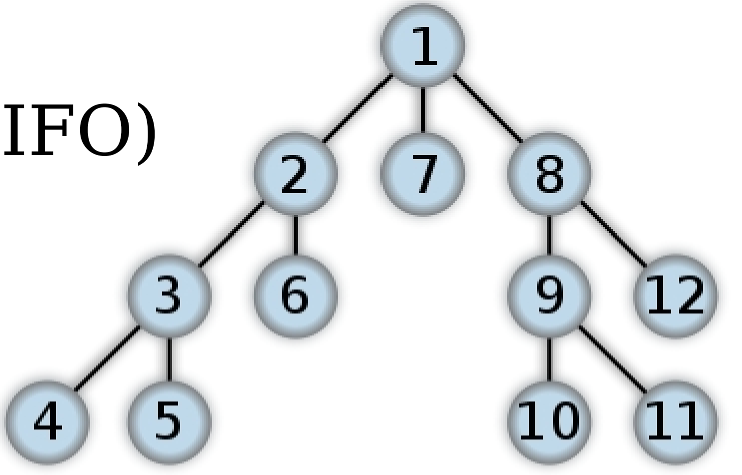
Knn-search: What do we want !

- We want to avoid full table scan – read only <right> tuples
  - So, we need index
- We want to avoid sorting – read <right> tuples in <right> order
  - So, we need special strategy to traverse index
- We want to support tuples visibility
  - So, we should be able to resume index traverse

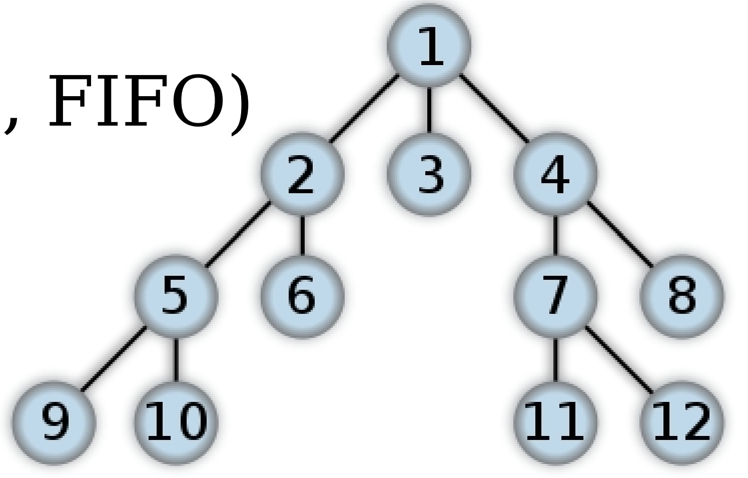
Knn-search: Index traverse

- Depth First Search (stack, LIFO)

  R-tree search



- Breadth First Search (queue, FIFO)



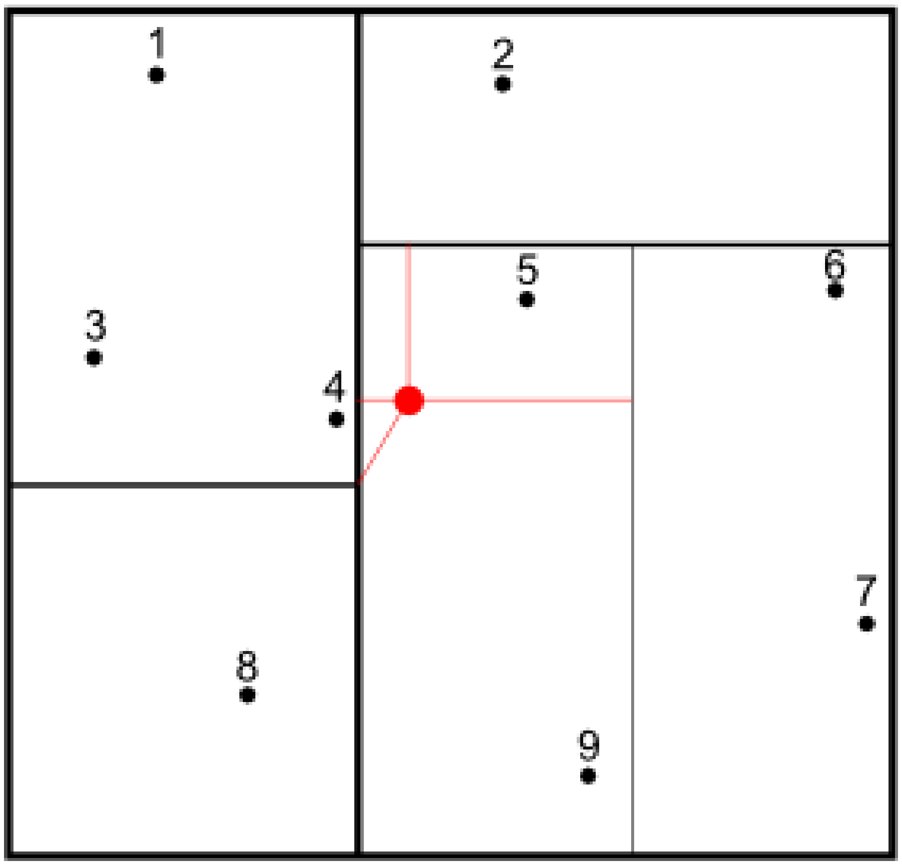
- Both strategies are not good for us – full index scan

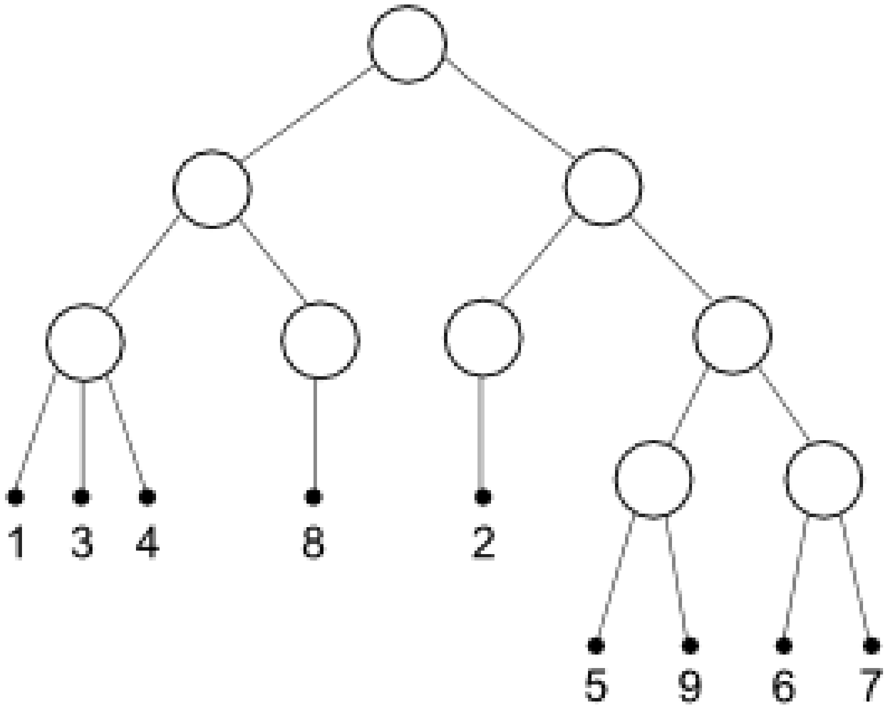
Knn-search: Index traverse

- Best First Search (PQ, priority queue). Maintain order of items in PQ according their distance from given point
  - Distance to MBR (rectangle for Rtree) for internal pages
    - minimum distance of all items in that MBR
  - Distance = 0 for MBR with given point
  - Distance to point for leaf pages
- Each time we extract point from PQ we output it – it is next closest point ! If we extract rectangle, we expand it by pushing their children (rectangles and points) into the queue.
- We traverse index by visiting only interesting nodes !

Knn-search: Index traverse

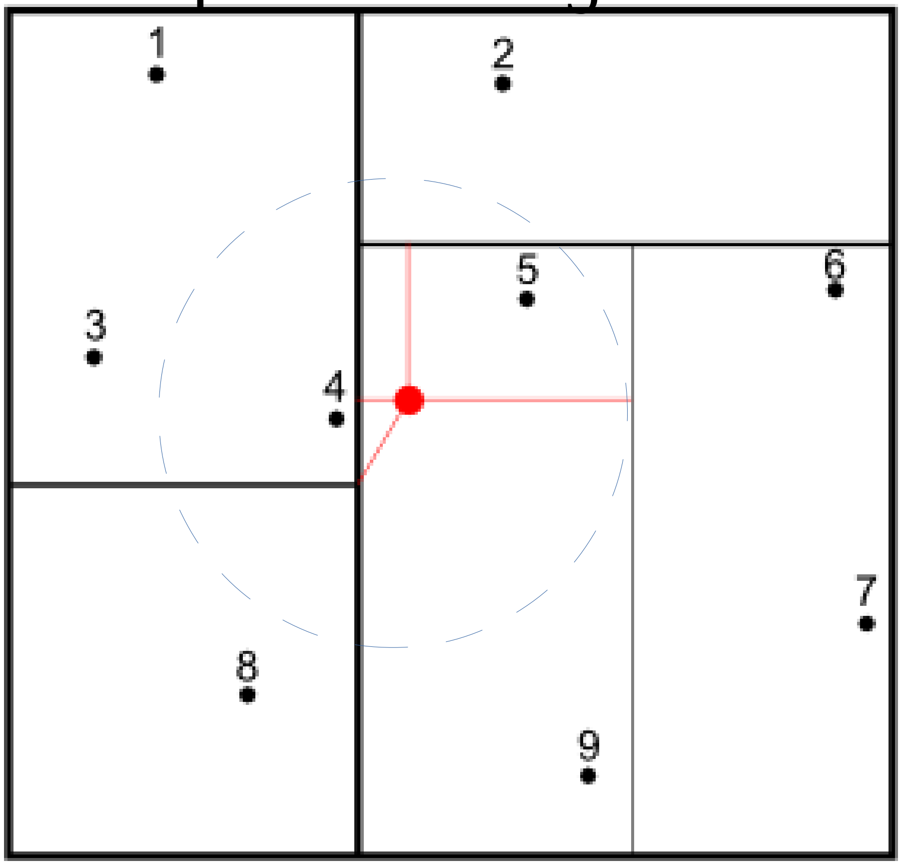
- Simple example – non-overlapped partitioning





