Performance improvements and new usage scenarios for SPGiST access method

Pavel Borisov

www.postgrespro.com

About author

Pavel Borisov

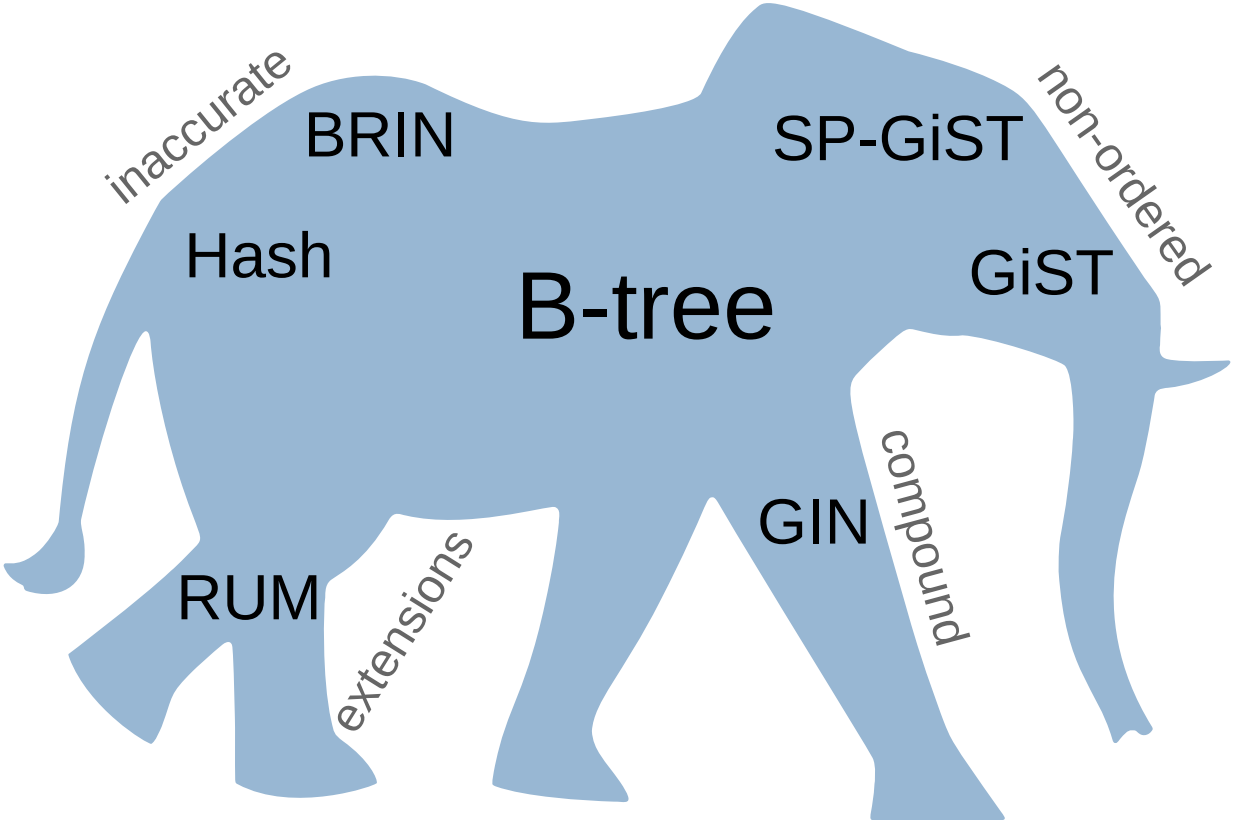
p.borisov@postgrespro.com

- Works on the PostgreSQL core features improvement:
  - indexing,
  - fast text search,
  - declarative partitioning
- Reviewer in the community of PostgreSQL developers.
- Involved in the development of Postgres Pro DBMS, the fork of a PostgresSQL.

There is no best index for every case

...that is why PostgreSQL has so many of them!

Index is based on an idea about data we have in the **column**.
(the data we request often consists of different column types)



- The most used index is B-tree but we don't have ordering for some data (spatial etc.)

Multicolumn index

All columns are index-ordered

```
CREATE INDEX idx ON db USING gist(key_column_1, key_column_2,
key_column_42);
```

PROS:

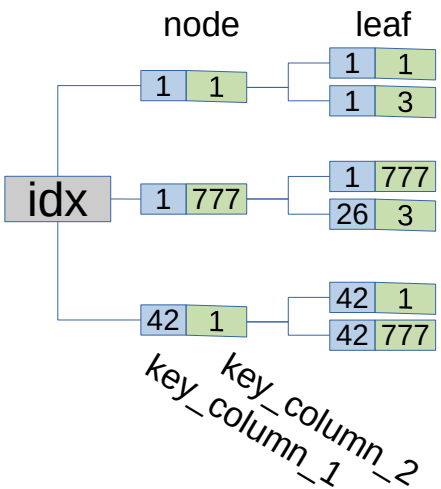
- index-only scans on several indexed columns
- can select with all indexed columns under WHERE clause

CONS:

- only columns of data types with the index opclass can be added
- index size and performance depend on columns order: lower cardinality column should be put first (we should know cardinality a priori)
- inefficient if the first columns have many unique values

Some indexes are single-column by design: SP-GiST

2-column index
(very simplified)



Covering index

Only key columns are index-ordered. The included columns are without order. They are present only in leaf tuples, not inside the search tree

