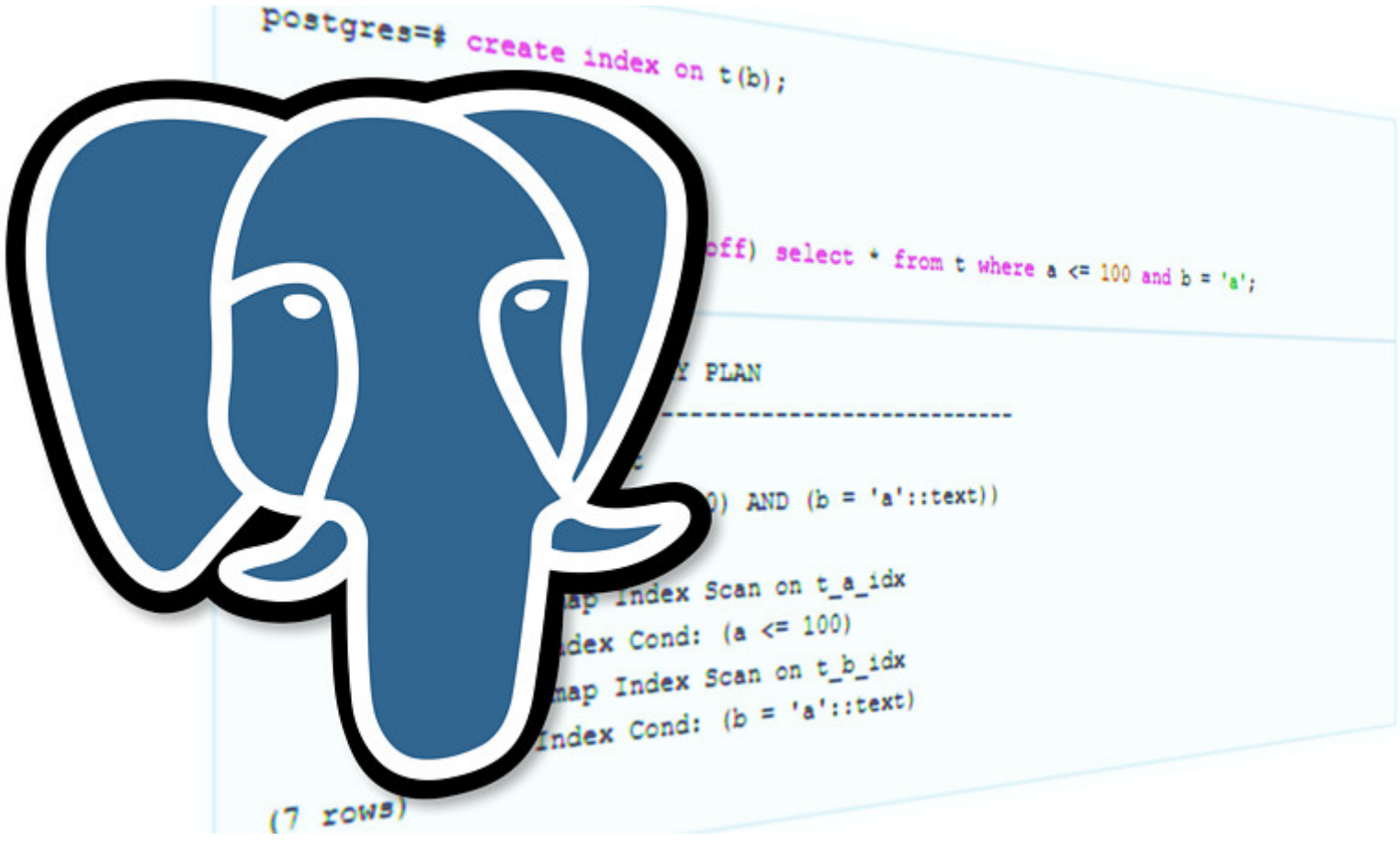
October 1, 2020 · PostgreSQL · By Egor Rogov

Indexes in PostgreSQL – 9 (BRIN)



Habr Article

In the previous articles we discussed PostgreSQL indexing engine, the interface of access methods, and the following methods: B-trees, GiST, SP-GiST, GIN, and RUM. The topic of this article is BRIN indexes.