Knn-search: Index traverse

- Simple example – non-overlapped partitioning



- Priority Queue

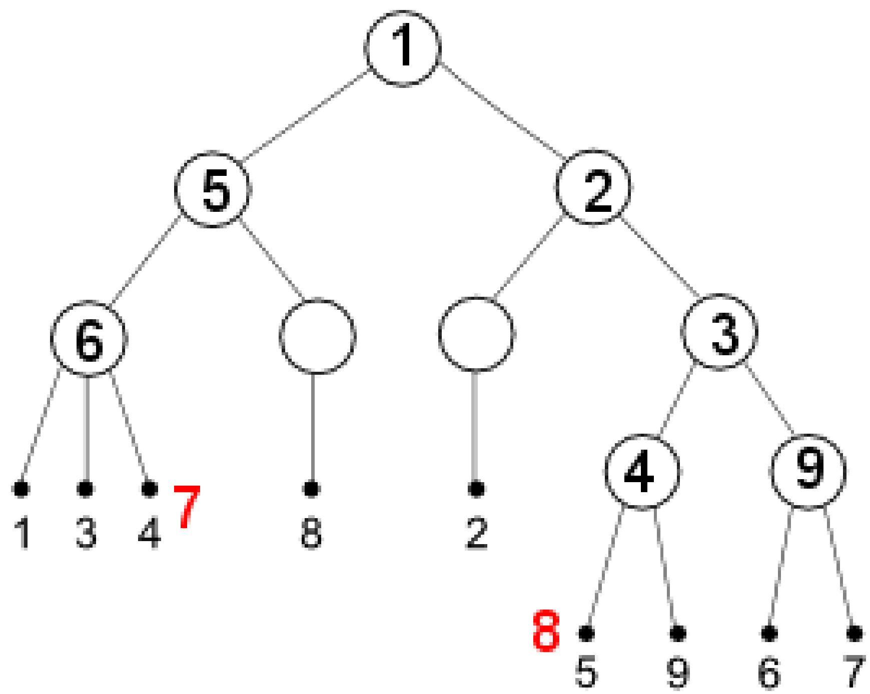
- 1: {1,2,3,4,5,6,7,8,9}
- 2: {2,5,6,7,9}, {1,3,4,8}
- 3: {5,6,7,9}, {1,3,4,8}, {2}
- 4: {5,9}, {1,3,4,8}, {2}, {6,7}
- 5: {1,3,4,8}, 5, {2}, {6,7}, 9
- 6: {1,3,4}, {8}, 5, {2}, {6,7}, 9
- 7: 4, {8}, 5, {2}, {6,7}, 3, 1, 9

  we can output 4 without visit other rectangles !

- 8: 5, {2}, {6,7}, 3, 8, 1, 9
- 9: {6,7}, 3, 2, 8, 1, 9
- 10: 3, 2, 8, 1, 9, 6, 7

Knn-search: Index traverse

- Simple example – non-overlapped partitioning



- Priority Queue
  - 1: {1,2,3,4,5,6,7,8,9}
  - 2: {2,5,6,7,9}, {1,3,4,8}
  - 3: {5,6,7,9}, {1,3,4,8}, {2}
  - 4: {5,9}, {1,3,4,8}, {2}, {6,7}
  - 5: {1,3,4,8}, 5, {2}, {6,7}, 9
  - 6: {1,3,4}, {8}, 5, {2}, {6,7}, 9
  - **7:** 4, {8}, 5, {2}, {6,7}, 3, 1, 9
  - 8: 5, {2}, {6,7}, 3, 8, 1, 9

Knn-search: Performance

- SEQ (no index) – base performance
  - Sequentially read full table + Sort full table (can be very bad, sort_mem !)
- DFS – very bad !
  - Full index scan + Random read full table + Sort full table
- BFS – the best for small k !
  - Partial index scan + Random read k-records
    - T(index scan) ~ Height of Search tree ~ log(n)
  - Performance win BFS/SEQ ~ Nrelpages/k, for small k. The more rows, the more benefit !
  - Can still win even for k=n (for large tables) - no sort !

K-nearest neighbour search

```
SELECT id, date, event FROM events
ORDER ABS(date - '1957-10-04'::date) ASC LIMIT 10;
Limit (actual time=54.481..54.485 rows=10 loops=1)
   Buffers: shared hit=1824
   ->  Sort (actual time=54.479..54.481 rows=10 loops=1)
         Sort Key: (abs((date - '1957-10-04'::date)))
         Sort Method: top-N heapsort  Memory: 26kB
         Buffers: shared hit=1824
         ->  Seq Scan on events (actual time=0.020..25.896 rows=151643 loops=1)
               Buffers: shared hit=1824
 Planning Time: 0.091 ms
 Execution Time: 54.513 ms
(10 rows)
```

KNN-GiST (Btree-GiST)

```
SELECT id, date, event FROM events
ORDER BY date <-> '1957-10-04'::date ASC LIMIT 10;
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Limit (actual time=0.128..0.145 rows=10 loops=1)
   ->  Index Scan using events_date_idx1 on events (actual time=0.128..0.142 rows=10 loops=1)
         Order By: (date <-> '1957-10-04'::date)
 Planning Time: 0.155 ms
 Execution Time: 0.186 ms
(5 rows)
```

KNN SP-GiST (committed)

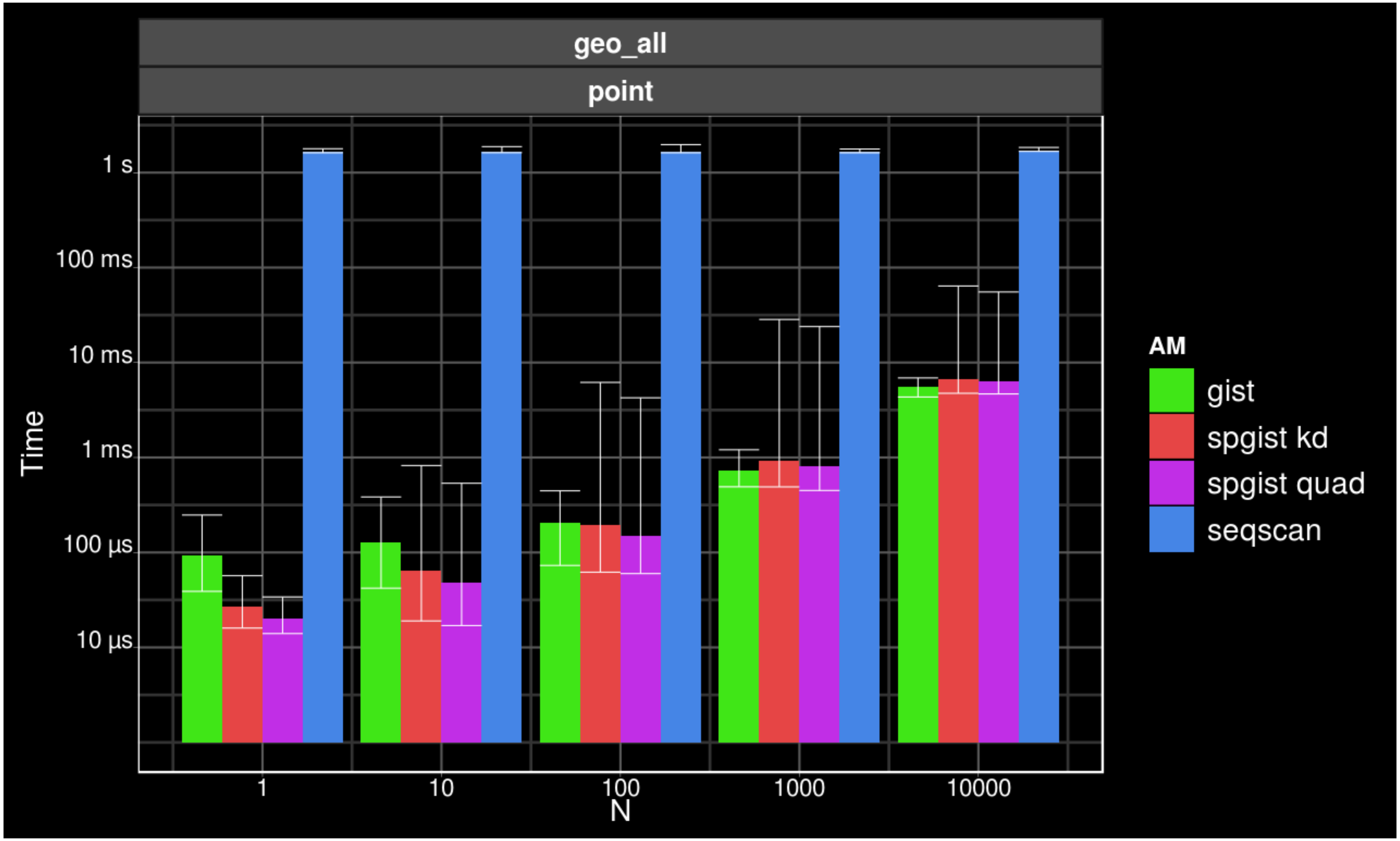
```
SELECT *
FROM knn_test
ORDER BY p <-> point(:x,:y) LIMIT :n;
```