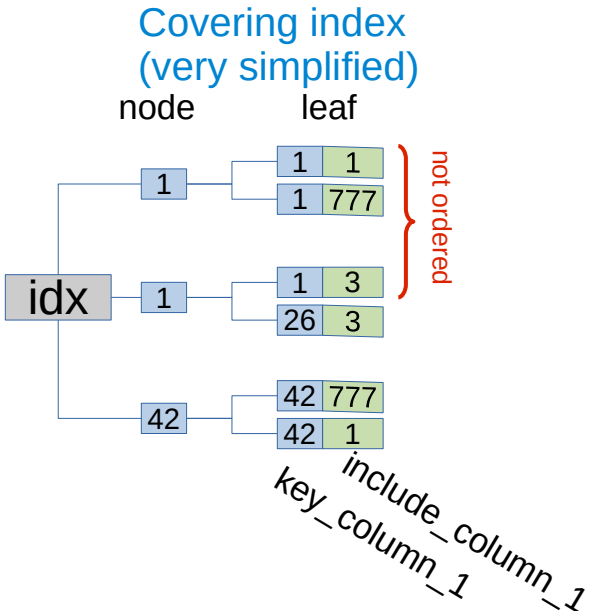
```
CREATE INDEX idx ON db USING spgist(key_column)
    INCLUDE(include_column_1, include_column_2);
```

PROS:

- Increases the number of columns for fast index-only SELECTs
- Included columns can be with any data type (index opclass is needed only for key columns)
- Is more lightweight than multicolumn one

CONS:

- Can not help for SELECTs with non-key columns under WHERE clause



Available in Btree, GiST, SP-GiST(not committed yet)

Generalized trees: GiST and SP-GiST

GiST

- MBRs are built around 'points' only
- Can be multicolumn and covering

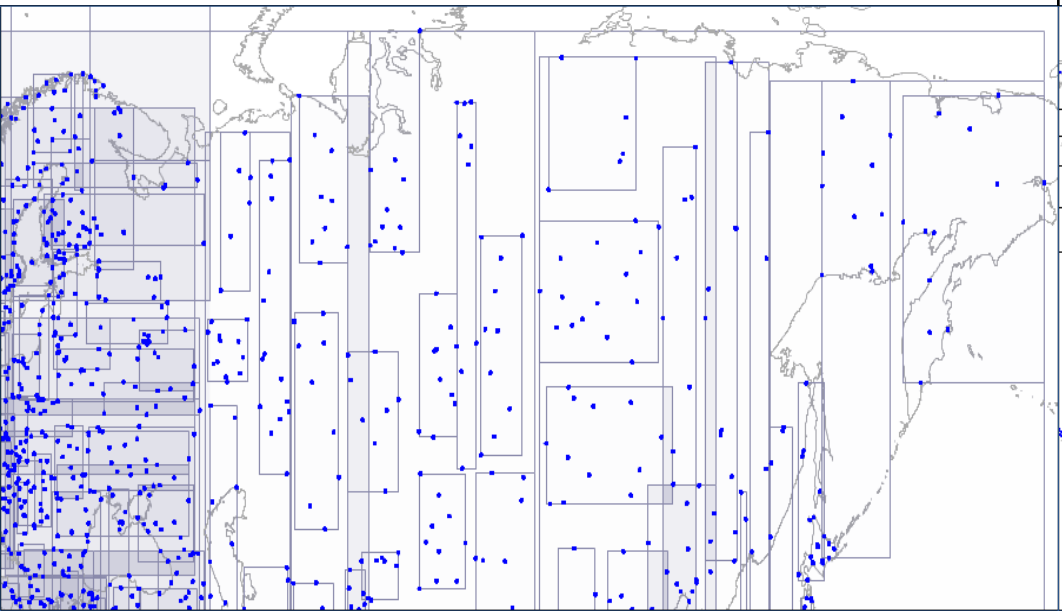
SP-GiST

- All space is to be partitioned
- Single-column
- Better Knn for many dimensions

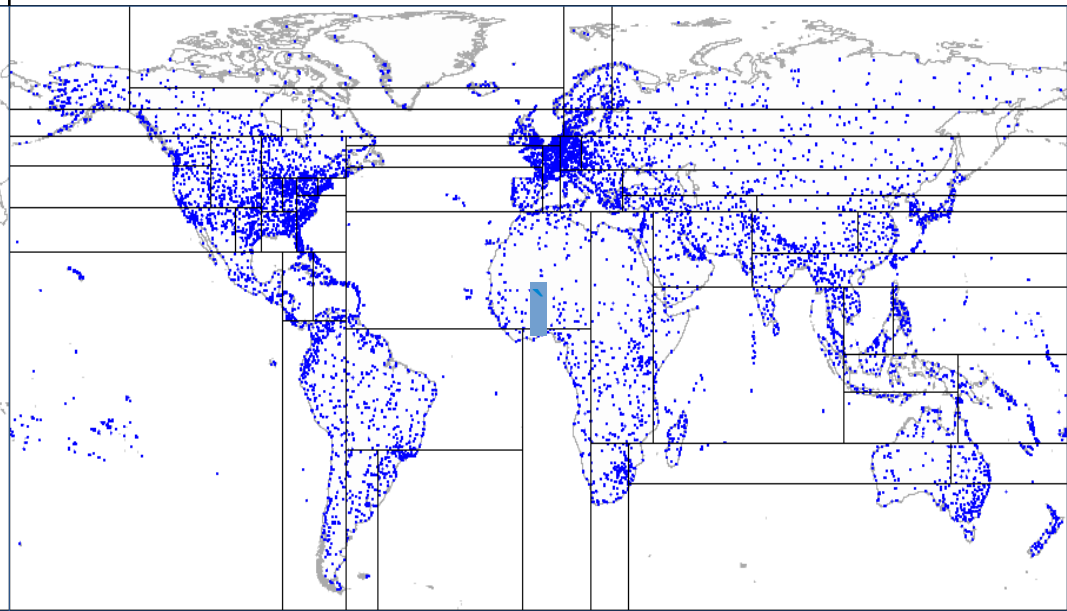
Do we really gain from the covering SP-GiST?