BRIN

General concept

Unlike indexes with which we've already got acquainted, the idea of BRIN is to avoid looking through definitely unsuited rows rather than quickly find the matching ones. This is always an inaccurate index: it does not contain TIDs of table rows at all.

Simplistically, BRIN works fine for columns where values correlate with their physical location in the table. In other words, if a query without ORDER BY clause returns the column values virtually in the increasing or decreasing order (and there are no indexes on that column).

This access method was created in scope of Axle, the European project for extremely large analytical databases, with an eye on tables that are several terabyte or dozens of terabytes large. An important feature of BRIN that enables us to create indexes on such tables is a small size and minimal overhead costs of maintenance.

This works as follows. The table is split into *ranges* that are several pages large (or several blocks large, which is the same) - hence the name: Block Range Index, BRIN. The index stores *summary information* on the data in each range. As a rule, this is the minimal and maximal values, but it happens to be different, as shown further. Assume that a query is performed that contains the condition for a column; if the sought values do not get into the interval, the whole range can be skipped; but if they do get, all rows in all blocks will have to be looked through to choose the matching ones among them.

It will not be a mistake to treat BRIN not as an index, but as an accelerator of sequential scan. We can regard BRIN as an alternative to partitioning if we consider each range as a "virtual" partition.

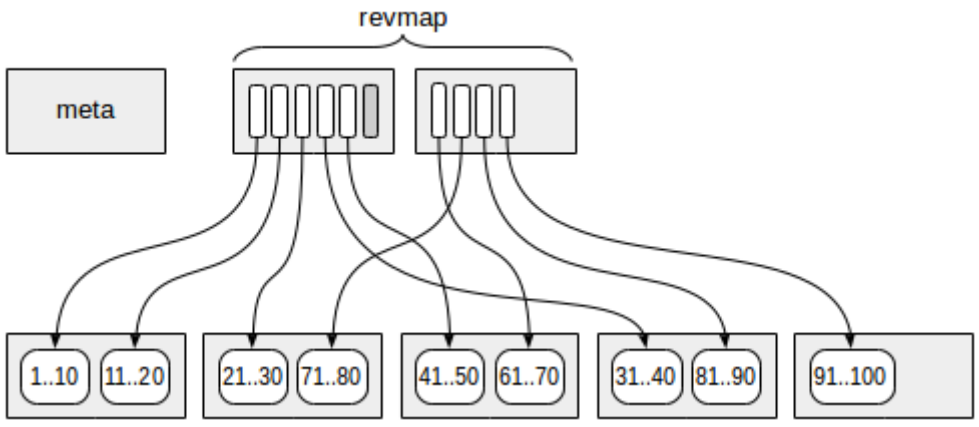
Now let's discuss the structure of the index in more detail.

Structure

The first (more exactly, zero) page contains the metadata.

Pages with the summary information are located at a certain offset from the metadata. Each index row on those pages contains summary information on one range.

Between the meta page and summary data, pages with the *reverse range map* (abbreviated as "revmap") are located. Actually, this is an array of pointers (TIDs) to the corresponding index rows.



For some ranges, the pointer in "revmap" can lead to no index row (one is marked in gray in the figure). In such a case, the range is considered to have no summary information yet.

Scanning the index

How is the index used if it does not contain references to table rows? This access method certainly cannot return rows TID by TID, but it can build a bitmap. There can be two kinds of bitmap pages: accurate, to the row, and inaccurate, to the page. It's an inaccurate bitmap that is used.