	GiST		SP-GiST	
n	time, ms	buffers	time, ms	buffers
10	0,12	14	0,07	18
100	0,27	110	0,2	118
1000	1,58	1231	1,51	1264

KNN-SPGiST (committed)

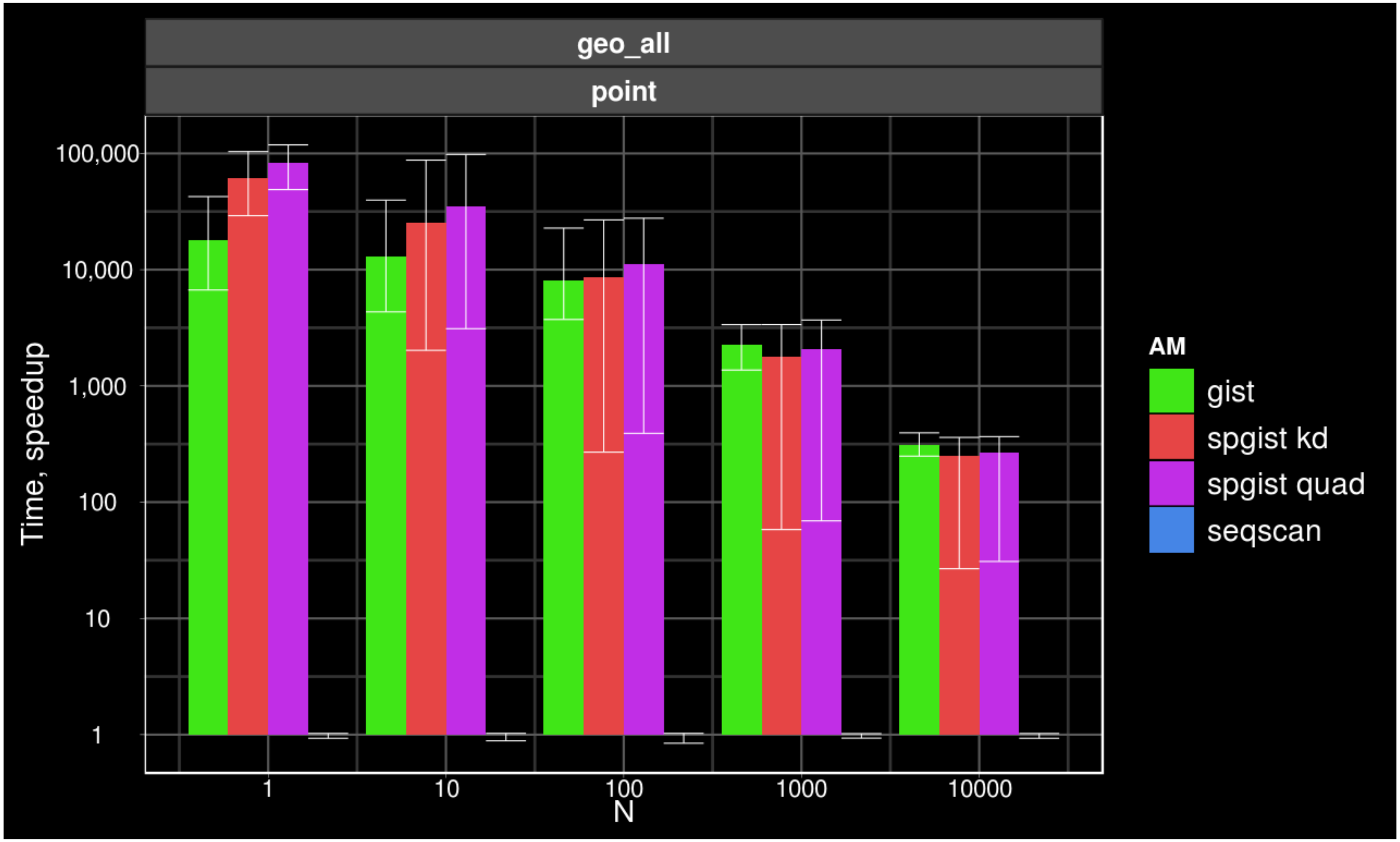
7240858 points (geonames)

SELECT point, point <-> ? FROM geo_all ORDER BY 2 LIMIT ?
KD-tree, Quad-tree



KNN-SPGiST (committed)

KNN Speedup



KNN B-tree (in-progress)

```
SELECT * FROM events
ORDER BY date <-> '2000-01-01'::date ASC
LIMIT 100;
```

	KNN B-tree		btree_gist		union		seq scan	
k	time, ms	buffers	time, ms	buffers	time, ms	buffers	time, ms	buffers
1	0.041	4	0.079	4	0.060	8	41.1	1824
10	0.048	7	0.091	9	0.097	17	41.8	1824
100	0.107	47	0.192	52	0.342	104	42.3	1824
1000	0.735	573	0.913	650	2.970	1160	43.5	1824
10000	5.070	5622	6.240	6760	36.300	11031	54.1	1824
100000	49.600	51608	61.900	64194	295.100	94980	115.0	1824

Covering GiST

- Include non-indexed columns into index to greatly improve Index-only scan (index should contains all columns from query)
  - Index is smaller than composite index
  - No need opclass for column

- PG11: INCLUDE for B-tree
  One index for UNIQUE/PRIMARY and INCLUDE to use Index-only scan

```
CREATE TABLE foo (id int, col1 text, col2 text,  primary key (id) include (col1,col2));
```

- PG12: INCLUDE for GiST

```
CREATE INDEX ON mowboxes USING  gist(bounds) INCLUDING (ip);
```

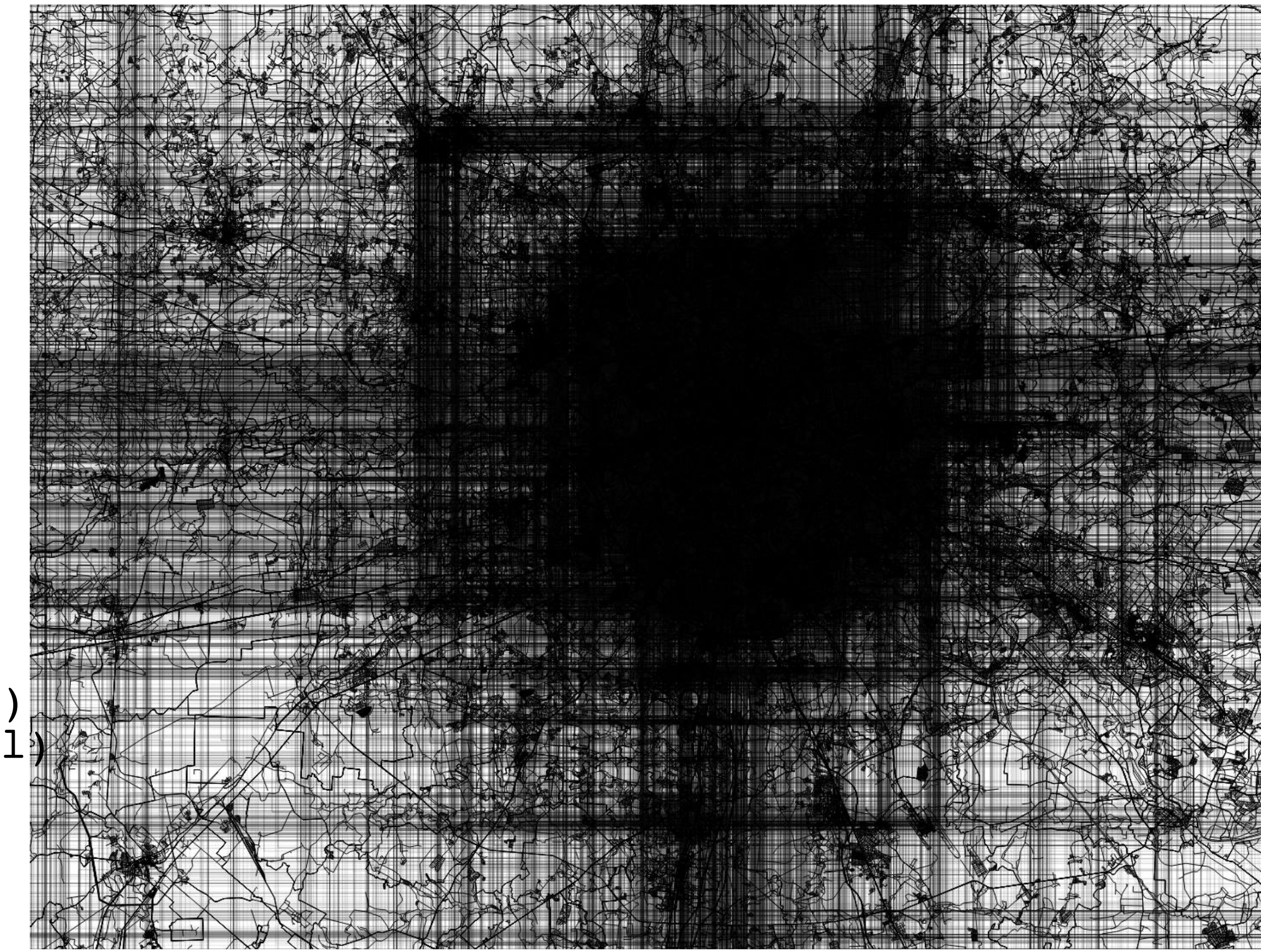
Covering GiST

Test data — 7803499 boxes with additional columns

```
\d mowboxes

  Column    |   Type
----------+---------
 Ip         | cidr
 num        | integer
 center     | point
 bounds     | box
 Tsbounds   | tsrange

Indexes:
gist (bounds)
gist (bounds,ip)
gist (bounds)INCLUDE(ip)
gist (bounds)INCLUDE(all)
```