(it is still a proposal: https://commitfest.postgresql.org/31/2675/)

r-tree



k-D tree

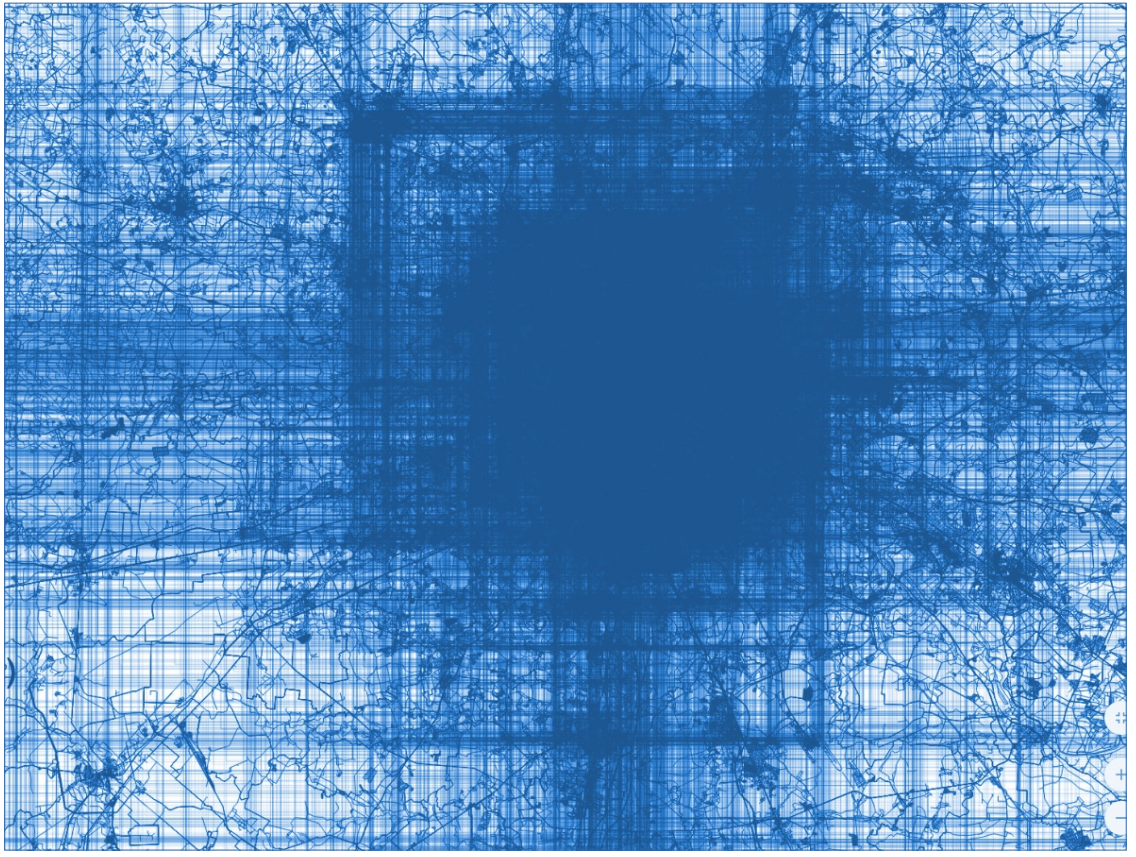


Benchmark GiST vs SP-GiST

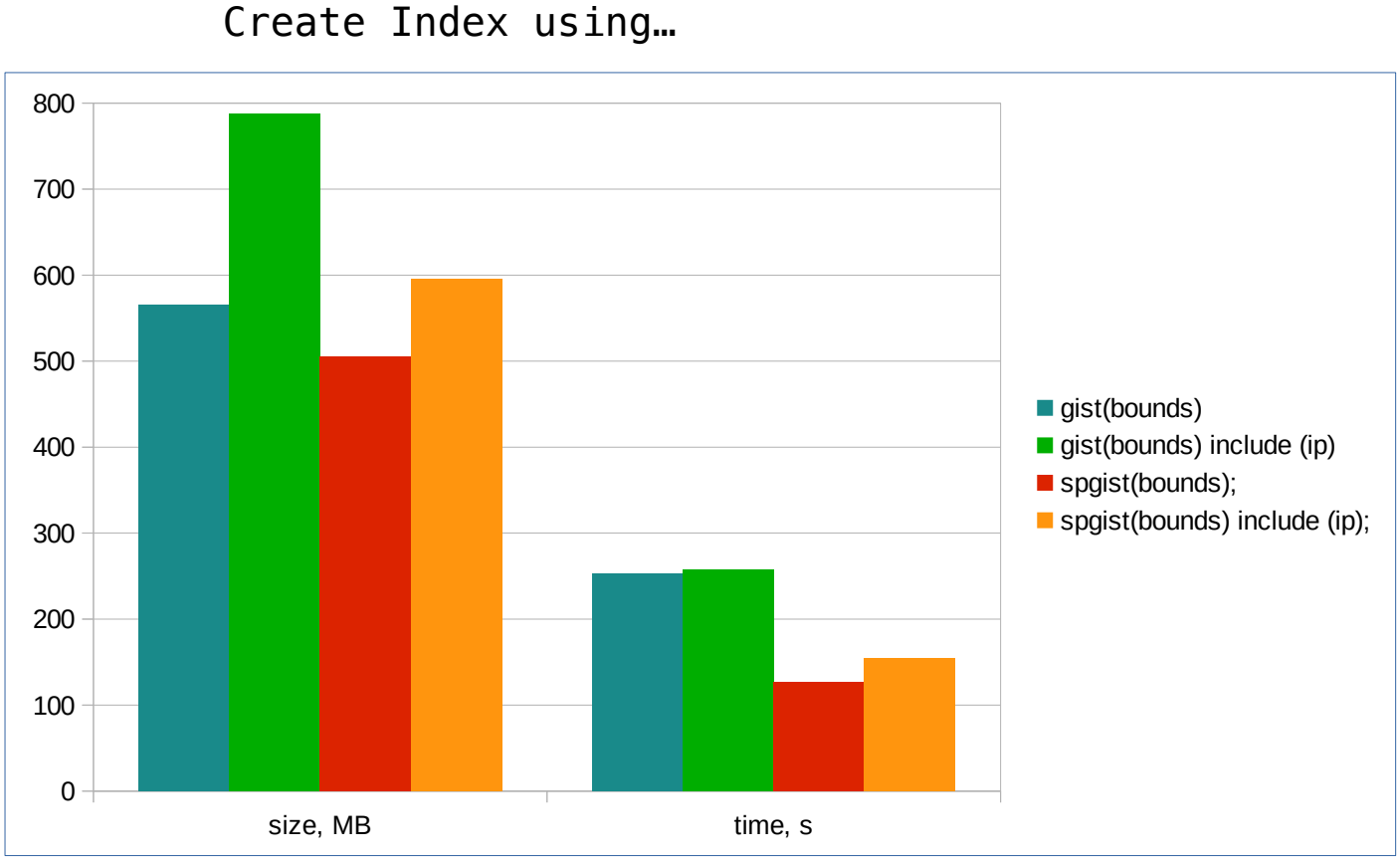
The problem: select all ip's and bounds, containing some given small region (or point).

```
Table "mowboxes_rnd"
  Column   |       Type
-----------+------------------
 ip        | cidr
 num       | integer
 center    | point
 bounds    | box
 tsbounds  | tsrange
 text      | character varying
```