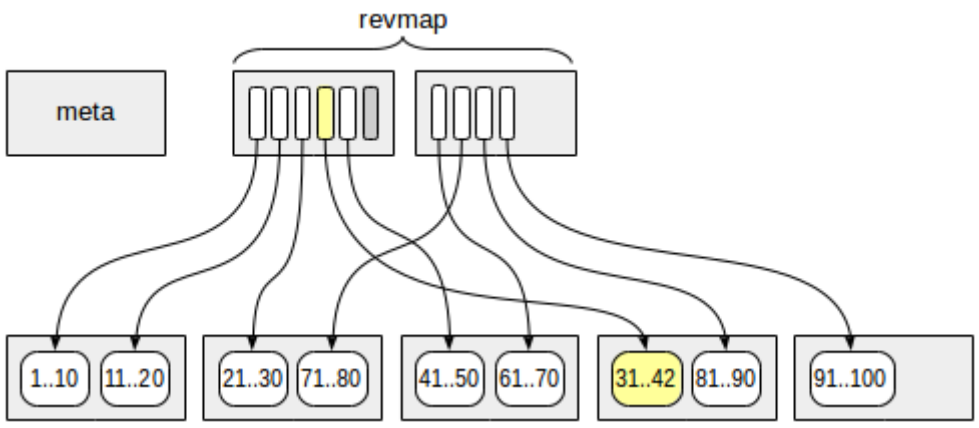
The algorithm is simple. The map of ranges is sequentially scanned (that is, the ranges are went through in the order of their location in the table). The pointers are used to determine index rows with summary information on each range. If a range does not contain the value sought, it is skipped, and if it can contain the value (or summary information is unavailable), all pages of the range are added to the bitmap. The resulting bitmap is then used as usual.

Updating the index

It is more interesting how the index is updated when the table is changed.

When **adding** a new version of a row to a table page, we determine which range it is contained in and use the map of ranges to find the index row with the summary information. All these are simple arithmetic operations. Let, for instance, the size of a range be four and on page 13, a row version with the value of 42 occur. The number of the range (starting with zero) is 13 / 4 = 3, therefore, in "revmap" we take the pointer with the offset of 3 (its order number is four).

The minimal value for this range is 31, and the maximal one is 40. Since the new value of 42 is out of the interval, we update the maximal value (see the figure). But if the new value is still within the stored limits, the index does not need to be updated.



All this relates to the situation when the new version of the page occurs in a range for which the summary information is available. When the index is created, the summary information is computed for all ranges available, but while the table is further expanded, new pages can occur that fall out of the limits. Two options are available here:

1. Usually the index is not updated immediately. This is not a big deal: as already mentioned, when scanning the index, the whole range will be looked through. Actual update is done during "vacuum", or it can be done manually by calling "brin_summarize_new_values" function.
2. If we create the index with "autosummarize" parameter, the update will be done immediately. But when pages of the range are populated with new values, updates can happen too often, therefore, this parameter is turned off by default.

When new ranges occur, the size of "revmap" can increase. Whenever the map, located between the meta page and summary data, needs to be extended by another page, existing row versions are moved to some other pages. So, the map of ranges is always located between the meta page and summary data.

When a row is **deleted**, ... nothing happens. We can notice that sometimes the minimal or maximal value will be deleted, in which case the interval could be reduced. But to detect this, we would have to read all values in the range, and this is costly.

The correctness of the index is not affected, but search may require looking through more ranges than is actually needed. In general, summary information can be manually recalculated for such a zone (by calling "brin_desummarize_range" and "brin_summarize_new_values" functions), but how can we detect such a need? Anyway, no conventional procedure is available to this end.

Finally, **updating a row** is just a deletion of the outdated version and addition of a new one.

Example

Let's try to build our own mini data warehouse for the data from tables of the demo database. Let's assume that for the purpose of BI reporting, a denormalized table is needed to reflect the flights departed from an airport or landed in the airport to the accuracy of a seat in the cabin. The data for each airport will be added to the table once a day, when it is midnight in the appropriate time zone. The data will be neither updated nor deleted.

The table will look as follows:

```
demo=# create table flights_bi(
  airport_code char(3),
  airport_coord point,         -- geo coordinates of airport
  airport_utc_offset interval, -- time zone
  flight_no char(6),           -- flight number
  flight_type text.            -- flight type: departure / arrival
  scheduled_time timestamptz,  -- scheduled departure/arrival time of flight
  actual_time timestamptz,     -- actual time of flight
  aircraft_code char(3),
  seat_no varchar(4),          -- seat number
  fare_conditions varchar(10), -- travel class
  passenger_id varchar(20),
  passenger_name text
);
```

We can simulate the procedure of loading the data using nested loops: an external one - by days (we will consider a large database, therefore 365 days), and an internal loop - by time zones (from UTC+02 to UTC+12). The query is pretty long and not of particular interest, so I'll hide it under the spoiler.