SELECT ip,bounds FROM mowboxes WHERE bounds @> some::point

Covering GiST

Test data — 7803499 boxes with additional columns

```
\d mowboxes

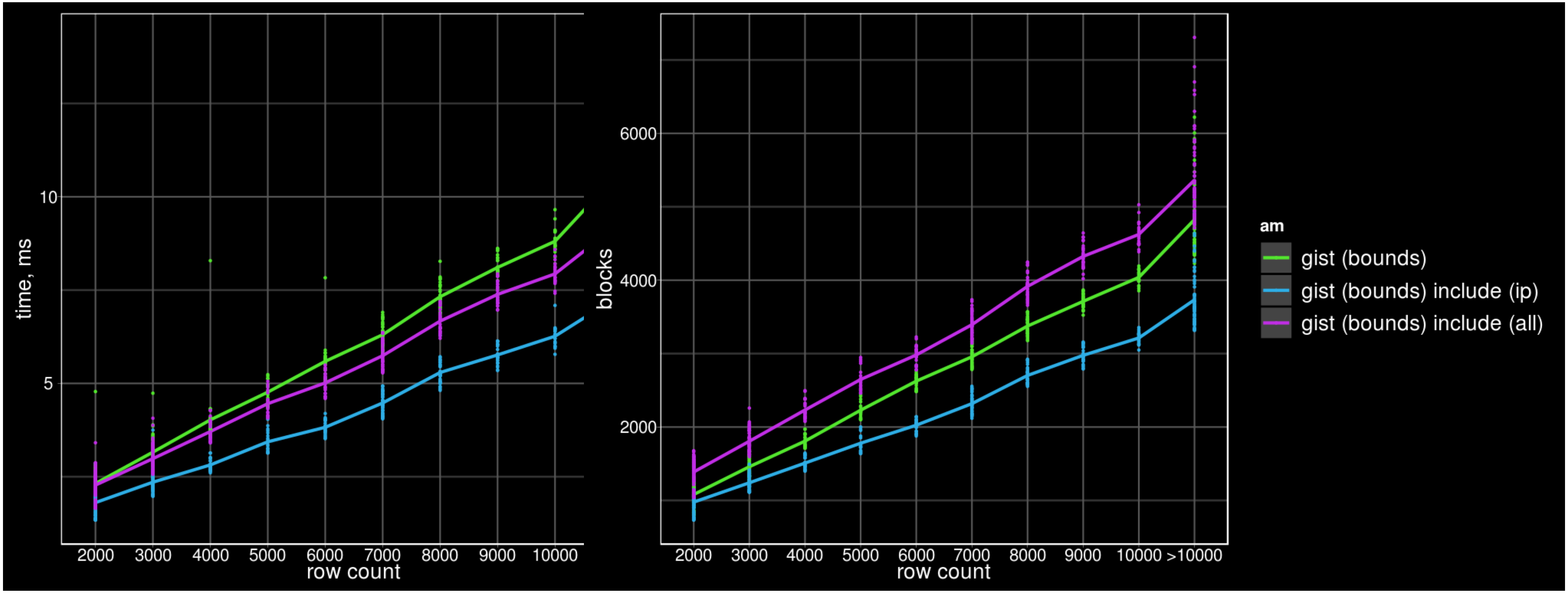
  Column   |  Type
----------+---------
 Ip        | cidr
 num       | integer
 center    | point
 bounds    | box
 Tsbounds  | tsrange

Indexes:
gist (bounds)                  665 MB
gist (bounds,ip)               876 MB
gist (bounds)INCLUDE(ip)       788 MB
gist (bounds)INCLUDE(all)     1498 MB
```

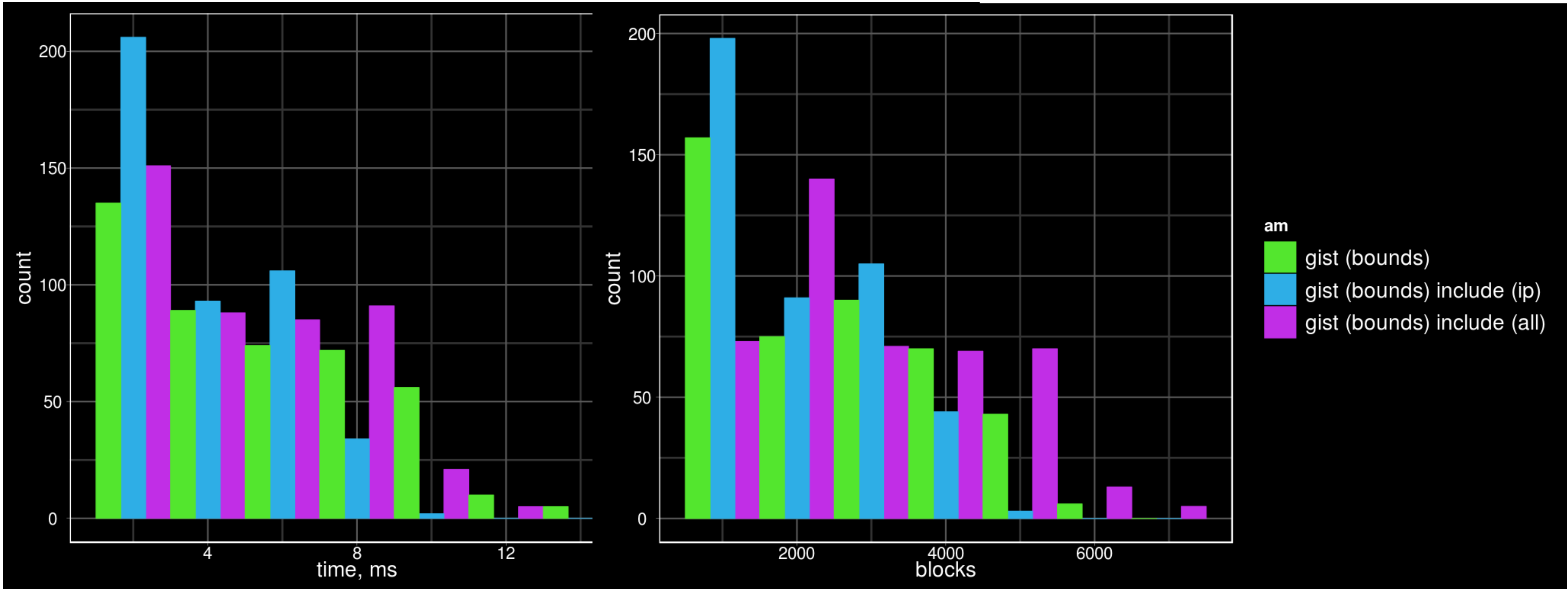
TEST QUERY (POINTs from (37.0, 55.0) - (47.5, 65.0) , step 0.5):

