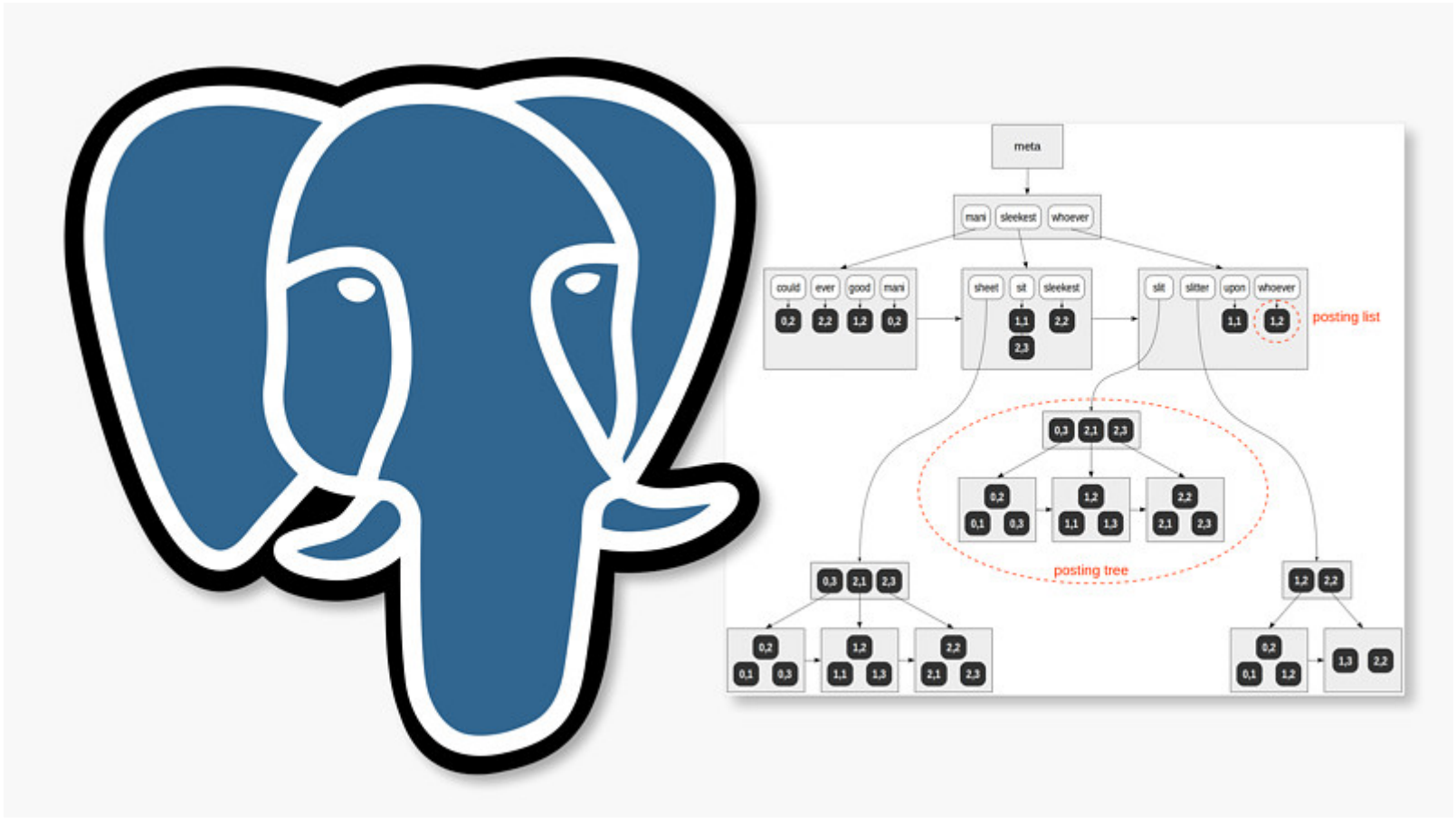
September 17, 2020 · PostgreSQL · By Egor Rogov

Indexes in PostgreSQL – 7 (GIN)



Habr Article

We have already got acquainted with PostgreSQL indexing engine and the interface of access methods and discussed hash indexes, B-trees, as well as GiST and SP-GiST indexes. And this article will feature GIN index.