8M rows, 5.6 Gb, random order



Benchmark GiST vs SP-GiST



Since v.14, 2-times faster ordered GiST index can be built for points, but not for boxes. (A. Borodin, https://commitfest.postgresql.org/29/2276/)

Select's Benchmark Setup

The problem: select all ip's and bounds, containing some given small region.

```
SELECT ip, bounds FROM mowboxes_rnd WHERE bounds @> small::box
```

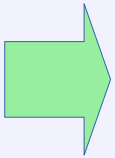
```
small is a box (0.001x0.001) size and with random coordinate in a square area (1x1) size
```

In a set of 10000 of **small**'s some will be in a

- densely populated area (up to many thousand rows result)
- more deserted region (from hundreds of rows result)

Next, we will look at the **query time vs number of rows** graph.

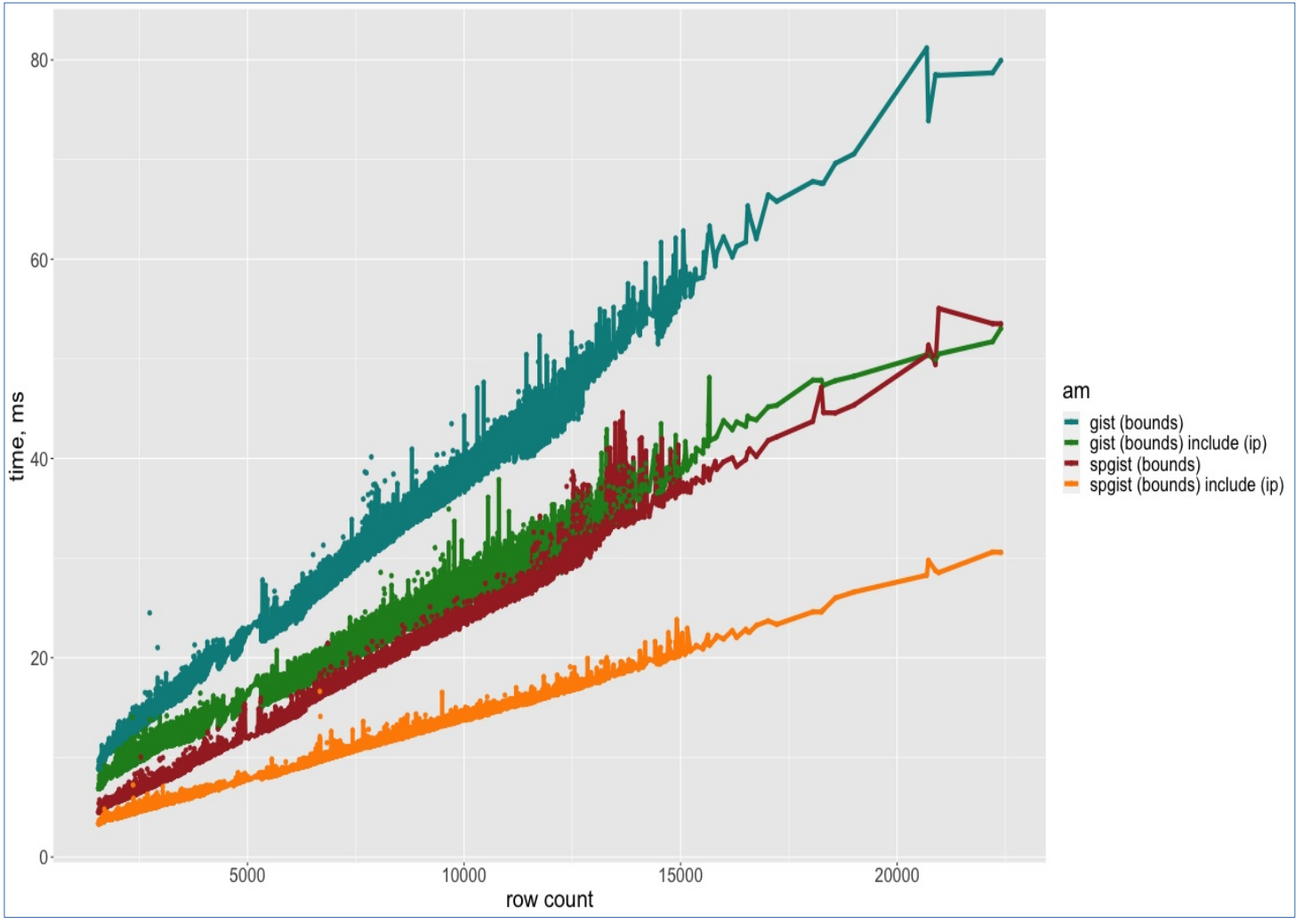
```
explain (ANALYZE, BUFFERS) SELECT bounds, ip FROM mowboxes_rnd WHERE bounds @> box(point(37.6, 55.7), point(37.601, 55.701));
QUERY PLAN
----------------------------------
 Bitmap Heap Scan on mowboxes_rnd
(cost=308.90..28778.60 rows=7804 width=39)
(actual time=13.489..33.109 rows=22203 loops=1)
   Recheck Cond: (bounds @> '(37.601,55.701),(37.6,55.7)'::box)
   Heap Blocks: exact=21851
   Buffers: shared hit=23528
   ->  Bitmap Index Scan on mowboxes_spgist
(cost=0.00..306.94 rows=7804 width=0)
(actual time=9.328..9.328 rows=22203 loops=1)
         Index Cond: (bounds @> '(37.601,55.701),(37.6,55.7)'::box)
         Buffers: shared hit=1677
 Planning Time: 0.179 ms
 Execution Time: 34.834 ms
```



```
QUERY PLAN
----------------------------------
 Index Only Scan using mowboxes_spgist_include on mowboxes_rnd
(cost=0.41..428.99 rows=7804 width=39)
 (actual time=0.059..13.729 rows=22203 loops=1)
   Index Cond: (bounds @> '(37.601,55.701),(37.6,55.7)'::box)
   Heap Fetches: 0
   Buffers: shared hit=16337
 Planning Time: 0.150 ms
 Execution Time: 15.620 ms
```

