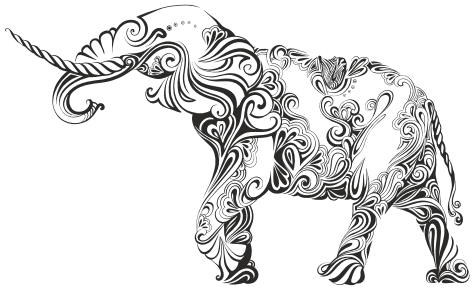
K-Nearest Neighbors Search in PostgreSQL



Oleg Bartunov, Nikita Glukhov
Postgres Professional

http://www.sai.msu.su/~megera/postgres/talks/knn-pgconf-2021.pdf

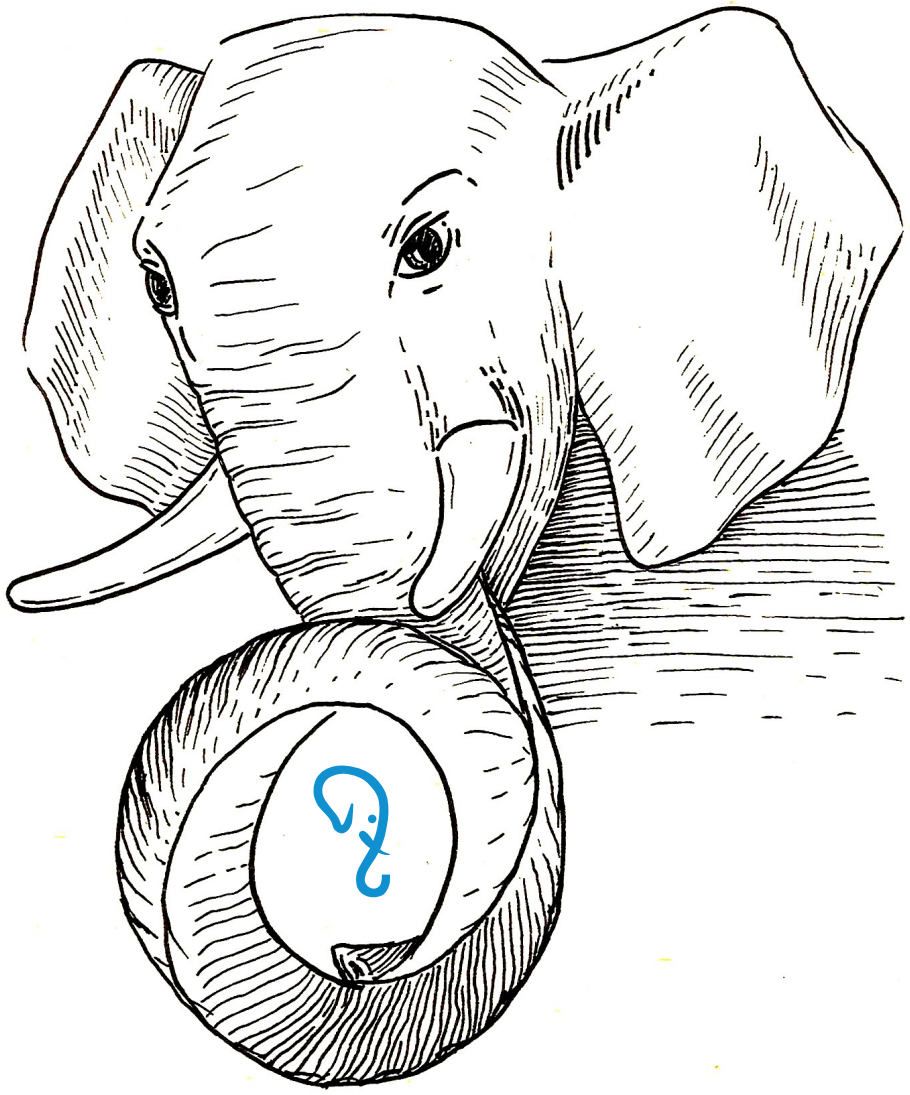
PostgresPro

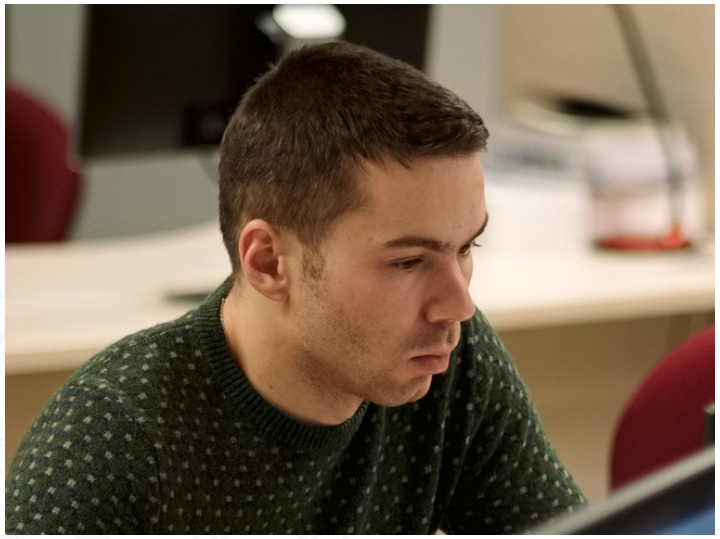
Since Postgres95

Research scientist @
Moscow University
CEO Postgres Professional
Major PostgreSQL contributor

PostgresPro

Nikita Glukhov





Senior developer @Postgres Professional
PostgreSQL contributor

Major CORE contributions:

- Jsonb improvements
- SQL/JSON (Jsonpath)
- KNN SP-GiST
- Opclass parameters

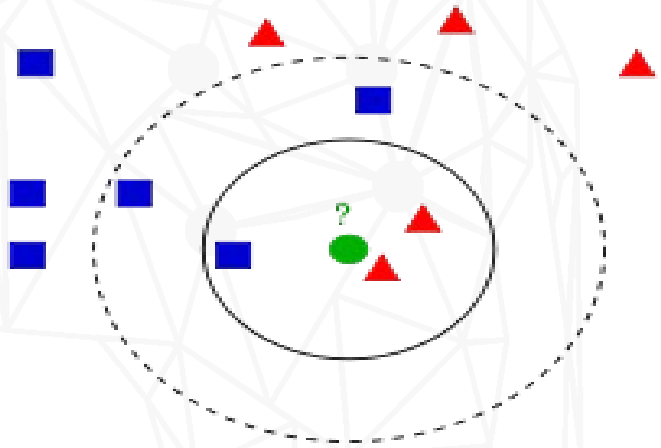
Current development:

- SQL/JSON functions
- Jsonb performance

PostgresPro

Knn-search: The problem

- What are interesting points near Royal Oak pub in Ottawa ?
- What are the closest events to the May 20, 2009 in Ottawa ?
- Similar images – feature extraction, Hamming distance
- Classification problem (major voting)
- .....
- GIS, Science (high-dimensional data)



PostgresPro

K-nearest neighbour search

- 10 closest events to the launch of Sputnik ?

```
SELECT id, date, event FROM events ORDER BY 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)
```

- Before 9.1: Slow: Index is useless (no WHERE!), 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)
```

PostgresPro

K-nearest neighbour search

- 10 closest events to the launch of Sputnik
- Before 9.1: 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)
```

- After 9.1: Amazingly FAST using index !

```
Limit (actual time=0.123..0.143 rows=10 loops=1)
   Buffers: shared hit=9
   ->  Index Scan using event_date_idx on events (actual time=0.122..0.138 rows=10 loops=1)
         Order By: (date <-> '1957-10-04'::date)
         Buffers: shared hit=9
 Planning Time: 0.105 ms
 Execution Time: 0.172 ms
(7 rows)
```

~ 300 times faster !

Index can be used for

- Filtering - WHERE expr opr value
- Ordering - ORDER BY expr [ASC|DESC]
- **K-NN - ORDER BY expr opr value [ASC]**

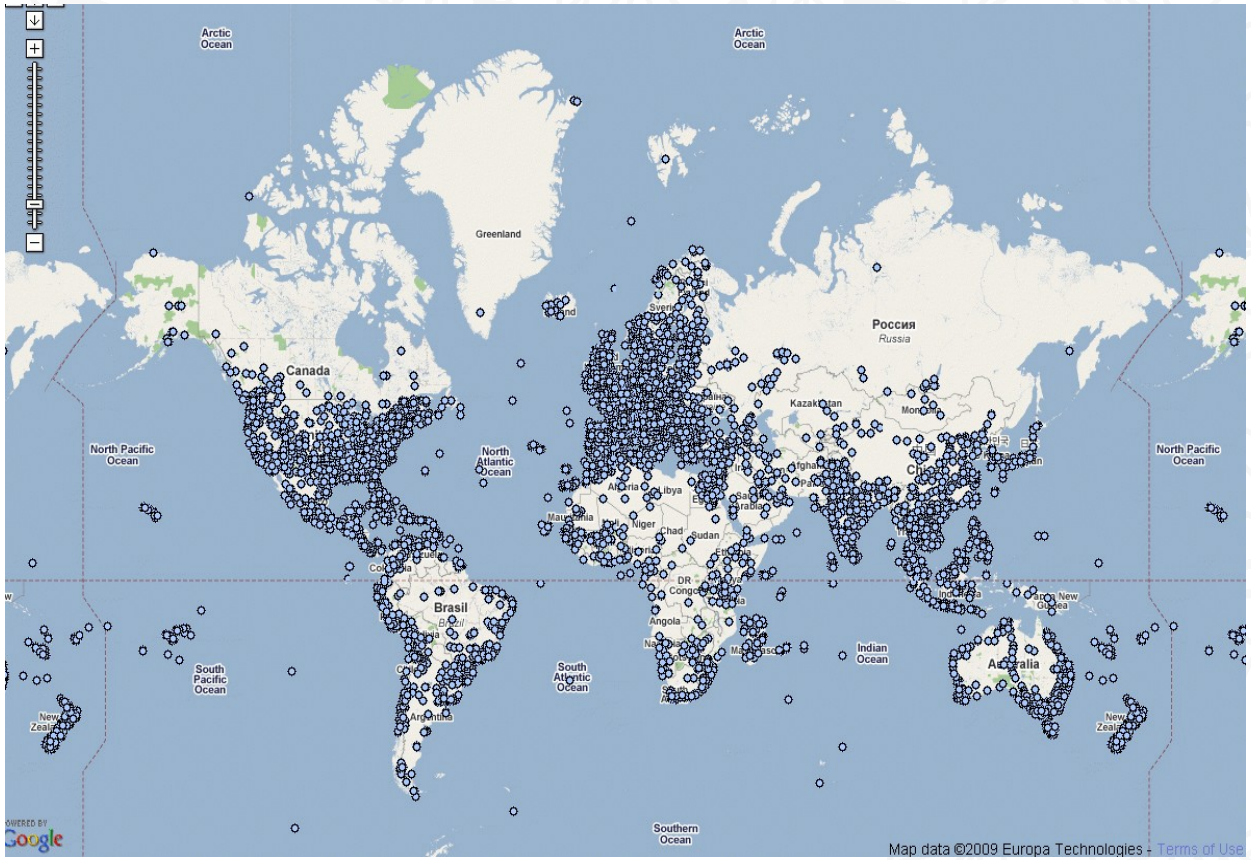
PostgresPro

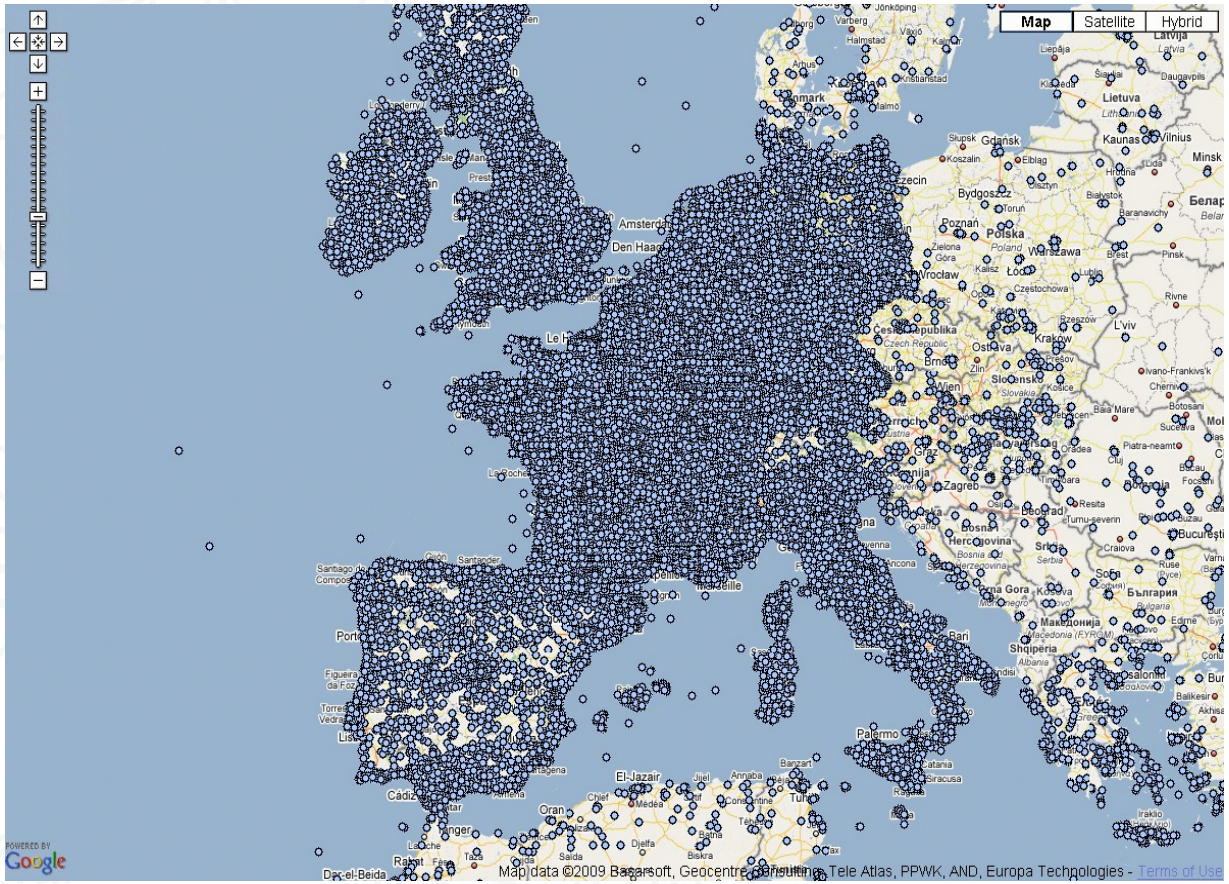
Knn-search: Existing solutions

Traditional way to speedup such query

- Use indexes - very inefficient (no search query !)
  - Scan full index
  - Full table scan, but in random order !
  - Sort full table
  - Better not to use index at all !
- Constrain data space (range search) - "Костыль"
  - 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' ?

Difficult choice for initial radius



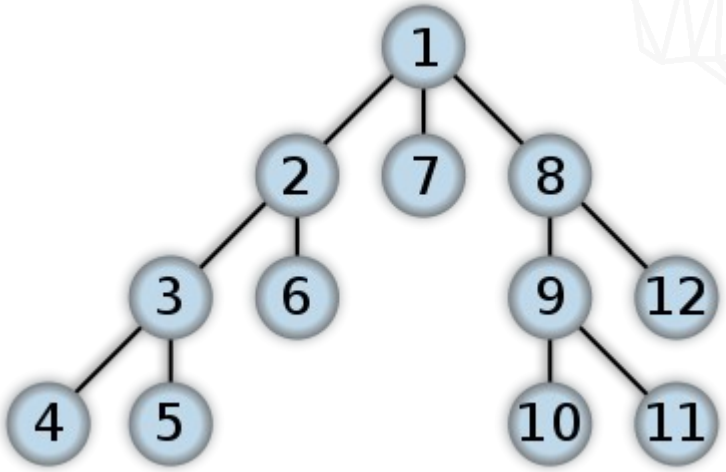


PostgresPro

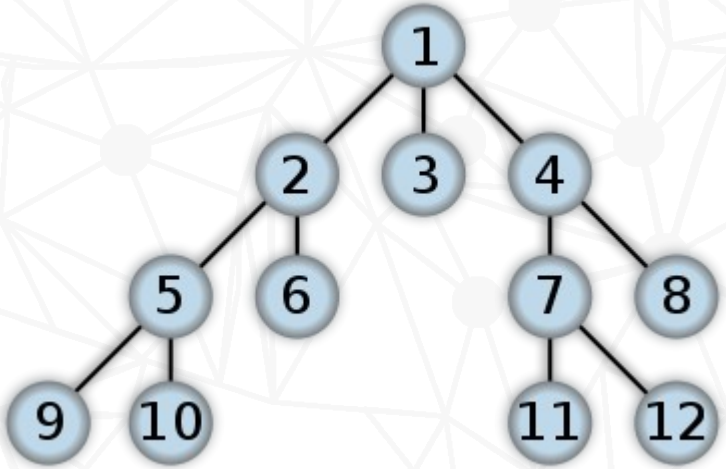
Regular Index traverse

- Index is a search tree

Depth First Search
(stack, LIFO)



Breadth First Search
(queue, FIFO)



Both strategies require WHERE clause, for K-NN it means full index scan

PostgresPro

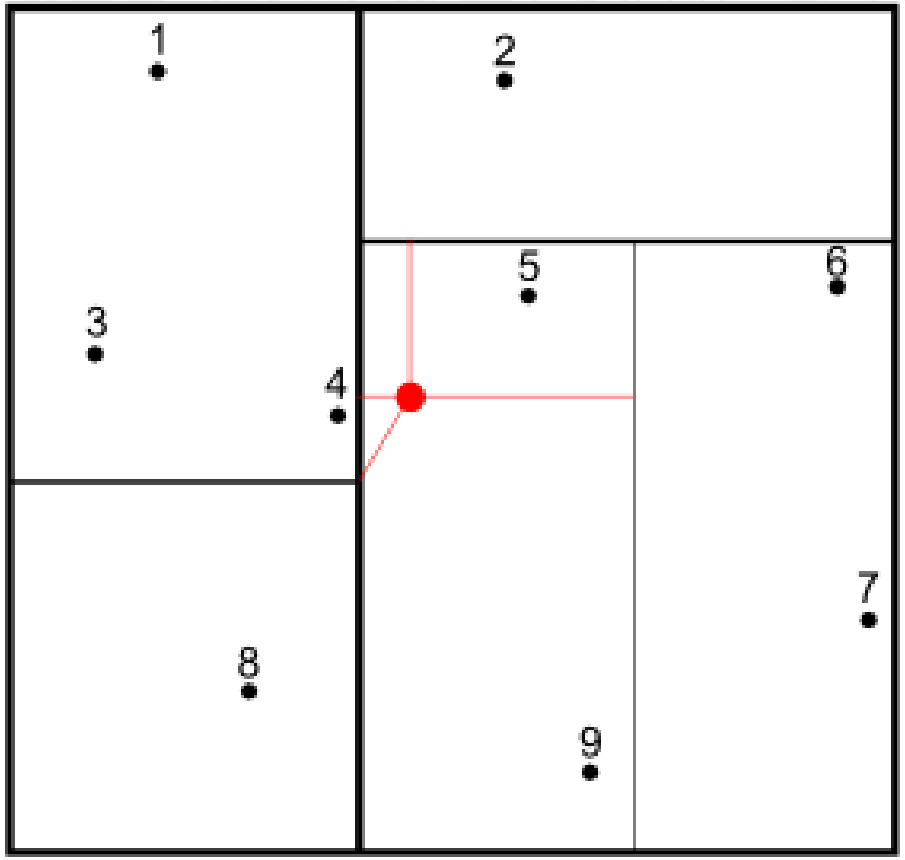
K-NN Index traverse — Best First Search

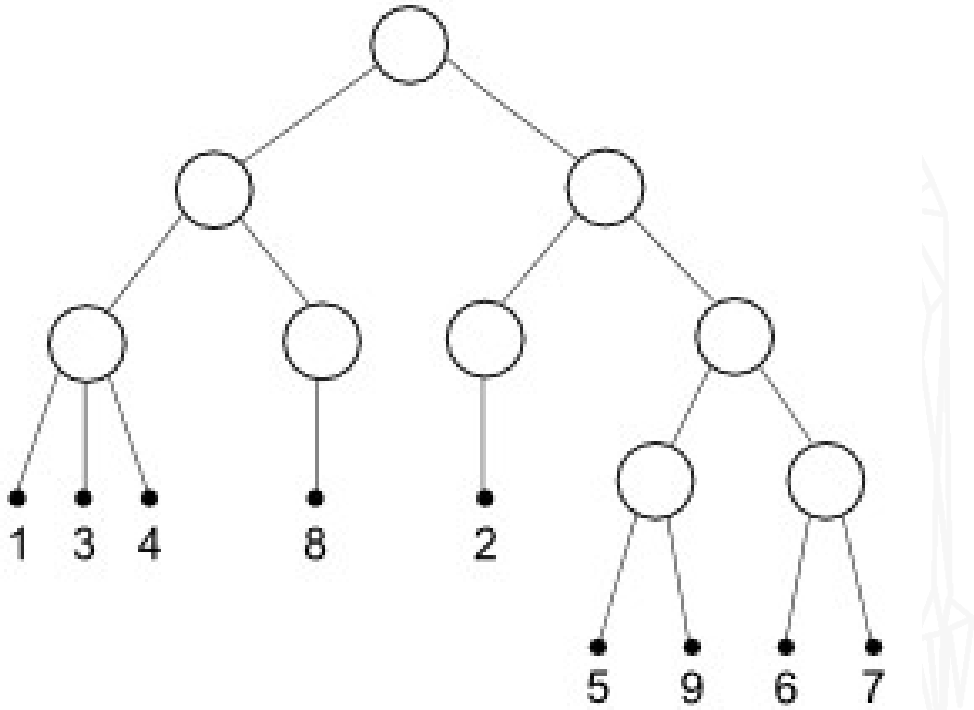
- Best First Search (PQ, priority queue). Maintain order of items in PQ according their distance from given point
  - Distance to MBR (rectangle for Rtree) for internal pages – minimum distance of all items in that MBR
  - Distance = 0 for MBR with given point
  - Distance to point for leaf pages
- Each time we extract point from PQ we output it – it is the 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 !