GIN

"Gin?.. Gin is, it seems, such an American liquor?.."
"I'm not a drink, oh, inquisitive boy!" again the old man flared up, again he realized himself and again took himself in hand. "I am not a drink, but a powerful and undaunted spirit, and there is no such magic in the world that I would not be able to do."
– Lazar Lagin, "Old Khottabych".

Gin stands for Generalized Inverted Index and should be considered as a genie, not a drink.
– README

General concept

GIN is the abbreviated Generalized Inverted Index. This is a so-called *inverted index*. It manipulates data types whose values are not atomic, but consist of elements. We will call these types compound. And these are not the values that get indexed, but individual elements; each element references the values in which it occurs.

A good analogy to this method is the index at the end of a book, which for each term, provides a list of pages where this term occurs. The access method must ensure fast search of indexed elements, just like the index in a book. Therefore, these elements are stored as a familiar B-tree (a different, simpler, implementation is used for it, but it does not matter in this case). An ordered set of references to table rows that contain compound values with the element is linked to each element. Orderliness is inessential for data retrieval (the sort order of TIDs does not mean much), but important for the internal structure of the index.

Elements are never deleted from GIN index. It is taken to be that values containing elements can disappear, arise, or vary, but the set of elements of which they are composed is more or less stable. This solution considerably simplifies algorithms for concurrent work of several processes with the index.

If the list of TIDs is pretty small, it can fit into the same page as the element (and is called "the posting list"). But if the list is large, a more efficient data structure is needed, and we are already aware of it – it is B-tree again. Such a tree is located on separate data pages (and is called "the posting tree").

So, GIN index consists of the B-tree of elements, and B-trees or flat lists of TIDs are linked to leaf rows of that B-tree.

Just like GiST and SP-GiST indexes, discussed earlier, GIN provides an application developer with the interface to support various operations over compound data types.

Full-text search

The main application area for GIN method is speeding up full-text search, which is, therefore, reasonable to be used as an example in a more detailed discussion of this index.

The article related to GiST has already provided a small introduction to full-text search, so let's get straight to the point without repetitions. It is clear that compound values in this case are *documents*, and elements of these documents are *lexemes*.

Let's consider the example from GiST-related article:

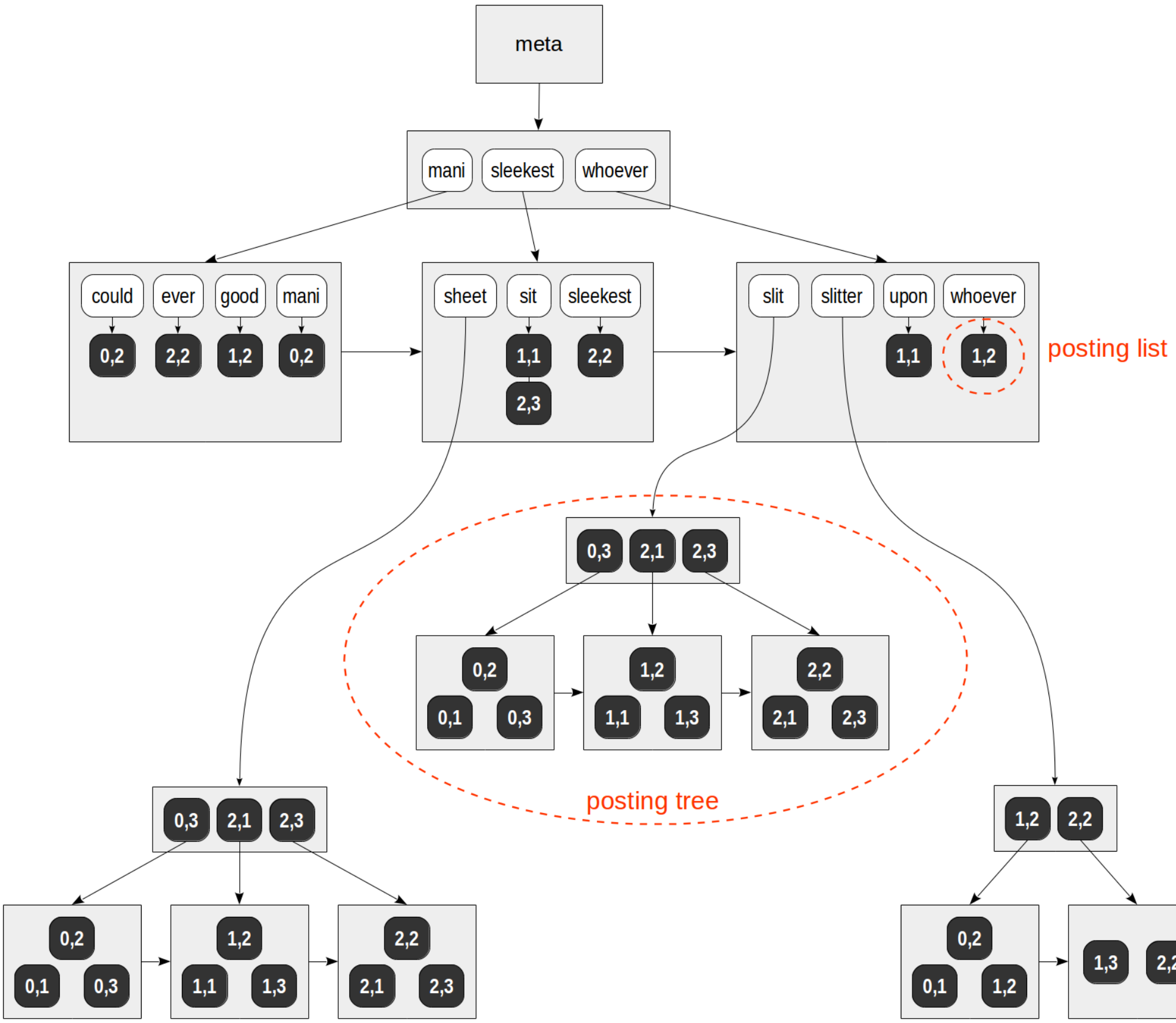
```
postgres=# create table ts(doc text, doc_tsv tsvector);

postgres=# insert into ts(doc) values
  ('Can a sheet slitter slit sheets?'),
  ('How many sheets could a sheet slitter slit?'),
  ('I slit a sheet, a sheet I slit.'),
  ('Upon a slitted sheet I sit.'),
  ('Whoever slit the sheets is a good sheet slitter.'),
  ('I am a sheet slitter.'),
  ('I slit sheets.'),
  ('I am the sleekest sheet slitter that ever slit sheets.'),
  ('She slits the sheet she sits on.');

postgres=# update ts set doc_tsv = to_tsvector(doc);

postgres=# create index on ts using gin(doc_tsv);
```

A possible structure of this index is shown in the figure:



Unlike in all the previous figures, references to table rows (TIDs) are denoted with numeric values on a dark background (the page number and position on the page) rather than with arrows.

```
postgres=# select ctid, left(doc,20), doc_tsv from ts;
```

```
  ctid |         left         |                          doc_tsv
-------+----------------------+-------------------------------------------------------------
 (0,1) | Can a sheet slitter  | 'sheet':3,6 'slit':5 'slitter':4
 (0,2) | How many sheets coul | 'could':4 'mani':2 'sheet':3,6 'slit':8 'slitter':7
 (0,3) | I slit a sheet, a sh | 'sheet':4,6 'slit':2,8
 (1,1) | Upon a slitted sheet | 'sheet':4 'sit':6 'slit':3 'upon':1
 (1,2) | Whoever slit the she | 'good':7 'sheet':4,8 'slit':2 'slitter':9 'whoever':1
 (1,3) | I am a sheet slitter | 'sheet':4 'slitter':5
 (2,1) | I slit sheets.       | 'sheet':3 'slit':2
 (2,2) | I am the sleekest sh | 'ever':8 'sheet':5,10 'sleekest':4 'slit':9 'slitter':6
 (2,3) | She slits the sheet  | 'sheet':4 'sit':6 'slit':2
(9 rows)
```

In this speculative example, the list of TIDs fit into regular pages for all lexemes but "sheet", "slit", and "slitter". These lexemes occurred in many documents, and the lists of TIDs for them have been placed into individual B-trees.

By the way, how can we figure out how many documents contain a lexeme? For a small table, a "direct" technique, shown below, will work, but we will learn further what to do with larger ones.

```
postgres=# select (unnest(doc_tsv)).lexeme, count(*) from ts
group by 1 order by 2 desc;
```

```
  lexeme   | count
-----------+-------
 sheet     |     9
 slit      |     8
 slitter   |     5
 sit       |     2
 upon      |     1
 mani      |     1
 whoever   |     1
 sleekest  |     1
 good      |     1
 could     |     1
 ever      |     1
(11 rows)
```

Note also that unlike a regular B-tree, pages of GIN index are connected by a unidirectional list rather than a bidirectional one. This is sufficient since a tree traverse is done only one way.