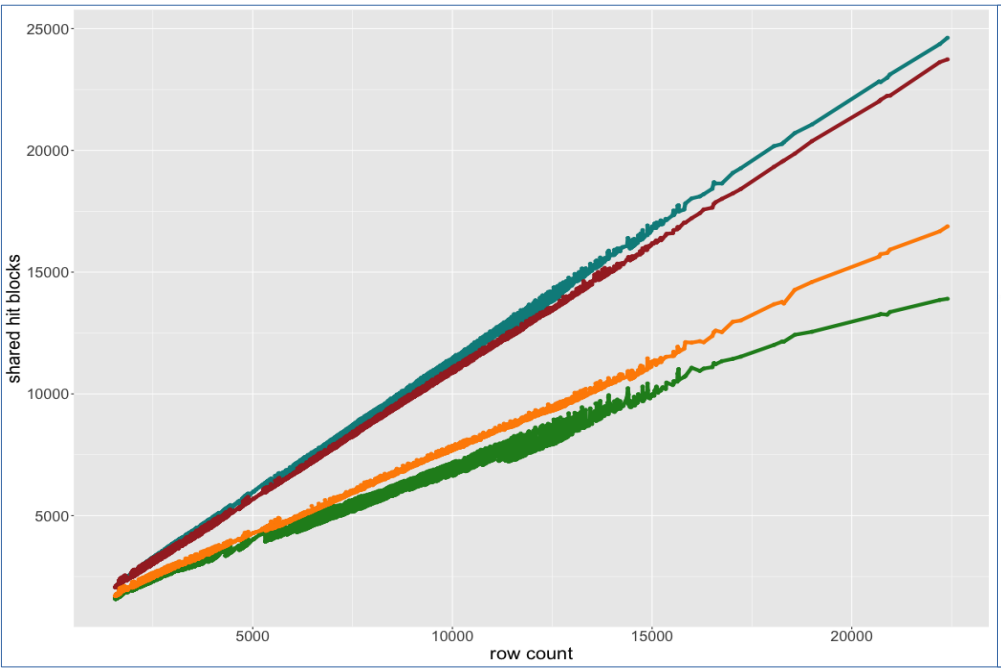
Benchmark GiST vs SP-GiST

Index and table in memory

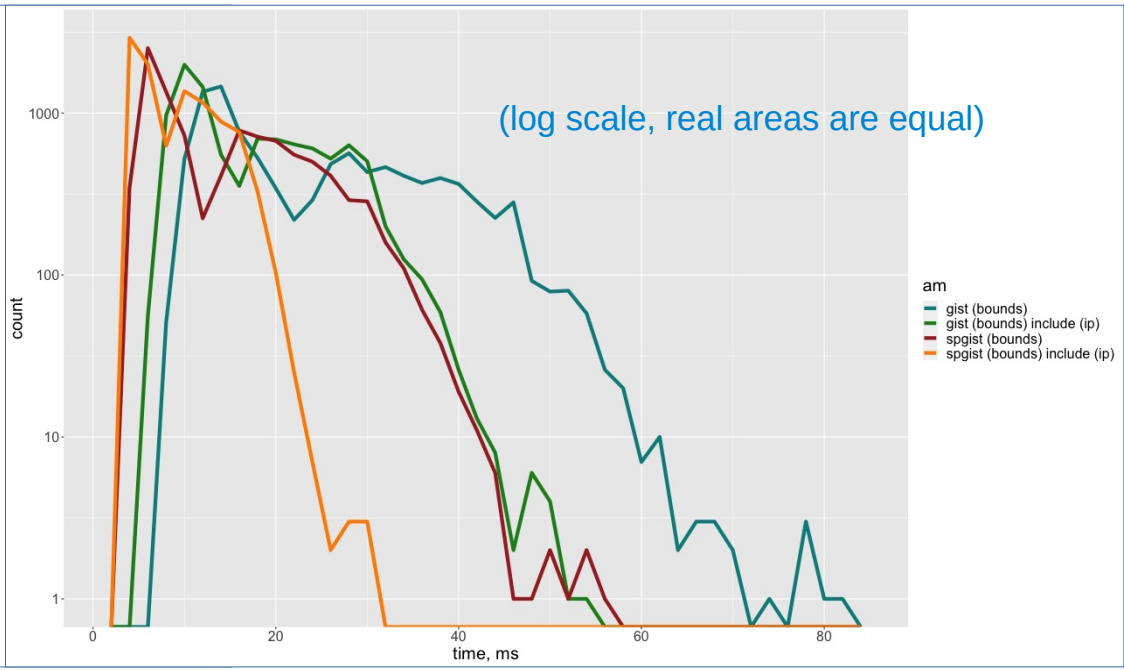


In-memory benchmark