SELECT ip,bounds FROM mowboxes WHERE bounds @> POINT::point

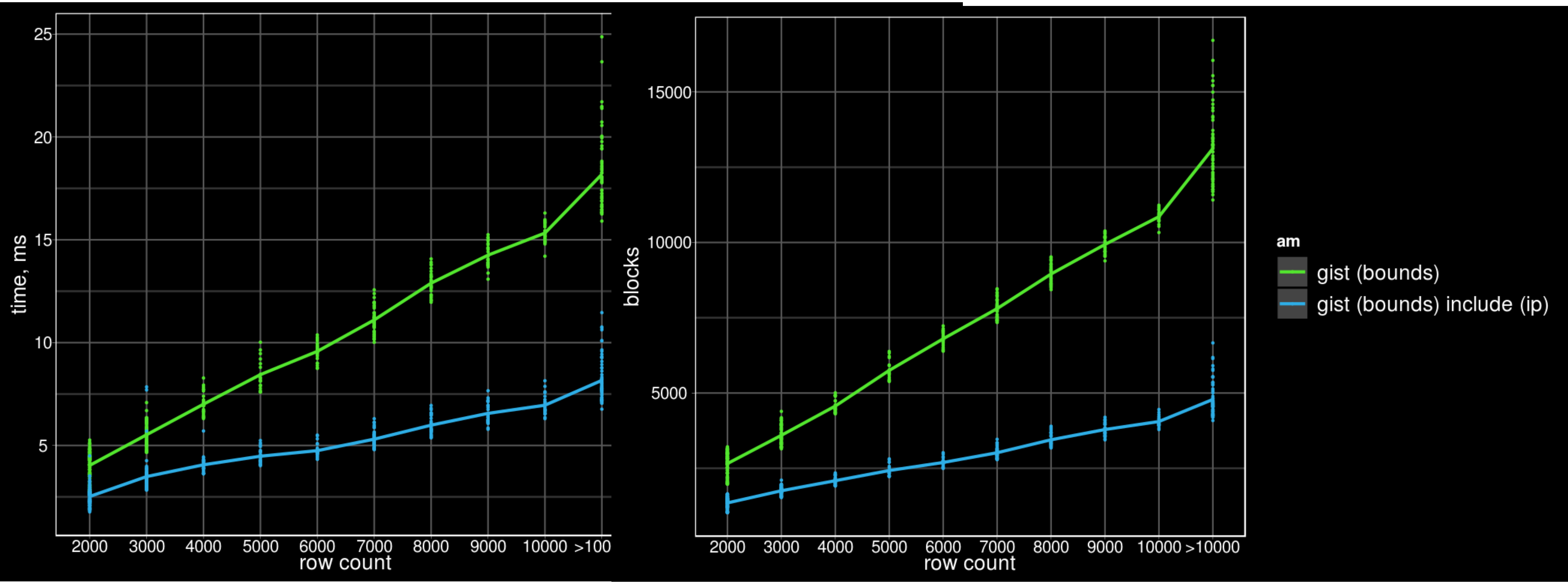
Covering GiST



Covering GiST



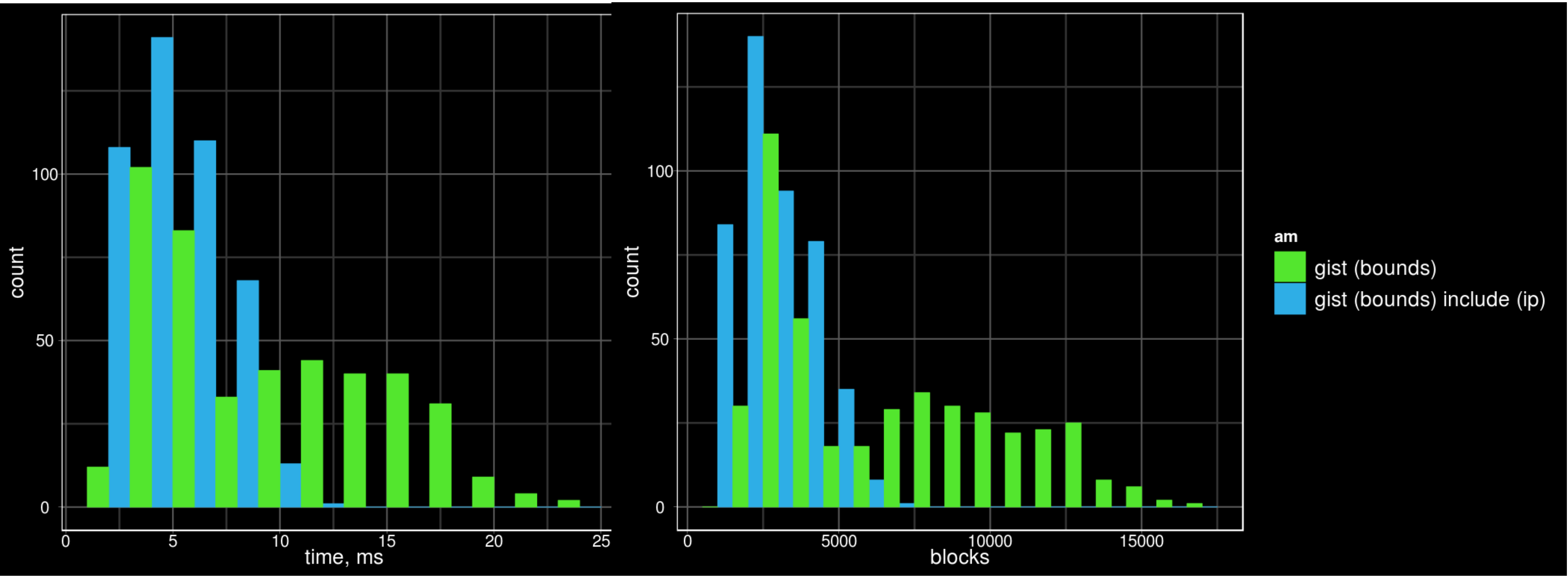
Covering GiST (randomize)



Randomize table:

CREATE TABLE mowboxes_rnd AS SELECT * FROM mowboxes ORDER BY random();

Covering GiST (randomize)



Randomize table:

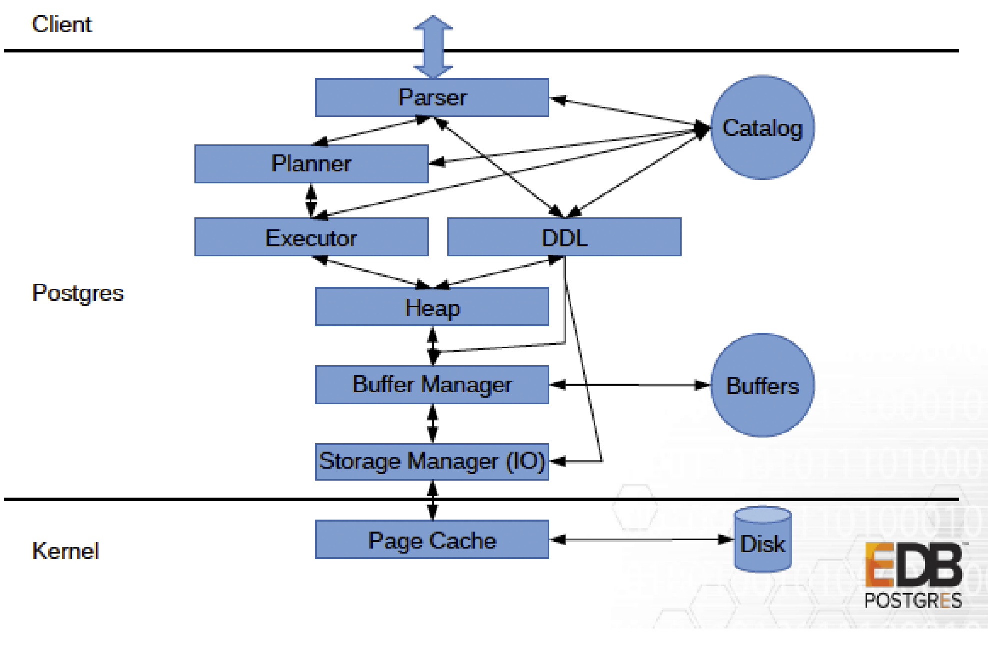
CREATE TABLE mowboxes_rnd AS SELECT * FROM mowboxes ORDER BY random();

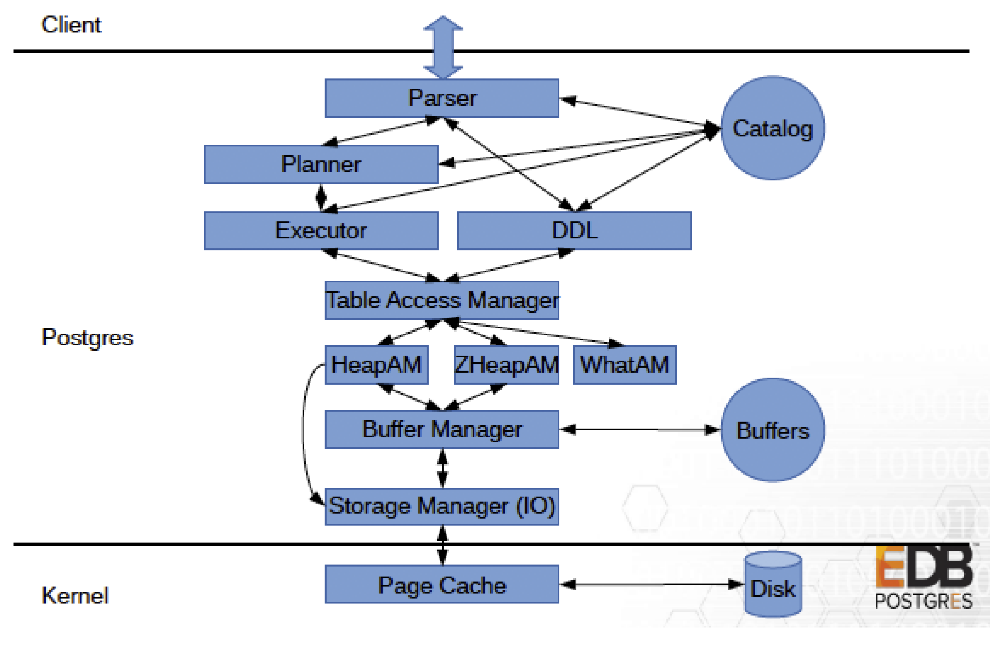
Covering GiST

- Covering GiST improves utility and performance of index-only scan

Pluggable storage

- Better Postgres extensibility
  - Storage is about tables/mat.views
  - Replace hardcoded *heap* by Table Access Manager
  - Several Table AMs coexists, could be added online
  - Examples: columnar, append-only, ZHeap, in-





Andres Freund, http://anarazel.de/talks/2018-10-25-pgconfeu-pluggable-storage/pluggable.pdf

Pluggable storage

- Better Postgres extensibility
  - Table access method
    CREATE ACCESS METHOD ... TYPE TABLE

PG 11

```
select amname, amtype from pg_am;
 amname | amtype
--------+--------
 btree  | i
 hash   | i
 gist   | i
 gin    | i
 spgist | i
 brin   | i
(6 rows)
```

PG 12

```
select amname, amtype from pg_am;
 amname | amtype
--------+--------
 heap   | t
 btree  | i
 hash   | i
 gist   | i
 gin    | i
 spgist | i
 brin   | i
(7 rows)
```

Pluggable storage

- Better Postgres extensibility
  - CREATE EXTENSION my_storage;
  - CREATE TABLE ... USING my_storage;
  - SET default_table_access_method = 'my_storage';