Example of a query

How will the following query be performed for our example?

```
postgres=# explain(costs off)
select doc from ts where doc_tsv @@ to_tsquery('many & slitter');
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on ts
   Recheck Cond: (doc_tsv @@ to_tsquery('many & slitter'::text))
   ->  Bitmap Index Scan on ts_doc_tsv_idx
         Index Cond: (doc_tsv @@ to_tsquery('many & slitter'::text))
(4 rows)
```

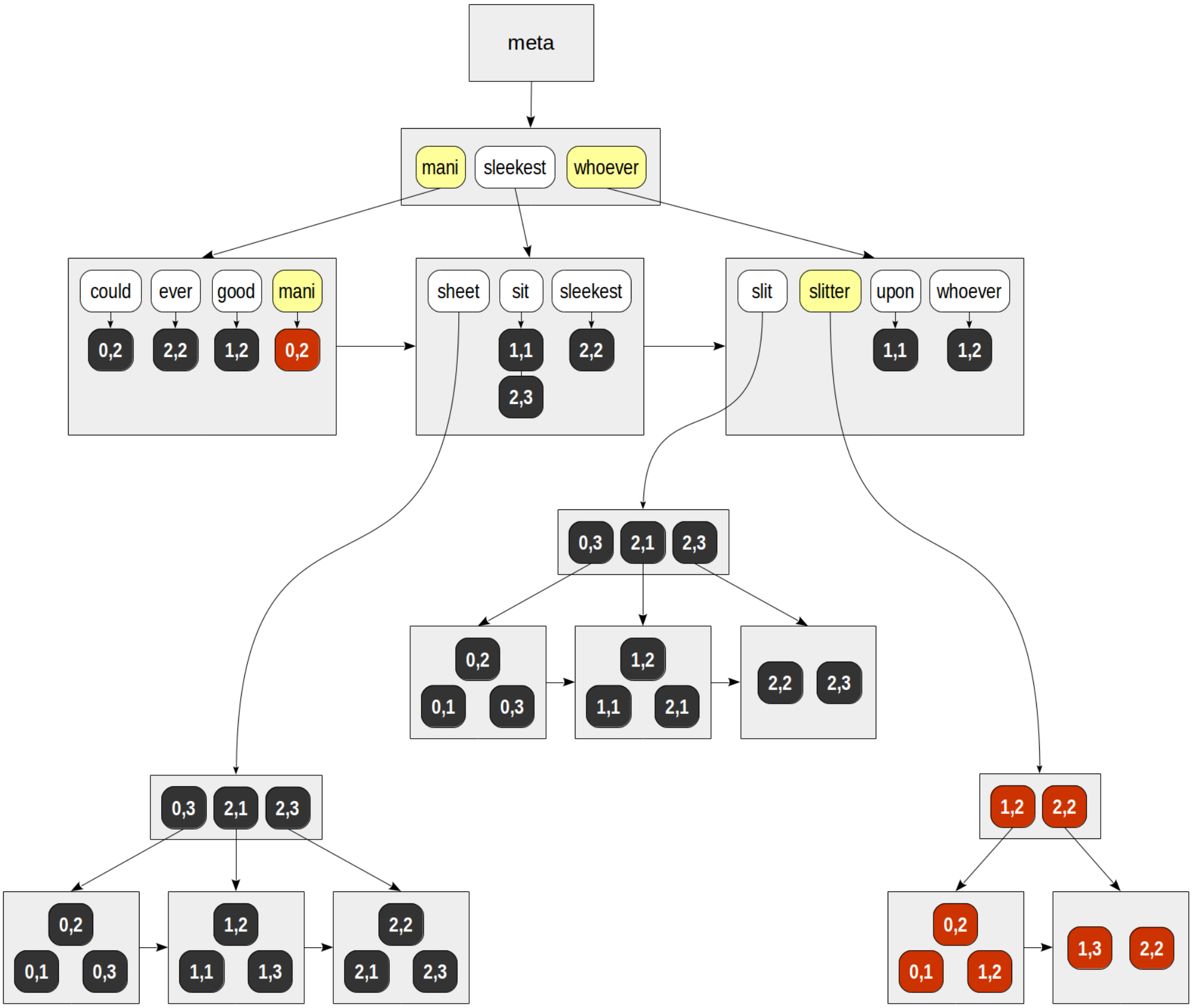
Individual lexemes (search keys) are extracted from the query first: "mani" and "slitter". This is done by a specialized API function that takes into account the data type and strategy determined by the operator class:

```
postgres=# select amop.amopopr::regoperator, amop.amopstrategy
from pg_opclass opc, pg_opfamily opf, pg_am am, pg_amop amop
where opc.opcname = 'tsvector_ops'
and opf.oid = opc.opcfamily
and am.oid = opf.opfmethod
and amop.amopfamily = opc.opcfamily
and am.amname = 'gin'
and amop.amoplefttype = opc.opcintype;
```

```
        amopopr         | amopstrategy
------------------------+--------------
 @@(tsvector,tsquery)   |            1  matching search query
 @@@(tsvector,tsquery)  |            2  synonym for @@ (for backward compatibility)
(2 rows)
```