PostgresPro

K-NN Index traverse — Best First Search

- Simple example – non-overlapped partitioning

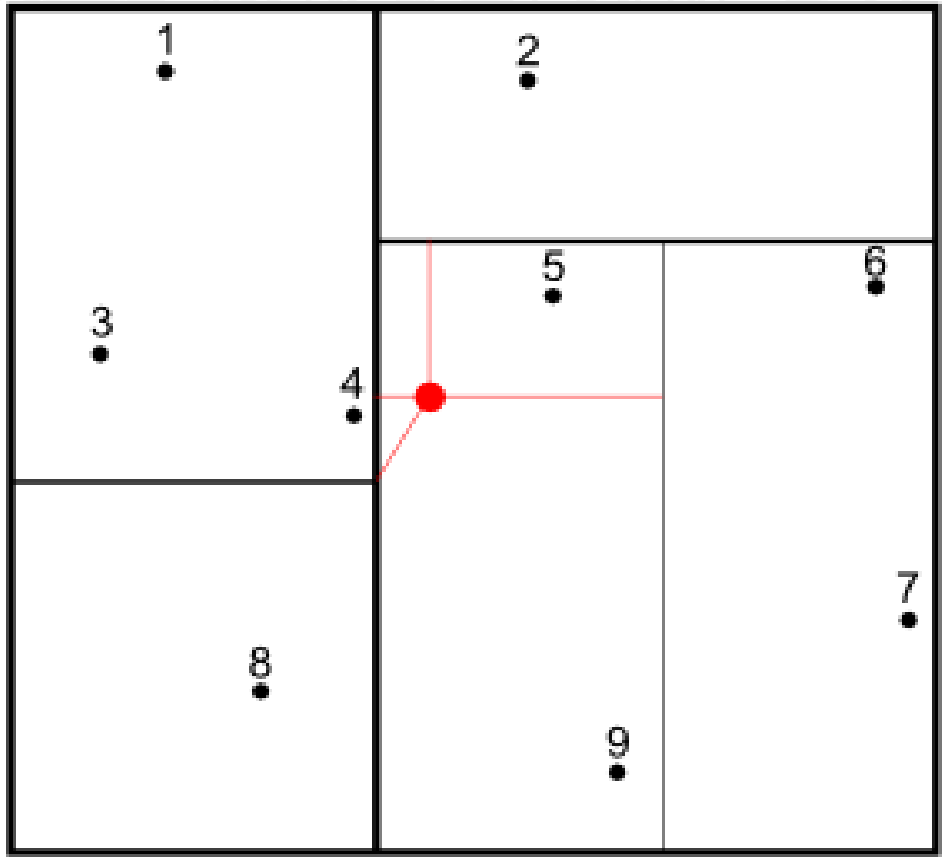




PostgresPro

K-NN Index traverse — Best First Search

- Simple example – non-overlapped partitioning



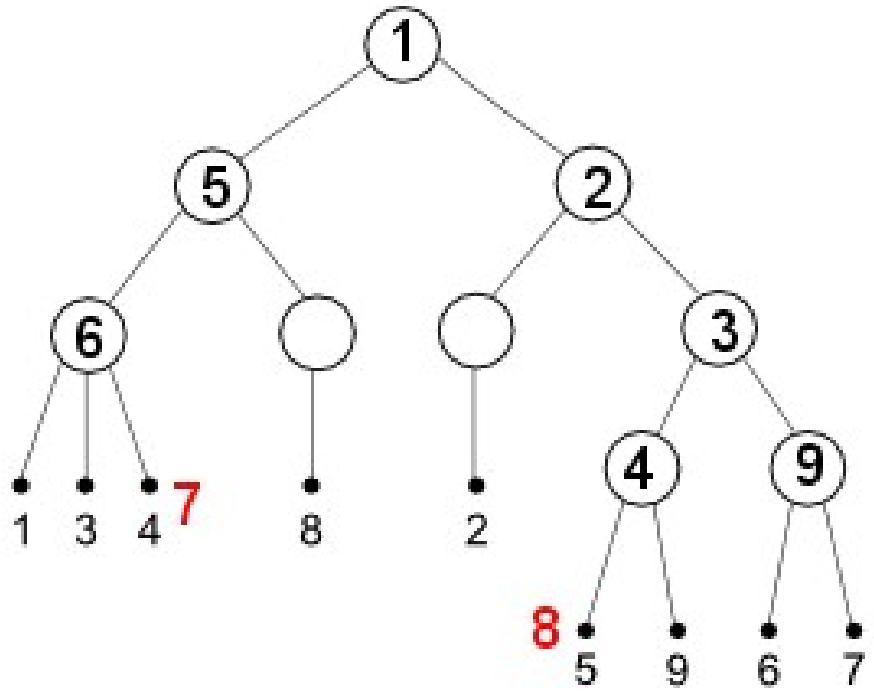
- Priority Queue
- 1: {1,2,3,4,5,6,7,8,9}
- 2: {2,5,6,7,9}, {1,3,4,8}
- 3: {5,6,7,9}, {1,3,4,8}, {2}
- 4: {5,9}, {1,3,4,8}, {2}, {6,7}
- 5: {1,3,4,8}, 5, {2}, {6,7}, 9
- 6: {1,3,4}, {8}, 5, {2}, {6,7}, 9
- 7: 4, {8}, 5, {2}, {6,7}, 3, 1, 9

  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

PostgresPro

K-NN Index traverse — Best First Search

- 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

PostgresPro

Knn-search: What we did !

- + 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
- + We want to support many data types
  - So, we need to modify GiST

PostgresPro

GiST Modification

Depth First Search

```
push Stack, Root;
While Stack {
   If p is heap {
              output p;
   else {
              children =
get_children(p);
              push Stack, children;
        }
}
```

Best First Search

```
push PQ, Root;
While PQ {
   If p is heap {
              output p;
   else {
              Children = get_children(p);
              push PQ, children;
        }
}
```

- For non-knn search all distances are zero, so PQ => Stack and BFS => DFS
- We can use only one strategy for both – normal search and knn-search !

PostgresPro

Knn-search: Performance

- SEQ (no index) – base performance
  - Sequentually read full table + Sort full table (can be very bad, sort_mem !)
- DFS — very bas !
  - 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 !

PostgresPro

Example: GiST R-Tree for Greece

- Create two tables with a subset of geo_all data belonging to Greece:
  - `CREATE TABLE geo_greece AS SELECT * FROM geo_all WHERE country_code = 'GR';`
  - `CREATE TABLE geo_greece_rnd AS SELECT * FROM geo_greece ORDER BY random();`

- Create three GiST indexes:
  - Z-order sorted which is default for PG14 (`WITH (buffering = off)` is by default)

    ```
    CREATE INDEX geo_greece_zorder_idx ON geo_greece USING gist (location);
    ```

  - Buffered on unsorted data

    ```
    CREATE INDEX geo_greece_unsorted_idx ON geo_greece USING gist(location) WITH (buffering=on);
    ```

  - Buffered on randomized data

    ```
    CREATE INDEX geo_greece_rnd_idx  ON geo_greece_rnd USING gist(location) WITH (buffering=on);
    ```

Example: GiST R-Tree for Greece

- Index statistics:

```
           index            |   size   | build time
----------------------------+----------+------------
 geo_greece_rnd_idx         |  2632 kB |     220 ms
 geo_greece_unsorted_idx    |  2736 kB |     240 ms
 geo_greece_zorder_idx      |  1768 kB |      37 ms
```

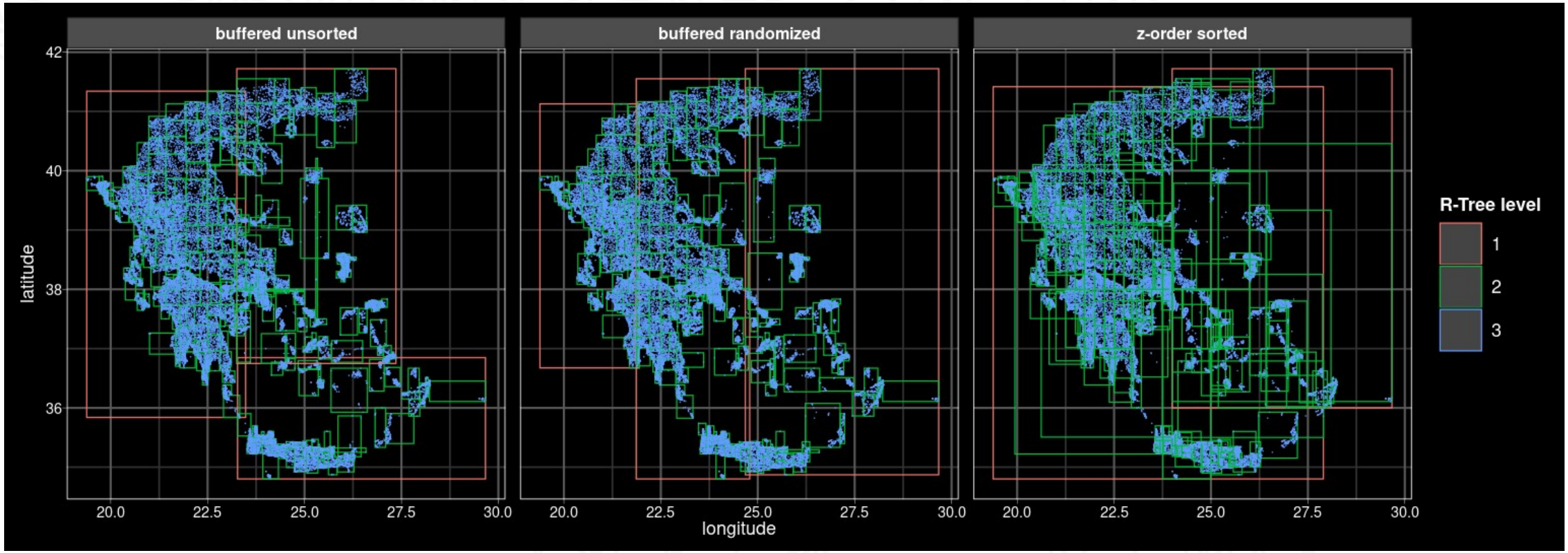
- R-Tree levels statistics (Gevel extension):

```
SELECT relname, level, count(*)
FROM pg_class, gist_print(relname) AS t(level int, valid bool, a box)
WHERE relname LIKE 'geo_greece_%_idx'
GROUP BY 1, 2 ORDER BY 1, 2;
```

```
          relname          | level | count
---------------------------+-------+-------
 geo_greece_rnd_idx        |     1 |     3
 geo_greece_rnd_idx        |     2 |   324
 geo_greece_rnd_idx        |     3 | 36124
 geo_greece_unsorted_idx   |     1 |     3
 geo_greece_unsorted_idx   |     2 |   335
 geo_greece_unsorted_idx   |     3 | 36124
 geo_greece_zorder_idx     |     1 |     2
 geo_greece_zorder_idx     |     2 |   218     1.5x more leaves per internal node
 geo_greece_zorder_idx     |     3 | 36124
```

Example: GiST R-Tree for Greece

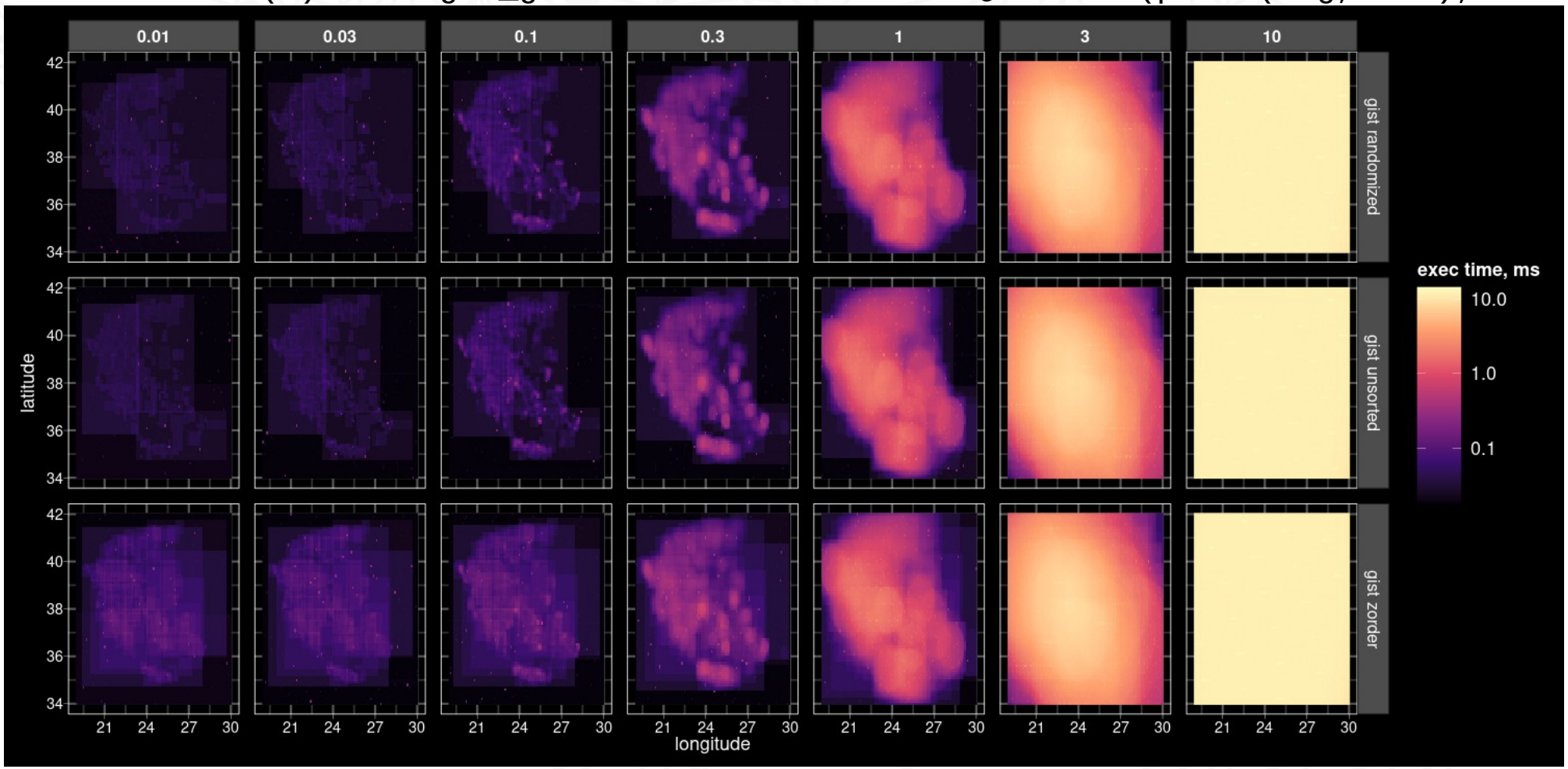
Z-order index boxes overlaps more strongly



Example: GiST R-Tree for Greece – search performance

Search speed depending on location and search area.

```
SELECT count(*) FROM geo_greece WHERE location <@ circle(point(lng, lat), radius);
```

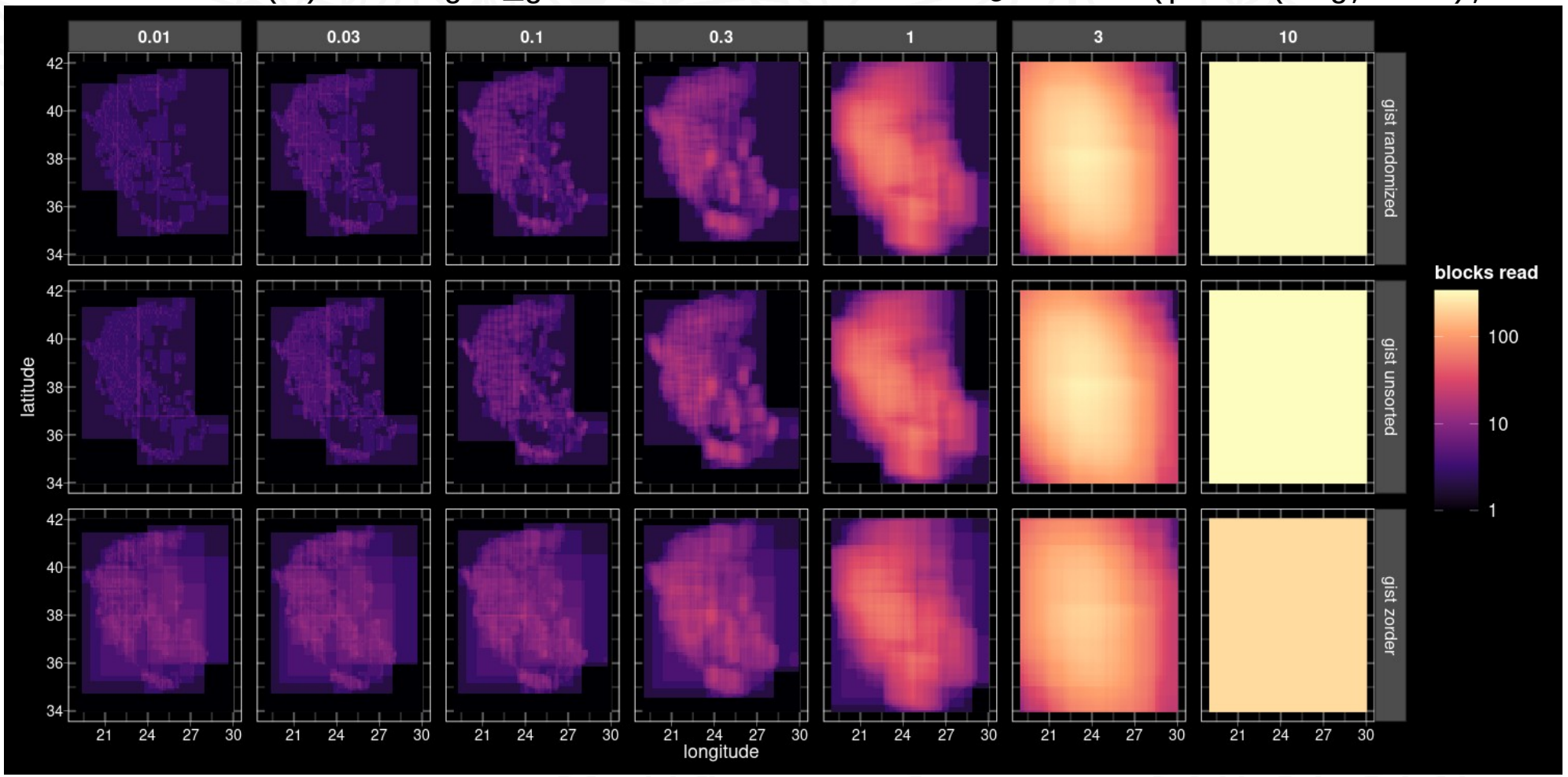


PostgresPro

Example: GiST R-Tree for Greece – search performance

Amount of blocks read depending on location and search area.