Simulation of loading the data to the storage

```
demo=# select count(*) from flights_bi;
```

```
  count
----------
 30517076
(1 row)
```

```
demo=# select pg_size_pretty(pg_total_relation_size('flights_bi'));
```

```
 pg_size_pretty
----------------
 4127 MB
(1 row)
```

We get 30 million rows and 4 GB. Not so large a size, but good enough for a laptop: sequential scan took me about 10 seconds.

On what columns should we create the index?

Since BRIN indexes have a small size and moderate overhead costs and updates happen infrequently, if any, a rare opportunity arises to build many indexes "just in case", for example, on all fields on which analyst users can create their ad-hoc queries. Won't come useful - never mind, but even an index that is not very efficient will work better than sequential scan for sure. Of course, there are fields on which it is absolutely useless to build an index; pure common sense will prompt them.

But it should be odd to limit ourselves to this piece of advice, therefore, let's try to state a more accurate criterion.

We've already mentioned that the data must somewhat correlate with its physical location. Here it makes sense to remember that PostgreSQL gathers table column statistics, which include the correlation value. The planner uses this value to select between a regular index scan and bitmap scan, and we can use it to estimate the applicability of BRIN index.

In the above example, the data is evidently ordered by days (by "scheduled_time", as well as by "actual_time" - there is no much difference). This is because when rows are added to the table (without deletions and updates), they are laid out in the file one after another. In the simulation of data loading we did not even use ORDER BY clause, therefore, dates within a day can be, in general, mixed up in an arbitrary way, but ordering must be in place. Let's check this:

```
demo=# analyze flights_bi;
demo=# select attname, correlation from pg_stats where tablename='flights_bi'
order by correlation desc nulls last;
```

```
      attname       | correlation
--------------------+-------------
 scheduled_time     |    0.999994
 actual_time        |    0.999994
 fare_conditions    |    0.796719
 flight_type        |    0.495937
 airport_utc_offset |    0.438443
 aircraft_code      |    0.172262
 airport_code       |   0.0543143
 flight_no          |   0.0121366
 seat_no            |  0.00568042
 passenger_name     |   0.0046387
 passenger_id       | -0.00281272
 airport_coord      |
(12 rows)
```

The value that is not too close to zero (ideally, near plus-minus one, as in this case), tells us that BRIN index will be appropriate.

The travel class "fare_condition" (the column contains three unique values) and type of the flight "flight_type" (two unique values) unexpectedly appeared to be in the second and third places. This is an illusion: formally the correlation is high, while actually on several successive pages all possible values will be encountered for sure, which means that BRIN won't do any good.

The time zone "airport_utc_offset" goes next: in the considered example, within a day cycle, airports are ordered by time zones "by construction".

It's these two fields, time and time zone, that we will further experiment with.

Possible weakening of the correlation

The correlation that is place "by construction" can be easily weakened when the data is changed. And the matter here is not in a change to a particular value, but in the structure of the multiversion concurrency control: the outdated row version is deleted on one page, but a new version may be inserted wherever free space is available. Due to this, whole rows get mixed up during updates.

We can partially control this effect by reducing the value of "fillfactor" storage parameter and this way leaving free space on a page for future updates. But do we want to increase the size of an already huge table? Besides, this does not resolve the issue of deletions: they also "set traps" for new rows by freeing the space somewhere inside existing pages. Due to this, rows that otherwise would get to the end of file, will be inserted at some arbitrary place.

By the way, this is a curious fact. Since BRIN index does not contain references to table rows, its availability should not hinder HOT updates at all, but it does.

So, BRIN is mainly designed for tables of large and even huge sizes that are either not updated at all or updated very slightly. However, it perfectly copes with the addition of new rows (to the end of the table). This is not surprising since this access method was created with a view to data warehouses and analytical reporting.

What size of a range do we need to select?

If we deal with a terabyte table, our main concern when selecting the size of a range will probably be not to make BRIN index too large. However, in our situation, we can afford analyzing data more accurately.

To do this, we can select unique values of a column and see on how many pages they occur. Localization of the values increases the chances of success in applying BRIN index. Moreover, the found number of pages will prompt the size of a range. But if the value is "spread" over all pages, BRIN is useless.