In the B-tree of lexemes, we next find both keys and go through ready lists of TIDs. We get:

for "mani" – (0,2).
for "slitter" – (0,1), (0,2), (1,2), (1,3), (2,2).



Finally, for each TID found, an API consistency function is called, which must determine which of the rows found match the search query. Since the lexemes in our query are joined by Boolean "and", the only row returned is (0,2):

```
      |      |         |  consistency
      |      |         |    function
  TID  | mani | slitter | slit & slitter
-------+------+---------+----------------
 (0,1) |    f |       T |              f
 (0,2) |    T |       T |              T
 (1,2) |    f |       T |              f
 (1,3) |    f |       T |              f
 (2,2) |    f |       T |              f
```

And the result is:

```
postgres=# select doc from ts where doc_tsv @@ to_tsquery('many & slitter');
```

```
                   doc
-----------------------------------------
 How many sheets could a sheet slitter slit?
(1 row)
```

If we compare this approach with the one already discussed for GiST, the advantage of GIN for full-text search appears evident. But there is more in this than meets the eye.

The issue of a slow update

The thing is that data insertion or update in GIN index is pretty slow. Each document usually contains many lexemes to be indexed. Therefore, when only one document is added or updated, we have to massively update the index tree.

On the other hand, if several documents are simultaneously updated, some of their lexemes may be the same, and the total amount of work will be smaller than when updating documents one by one.

GIN index has "fastupdate" storage parameter, which we can specify during index creation and update later:

```
postgres=# create index on ts using gin(doc_tsv) with (fastupdate = true);
```

With this parameter turned on, updates will be accumulated in a separate unordered list (on individual connected pages). When this list gets large enough or during vacuuming, all accumulated updates are instantaneously made to the index. What list to consider "large enough" is determined by "gin_pending_list_limit" configuration parameter or by the same-name storage parameter of the index.

But this approach has drawbacks: first, search is slowed down (since the unordered list needs to be looked through in addition to the tree), and second, a next update can unexpectedly take much time if the unordered list has been overflowed.

Search of a partial match

We can use partial match in full-text search. For example, consider the following query:

```
gin=# select doc from ts where doc_tsv @@ to_tsquery('slit:*');
```

```
                          doc
------------------------------------------------------
 Can a sheet slitter slit sheets?
 How many sheets could a sheet slitter slit?
 I slit a sheet, a sheet I slit.
 Upon a slitted sheet I sit.
 Whoever slit the sheets is a good sheet slitter.
 I am a sheet slitter.
 I slit sheets.
 I am the sleekest sheet slitter that ever slit sheets.
 She slits the sheet she sits on.
(9 rows)
```

This query will find documents that contain lexemes starting with "slit". In this example, such lexemes are "slit" and "slitter".

A query will certainly work anyway, even without indexes, but GIN also permits to speed up the following search:

```
postgres=# explain (costs off)
select doc from ts where doc_tsv @@ to_tsquery('slit:*');
```

```
                        QUERY PLAN
----------------------------------------------------------
 Bitmap Heap Scan on ts
   Recheck Cond: (doc_tsv @@ to_tsquery('slit:*'::text))
   ->  Bitmap Index Scan on ts_doc_tsv_idx
         Index Cond: (doc_tsv @@ to_tsquery('slit:*'::text))
(4 rows)
```

Here all lexemes having the prefix specified in the search query are looked up in the tree and joined by Boolean "or".