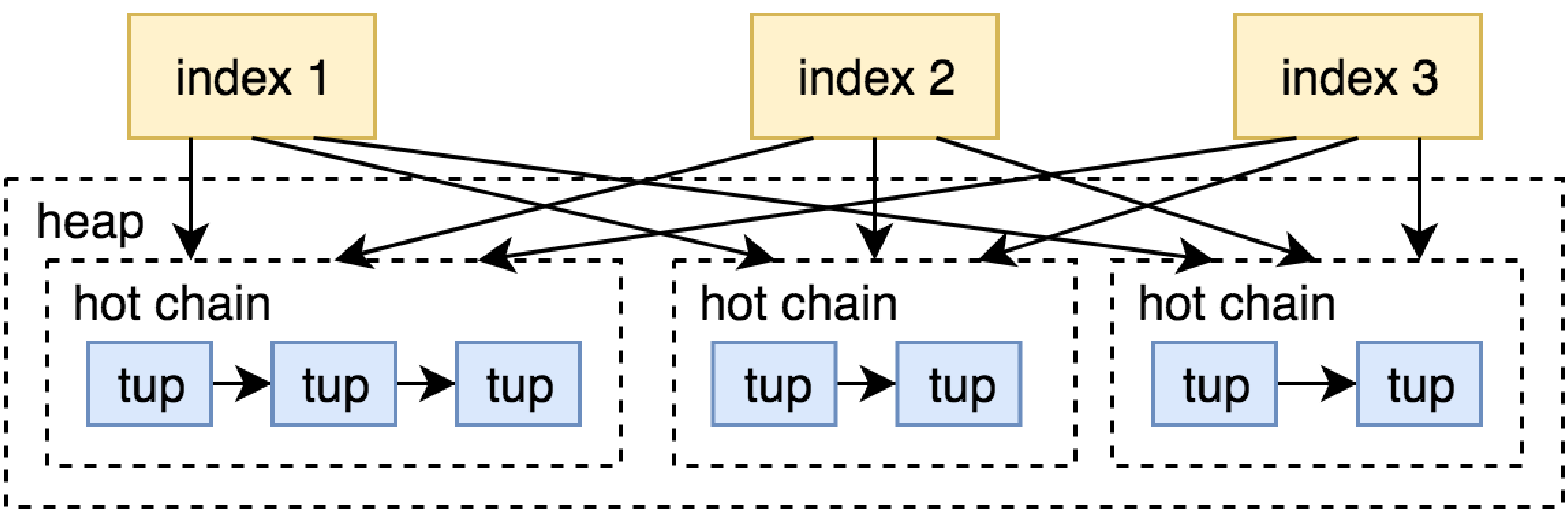
```
 =# CREATE TABLE bar() USING HEAP;
CREATE TABLE
=# show default_table_access_method;
 default_table_access_method
-----------------------------
 heap
(1 row)
```

Pluggable storage (in-progress)

- Support for INSERT/UPDATE/DELETE, triggers etc.
- Support for custom maintenance (own vacuum).
- Support for table rewrite.
- Support for custom tuple format.
- Support for custom tuple storage.
- Index-heap relationship must be the same. Only HOT-like update OR insertion to EVERY index.
- Row must be identified by 6-byte TID.
- System catalog must be heap.

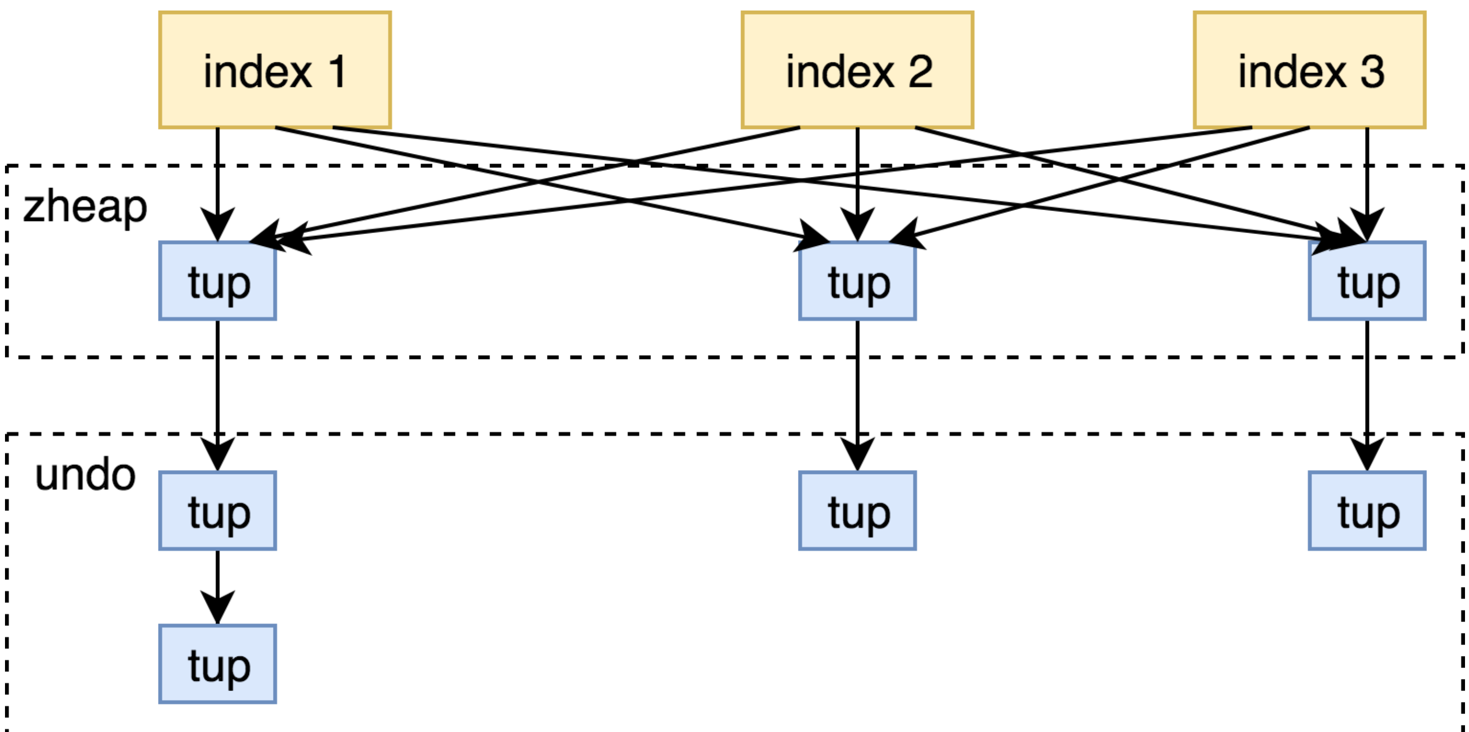
ZHeap (in-progress)

- MVCC implementation:
  - Oracle, MySQL, SQL Server: old versions are in other place
  - MVCC in Postgres: all row versions are in table
    - Table bloat, write amplification



ZHeap (in-progress)

- ZHeap — new storage for PostgreSQL with UNDO (No Vacuum storage)
  - The old versions of rows are in undo log
  - Reverse all changes made by aborted transactions



ZHeap (in-progress)

- ZHeap — new storage for PostgreSQL with UNDO
  - In-place updates (when possible) — less bloat
    - But, In-place update don“t need an extra space for new tuple on page as HOT, only if new tuple is wider.
    - In-place update like a HOT update (can“t modify any indexed columns)
  - Reclame space after transaction (committed or aborted)
  - Avoid non-modification data writes, like hint-bits
  - Shorter tuple header (no xmin,xmax, cmin,cmax)
    - UNDO log contains most of data for MVCC
    - Zheap is smaller on disk

SQL/Foundation recognizes JSON after 8 years

jsonpath (committed)

Jsonpath provides an ability to operate (in standard specified way) with json structure at SQL-language level

- `Dot notation – $.a.b.c`
  `$ -  the current context element`
- `Array - [*]`
- `Filter ? - $.a.b.c ? (@.x > 10)`
  `@ - current context in filter expression`
- `Methods - $.a.b.c.x.type()`
  `type(), size(), double(), ceiling(), floor(), abs(), datetime(), keyvalue()`

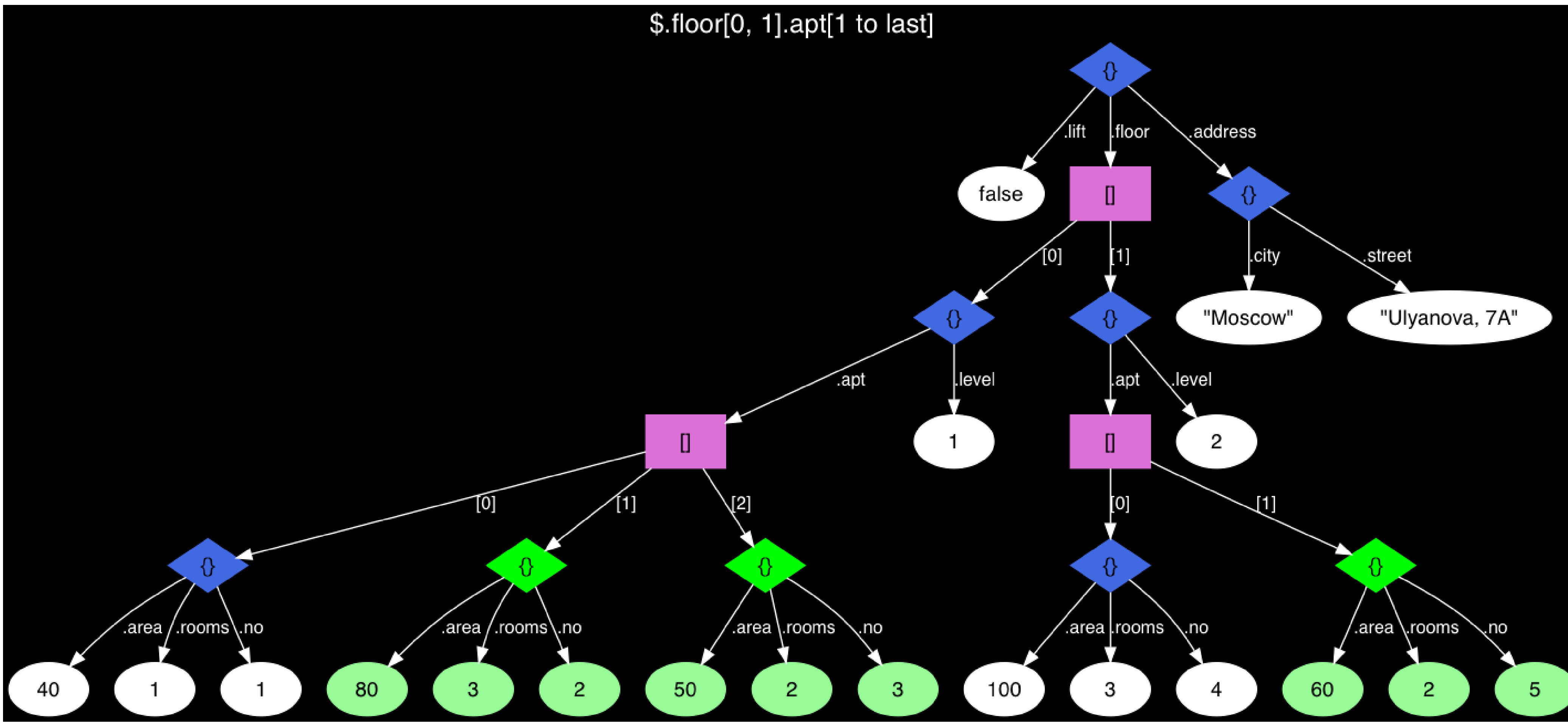
```
'$.floor[*].apt[*] ? (@.area > 40 && @.area < 90)'
```