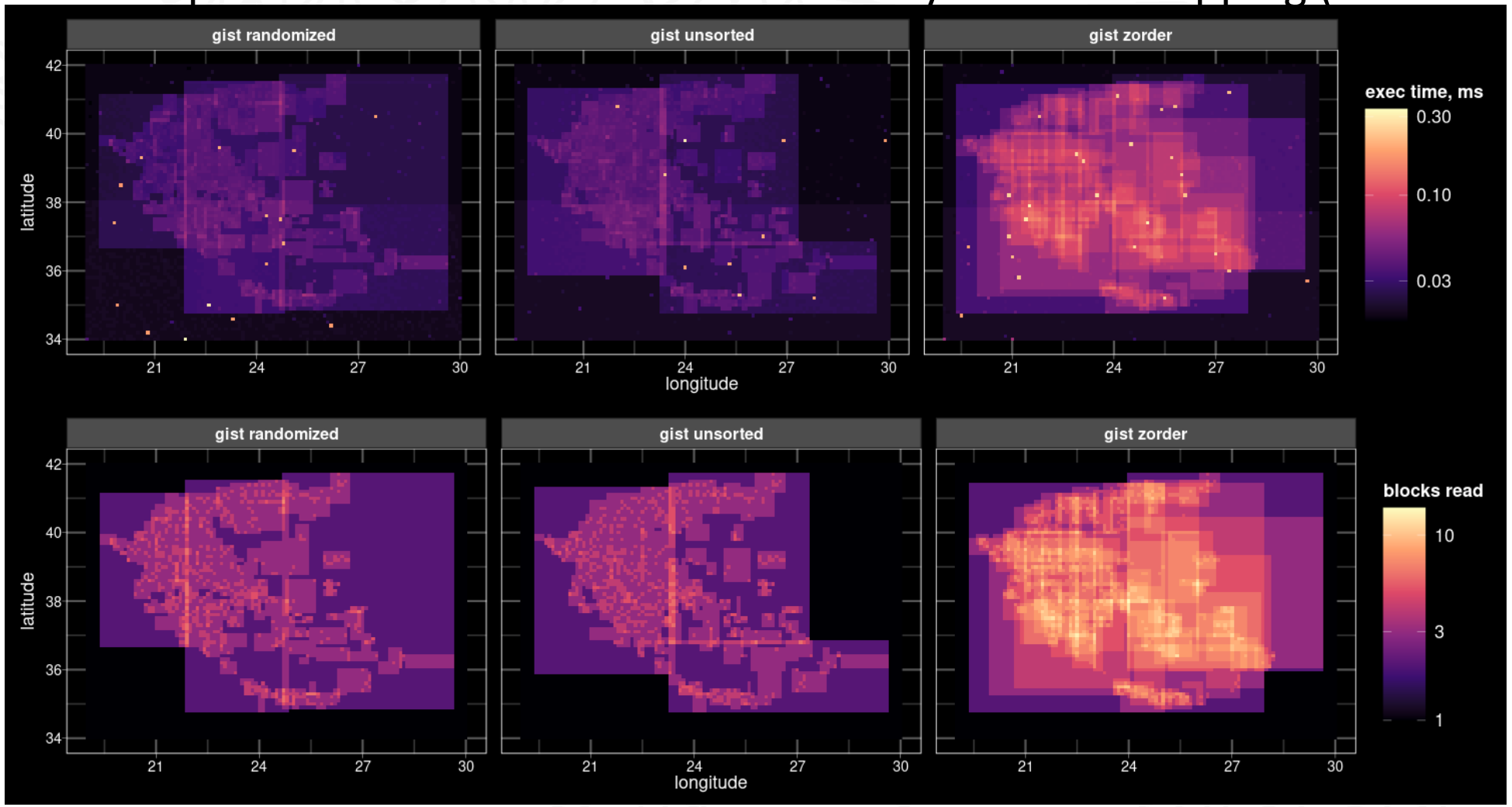
```
SELECT count(*) FROM geo_greece WHERE location <@ circle(point(lng, lat), radius);
```



PostgresPro

Example: GiST R-Tree for Greece – search performance

Search speed for small areas is determined by node overlapping (radius = 0.01):

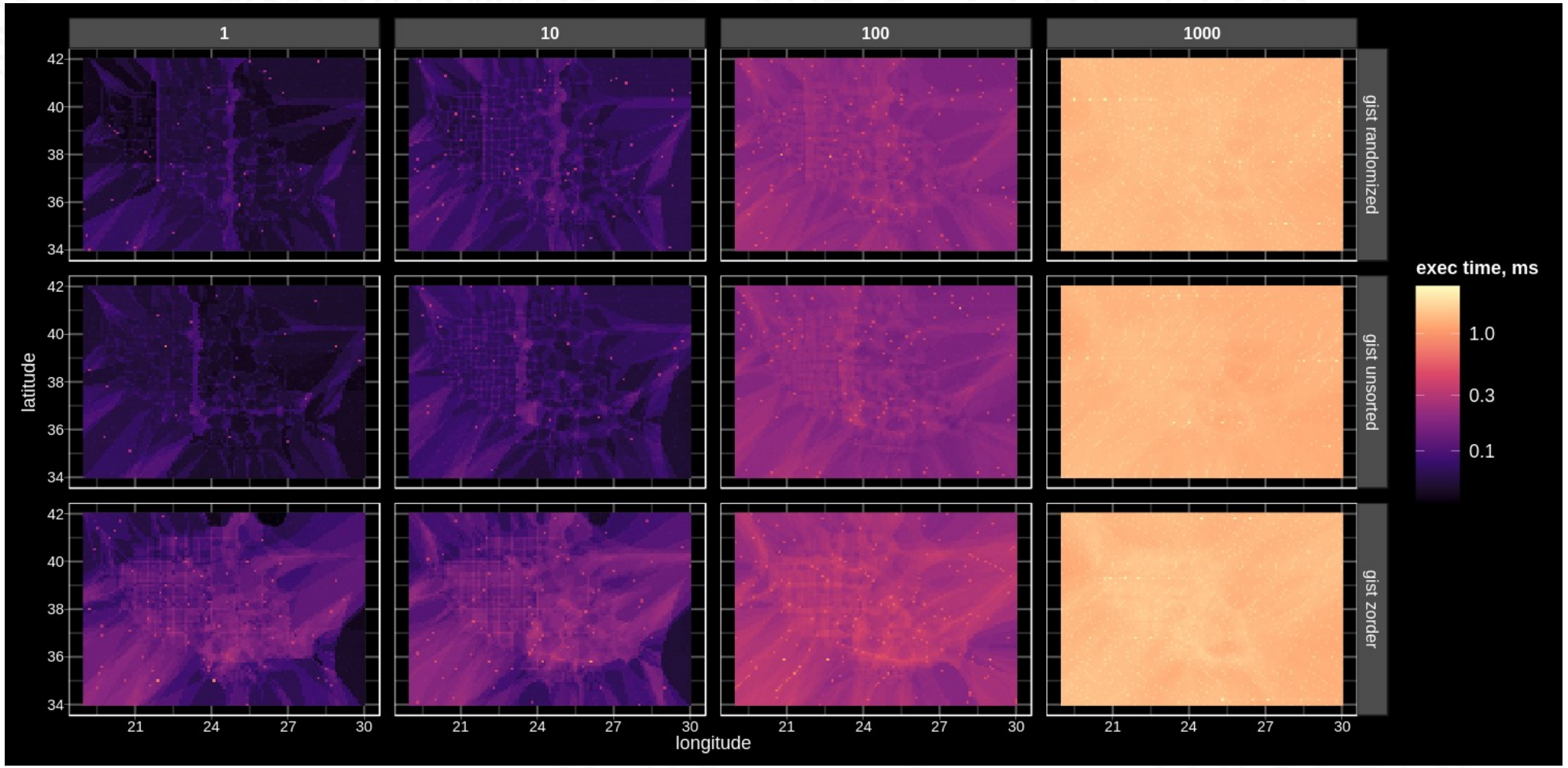


PostgresPro

Example: GiST R-Tree for Greece – kNN performance

kNN search time is not much tied to tree partitioning, but kNN still suffers from overlapping.

```
SELECT count(*) FROM geo_greece ORDER BY location <-> point(lng, lat) LIMIT n;
```

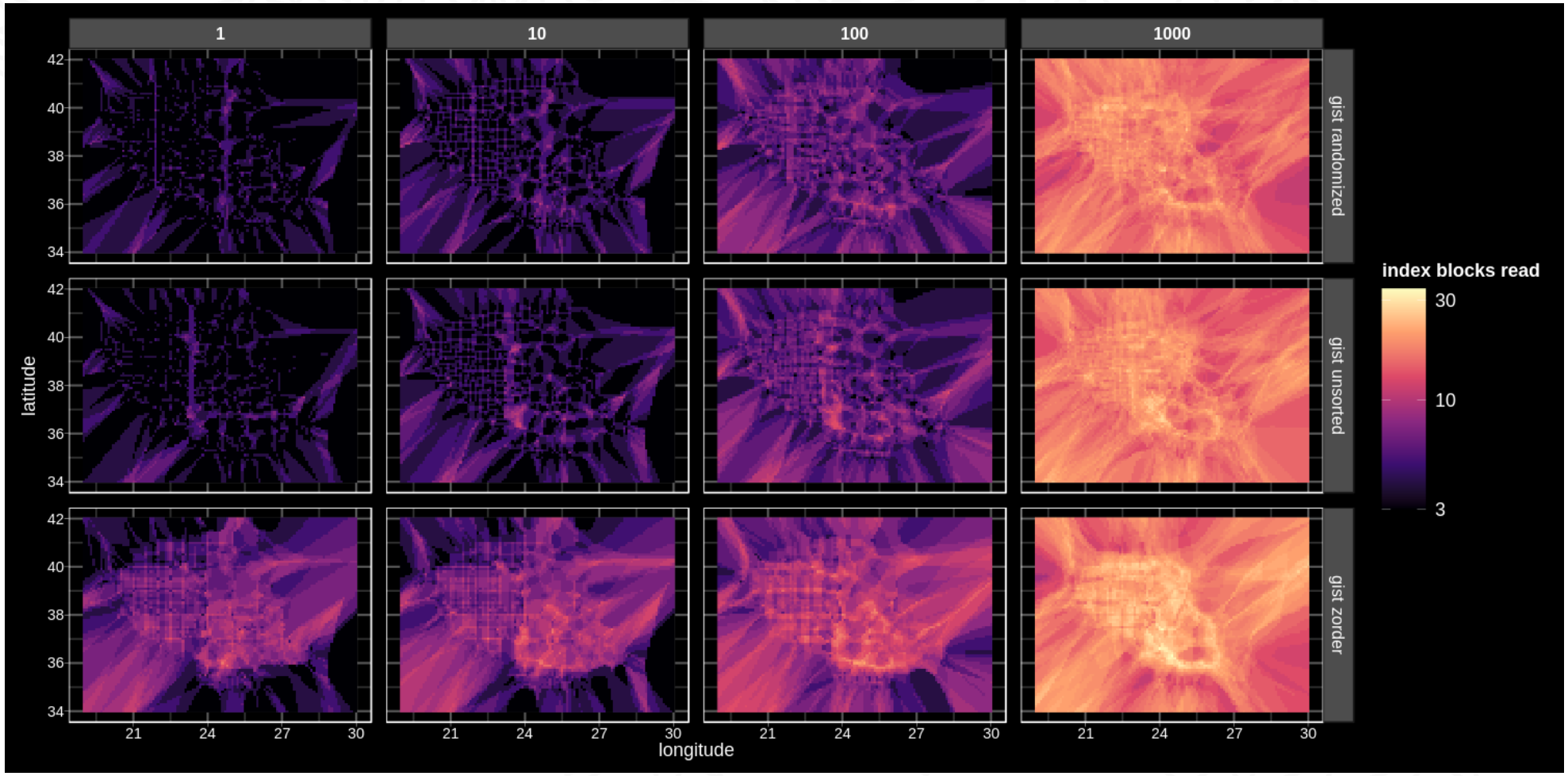


PostgresPro

Example: GiST R-Tree for Greece – kNN performance

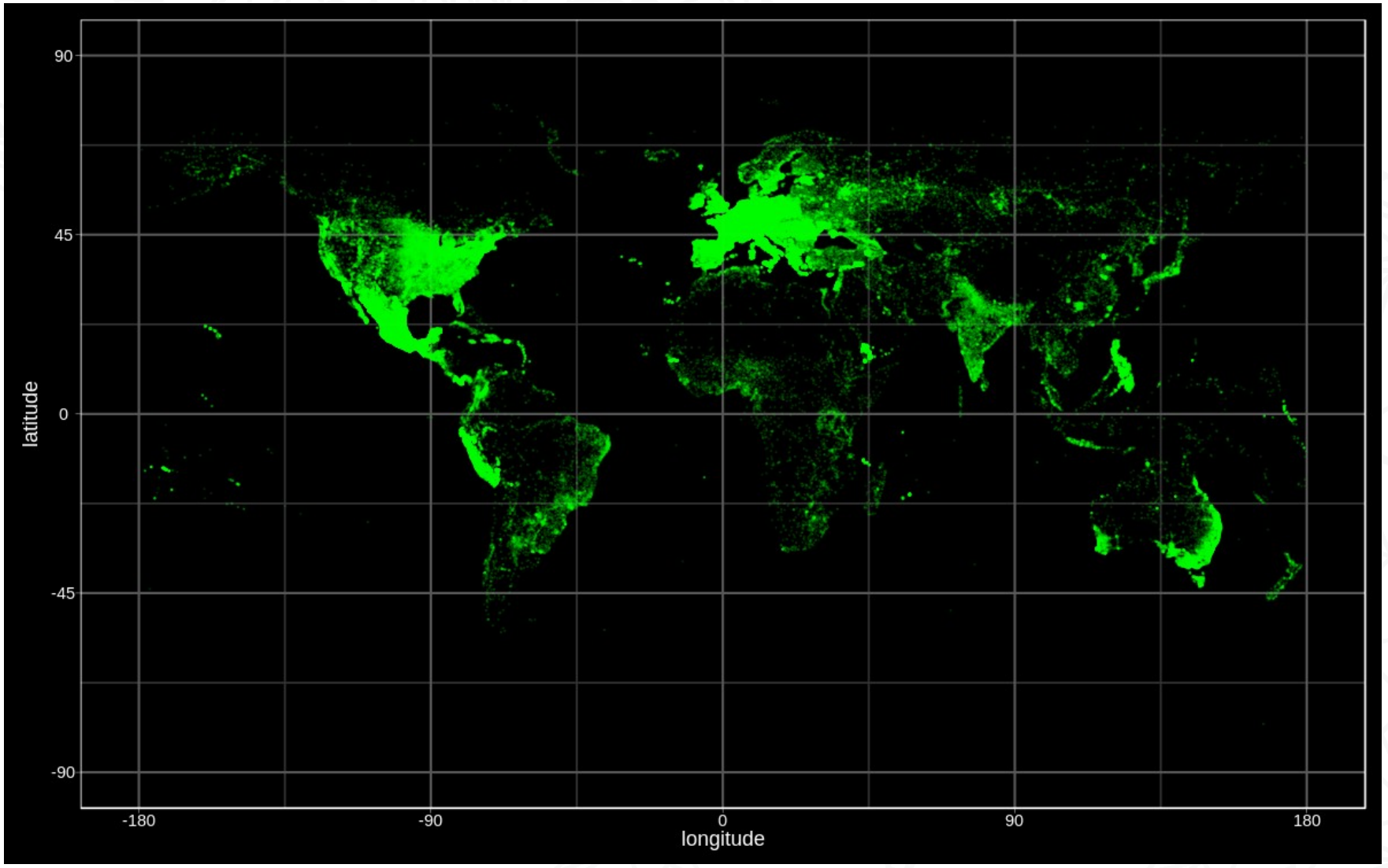
Number of index blocks read by lower-limit kNN is grealy increased by tree overlapping.

```
SELECT count(*) FROM geo_greece ORDER BY location <-> point(lng, lat) LIMIT n;
```



PostgresPro

Whole world kNN – Geonames dataset



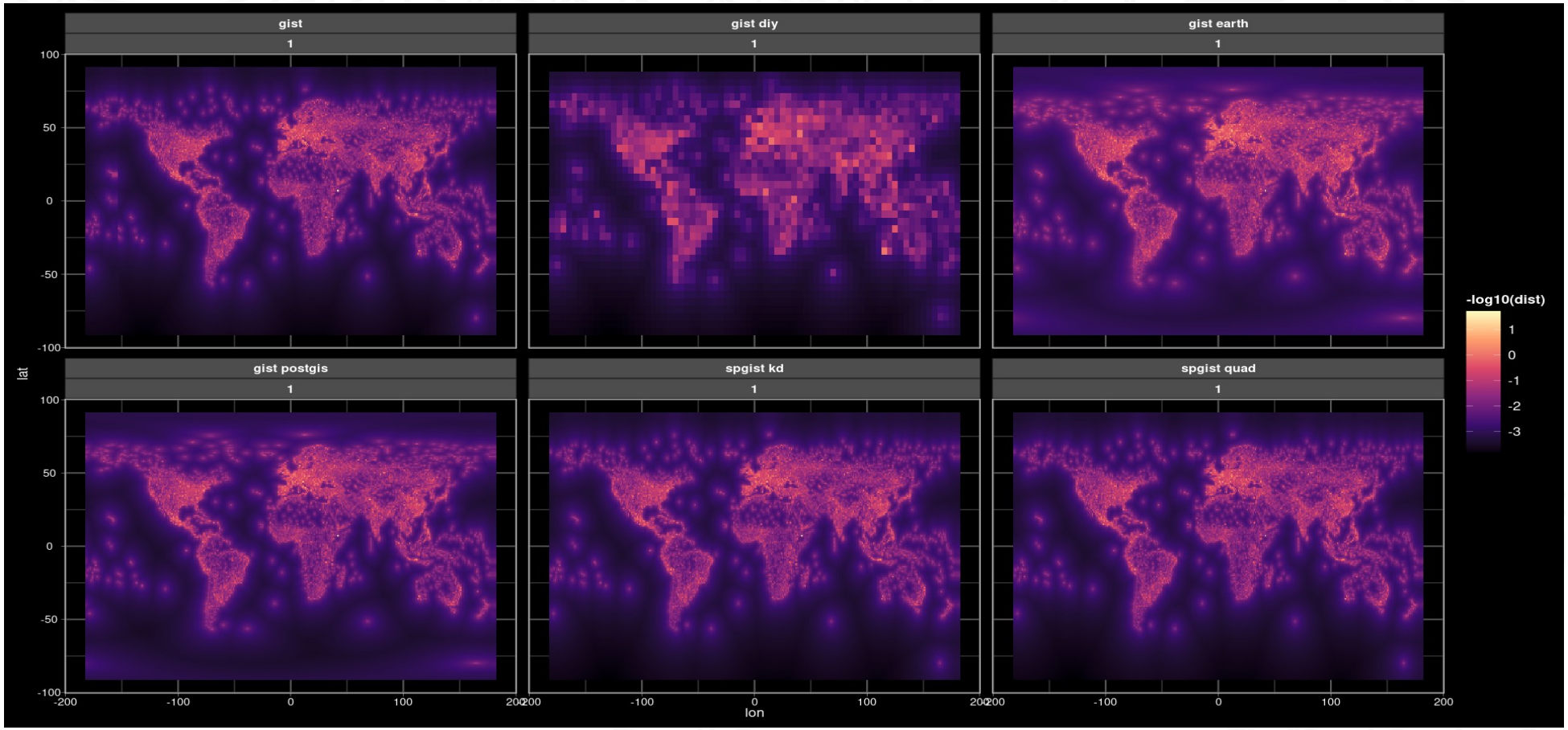
- Geonames dataset contains coordinates of 422523 cities

PostgresPro

How indexes see the Earth - Distances to the nearest city (1° step)

McMurdo Station, Antarctic research center — the farthest settlement in geonames db

```
 geonameid |                | earth_location |       dist (postgis)
-----------+----------------+----------------+-----------------------
   6696480 | McMurdo Station| (166.676,-77.846)      |    2159.347354917357
```