Of course, we should use this technique keeping a watchful eye on an internal structure of the data. For example, it makes no sense to consider each date (more exactly, a timestamp, also including time) as a unique value - we need to round it to days.

Technically, this analysis can be done by looking at the value of the hidden "ctid" column, which provides the pointer to a row version (TID): the number of the page and the number of the row inside the page. Unfortunately, there is no conventional technique to decompose TID into its two components, therefore, we have to cast types through the text representation:

```
demo=# select min(numblk), round(avg(numblk)) avg, max(numblk)
from (
  select count(distinct (ctid::text::point)[0]) numblk
  from flights_bi
  group by scheduled_time::date
) t;
```

```
 min  | avg  | max
------+------+------
 1192 | 1500 | 1796
(1 row)
```

```
demo=# select relpages from pg_class where relname = 'flights_bi';
```

```
 relpages
----------
   528172
(1 row)
```

We can see that each day is distributed across pages pretty evenly, and days are slightly mixed up with each other (1500 × 365 = 547500, which is only a little larger than the number of pages in the table 528172). This is actually clear "by construction" anyway.

Valuable information here is a specific number of pages. With a conventional range size of 128 pages, each day will populate 9–14 ranges. This seems realistic: with a query for a specific day, we can expect an error around 10%.

Let's try:

```
demo=# create index on flights_bi using brin(scheduled_time);
```

The size of the index is as small as 184 KB:

```
demo=# select pg_size_pretty(pg_total_relation_size('flights_bi_scheduled_time_idx'));
```

```
 pg_size_pretty
----------------
 184 kB
(1 row)
```

In this case, it hardly makes sense to increase the size of a range at the cost of losing the accuracy. But we can reduce the size if required, and the accuracy will, on the contrary, increase (along with the size of the index).

Now let's look at time zones. Here we cannot use a brute-force approach either. All values should be divided by the number of day cycles instead since the distribution is repeated within each day. Besides, since there are few time zones only, we can look at the entire distribution:

```
demo=# select airport_utc_offset, count(distinct (ctid::text::point)[0])/365 numblk
from flights_bi
group by airport_utc_offset
order by 2;
```

```
 airport_utc_offset | numblk
--------------------+--------
 12:00:00           |      6
 06:00:00           |      8
 02:00:00           |     10
 11:00:00           |     13
 08:00:00           |     28
 09:00:00           |     29
 10:00:00           |     40
 04:00:00           |     47
 07:00:00           |    110
 05:00:00           |    231
 03:00:00           |    932
(11 rows)
```

On average, the data for each time zone populates 133 pages a day, but the distribution is highly non-uniform: Petropavlovsk-Kamchatskiy and Anadyr fit as few as six pages, while Moscow and its neighborhood require hundreds of them. The default size of a range is no good here; let's, for example, set it to four pages.

```
demo=# create index on flights_bi using brin(airport_utc_offset) with (pages_per_range=4);
demo=# select pg_size_pretty(pg_total_relation_size('flights_bi_airport_utc_offset_idx'));
```

```
 pg_size_pretty
----------------
 6528 kB
(1 row)
```

Execution plan

Let's look at how our indexes work. Let's select some day, say, a week ago (in the demo database, "today" is determined by "booking.now" function):

```
demo=# \set d 'bookings.now()::date - interval \'7 days\''
demo=# explain (costs off,analyze)
  select *
  from flights_bi
  where scheduled_time >= :d and scheduled_time < :d + interval '1 day';
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi (actual time=10.282..94.328 rows=83954 loops=1)
   Recheck Cond: ...
   Rows Removed by Index Recheck: 12045
   Heap Blocks: lossy=1664
   ->  Bitmap Index Scan on flights_bi_scheduled_time_idx
       (actual time=3.013..3.013 rows=16640 loops=1)
         Index Cond: ...
 Planning time: 0.375 ms
 Execution time: 97.805 ms
```

As we can see, the planner used the index created. How accurate is it? The ratio of the number of rows that meet the query conditions ("rows" of Bitmap Heap Scan node) to the total number of rows returned using the index (the same value plus Rows Removed by Index Recheck) tells us about this. In this case 83954 / (83954 + 12045), which is approximately 90%, as

expected (this value will change from one day to another).