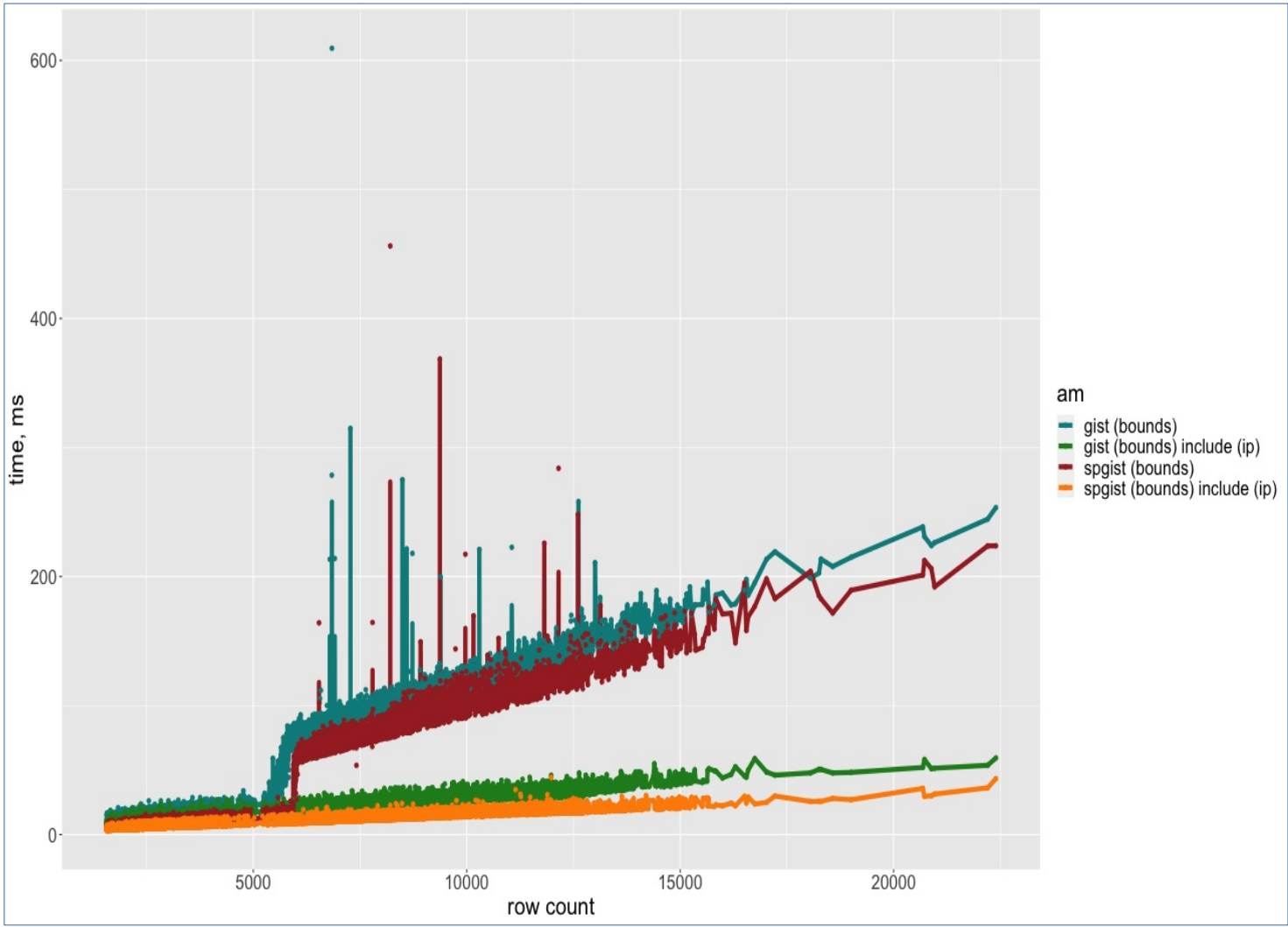
- Index-only scan is faster than bitmap scan even for in-memory case.
- SP-GiST is faster than GiST and is more lightweight.



