Performance improvements and new usage scenarios for SPGiST access method

Pavel Borisov
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 PostgreSQL.
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
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
<table>
<thead>
<tr>
<th>GiST</th>
<th>SP-GiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>• MBRs are built around 'points' only</td>
<td>• All space is to be partitioned</td>
</tr>
<tr>
<td>• Can be multicolumn and covering</td>
<td>• Single-column</td>
</tr>
<tr>
<td></td>
<td>• Better Knn for many dimensions</td>
</tr>
<tr>
<td></td>
<td><strong>Do we really gain from the covering SP-GiST?</strong></td>
</tr>
<tr>
<td></td>
<td>(it is still a proposal: <a href="https://commitfest.postgresql.org/31/2675/">https://commitfest.postgresql.org/31/2675/</a>)</td>
</tr>
</tbody>
</table>
Benchmark GiST vs SP-GiST

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

Table "mowboxes_rnd"

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>cidr</td>
</tr>
<tr>
<td>num</td>
<td>integer</td>
</tr>
<tr>
<td>center</td>
<td>point</td>
</tr>
<tr>
<td>bounds</td>
<td>box</td>
</tr>
<tr>
<td>tsbounds</td>
<td>tsrange</td>
</tr>
<tr>
<td>text</td>
<td>character varying</td>
</tr>
</tbody>
</table>

8M rows, 5.6 Gb, random order
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.
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.
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 diagram)
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

```sql
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
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!