Where does the 16640 number in "actual rows" of Bitmap Index Scan node originate from? The thing is that this node of the plan builds an inaccurate (page-by-page) bitmap and is completely unaware of how many rows the bitmap will touch, while something needs to be shown. Therefore, in despair one page is assumed to contain 10 rows. The bitmap contains 1664 pages in total (this value is shown in "Heap Blocks: lossy=1664"); so, we just get 16640. Altogether, this is a senseless number, which we should not pay attention to.

How about airports? For example, let's take the time zone of Vladivostok, which populates 28 pages a day:

```
demo=# explain (costs off,analyze)
  select *
  from flights_bi
  where airport_utc_offset = interval '8 hours';
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi (actual time=75.151..192.210 rows=587353 loops=1)
   Recheck Cond: (airport_utc_offset = '08:00:00'::interval)
   Rows Removed by Index Recheck: 191318
   Heap Blocks: lossy=13380
   ->  Bitmap Index Scan on flights_bi_airport_utc_offset_idx
       (actual time=74.999..74.999 rows=133800 loops=1)
         Index Cond: (airport_utc_offset = '08:00:00'::interval)
 Planning time: 0.168 ms
 Execution time: 212.278 ms
```

The planner again uses the BRIN index created. The accuracy is worse (about 75% in this case), but this is expected since the correlation is lower.

Several BRIN indexes (just like any other ones) can certainly be joined at the bitmap level. For example, the following is the data on the selected time zone for a month (notice "BitmapAnd" node):

```
demo=# \set d 'bookings.now()::date - interval \'60 days\''
demo=# explain (costs off,analyze)
  select *
  from flights_bi
  where scheduled_time >= :d and scheduled_time < :d + interval '30 days'
    and airport_utc_offset = interval '8 hours';
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi (actual time=62.046..113.849 rows=48154 loops=1)
   Recheck Cond: ...
   Rows Removed by Index Recheck: 18856
   Heap Blocks: lossy=1152
   ->  BitmapAnd (actual time=61.777..61.777 rows=0 loops=1)
         ->  Bitmap Index Scan on flights_bi_scheduled_time_idx
             (actual time=5.490..5.490 rows=435200 loops=1)
               Index Cond: ...
         ->  Bitmap Index Scan on flights_bi_airport_utc_offset_idx
             (actual time=55.068..55.068 rows=133800 loops=1)
               Index Cond: ...
 Planning time: 0.408 ms
 Execution time: 115.475 ms
```

Comparison with B-tree

What if we create regular B-tree index on the same field as BRIN?

```
demo=# create index flights_bi_scheduled_time_btree on flights_bi(scheduled_time);
demo=# select pg_size_pretty(pg_total_relation_size('flights_bi_scheduled_time_btree'));
```

```
 pg_size_pretty
----------------
 654 MB
(1 row)
```

It appeared to be *several thousand times larger* than our BRIN! However, the query is performed a little faster: the planner used statistics to figure out that the data is physically ordered and it is not needed to build a bitmap and, mainly, that the index condition does not need to be rechecked:

```
demo=# explain (costs off,analyze)
  select *
  from flights_bi
  where scheduled_time >= :d and scheduled_time < :d + interval '1 day';
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using flights_bi_scheduled_time_btree on flights_bi
 (actual time=0.099..79.416 rows=83954 loops=1)
   Index Cond: ...
 Planning time: 0.500 ms
 Execution time: 85.044 ms
```

That's what is so wonderful about BRIN: we sacrifice the efficiency, but gain very much space.

Operator classes

minmax

For data types whose values can be compared with one another, summary information consists of *the minimal and maximal values*. Names of the corresponding operator classes contain "minmax", for example, "date_minmax_ops". Actually, these are data types that we were considering so far, and most of the types are of this kind.

inclusive

Comparison operators are defined not for all data types. For example, they are not defined for points ("point" type), which represent the geographical coordinates of airports. By the way, it's for this reason that the statistics do not show the correlation for this column.

```
demo=# select attname, correlation
from pg_stats
where tablename='flights_bi' and attname = 'airport_coord';
```

```
    attname    | correlation
---------------+-------------
 airport_coord |
(1 row)
```

But many of such types enable us to introduce a concept of a "bounding area", for example, a bounding rectangle for geometric shapes. We discussed in detail how GiST index uses this feature. Similarly, BRIN also enables gathering summary information on columns having data types like these: *the bounding area for all values inside a range* is just the summary value.