Query time distribution



Benchmark GiST vs SP-GiST

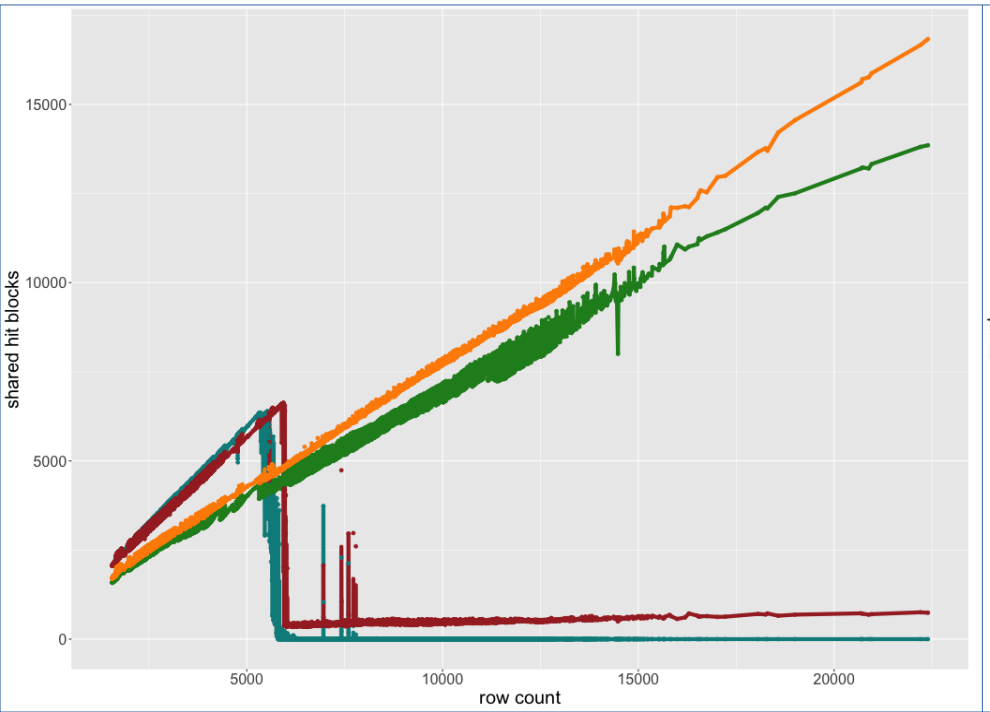


Index in memory.

Table needs disc access

Disc+memory benchmark

- Queries that return many results need disc access for bitmap scan and become many times slower.
- Index-only scan on covering index makes them much faster.



Query time distribution

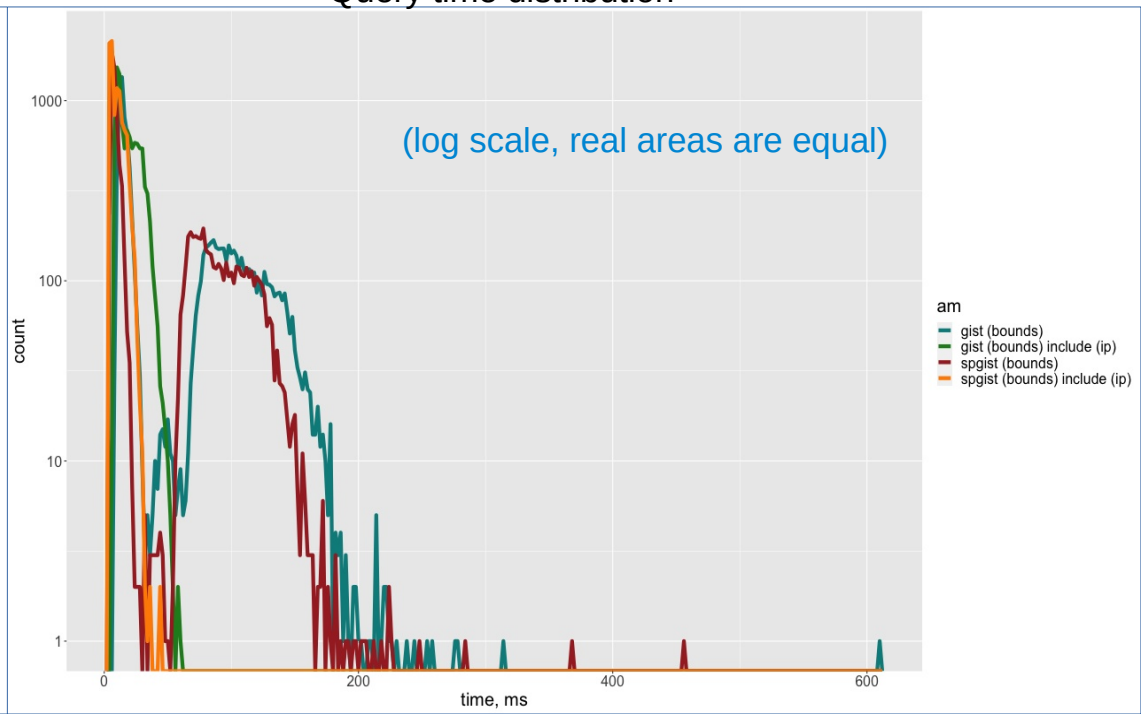


Table updates benchmark

When table records are updated, index should also be updated. This slows down the updates of indexed records.