McMurdo Station

Antarctic base

McMurdo Station from Observation Hill

Location of McMurdo Station in Antarctica

Coordinates: 77°50′47″S 166°40′06″E

PostgresPro

KNN whole world – DIY kNN

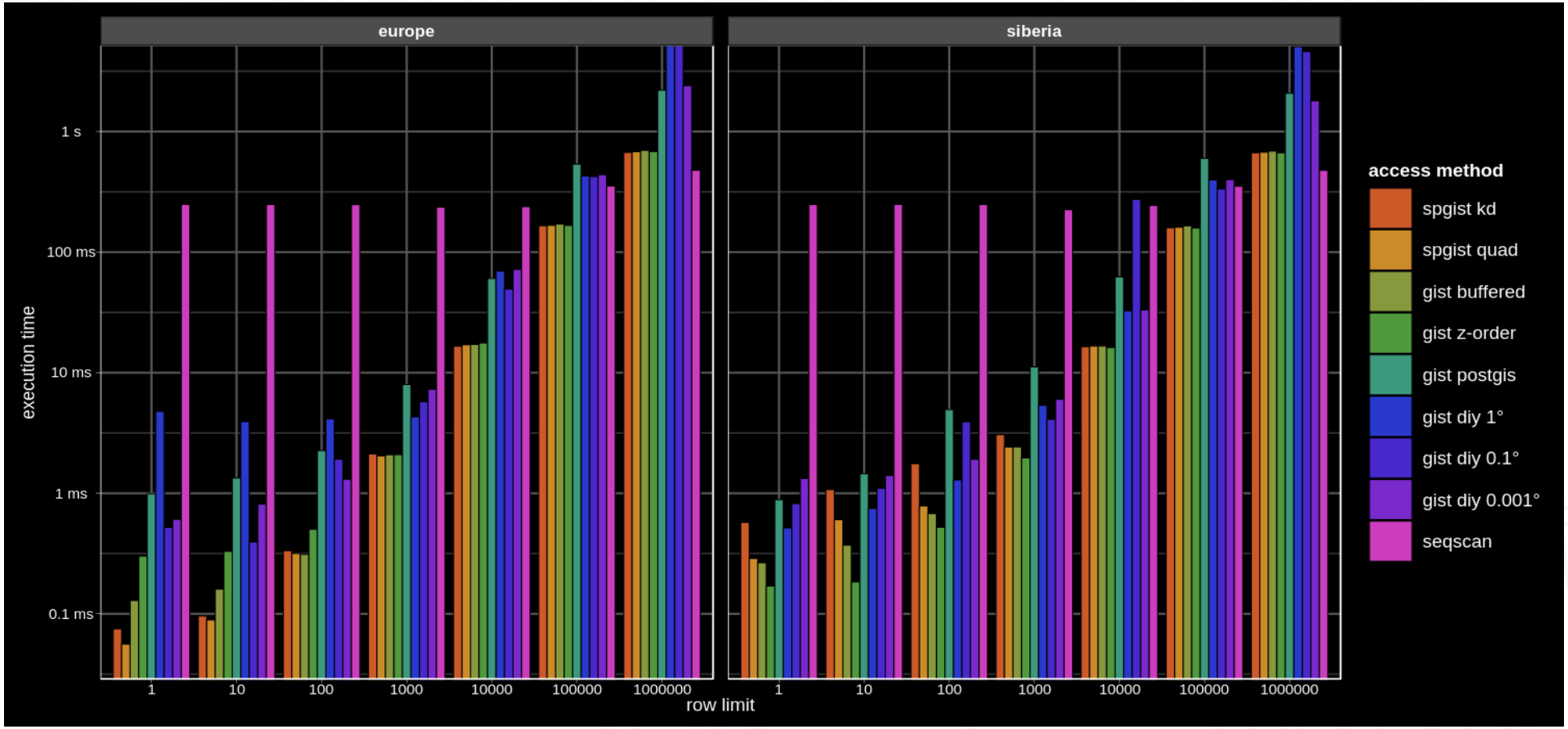
- KNN can be emulated by searching and sorting points inside the circles with exponentially increasing radius. Its performance is quite sensitive to the choice of initial radius, which can be estimated by point density and search limit.

```
SELECT points.*
FROM
  (VALUES (point($2, $1))) p(pt),
  generate_series(0.0, 20.0, 1.0) i,
  pow(2.0, i + $4) r,      -- increasing circle radius, starting from 2^$4
  LATERAL (
    SELECT * FROM (
      SELECT geonameid, name, location, location <-> pt dist
      FROM cities
      WHERE location <@ circle(pt, r)        -- search point inside circle
    ) tmp
    WHERE i = 0 OR dist > r * 0.5      -- skip points inside previous circle
    ORDER BY dist
  ) points
LIMIT $3;
```

PostgresPro

KNN whole world – Europe vs Siberia

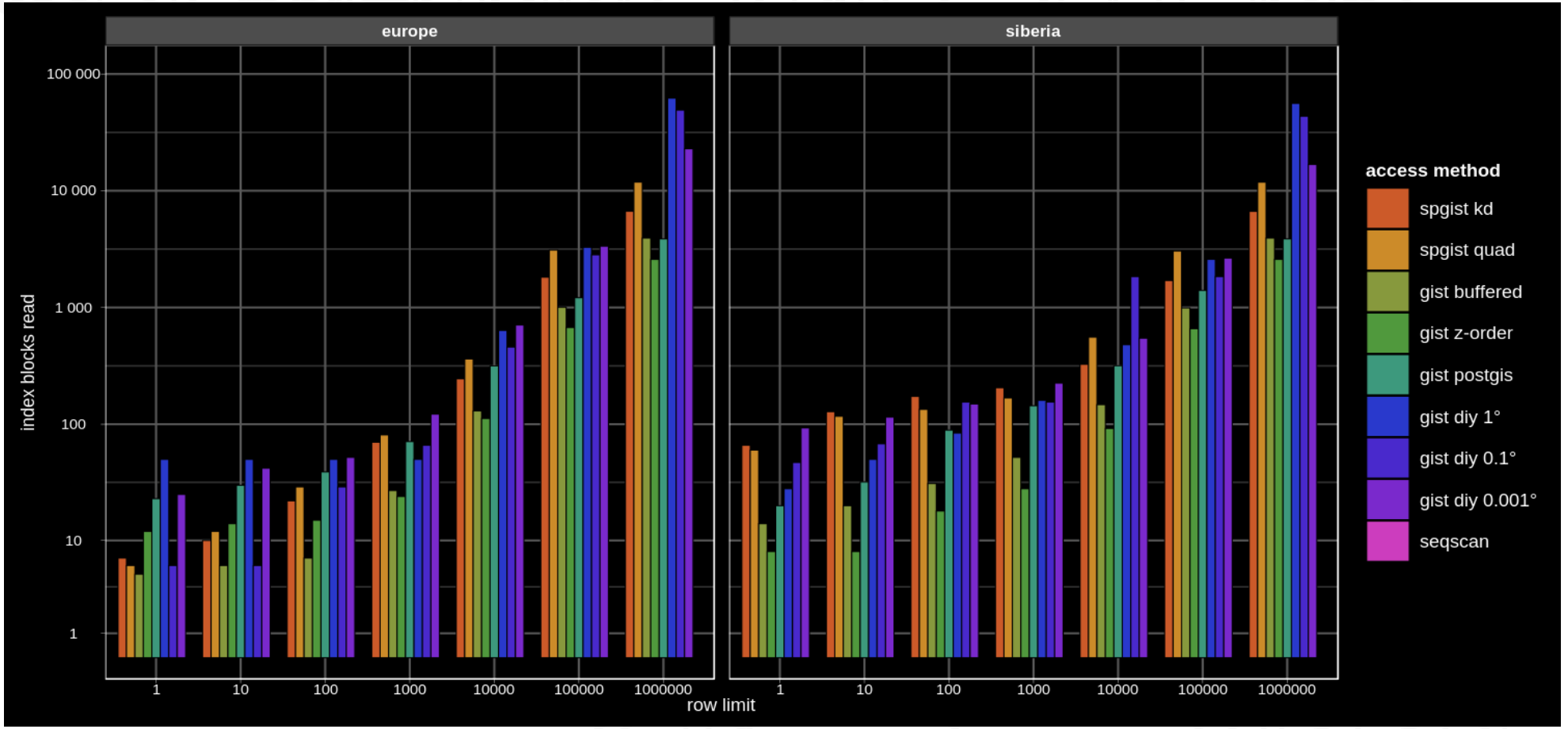
- GiST kNN is faster than SP-GiST kNN in lower-density areas
- Seq scan is a bit faster than kNN by index only when all rows are read
- DIY kNN scan speed is very sensitive to initial radius for low limits



PostgresPro

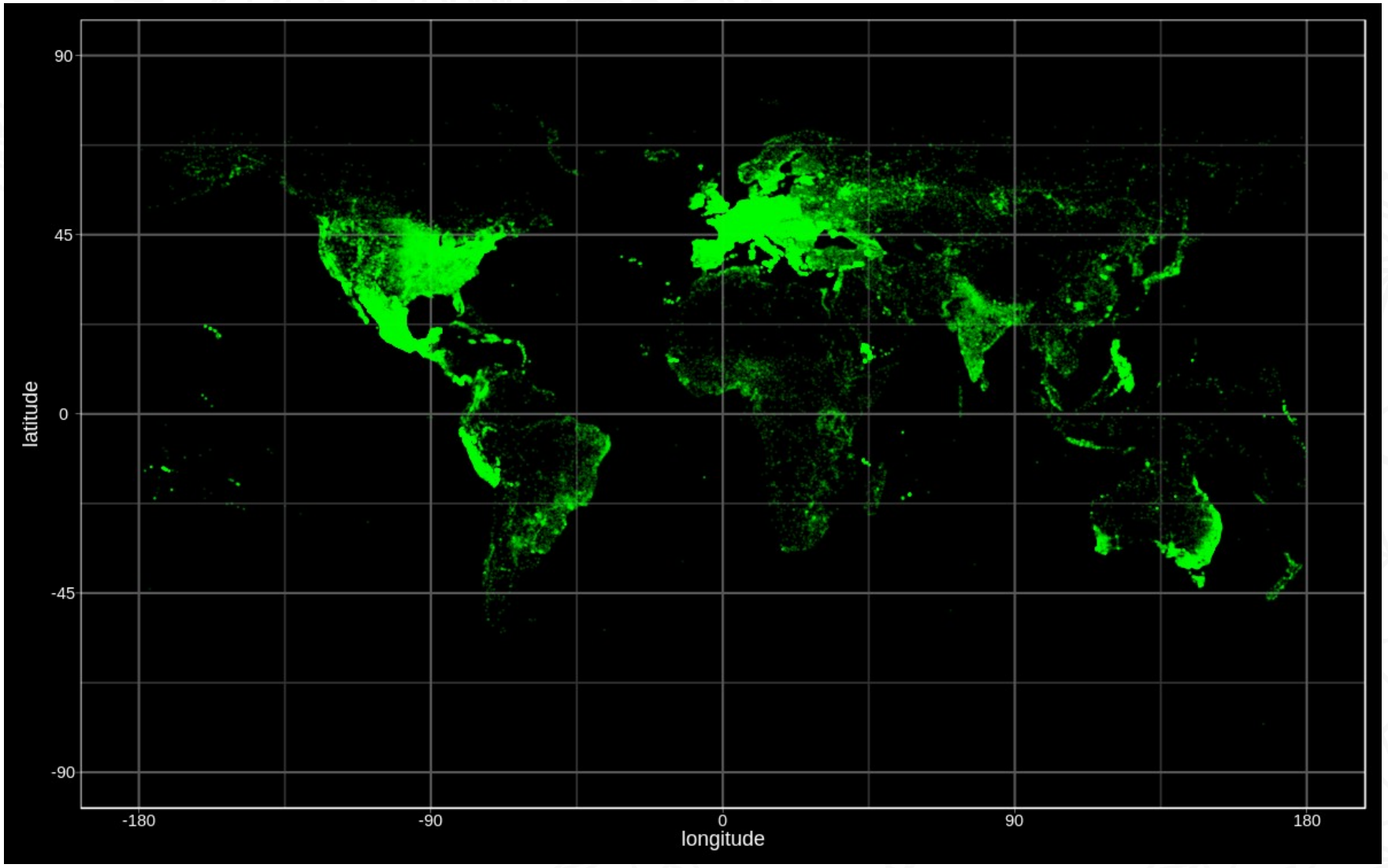
KNN whole world – Europe vs Siberia

- The same applies to the number of index blocks read.



PostgresPro

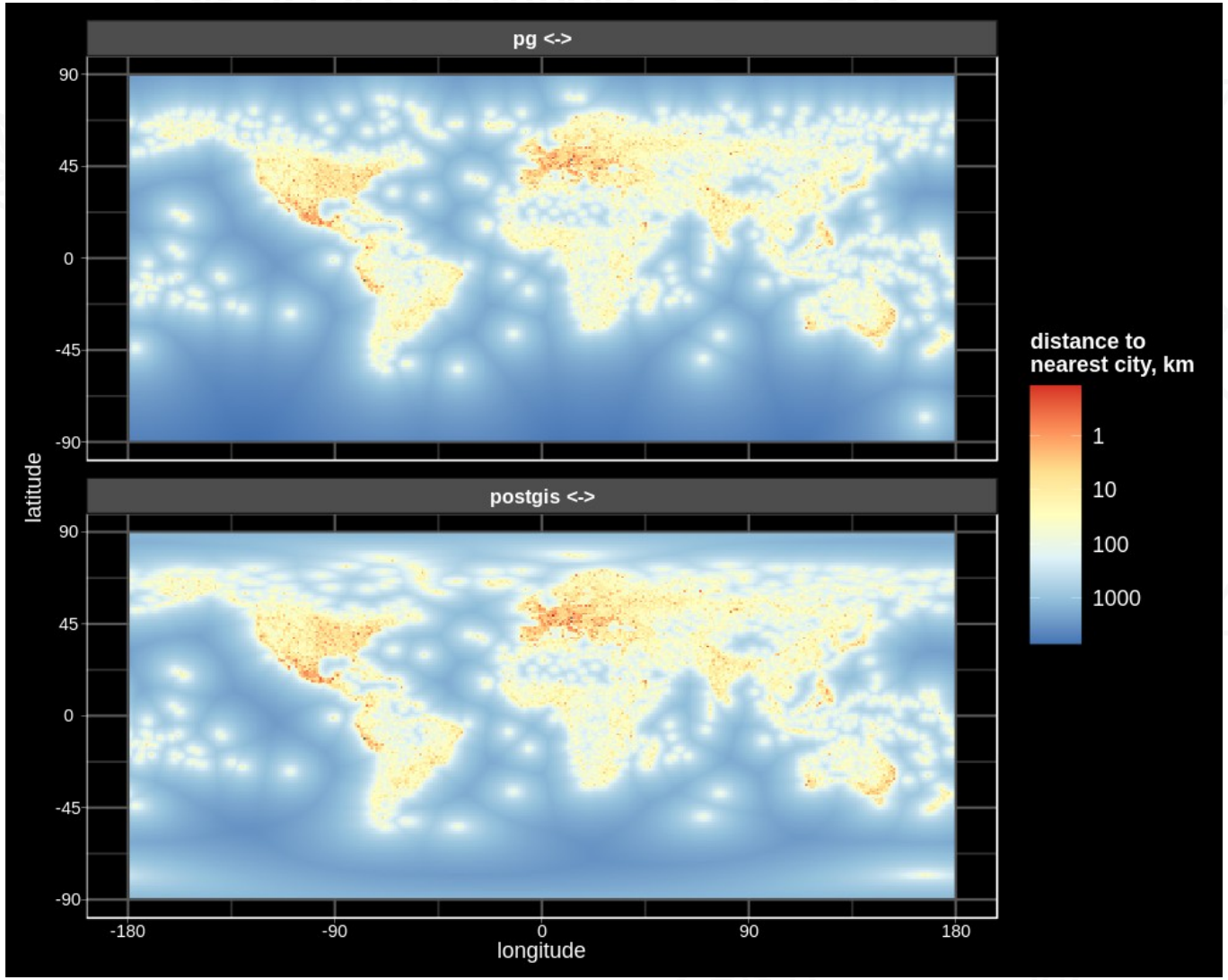
Whole world kNN – geonames dataset



- Geonames dataset contains coordinates of 422523 cities

PostgresPro

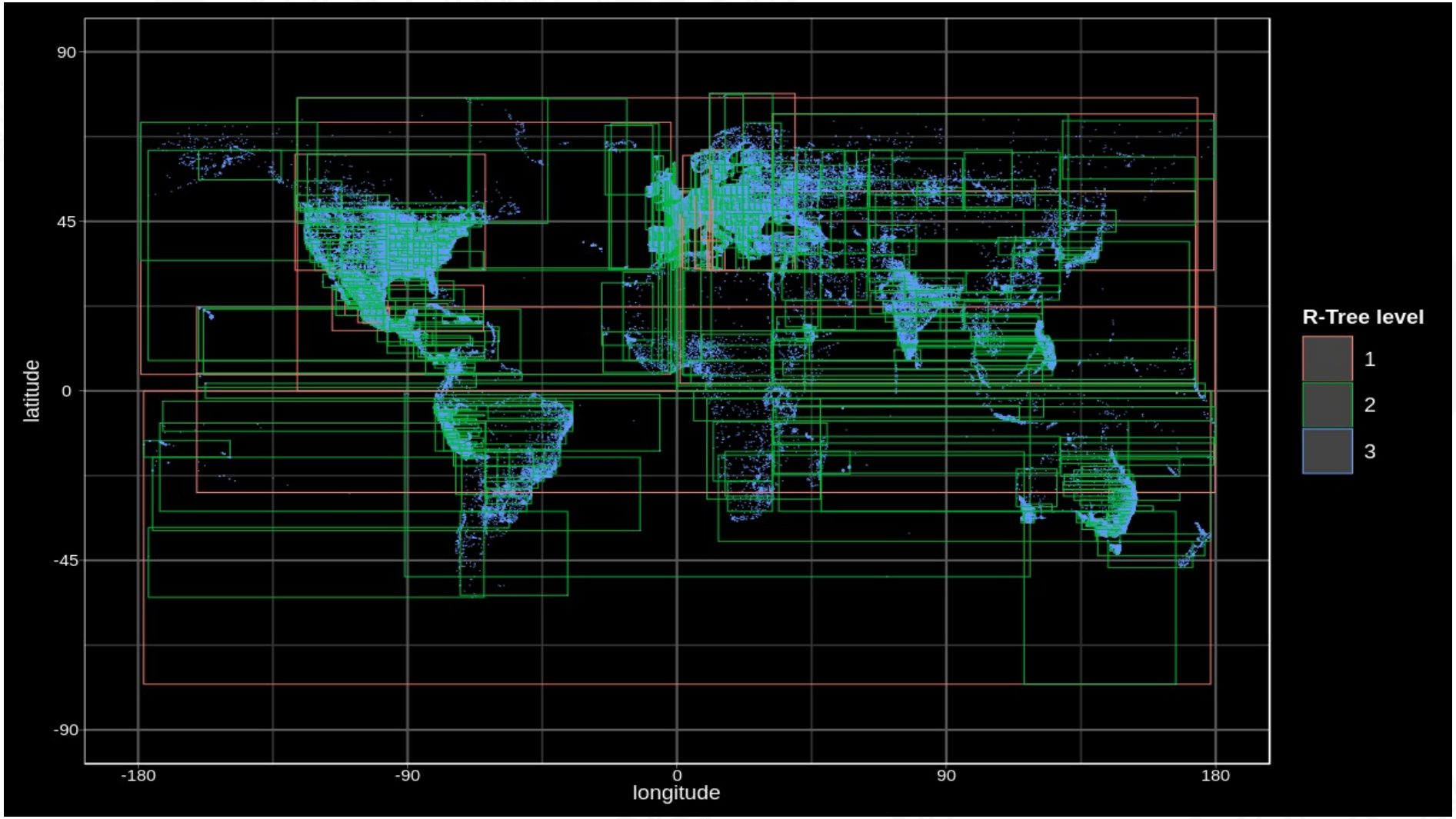
Whole world kNN – distance functions



- Built-in opclass GiST and SP-GiST opclasses for points use simple Euclid 2D distance:
  $sqrt((lat_1 — lat_2)^2 + (lng_1 - lng_2)^2)$
- PostGIS uses correct spherical distance function like contrib/earthdistance.
- The difference between them is absent for near points on equator , but it grows up with distance and latitude growth.

PostgresPro

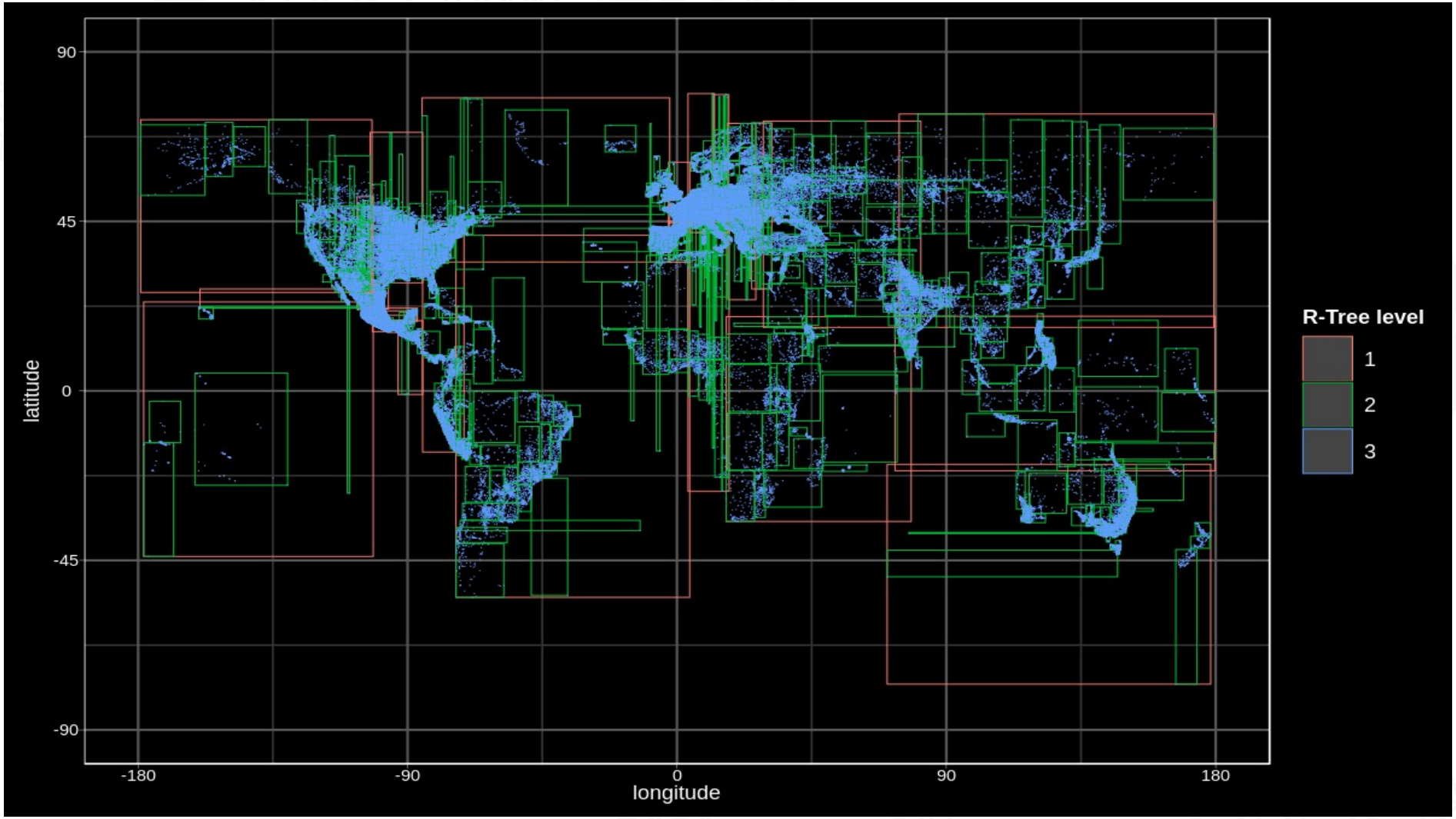
Whole world kNN – GiST Z-order sorted index



Z-order sorting index produces a lot of overlapping.