Why JSON path is a type ?

- Standard permits only string literals in JSON path specification.
- WHY a data type ?
- To accelerate JSON path queries using existing indexes for jsonb we need boolean operators for json[b] and jsonpath.
- Implementation as a type is much easier than integration of JSON path processing with executor (complication of grammar and executor).
- In simple cases, expressions with operators can be more concise than with SQL/JSON functions.
- It is Postgres-way to use operators with custom query types (tsquery for FTS, lquery for ltree, jsquery for jsonb,...)

```
{
        "address": {
                "city": "Moscow",
                "street": "Ulyanova, 7A"
        },
        "lift": false,
        "floor": [
                {
                        "level": 1,
                        "apt": [
                                {"no": 1, "area": 40, "rooms": 1},
                                {"no": 2, "area": 80, "rooms": 3},
                                {"no": 3, "area": 50, "rooms": 2}
                        ]
                },
                {
                        "level": 2,
                        "apt": [
                                {"no": 4, "area": 100, "rooms": 3},
                                {"no": 5, "area": 60, "rooms": 2}
                        ]
                }
        ]
}
```

$.floor[0,1].apt[1 to last]

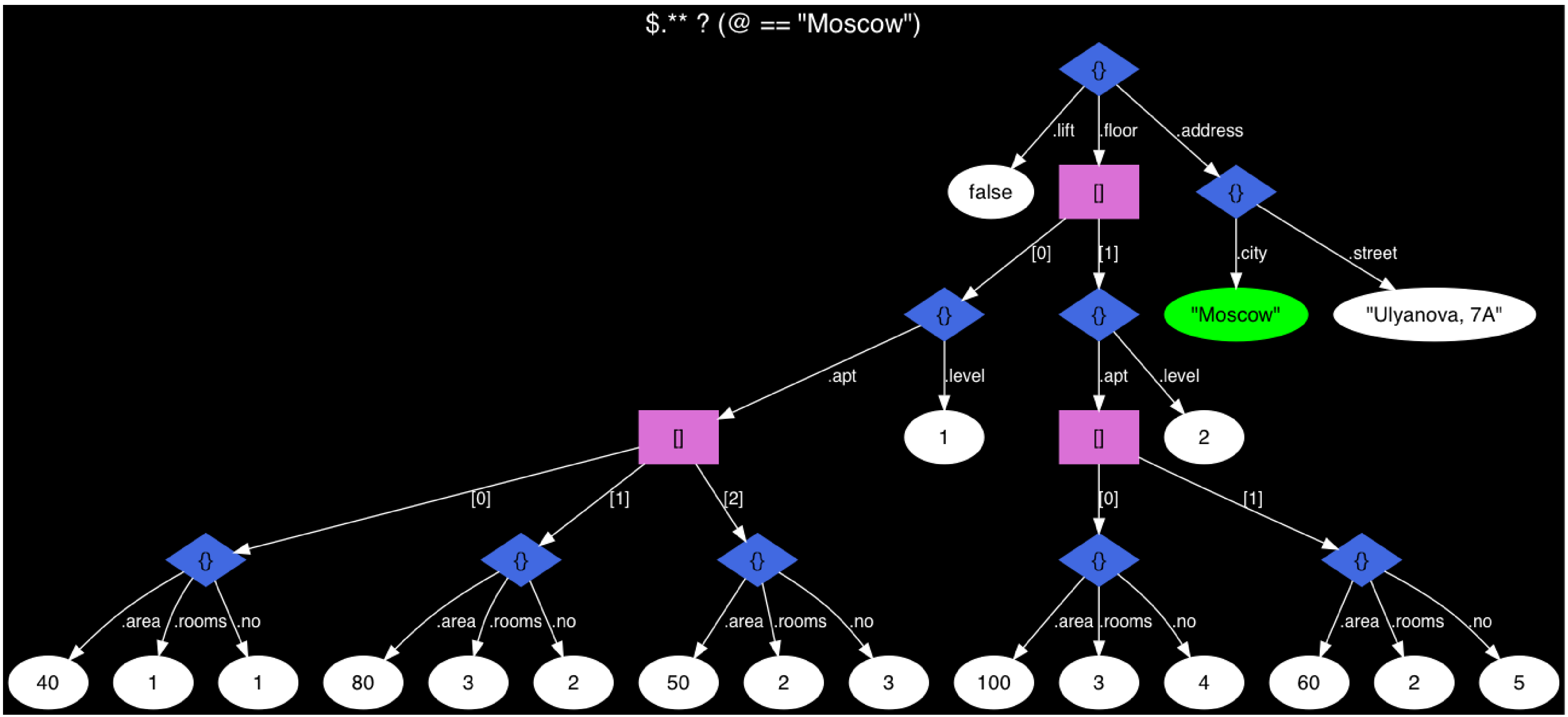


$.floor[0, 1].apt[1 to last]

```
SELECT jsonb_path_query_array(js, '$.floor[0, 1].apt[1 to last]')
FROM house;
```

```
SELECT jsonb_agg(apt)
FROM (SELECT apt->generate_series(1, jsonb_array_length(apt) - 1)
FROM (SELECT js->'floor'->unnest(array[0, 1])->'apt'
FROM house) apts(apt)) apts(apt);
```

Extension: wildcard search



$.** ? (@ == "Moscow")

```
SELECT jsonb_path_exists(js, '$.** ? (@ == "Moscow")') FROM house;
```

```
WITH RECURSIVE t(value) AS
(SELECT * FROM house
 UNION ALL
    ( SELECT
        COALESCE(kv.value, e.value) AS value
      FROM
        t
      LEFT JOIN LATERAL jsonb_each(
        CASE WHEN jsonb_typeof(t.value) = 'object' THEN t.value ELSE NULL END
      ) kv ON true
      LEFT JOIN LATERAL jsonb_array_elements(
        CASE WHEN jsonb_typeof(t.value) = 'array' THEN t.value ELSE NULL END
      ) e ON true
      WHERE
        kv.value IS NOT NULL OR e.value IS NOT NULL)
)
SELECT EXISTS (SELECT 1 FROM t WHERE value = '"Moscow"');
```

jsonpath (committed)

The jsonpath functions for jsonb:

- **`jsonb_path_exists()`** `=> boolean`
  Test whether a JSON path expression returns any SQL/JSON items (operator @?).
- **`jsonb_path_match()`** `=> boolean`
  Get the result of a JSON path predicate (operator @@).
- **`jsonb_path_query()`** `=> setof jsonb`
  Extract a sequence of SQL/JSON items from a JSON value.
- **`jsonb_path_query_array()`** `=> jsonb`

  Extract a sequence of SQL/JSON items wrapped into JSON array.
- **`jsonb_path_query_first()`** `=> jsonb`

  Extract the first SQL/JSON item from a JSON value.

jsonpath (committed)

All `jsonb_path_xxx()` functions have the same signature:

```
jsonb_path_xxx(
    js jsonb,
    jsp jsonpath,
    vars jsonb DEFAULT '{}',
    silent boolean DEFAULT false
)
```

- "vars" is a jsonb object used for passing jsonpath variables:

```
SELECT jsonb_path_query_array('[1,2,3,4,5]', '$[*] ? (@ > $x)',
                              vars => '{"x": 2}');
 jsonb_path_query_array
------------------------
 [3, 4, 5]
```

- "silent" flag enables suppression of errors:

```
SELECT jsonb_path_query('[]', 'strict $.a');
ERROR:  SQL/JSON member not found
DETAIL:  jsonpath member accessor can only be applied to an object

SELECT jsonb_path_query('[]', 'strict $.a', silent => true);
 jsonb_path_query
------------------
(0 rows)
```

jsonpath (committed)

Jsonpath function examples:

- ```
  jsonb_path_exists('{"a": 1}', '$.a') => true
  jsonb_path_exists('{"a": 1}', '$.b') => false
  ```
- ```
  jsonb_path_match('{"a": 1}', '$.a == 1') => true
  jsonb_path_match('{"a": 1}', '$.a >= 2') => false
  ```
- ```
  jsonb_path_query('{"a": [1,2,3,4,5]}',
                   '$.a[*] ? (@ > 2)') => 3, 4, 5 (3 rows)
  jsonb_path_query('{"a": [1,2,3,4,5]}',
                   '$.a[*] ? (@ > 5)') =>  (0 rows)
  ```
- ```
  jsonb_path_query_array('{"a": [1,2,3,4,5]}',
                         '$.a[*] ? (@ > 2)') => [3, 4, 5]
  jsonb_path_query_array('{"a": [1,2,3,4,5]}',
                         '$.a[*] ? (@ > 5)') => []
  ```
- ```
  jsonb_path_query_first('{"a": [1,2,3,4,5]}',
                         '$.a[*] ? (@ > 2)') => 3
  jsonb_path_query_first('{"a": [1,2,3,4,5]}',
                         '$.a[*] ? (@ > 5)') => NULL
  ```
  ```

jsonpath (committed)

Boolean jsonpath operators for jsonb:

- `jsonb @? jsonpath` (exists)
  Test whether a JSON path expression returns any SQL/JSON items.