Frequent and infrequent lexemes

To watch how the indexing works on live data, let's take the archive of "pgsql-hackers" email, which we've already used while discussing GiST. This version of the archive contains 356125 messages with the send date, subject, author, and text.

```
fts=# alter table mail_messages add column tsv tsvector;

fts=# update mail_messages set tsv = to_tsvector(body_plain);
```

```
NOTICE:  word is too long to be indexed
DETAIL:  Words longer than 2047 characters are ignored.
...
UPDATE 356125
```

```
fts=# create index on mail_messages using gin(tsv);
```

Let's consider a lexeme that occurs in many documents. The query using "unnest" will fail to work on such a large data size, and the correct technique is to use "ts_stat" function, which provides the information on lexemes, the number of documents where they occurred, and total number of occurrences.

```
fts=# select word, ndoc
from ts_stat('select tsv from mail_messages')
order by ndoc desc limit 3;
```

```
 word  |  ndoc
-------+--------
 re    | 322141
 wrote | 231174
 use   | 176917
(3 rows)
```

Let's choose "wrote".

And we will take some word that is infrequent for developers' email, say, "tattoo":

```
fts=# select word, ndoc from ts_stat('select tsv from mail_messages') where word = 'tattoo';
```

```
  word  | ndoc
--------+------
 tattoo |    2
(1 row)
```

Are there any documents where both these lexemes occur? It appears that there are:

```
fts=# select count(*) from mail_messages where tsv @@ to_tsquery('wrote & tattoo');
```

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

A question arises how to perform this query. If we get lists of TIDs for both lexemes, as described above, the search will evidently be inefficient: we will have to go through more than 200 thousand values, only one of which will be left. Fortunately, using the planner statistics, the algorithm understands that "wrote" lexeme occurs frequently, while "tatoo" occurs infrequently. Therefore, the search of the infrequent lexeme is performed, and the two documents retrieved are then checked for occurrence of "wrote" lexeme. And this is clear from the query, which is performed quickly:

```
fts=# \timing on

fts=# select count(*) from mail_messages where tsv @@ to_tsquery('wrote & tattoo');
```

```
 count
-------
     1
(1 row)
Time: 0,959 ms
```

The search of "wrote" alone takes considerably longer:

```
fts=# select count(*) from mail_messages where tsv @@ to_tsquery('wrote');
```

```
 count
--------
 231174
(1 row)
Time: 2875,543 ms (00:02,876)
```

This optimization certainly works not only for two lexemes, but in more complex cases too.

Limiting the query result

A feature of GIN access method is that the result is always returned as a bitmap: this method cannot return the result TID by TID. It's because of this, all query plans in this article use bitmap scan.

Therefore, limitation of the index scan result using LIMIT clause is not quite efficient. Pay attention to the predicted cost of the operation ("cost" field of "Limit" node):

```
fts=# explain (costs off)
select * from mail_messages where tsv @@ to_tsquery('wrote') limit 1;
```

```
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Limit  (cost=1283.61..1285.13 rows=1)
   ->  Bitmap Heap Scan on mail_messages  (cost=1283.61..209975.49 rows=137207)
         Recheck Cond: (tsv @@ to_tsquery('wrote'::text))
         ->  Bitmap Index Scan on mail_messages_tsv_idx  (cost=0.00..1249.30 rows=137207)
               Index Cond: (tsv @@ to_tsquery('wrote'::text))
(5 rows)
```

The cost is estimated as 1285.13, which is a little bit larger than the cost of building the whole bitmap 1249.30 ("cost" field of Bitmap Index Scan node).

Therefore, the index has a special capability to limit the number of results. The threshold value is specified in "gin_fuzzy_search_limit" configuration parameter and is equal to zero by default (no limitation takes place). But we can set the threshold value:

```
fts=# set gin_fuzzy_search_limit = 1000;

fts=# select count(*) from mail_messages where tsv @@ to_tsquery('wrote');
```

```
 count
-------
  5746
(1 row)
```

```
fts=# set gin_fuzzy_search_limit = 10000;

fts=# select count(*) from mail_messages where tsv @@ to_tsquery('wrote');
```

```
 count
-------
 14726
(1 row)
```

As we can see, the number of rows returned by the query differs for different parameter values (if index access is used). The limitation is not strict: more rows than specified can be returned, which justifies "fuzzy" part of the parameter name.