Benchmark setup:

- Reads should not use SP-GiST, only updates → add primary key (id)
- The change in bounds coordinates is big enough for the record to jump to another branch in a k-D tree
- Zipfian distribution of the updated records is used to emulate real load

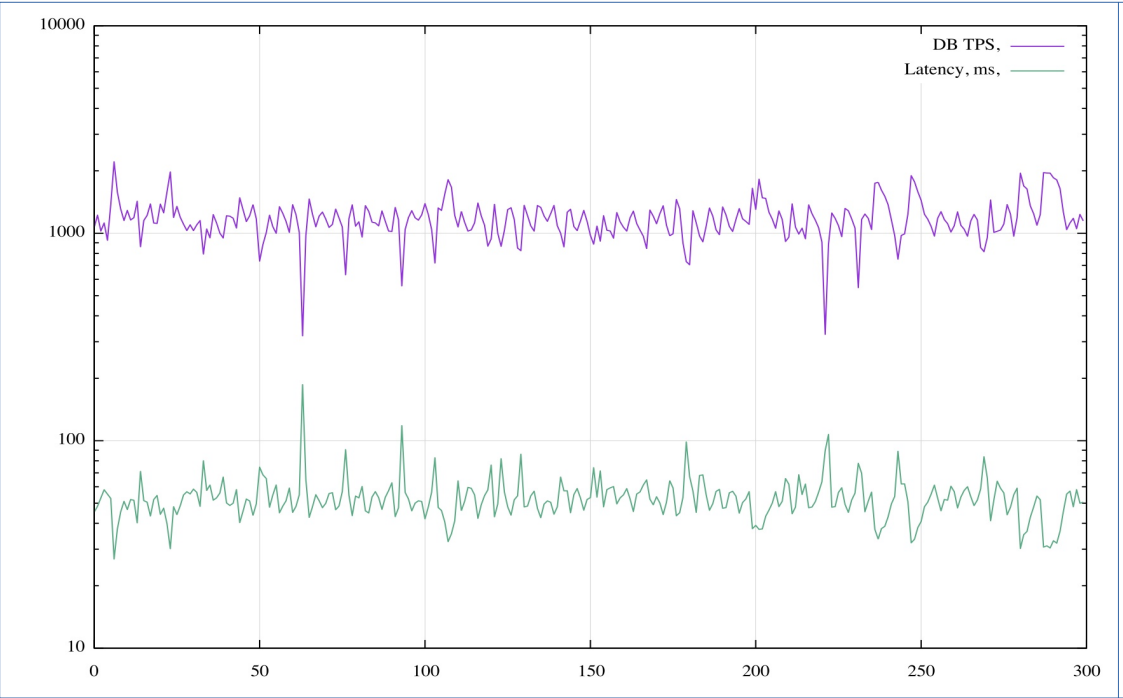
```
pgbench postgres -c 60 -j 60 -n -M prepared -T 300 -P 1 -f ./pgbench-update-zipfian.sql


\SET id random_zipfian(1, 8000000 * :scale, 2) # Zipfian factor up to 5
\SET delta ( random(1, 2000000)/1000000 - 1 )  # Coordinate shift in range +/- 1 degree
...
UPDATE mowboxes_rnd SET bounds = bounds + point(:delta,:delta) WHERE id = :id;
```

Table updates benchmark

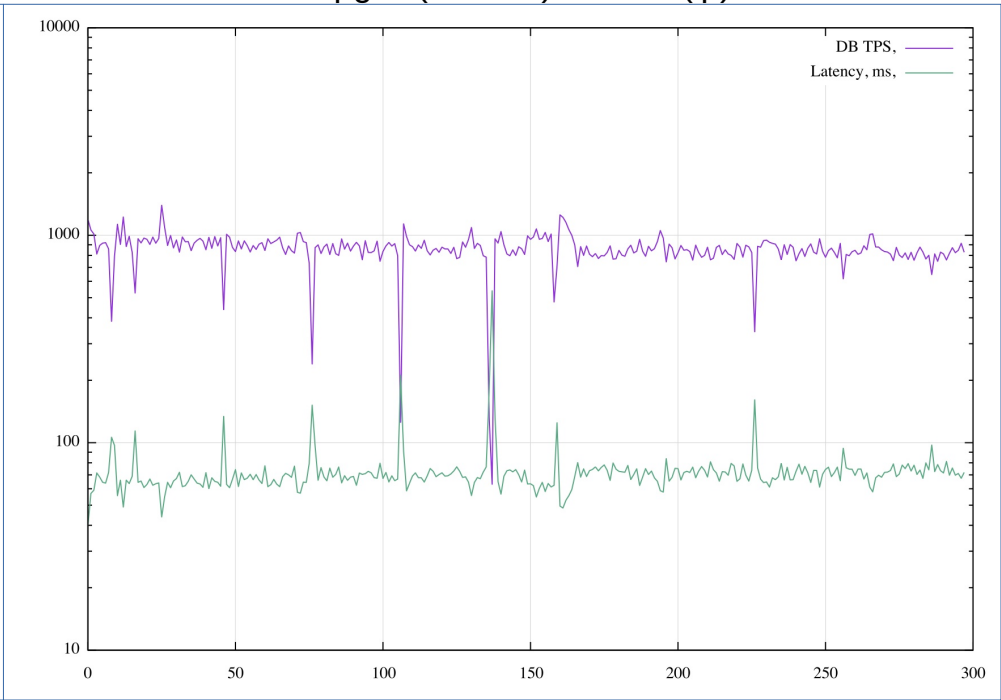
A little slowdown is seen as the index fits the shared memory
(no difference for covering and non-covering index)

NB: the test is limited to the laptop performance, generally results can differ more

no index



spgist (bounds) include (ip)



Use cases for SP-GiST covering

Consider covering SP-GiST when you expect:

- Selects with WHERE clause on single column:
  - of spatial data (2+ dimensional)
  - with the SP-GiST opclass and values close to unique (e.g. prefix tree on URL's etc.)
- Selects for several columns output

Also you may not care if you expect requests to the columns without the SP-GiST opclass.

`

Summary

- For spatial data SP-GiST index is often faster than GiST
- If index-only scans are possible, they can improve performance dramatically (this applies to any index)
- Included columns in a covering index allow data types without index-supported opclass
- Covering indexes generally have less overhead than multicolumn ones and work better when first column(s) has values close to unique.

Thanks! Your questions are welcome!