PostgresPro

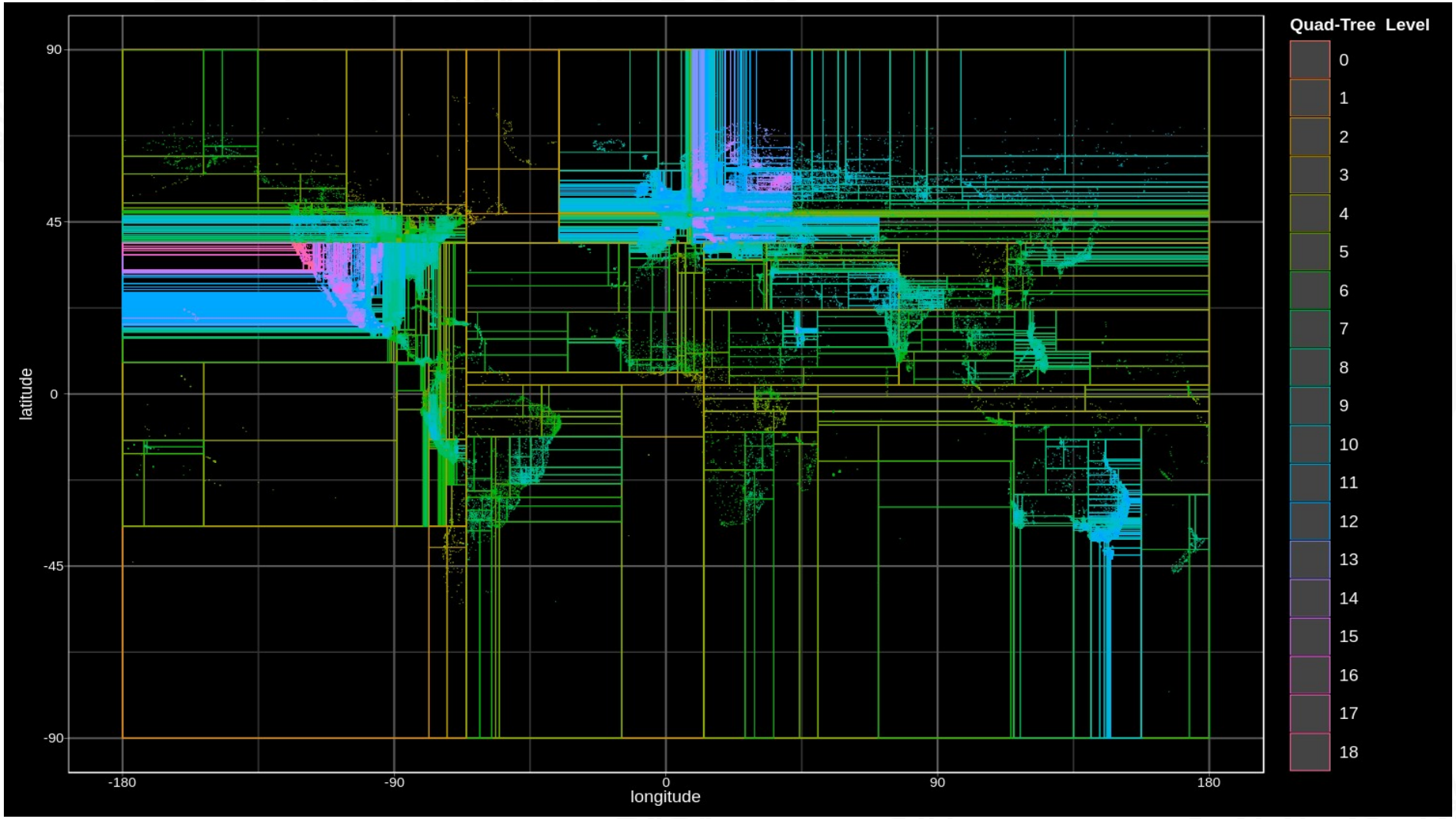
Whole world kNN – GiST on randomly sorted data



Buffered GiST building with randomly sorted data produces a much better tree partitioning.

PostgresPro

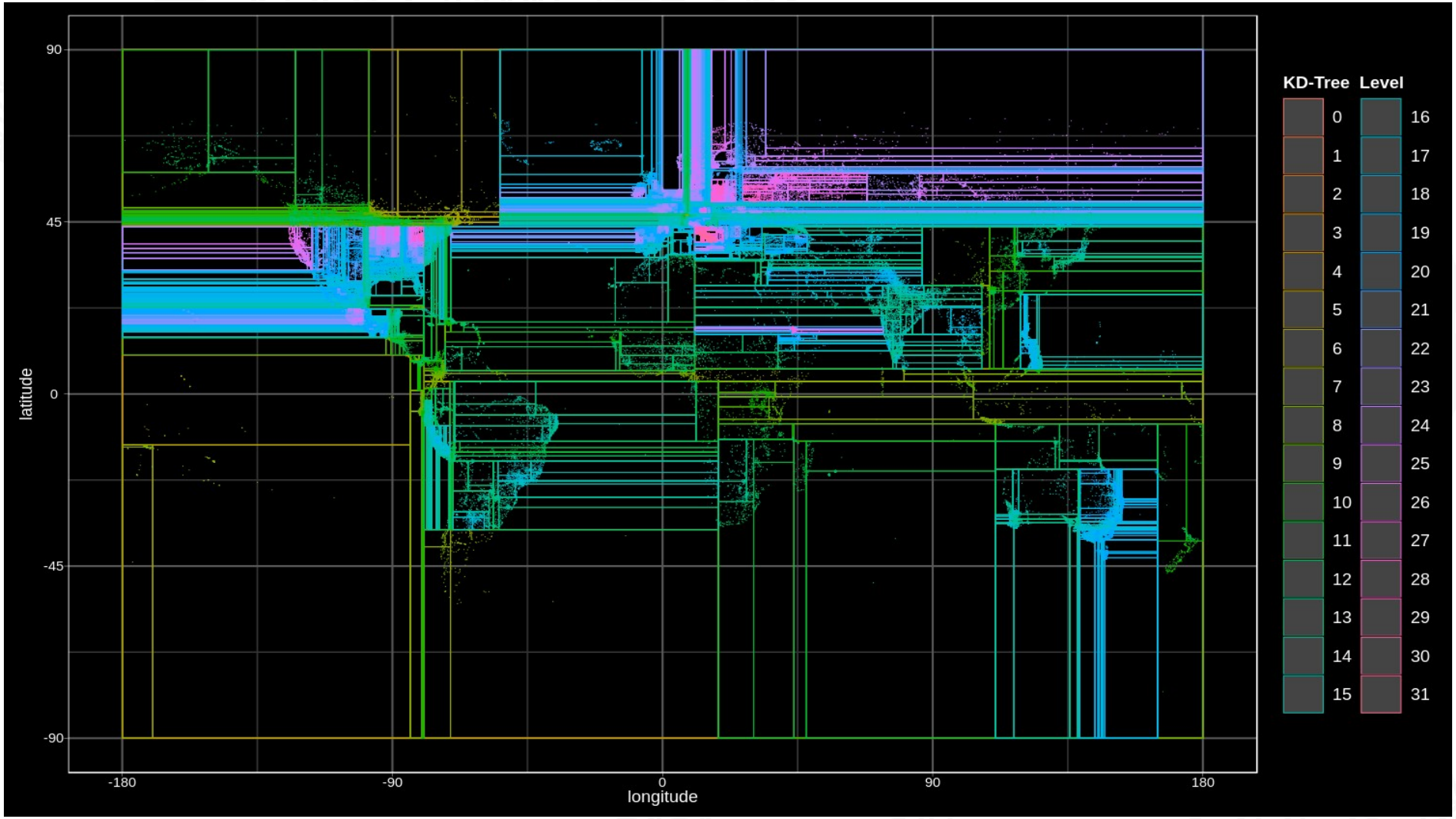
Whole world kNN – SP-GiST Quad-Tree index



SP-GiST Quad-Tree contains up to 19 levels in higher-density areas.

PostgresPro

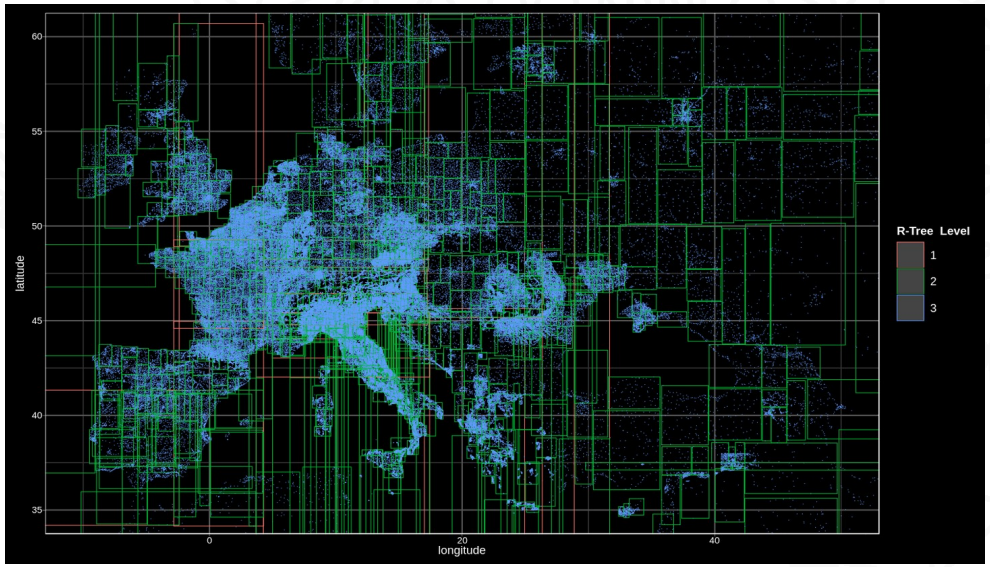
Whole world kNN – SP-GiST KD-Tree index

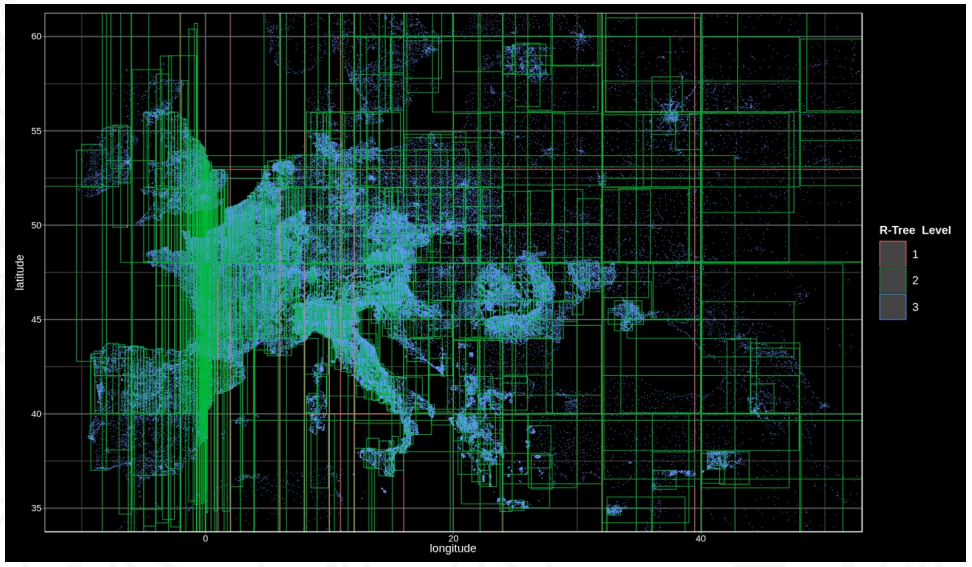


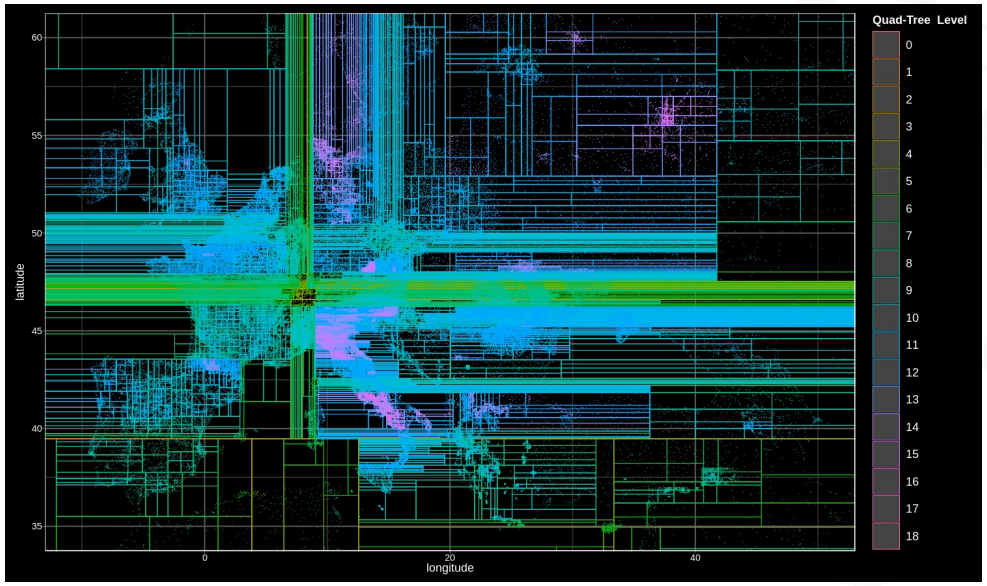
SP-GiST KD-Tree has even more levels: up to 32 levels.