  ```
  jsonb '[1,2,3]' @? '$[*] ? (@ == 3)'   => true
  ```

- `jsonb @@ jsonpath` (match)
  Get the result of a JSON path predicate.

  ```
  jsonb '[1,2,3]' @@ '$[*] == 3'   => true
  ```

- Operators are interchangeable:

  ```
  js @? '$.a'        <=>  js @@ 'exists($.a)'
  js @@ '$.a == 1' <=>  js @? '$ ? ($.a == 1)'
  ```

jsonpath (committed)

Boolena jsonpath operators are supported by GIN `jsonb_ops` and `jsonb_path_ops`:

```
CREATE INDEX ON house USING gin (js);

EXPLAIN (COSTS OFF)
SELECT * FROM house
WHERE js @? '$.floor[*].apt[*] ? (@.rooms == 3)'
```

```
                              QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on house
   Recheck Cond: (js @? '$."floor"[*]."apt"[*]?(@."rooms" == 3)'::jsonpath)
   ->  Bitmap Index Scan on house_js_idx
         Index Cond: (js @? '$."floor"[*]."apt"[*]?(@."rooms" == 3)'::jsonpath)
(4 rows)
```

jsonpath (committed)

Exists @? and match @~ operators can be speeded up by GIN index using built-in `jsonb_ops` or `jsonb_path_ops`.

```
EXPLAIN (ANALYZE)
SELECT COUNT(*) FROM bookmarks
WHERE jb @~ '$.tags[*].term == "NYC"';
```

```
                                         QUERY PLAN

-----------------------------------------------------------------------------------
Aggregate  (cost=4732.10..4732.11 rows=1 width=8) (actual time=1.238..1.238 rows=1 
loops=1)
   ->  Bitmap Heap Scan on bookmarks  (cost=33.71..4728.97 rows=1253 width=0) (actual 
time=0.128..1.196 rows=285 loops=1)
         Recheck Cond: (jb @~ '($."tags"[*]."term" == "NYC")'::jsonpath)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on bookmarks_jb_idx  (cost=0.00..33.40 rows=1253 width=0) 
(actual time=0.071..0.071 rows=285 loops=1)
               Index Cond: (jb @~ '($."tags"[*]."term" == "NYC")'::jsonpath)
 Planning time: 0.080 ms
 Execution time: 1.283 ms
(8 rows)
```

jsonpath (committed)

- `.datetime()` item method will not be implemented in PG12:

```
-- behavior required by standard
SELECT jsonb_path_query('"13.03.2019"', '$.datetime("DD.MM.YYYY")');
 jsonb_path_query
------------------
 "2019-03-13"
(1 row)

-- behavior of PG12
SELECT jsonb_path_query('"13.03.2019"', '$.datetime("DD.MM.YYYY")');
ERROR:  bad jsonpath representation
```

- Arithmetic errors in filters may be not suppressed:

```
-- behavior required by standard
SELECT jsonb_path_query('[1,0,2]', '$[*] ? (1 / @ >= 1)');
jsonb_path_query
------------------
 1
(1 row)

-- possible behavior of PG12
SELECT jsonb_path_query('[1,0,2]', '$[*] ? (1 / @ >= 1)');
ERROR:  division by zero
```

SQL/JSON (доп.материалы)

- Презентация по SQL/JSON
  http://www.sai.msu.su/~megera/postgres/talks/sqljson-china-2018.pdf

- Введение в SQL/JSON
  https://github.com/obartunov/sqljsondoc/blob/master/README.jsonpath.md

- Посты про SQL/JSON
  https://obartunov.livejournal.com/tag/sqljson

Generalized expression syntax for partition bounds (committed)

```
CREATE TABLE part (ts timestamp)
PARTITION BY RANGE(ts);
CREATE TABLE part1
PARTITION OF part FOR VALUES
FROM ('2018-01-01')
TO (current_timestamp + '1 day');
```

Support for expressions in partition bounds!

Run-time partition pruning for MergeAppend (committed)

```
# explain analyze  select * from news
  where category = (select category from hot_category)
  order by ts limit 10;

 Limit  (cost=36.79..37.26 rows=10 width=12) (actual time=0.0
   InitPlan 1 (returns $0)
     ->  Seq Scan on hot_category  (cost=0.00..35.50 rows=255
   ->  Merge Append  (cost=1.29..46833.10 rows=1000000 width=
         Sort Key: news_cat1.ts
         ->  Index Scan using news_cat1_ts_idx on news_cat1
             (cost=0.42..11302.75 rows=333333 width=12)
             (actual time=0.016..0.021 rows=10 loops=1)
               Filter: (category = $0)
         ->  Index Scan using news_cat2_ts_idx on news_cat2
             (cost=0.42..11302.77 rows=333334 width=12)
             (never executed)
               Filter: (category = $0)
         ->  Index Scan using news_cat3_ts_idx on news_cat3
             (cost=0.42..11302.75 rows=333333 width=12)
             (never executed)
               Filter: (category = $0)
```

Reduce partition tuple routing overheads (committed)

Inserts into 10k partitions table:

original: 96 TPS

patched: 17729 TPS

non-partitioned: 19121 TPS

PostgreSQL version in log (committed)

```
2019-02-02 09:23:11.711 MSK [59708] LOG:  starting
PostgreSQL 12devel on x86_64-apple-darwin17.7.0, compiled
by Apple LLVM version 10.0.0 (clang-1000.11.45.5), 64-bit
2019-02-02 09:23:11.715 MSK [59708] LOG:  listening on
IPv6 address "::1", port 5434
2019-02-02 09:23:11.715 MSK [59708] LOG:  listening on
IPv6 address "fe80::1%lo0", port 5434
2019-02-02 09:23:11.715 MSK [59708] LOG:  listening on
IPv4 address "127.0.0.1", port 5434
2019-02-02 09:23:11.716 MSK [59708] LOG:  listening on
Unix socket "/tmp/.s.PGSQL.5434"


.........................................................
```

Locking B-tree leafs immediately in exclusive mode (committed)

test	original, TPS	patched, TPS
unordered inserts	409 591	412 765
ordered inserts	252 796	314 541
duplicate inserts	44 811	202 325

Improve behavior of to_timestamp() / to_date() functions (committed)

Before

```
# select to_timestamp('2019-13-01', 'YYYYMMDD');
      to_timestamp
------------------------
 2018-11-03 00:00:00+03

# select to_timestamp('2019 -01-01', 'YYYY-MM-DD');
 ERROR:  date/time field value out of range: "2019 -01-01"
```

After

```
# select to_timestamp('2019-13-01', 'YYYYMMDD');
ERROR:  date/time field value out of range: "2019-13-01"

# select to_timestamp('2019 -01-01', 'YYYY-MM-DD');
      to_timestamp
------------------------
 2019-01-01 00:00:00+03
(1 row)
```

Now well documented!

Function to promote standby servers (committed)

How to promote a standby?

- Trigger file
- pg_ctl promote
- **SELECT pg_promote();**

Step towards managing cluster in pure SQL!

Speedup of relation deletes during recovery (committed)

Relation delete or truncate:

- Causes sequential scan of shared_buffers
- Slow with large shared_buffers
- Especially bad for standby, because of single-process recovery

Now, instead of

```
DELETE tab1; DELETE tab2; … DELETE tabN;
```

it's better to do

```
BEGIN;
DELETE tab1; DELETE tab2; … DELETE tabN;
COMMIT;
```

Single pass over shared_buffers instead of N.

Less replication lag!

Use the built-in float datatypes to implement geometric types (committed)

- Check for underflow, overflow and division by zero
- Consider NaN values to be equal
- Return NULL when the distance is NaN for all closest point operators
- Favour not-NaN over NaN where it makes sense

Before

```
# select point('NaN', 'NaN') ~= point('NaN', 'NaN');
 ?column?
----------
 f
```

After

```
# select point('NaN', 'NaN') ~= point('NaN', 'NaN');
 ?column?
----------
 t
```

Add log_statement_sample_rate parameter (committed)

- Logging all the statements consumes much of resources
- Logging only long statements may distort your picture
- Sample logging is the solution!

```
log_statement_sample_rate = 1 ; log every statement
log_statement_sample_rate = 0 ; log no statements
log_statement_sample_rate = 0.5 ; log half of statement
log_statement_sample_rate = 0.1 ; log one tenth of
                                ; statement
```

Hyperbolic functions

- SQL:2016 standard introduced
  - Sinh()
  - Cosh()
  - Tanh()
  - Only float8, no numeric support
  - Log10() - alias to log()

כל מה שאתה צריך זה פוסטגרס