Compact representation

Among the rest, GIN indexes are good thanks to their compactness. First, if one and the same lexeme occurs in several documents (and this is usually the case), it is stored in the index only once. Second, TIDs are stored in the index in an ordered fashion, and this enables us to use a simple compression: each next TID in the list is actually stored as its difference from the previous one; this is usually a small number, which requires much fewer bits that a complete six-byte TID.

To get an idea of the size, let's build B-tree from the text of the messages. But a fair comparison is not certainly going to happen:

- GIN is built a on different data type ("tsvector" rather than "text"), which is smaller,
- at the same time, the size of messages for B-tree has to be shortened to approximately two kilobytes.

Nevertheless, we continue:

```
fts=# create index mail_messages_btree on mail_messages(substring(body_plain for 2048));
```

We will build GiST index as well:

```
fts=# create index mail_messages_gist on mail_messages using gist(tsv);
```

The size of indexes upon "vacuum full":

```
fts=# select pg_size_pretty(pg_relation_size('mail_messages_tsv_idx')) as gin,
            pg_size_pretty(pg_relation_size('mail_messages_gist')) as gist,
            pg_size_pretty(pg_relation_size('mail_messages_btree')) as btree;
```

```
  gin   |  gist  | btree
--------+--------+--------
 179 MB | 125 MB | 546 MB
(1 row)
```

Because of representation compactness, we can try to use GIN index during migration from Oracle as a replacement for bitmap indexes (without going into details, I provide a reference to Lewis's post for inquisitive minds). As a rule, bitmap indexes are used for fields that have few unique values, which is excellent also for GIN. And, as shown in the first article, PostgreSQL can build a bitmap based on any index, including GIN, on the fly.

GiST or GIN?

For many data types, operator classes are available for both GiST and GIN, which raises a question which index to use. Perhaps, we can already make some conclusions.

As a rule, GIN beats GiST in accuracy and search speed. If the data is updated not frequently and fast search is needed, most likely GIN will be an option.

On the other hand, if the data is intensively updated, overhead costs of updating GIN may appear to be too large. In this case, we will have to compare both options and choose the one whose characteristics are better balanced.

Arrays

Another example of using GIN is indexing of arrays. In this case, array elements get into the index, which permits to speed up a number of operations over arrays:

```
postgres=# select amop.amopopr::regoperator, amop.amopstrategy
from pg_opclass opc, pg_opfamily opf, pg_am am, pg_amop amop
where opc.opcname = 'array_ops'
and opf.oid = opc.opcfamily
and am.oid = opf.opfmethod
and amop.amopfamily = opc.opcfamily
and am.amname = 'gin'
and amop.amoplefttype = opc.opcintype;
```

```
        amopopr        | amopstrategy
-----------------------+--------------
 &&(anyarray,anyarray) |            1  intersection
 @>(anyarray,anyarray) |            2  contains array
 <@(anyarray,anyarray) |            3  contained in array
 =(anyarray,anyarray)  |            4  equality
(4 rows)
```

Our demo database has "routes" view with information on flights. Among the rest, this view contains "days_of_week" column - an array of weekdays when flights take place. For example, a flight from Vnukovo to Gelendzhik leaves on Tuesdays, Thursdays, and Sundays:

```
demo=# select departure_airport_name, arrival_airport_name, days_of_week
from routes
where flight_no = 'PG0049';
```

```
 departure_airport_name | arrival_airport_name | days_of_week
------------------------+----------------------+--------------
 Vnukovo                | Gelendzhik           | {2,4,7}
(1 row)
```

To build the index, let's "materialize" the view into a table:

```
demo=# create table routes_t as select * from routes;

demo=# create index on routes_t using gin(days_of_week);
```

Now we can use the index to get to know all flights that leave on Tuesdays, Thursdays, and Sundays:

```
demo=# explain (costs off) select * from routes_t where days_of_week = ARRAY[2,4,7];
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Bitmap Heap Scan on routes_t
   Recheck Cond: (days_of_week = '{2,4,7}'::integer[])
   ->  Bitmap Index Scan on routes_t_days_of_week_idx
         Index Cond: (days_of_week = '{2,4,7}'::integer[])
(4 rows)
```