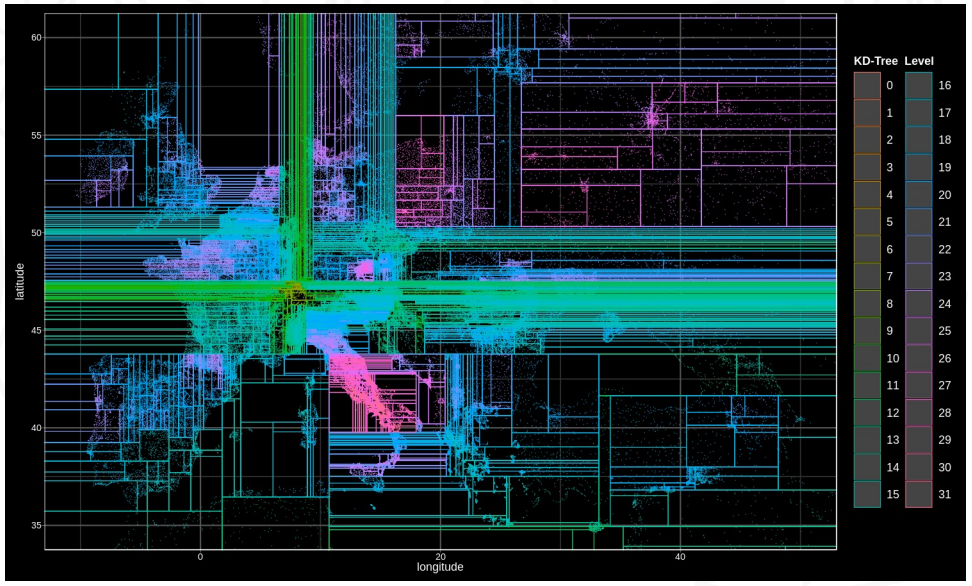
PostgresPro

Whole world kNN – 4 indexes, Europe only





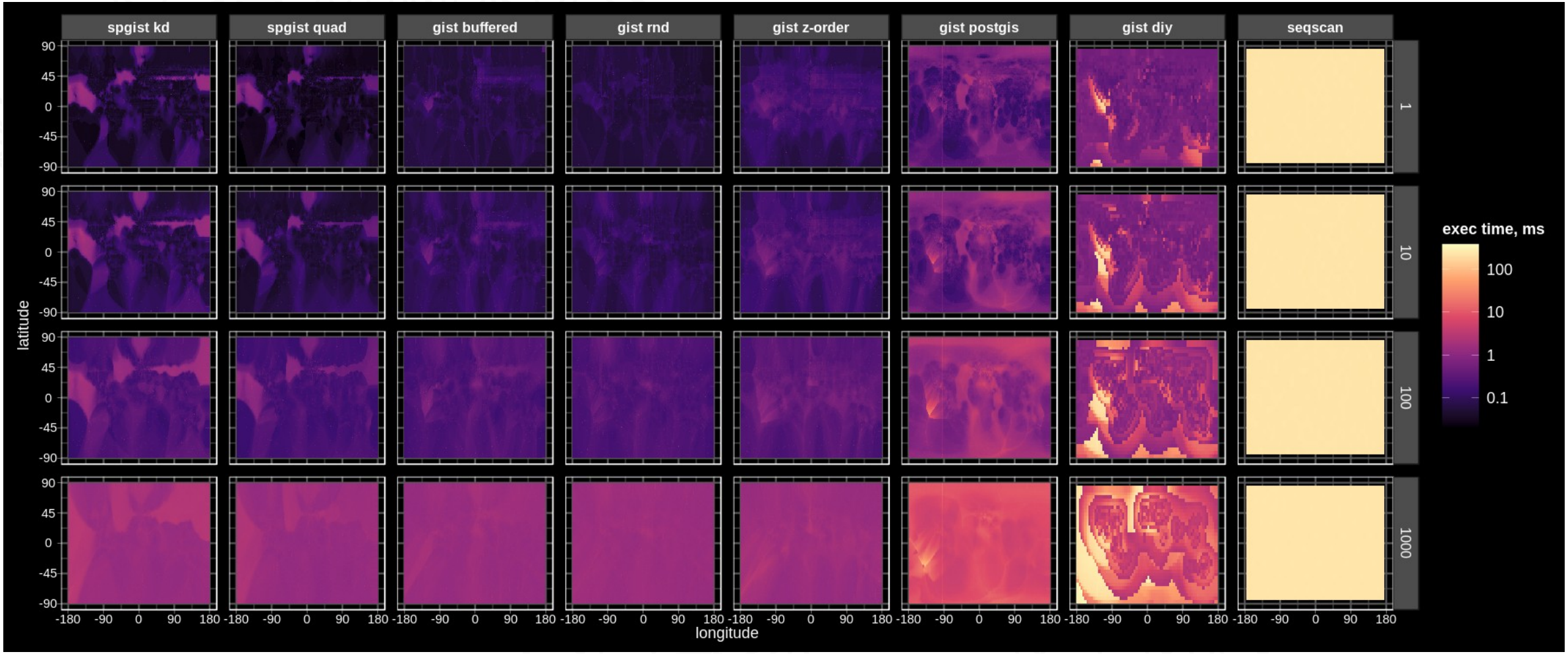




GiST randomized
GiST Z-order
SP-GiST Quad
SP-GiST KD

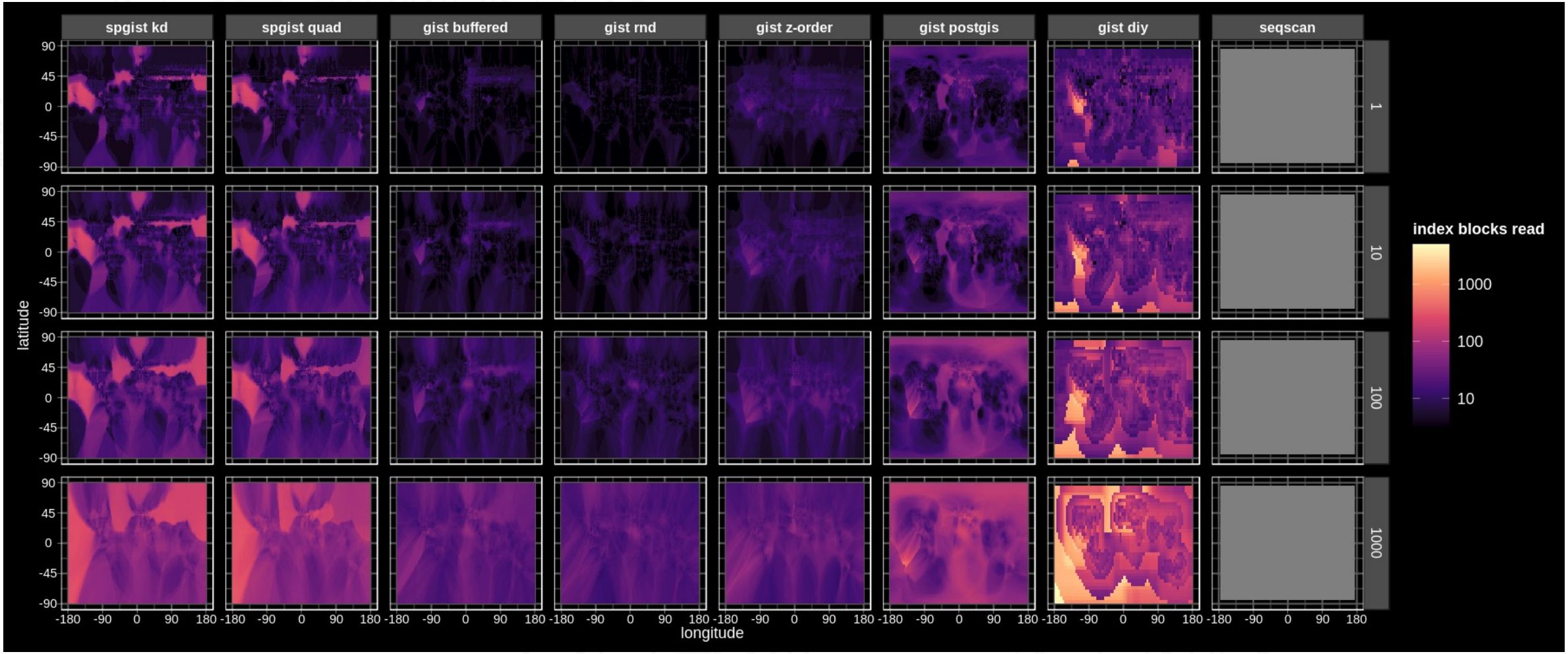
PostgresPro

KNN whole world – execution time map



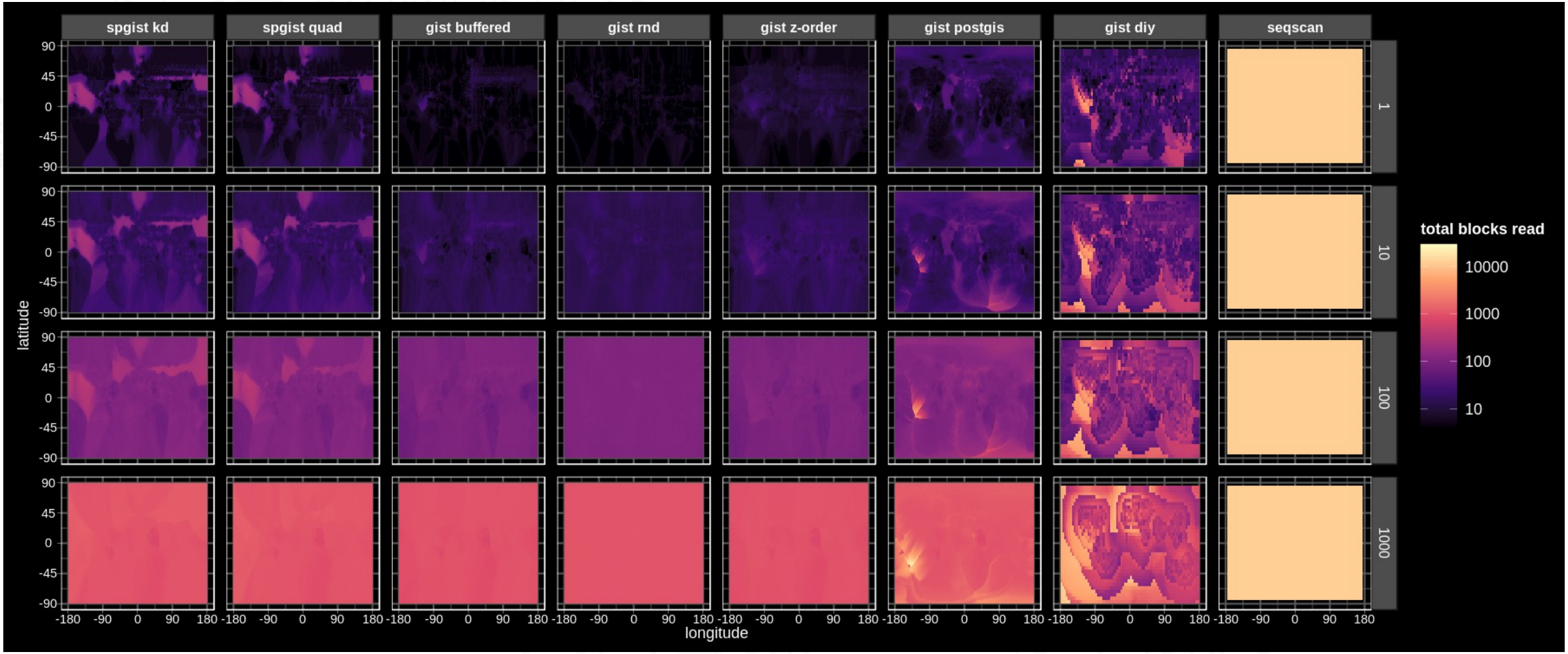
PostgresPro

KNN whole world – index blocks map



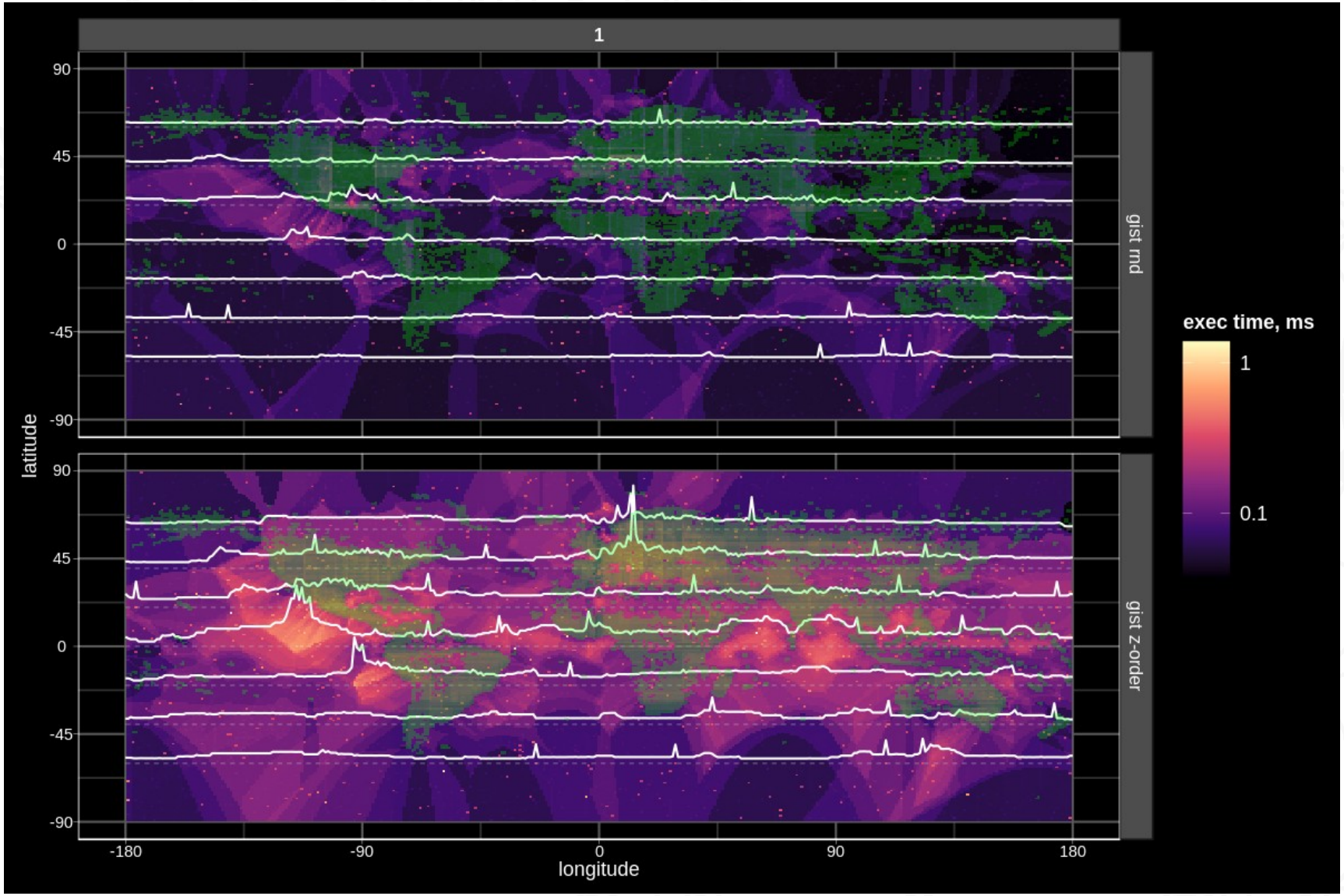
PostgresPro

KNN whole world – total blocks map



PostgresPro

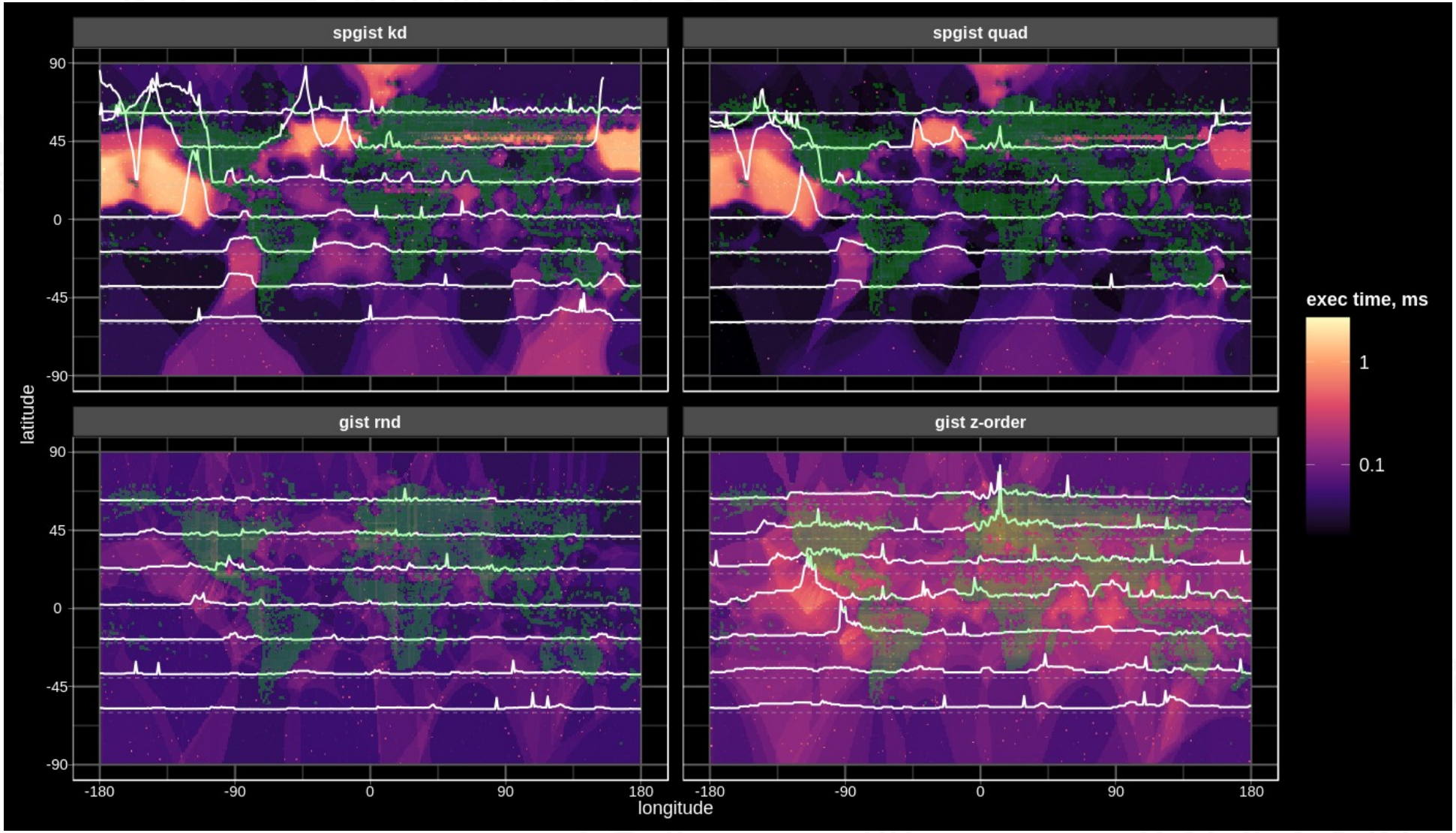
KNN whole world – Z-order sorted vs randomized



- Z-order performance is more unstable

PostgresPro

KNN whole world – 4 indexes



- GiST rnd has the best predictable performance

PostgresPro

KNN on trigrams

- `contrib/pg_trgm`
- GiST and GIN indexes
- Supports kNN only on GiST
- Distance operators:
  - Similarity: text <-> text
  - Word similarity: text <->> text, text <<-> text

$$word1 <-> word2 = 1 - N_{commmon\ trigrams} / (Len_{word1} + Len_{word2} - N_{commmon\ trigrams})$$

PostgresPro

KNN on trigrams

`pg_gtrm` GiST index contains:

- Unmodified arrays of trigrams in the leaves.
- Fixed-length signatures in the inner nodes. Each trigram is mapped to the one bit in signature using simple hash function. So trigrams can collide and signature lengths equal to power of 2 are not effective, because hash function will be not sensitive to all chars.

$$Hash(trg) = (char_0 + char_1 << 8 + char_2 << 16) \% Len_{signature}$$

$$Signature(word) = OR_{i=1..Ntrg}(1 << Hash(trg_i))$$

$$Signature(leaf\ page) = OR_{i=1..Nwords}\ (Signature(word_i))$$

$$Signature(inner\ page) = OR_{i=1..Nitems}\ (Signature_i)$$

- kNN uses the following minimal distance estimation for signatures:

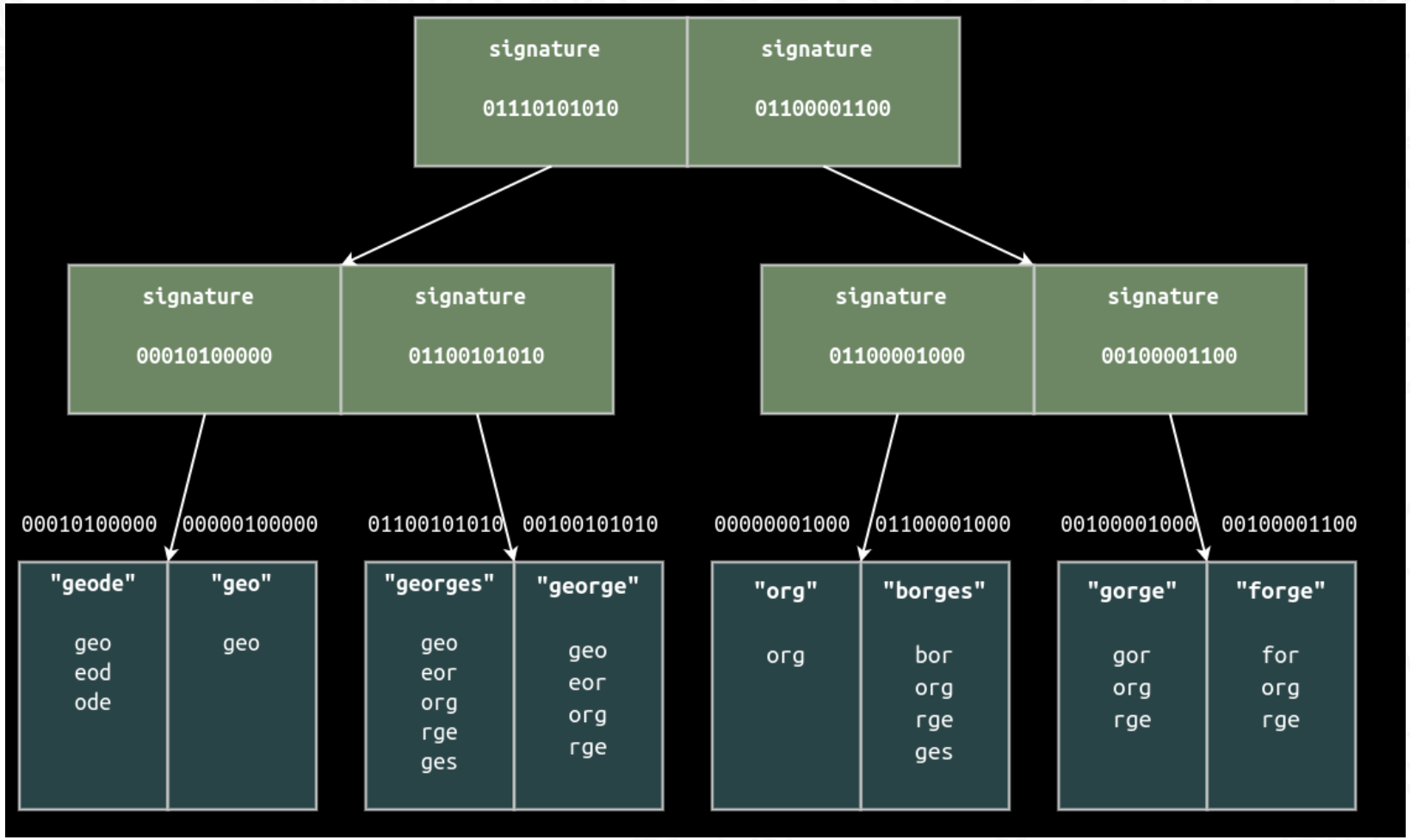
$$word <-> signature = 1 - N_{common\ bits} / N_{word\ trigrams}$$

PostgresPro

KNN on trigrams

Sample GiST tree with 8 words and query:

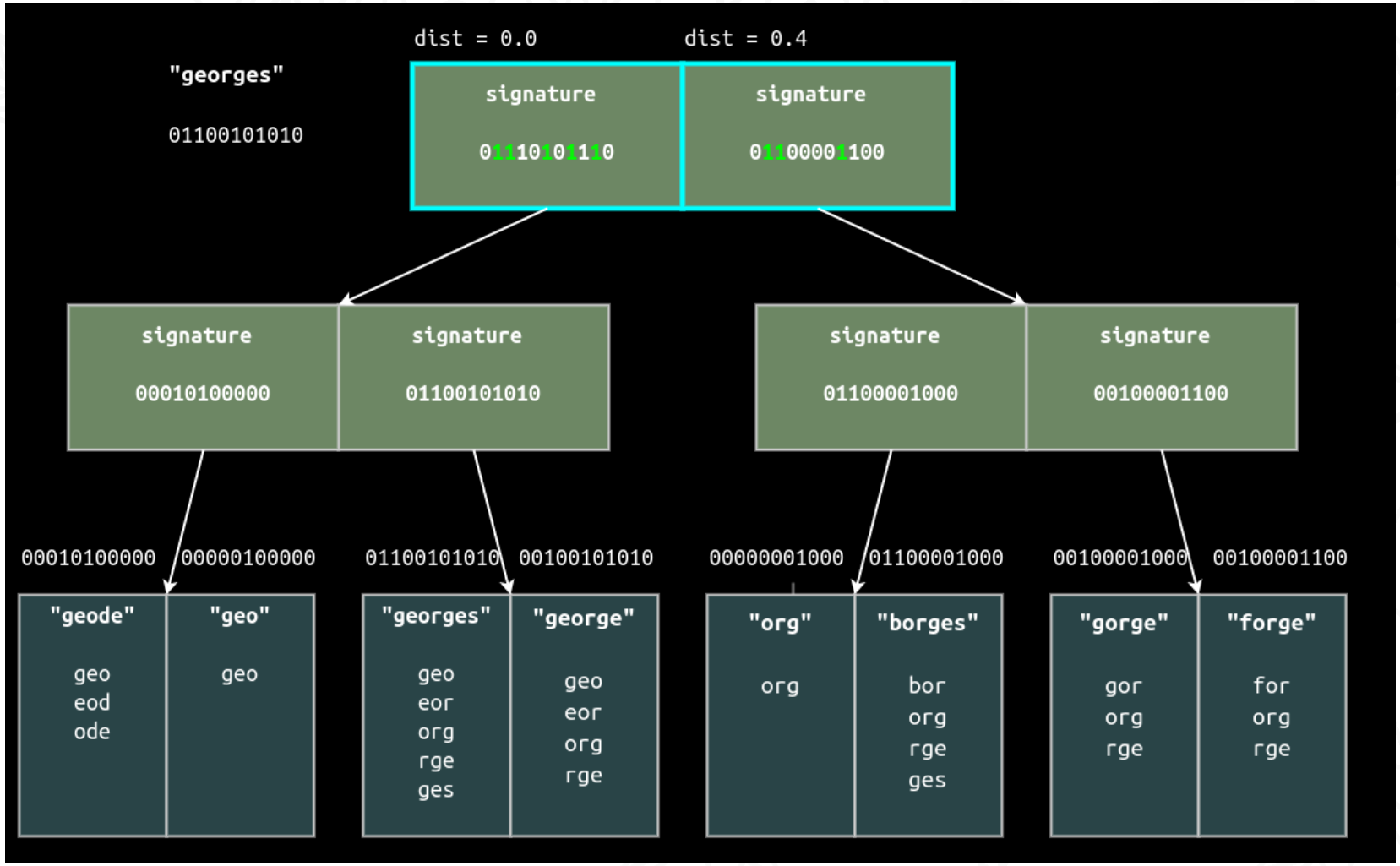
```
SELECT * FROM words ORDER BY word <-> 'georges' LIMIT 3;
```



PostgresPro

KNN on trigrams

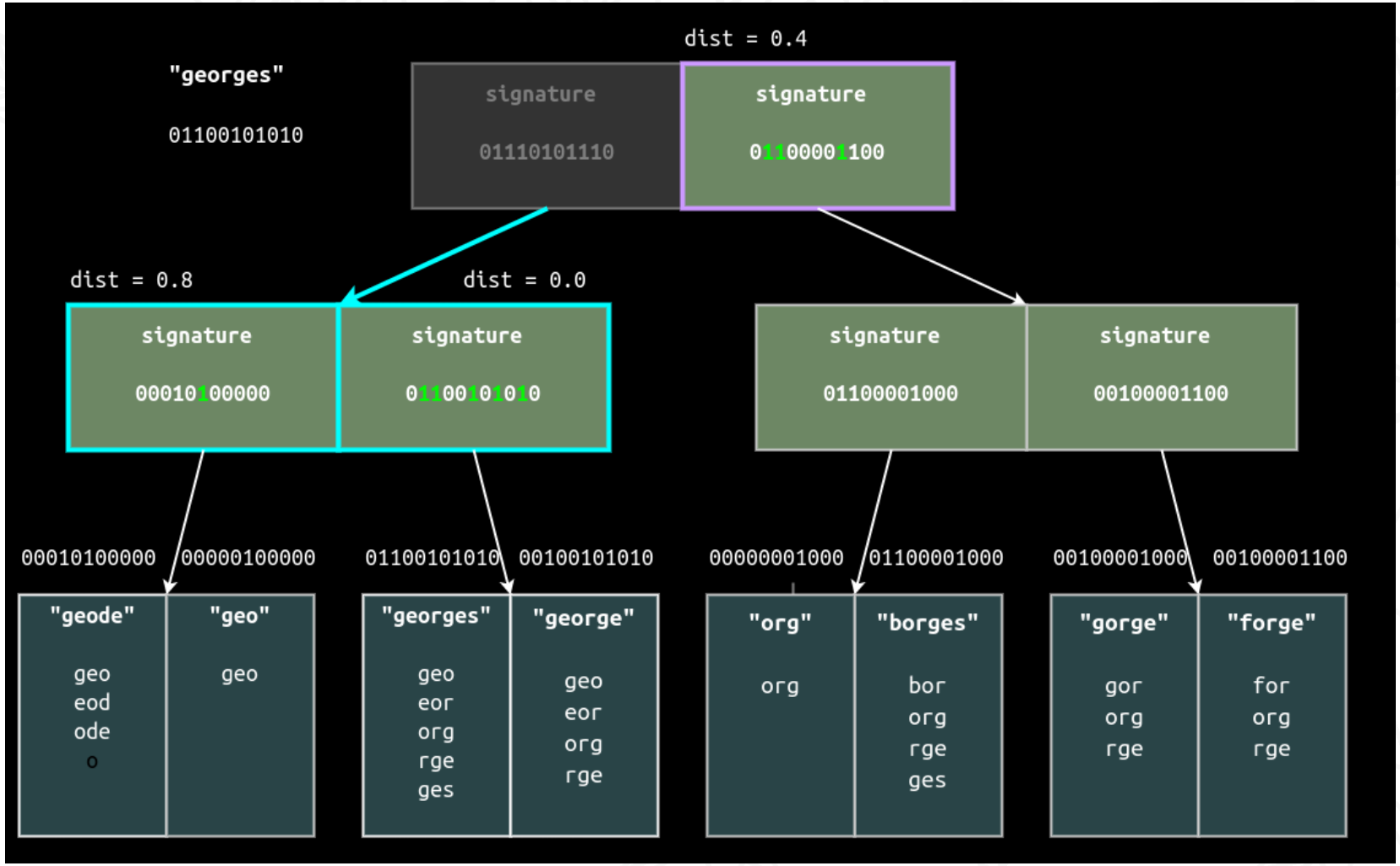
Step 1: calculate distance to root items



PostgresPro

KNN on trigrams

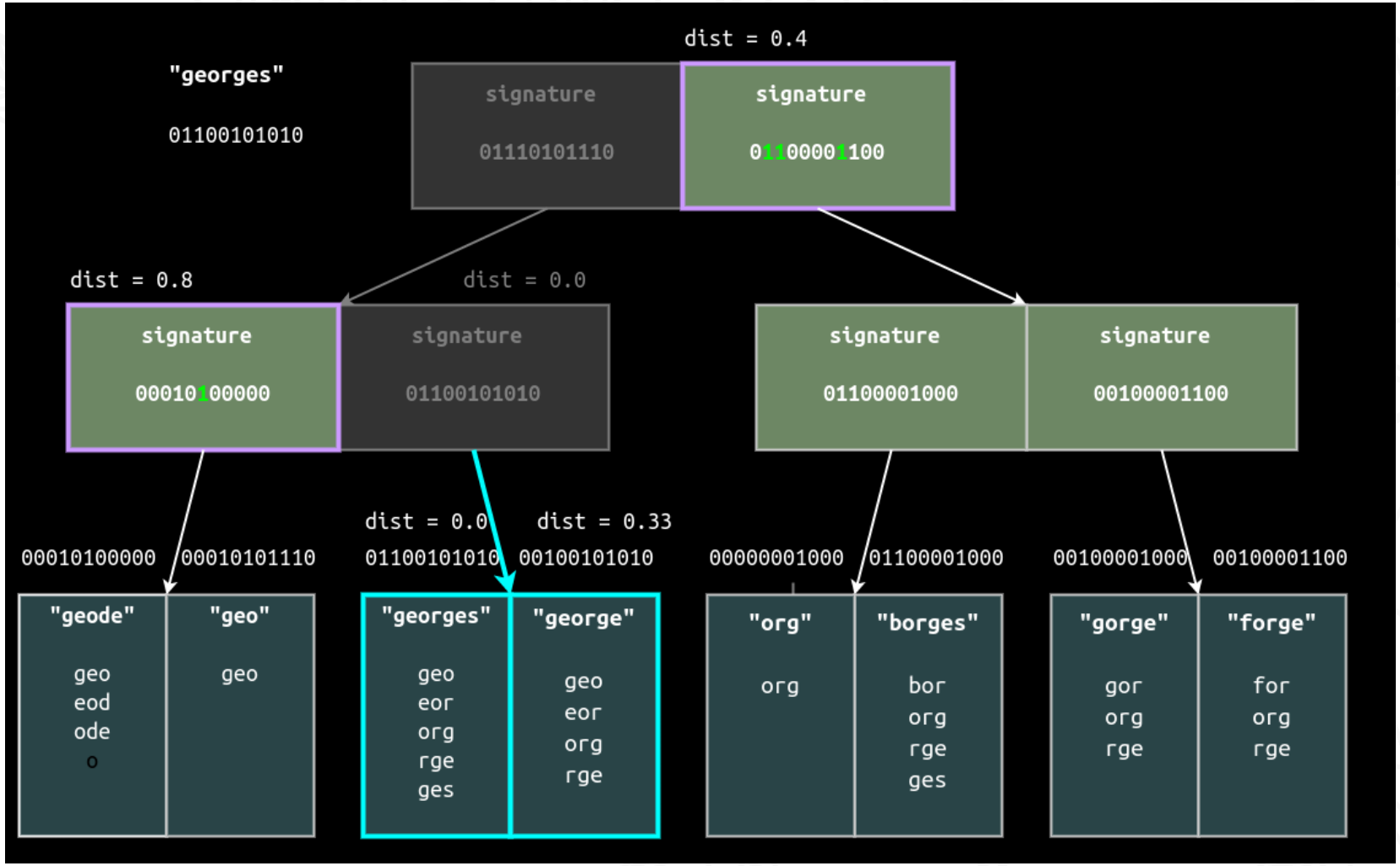
Step 2: descent to children of nearest root item, other items are queued



PostgresPro

KNN on trigrams

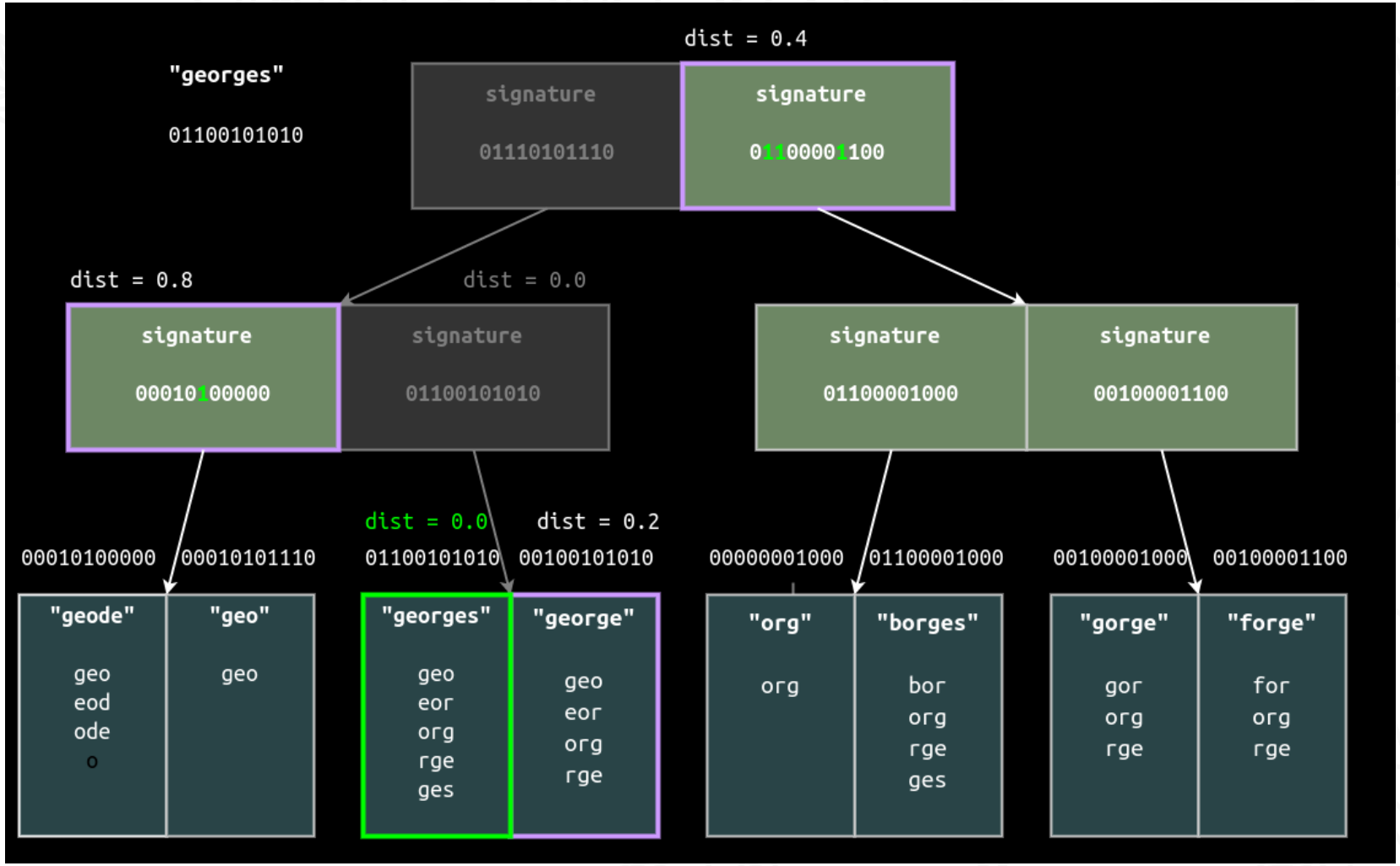
Step 3: descent to 2nd leaf page of the nearest inner item



PostgresPro

KNN on trigrams

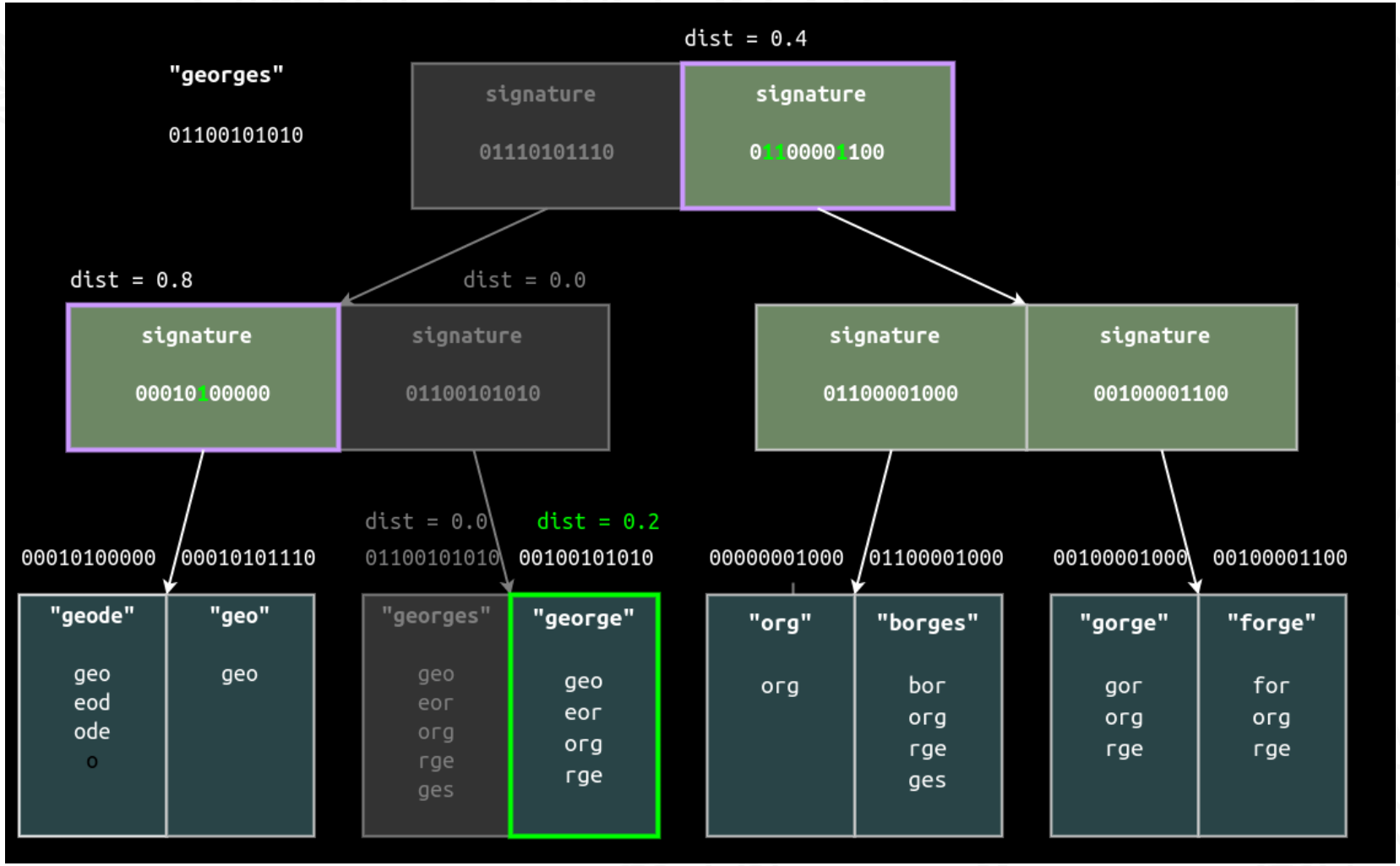
Step 4: emit first leaf item because it is the nearest in the queue



PostgresPro

KNN on trigrams

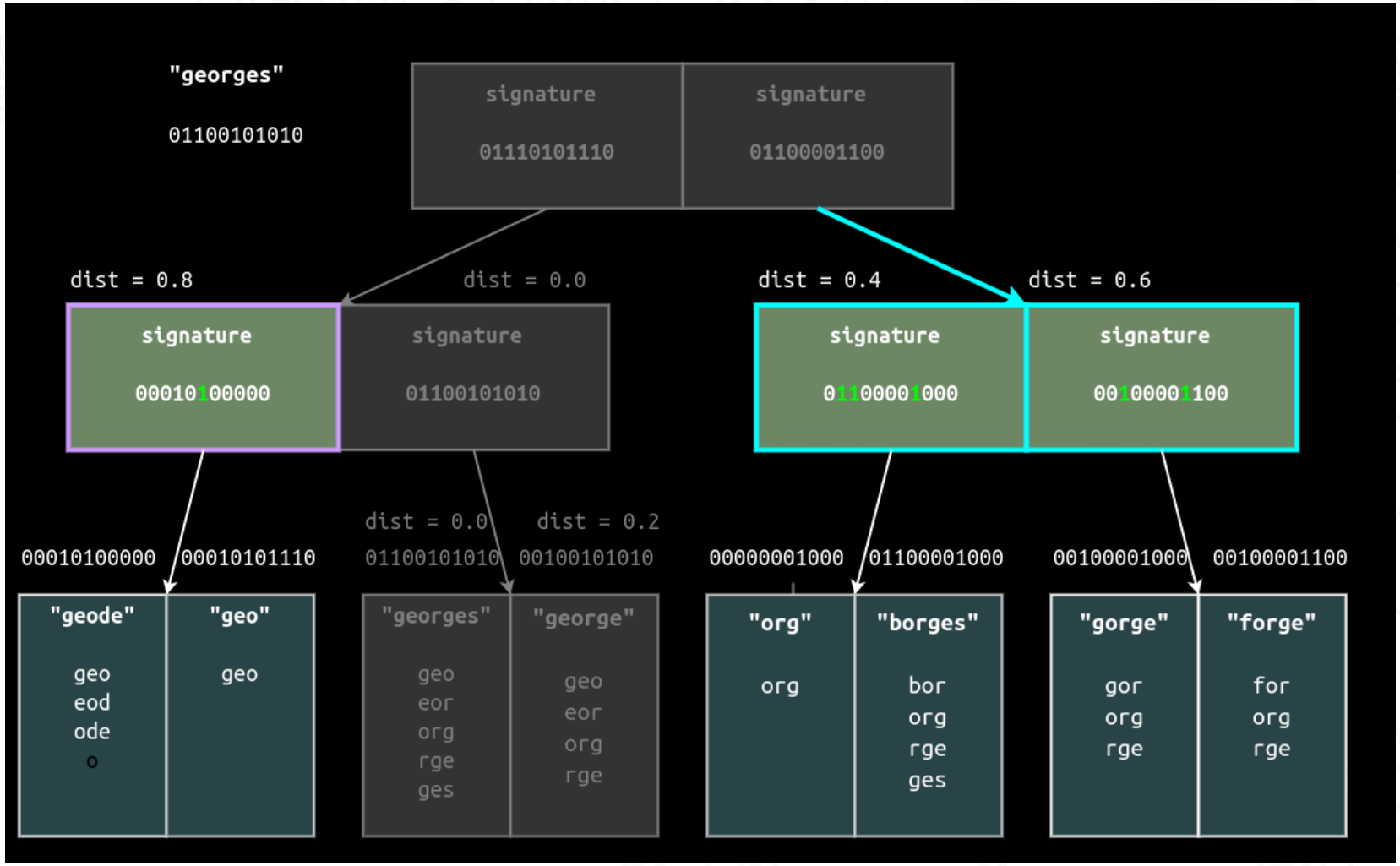
Step 5: emit second leaf item, it is also the nearest in the queue