Unlike for GiST, the summary value for BRIN must be of the same type as the values being indexed. Therefore, we cannot build the index for points, although it is clear that the coordinates could work in BRIN: the longitude is closely connected with the time zone. Fortunately, nothing hinders creation of the index on an expression after transforming points into degenerate rectangles. At the same time, we will set the size of a range to one page, just to show the limit case:

```
demo=# create index on flights_bi using brin (box(airport_coord)) with (pages_per_range=1);
```

The size of the index is as small as 30 MB even in such an extreme situation:

```
demo=# select pg_size_pretty(pg_total_relation_size('flights_bi_box_idx'));
```

```
 pg_size_pretty
----------------
 30 MB
(1 row)
```

Now we can make up queries that limit the airports by coordinates. For example:

```
demo=# select airport_code, airport_name
from airports
where box(coordinates) <@ box '120,40,140,50';
```

```
 airport_code |  airport_name
--------------+----------------
 KHV          | Khabarovsk-Novyi
 VVO          | Vladivostok
(2 rows)
```

The planner will, however, refuse to use our index.

```
demo=# analyze flights_bi;
demo=# explain select * from flights_bi
where box(airport_coord) <@ box '120,40,140,50';
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on flights_bi  (cost=0.00..985928.14 rows=30517 width=111)
   Filter: (box(airport_coord) <@ '(140,50),(120,40)'::box)
```

Why? Let's disable sequential scan and see what happens:

```
demo=# set enable_seqscan = off;
demo=# explain select * from flights_bi
where box(airport_coord) <@ box '120,40,140,50';
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi  (cost=14079.67..1000007.81 rows=30517 width=111)
   Recheck Cond: (box(airport_coord) <@ '(140,50),(120,40)'::box)
   ->  Bitmap Index Scan on flights_bi_box_idx
       (cost=0.00..14072.04 rows=30517076 width=0)
         Index Cond: (box(airport_coord) <@ '(140,50),(120,40)'::box)
```

It appears that the index *can be* used, but the planner supposes that the bitmap will have to be built on the whole table (look at "rows" of Bitmap Index Scan node), and it is no wonder that the planner chooses sequential scan in this case. The issue here is that for geometric types, PostgreSQL does not gather any statistics, and the planner has to go blindly:

```
demo=# select * from pg_stats where tablename = 'flights_bi_box_idx' \gx
```

```
-[ RECORD 1 ]----------+------------------
schemaname             | bookings
tablename              | flights_bi_box_idx
attname                | box
inherited              | f
null_frac              | 0
avg_width              | 32
n_distinct             | 0
most_common_vals       |
most_common_freqs      |
histogram_bounds       |
correlation            |
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

Alas. But there are no complaints about the index - it does work and works fine:

```
demo=# explain (costs off,analyze)
select * from flights_bi where box(airport_coord) <@ box '120,40,140,50';
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on flights_bi (actual time=158.142..315.445 rows=781790 loops=1)
   Recheck Cond: (box(airport_coord) <@ '(140,50),(120,40)'::box)
   Rows Removed by Index Recheck: 70726
   Heap Blocks: lossy=14772
   ->  Bitmap Index Scan on flights_bi_box_idx
       (actual time=158.083..158.083 rows=147720 loops=1)
         Index Cond: (box(airport_coord) <@ '(140,50),(120,40)'::box)
 Planning time: 0.137 ms
 Execution time: 340.593 ms
```

The conclusion must be like this: PostGIS is needed if anything nontrivial is required of the geometry. It can gather statistics anyway.

Internals

The conventional extension "pageinspect" enables us to look inside BRIN index.

First, the metainformation will prompt us the size of a range and how many pages are allocated for "revmap":

```
demo=# select *
from brin_metapage_info(get_raw_page('flights_bi_scheduled_time_idx',0));
```

```
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+----------------
 0xA8109CFA |       1 |           128 |              3
(1 row)
```

Pages 1–3 here are allocated for "revmap", while the rest contain summary data. From "revmap" we can get references to summary data for each range. Say, the information on the first range, incorporating first 128 pages, is located here:

```
demo=# select *
from brin_revmap_data(get_raw_page('flights_bi_scheduled_time_idx',1))
limit 1;
```

```
  pages
---------
 (6,197)
(1 row)
```

And this is the summary data itself:

```
demo=# select allnulls, hasnulls, value
from brin_page_items(
  get_raw_page('flights_bi_scheduled_time_idx',6),
  'flights_bi_scheduled_time_idx'
)
where itemoffset = 197;
```

```
 allnulls | hasnulls |                       value
----------+----------+-------------------------------------------------
 f        | f        | {2016-08-15 02:45:00+03 .. 2016-08-15 17:15:00+03}
(1 row)
```

Next range:

```
demo=# select *
from brin_revmap_data(get_raw_page('flights_bi_scheduled_time_idx',1))
offset 1 limit 1;
```

```
  pages
---------
 (6,198)
(1 row)
```

```
demo=# select allnulls, hasnulls, value
from brin_page_items(
  get_raw_page('flights_bi_scheduled_time_idx',6),
  'flights_bi_scheduled_time_idx'
)
where itemoffset = 198;
```

```
 allnulls | hasnulls |                       value
----------+----------+-------------------------------------------------
 f        | f        | {2016-08-15 06:00:00+03 .. 2016-08-15 18:55:00+03}
(1 row)
```

And so on.

For "inclusion" classes, the "value" field will display something like

```
 {(94.4005966186523,69.3110961914062),(77.6600036621,51.6693992614746) .. f .. f}
```

The first value is the embedding rectangle, and "f" letters at the end denote lacking empty elements (the first one) and lacking unmergeable values (the second one). Actually, the only unmergeable values are "IPv4" and "IPv6" addresses ("inet" data type).

Properties

Reminding you of the queries that have already been provided.

The following are the properties of the access method:

```
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 brin   | can_order     | f
 brin   | can_unique    | f
 brin   | can_multi_col | t
 brin   | can_exclude   | f
```

Indexes can be created on several columns. In this case, its own summary statistics are gathered for each column, but they are stored together for each range. Of course, this index makes sense if one and the same size of a range is suitable for all columns.

The following index-layer properties are available:

```
    name      | pg_index_has_property
--------------+-----------------------
 clusterable  | f
 index_scan   | f
 bitmap_scan  | t
 backward_scan | f
```

Evidently, only bitmap scan is supported.

However, lack of clustering may seem confusing. Seemingly, since BRIN index is sensitive to physical order of rows, it would be logical to be able to cluster data according to the index. But this is not so. We can only create a "regular" index (B-tree or GiST, depending on the data type) and cluster according to it. By the way, do you want to cluster a supposedly huge table taking into account Exclusive locks, execution time, and consumption of disk space during rebuilding?

The following are the column-layer properties:

```
       name        | pg_index_column_has_property
-------------------+------------------------------
 asc               | f
 desc              | f
 nulls_first       | f
 nulls_last        | f
 orderable         | f
 distance_orderable | f
 returnable        | f
 search_array      | f
 search_nulls      | t
```

The only available property is the ability to manipulate NULLs.

Previous article

Next article

Egor Rogov

← Back to all articles

Egor Rogov

Your e-mail

Subscribe

Having clicked “Subscribe” I agree to receive blog updates and other communications (i.e. event invitations) from Postgres Professional Europe Limited. I am free to opt out at any time. Privacy Policy

Products

Postgres Pro Enterprise
Postgres Pro Standard
Cloud Solutions
Postgres Extensions

Resources

Blog
Documentation
Webinars
Videos
Presentations

Services

24×7×365 Technical Support
Migration to Postgres
High Availability Deployment
Database Audit
Remote DBA for PostgreSQL

Community

Events
Training Courses
Intro Book
Demo Database
Mailing List Archives

About

Leadership team
Partners
Customers
In the News
Press Releases
Press Info

Contacts

Neptune House, Marina Bay, office 207, Gibraltar, GX11 1AA
info@postgrespro.com

Get in touch!

Your First and Last Name

Company

E-mail

Message

I confirm that I have read and accepted PostgresPro’s Privacy Policy.

I agree to get Postgres Pro discount offers and other marketing communications.

Send a message

EULA
EULA for Cloud Environments
Privacy Policy