PostgresPro

KNN on trigrams

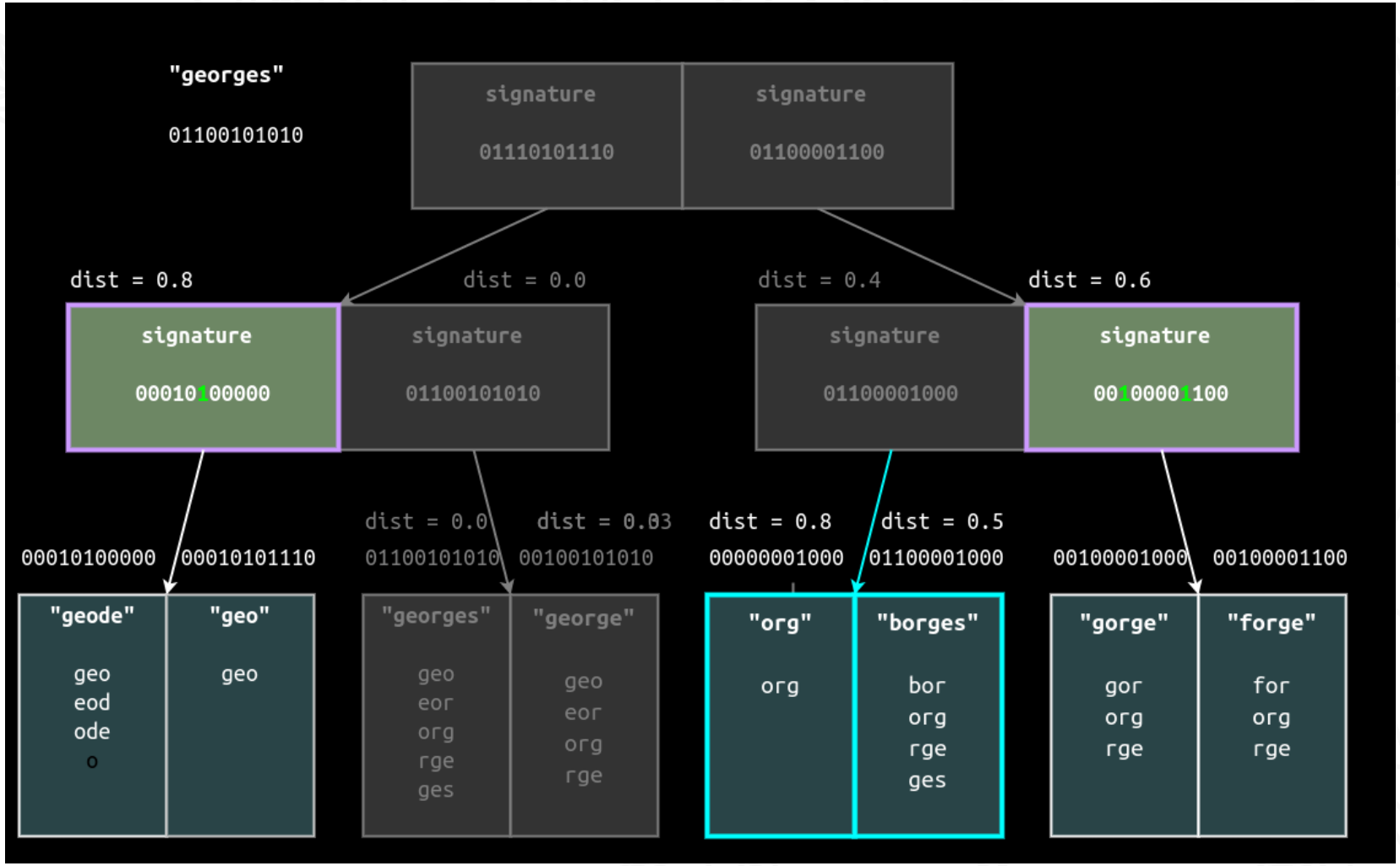
Step 6: descent to the remaining inner node



PostgresPro

KNN on trigrams

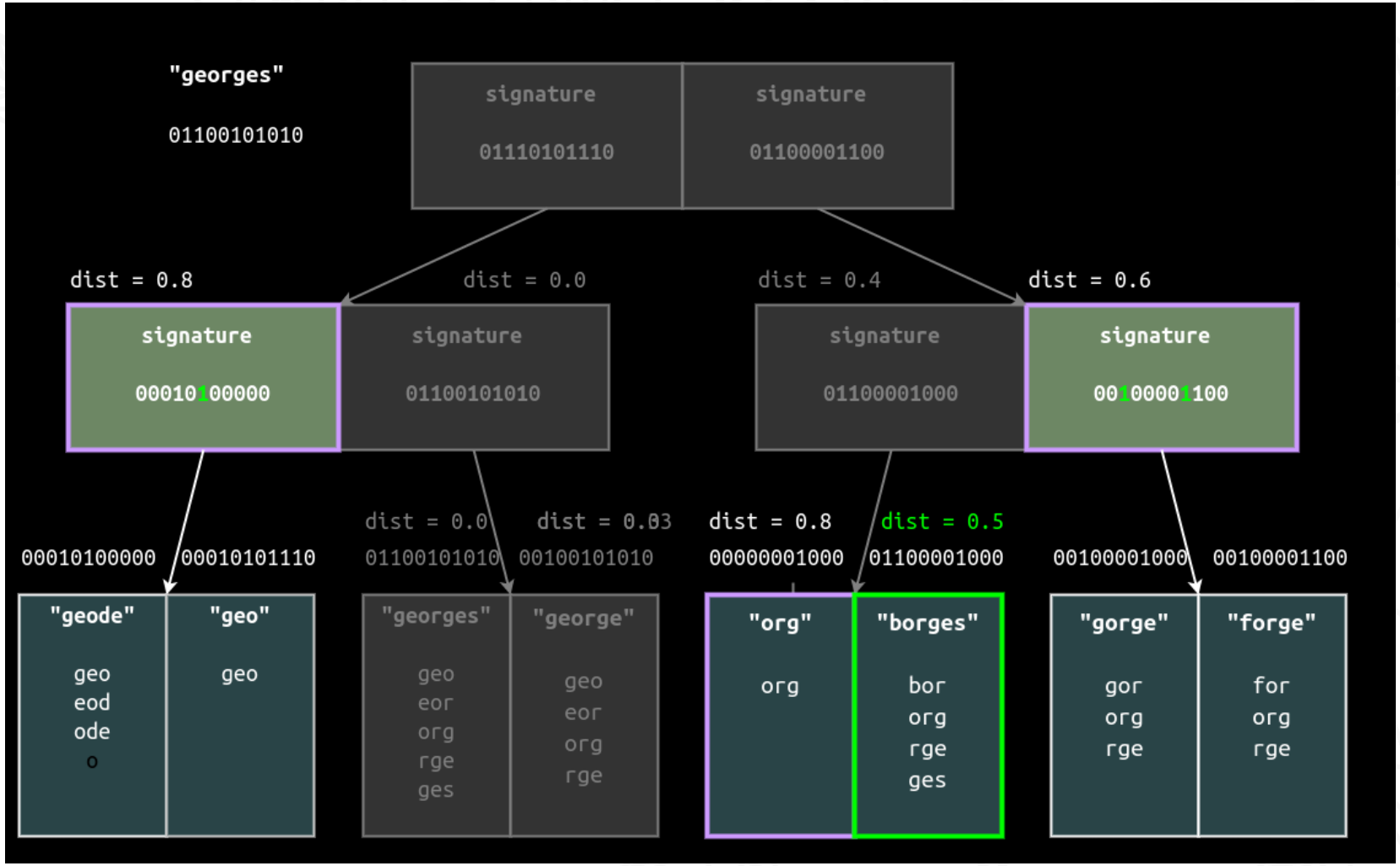
Step 7: descent to the 3rd leaf page



PostgresPro

KNN on trigrams

Step 8: emit the second leaf item of 3rd leaf page

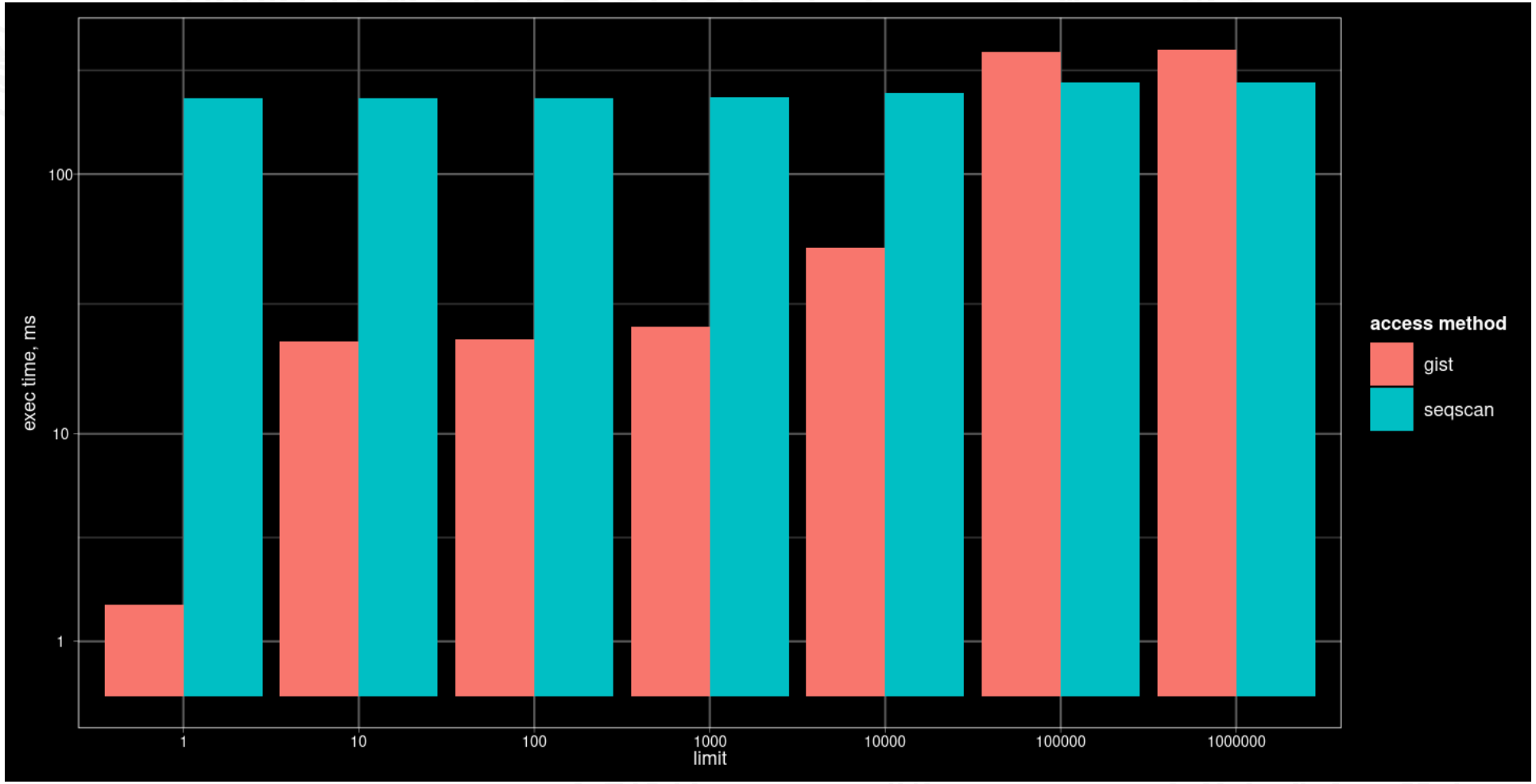


1: {R1,R2}
2:{I2,R2,I1}
3:{L3,L4,R2,I1}
4:L3,{L4,R2,I1}
5:L4,{R2,I1}
6:{I3,I4,I1}
7:{L6,I4,L5,I1}
8:L6,{I4,L5,I1}

PostgresPro

KNN on trigrams

Results (average time, 100 random sample words)



PostgresPro

Search Similar images - OK

Query:

Similar:

PostgresPro

Search Similar images - OK

Query:

Similar:

PostgresPro

Search Similar images — NOT OK

PostgresPro

Similar images: Preprocessing

```
CREATE TABLE pat AS (
        SELECT
                id,
                shuffle_pattern(pattern) AS pattern,
                pattern2signature(pattern) AS signature
        FROM (
                SELECT
                        id,
                        jpeg2pattern(data) AS pattern
                FROM
                        image
        ) x
);
CREATE INDEX pat_signature_idx ON pat USING gist (signature);
CREATE INDEX pat_id_idx ON pat(id);
```

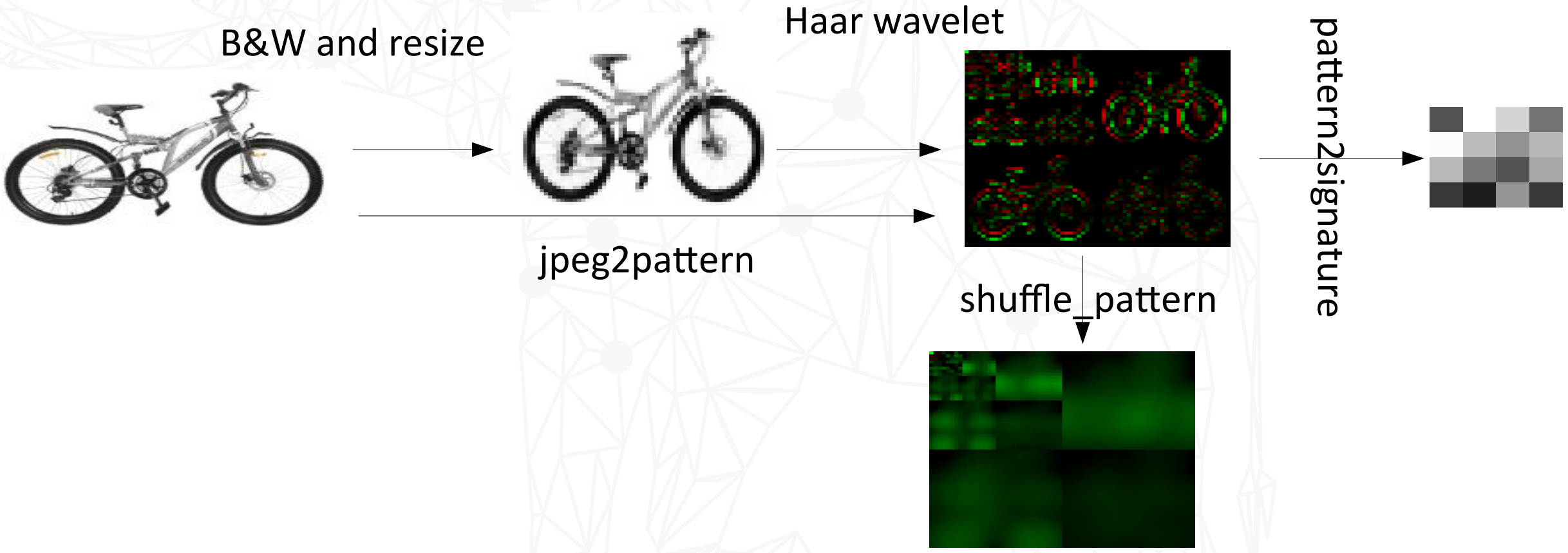
PostgresPro

Similar images: The Query

```
SELECT
        id,
        smlr
FROM
(
        SELECT
                id,
                pattern <-> (SELECT pattern FROM pat WHERE id = :id) AS smlr
        FROM pat
        WHERE id <> :id
        ORDER BY
                signature <-> (SELECT signature FROM pat WHERE id = :id)
        LIMIT 100
) x
ORDER BY x.smlr ASC
LIMIT 10
```

PostgresPro

Search similar: Inside



PostgresPro

K-NN Corner Case

- Corner case for Best First Strategy - all data are on the same distance from point Q !

```
CREATE table circle (id serial, p point, s int4);
INSEART INTO circle (p,s)
   SELECT point( p.x, p.y), (random()*1000)::int
   FROM ( select  t.x, sqrt(1- t.x*t.x)  AS y
        FROM ( select random() as x, generate_series(1,1000000) ) as t
) AS p;
CREATE index circle_p_idx ON circle USING gist(p);
ANALYZING circle;

Number of levels:          3
Number of pages:          8266
Number of leaf pages:     8201
```

PostgresPro

K-NN Corner Case

- Corner case for Best First Strategy - all data are on the same distance from point Q !

```
explain (analyze on, buffers, costs off) select * from circle order by (p <-> '(0,0)') asc limit 10;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit (actual time=252.242..252.244 rows=10 loops=1)
   Buffers: shared hit=7353
   ->  Sort (actual time=252.240..252.241 rows=10 loops=1)
         Sort Key: ((p <-> '(0,0)'::point))
         Sort Method: top-N heapsort  Memory: 26kB
         Buffers: shared hit=7353
         ->  Seq Scan on circle (actual time=0.015..140.504 rows=1000000 loops=1)
               Buffers: shared hit=7353
 Planning Time: 0.103 ms
 Execution Time: 252.272 ms
(10 rows)
```

K-NN Corner Case

- Corner case for Best First Strategy - all data are on the same distance from point Q !

```
explain (analyze on, buffers, costs off) select * from circle order by (p <-> '(0,0)') asc limit 10;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Limit (actual time=96.037..96.057 rows=10 loops=1)
   Buffers: shared hit=6073
   ->  Index Scan using circle_p_idx on circle (actual time=96.035..96.054 rows=10 loops=1)
         Order By: (p <-> '(0,0)'::point)
         Buffers: shared hit=6073
 Planning Time: 0.065 ms
 Execution Time: 125.737 ms VS 252.272 ms (seq scan)
(7 rows)
```

K-NN Corner Case

- Corner case for Best First Strategy - all data are on the same distance from point Q !

```
Limit (actual time=97.266..99.453 rows=10 loops=1)
  Buffers: shared hit=7431
  ->  Gather Merge (actual time=97.265..99.449 rows=10 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        Buffers: shared hit=7431
        ->  Sort (actual time=92.849..92.850 rows=10 loops=3)
              Sort Key: ((p <-> '(0,0)'::point))
              Sort Method: top-N heapsort  Memory: 26kB
              Buffers: shared hit=7431
              Worker 0:  Sort Method: top-N heapsort  Memory: 26kB
              Worker 1:  Sort Method: top-N heapsort  Memory: 25kB
              ->  Parallel Seq Scan on circle (actual time=0.013..53.711 rows=333333 loops=3)
                    Buffers: shared hit=7353
 Planning Time: 0.076 ms
 Execution Time: 99.477 ms  (seq scan parallel) < 125.737 (knn) ms VS 252.272 ms (seq scan)
(16 rows)
```

PostgresPro

TODO

- Modify opclass API to support different distance operators independently, without need to rewrite support function shared between them
- Support for reverse distance ordering, NULLS FIRST:
  `ORDER BY distance DESC [NULLS FIRST]`
  `ORDER BY distance ASC NULLS FIRST`
- Replace sortsupport opclass function with opclass parameter:
  `CREATE INDEX … USING (col1 opclass1 (sort=zorder_point_cmp), …)`
- Investigate hot spots on kNN map
- Try Z-order sorting for Quad-Tree SP-GiST indexes building

TODO – multiple distance operators support

- Currently, GiST needs support function to opclass, that will handle all possible kNN strategies. So, you can't easily add new ordering operator to opclass.

```
CREATE OPERATOR CLASS foo_ops …
  OPERATOR 1 <-> FOR ORDER BY bar_pos,
  OPERATOR 2 <@> FOR ORDER BY baz_ops,
  FUNCTION 8 gist_foo_distance …        -- should handle both operators
```

- The possible solution is support functions for strategies:

```
  FUNCTION 8 gist_foo_bar_distance FOR STRATEGY 1,
  FUNCTION 8 gist_foo_baz_distance FOR STRATEGY 2,
```

- The same applies to consistent function, used for search operators.
- In SP-GiST it is even worse: kNN support is hardcoded into consistent support function. At fist, we need to extract distance support function.

References

- Original K-NN post in hackers, we introduced >< operator, now it is <->
  - https://www.postgresql.org/message-id/4B0AC9E1.5050509@sigaev.ru
- Original K-NN talk at PGCON-2010
  - https://www.pgcon.org/2010/schedule/events/227.en.html
- Slides of this talk (PDF)
- Geonames database dump
  - http://download.geonames.org/export/dump/
- Events database dump
  - http://www.sai.msu.su/~megera/postgres/files/events.dump.gz
- Gevel extension – a tool for inspecting indexes
  - git://sigaev.ru/gevel

PostgresPro


ALL
YOU
NEED
POSTGRES
IS