It appears that there are six of them:

```
demo=# select flight_no, departure_airport_name, arrival_airport_name, days_of_week from routes_t where
days_of_week = ARRAY[2,4,7];
```

```
 flight_no | departure_airport_name | arrival_airport_name | days_of_week
-----------+------------------------+----------------------+-------------
 PG0005    | Domodedovo             | Pskov                | {2,4,7}
 PG0049    | Vnukovo                | Gelendzhik           | {2,4,7}
 PG0113    | Naryan-Mar             | Domodedovo           | {2,4,7}
 PG0249    | Domodedovo             | Gelendzhik           | {2,4,7}
 PG0449    | Stavropol              | Vnukovo              | {2,4,7}
 PG0540    | Barnaul                | Vnukovo              | {2,4,7}
(6 rows)
```

How is this query performed? Exactly the same way as described above:

1. From the array {2,4,7}, which plays the role of the search query here, elements (search keywords) are extracted. Evidently, these are the values of "2", "4", and "7".
2. In the tree of elements, the extracted keys are found, and for each of them the list of TIDs is selected.
3. Of all TIDs found, the consistency function selects those that match the operator from the query. For = operator, only those TIDs match it that occurred in all the three lists (in other words, the initial array must contain all the elements). But this is not sufficient: it is also needed for the array not to contain any other values, and we cannot check this condition with the index. Therefore, in this case, the access method asks the indexing engine to recheck all TIDs returned with the table.

Interestingly, there are strategies (for example, "contained in array") that cannot check anything and have to recheck all the TIDs found with the table.

But what to do if we need to know the flights that leave from Moscow on Tuesdays, Thursdays, and Sundays? The index will not support the additional condition, which will get into "Filter" column.

```
demo=# explain (costs off)
select * from routes_t where days_of_week = ARRAY[2,4,7] and departure_city = 'Moscow';
```

```
                        QUERY PLAN
----------------------------------------------------------
 Bitmap Heap Scan on routes_t
   Recheck Cond: (days_of_week = '{2,4,7}'::integer[])
   Filter: (departure_city = 'Moscow'::text)
   ->  Bitmap Index Scan on routes_t_days_of_week_idx
         Index Cond: (days_of_week = '{2,4,7}'::integer[])
(5 rows)
```

Here this is OK (the index selects only six rows anyway), but in cases where the additional condition increases the selective capability, it is desired to have such a support. However, we cannot just create the index:

```
demo=# create index on routes_t using gin(days_of_week,departure_city);
```

```
ERROR:  data type text has no default operator class for access method "gin"
HINT:  You must specify an operator class for the index or define a default operator class for the data type.
```

But "btree_gin" extension will help, which adds GIN operator classes that simulate work of a regular B-tree.

```
demo=# create extension btree_gin;

demo=# create index on routes_t using gin(days_of_week,departure_city);

demo=# explain (costs off)
select * from routes_t where days_of_week = ARRAY[2,4,7] and departure_city = 'Moscow';
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on routes_t
   Recheck Cond: ((days_of_week = '{2,4,7}'::integer[]) AND
                  (departure_city = 'Moscow'::text))
   ->  Bitmap Index Scan on routes_t_days_of_week_departure_city_idx
         Index Cond: ((days_of_week = '{2,4,7}'::integer[]) AND
                      (departure_city = 'Moscow'::text))
(4 rows)
```

JSONB

One more example of a compound data type that has built-in GIN support is JSON. To work with JSON values, a number of operators and functions are defined at present, some of which can be sped up using indexes:

```
postgres=# select opc.opcname, amop.amopopr::regoperator, amop.amopstrategy as str
from pg_opclass opc, pg_opfamily opf, pg_am am, pg_amop amop
where opc.opcname in ('jsonb_ops','jsonb_path_ops')
and opf.oid = opc.opcfamily
and am.oid = opf.opfmethod
and amop.amopfamily = opc.opcfamily
and am.amname = 'gin'
and amop.amoplefttype = opc.opcintype;
```

```
    opcname     |     amopopr      | str
----------------+------------------+-----
 jsonb_ops      | ?(jsonb,text)    |   9  top-level key exists
 jsonb_ops      | ?|(jsonb,text[]) |  10  some top-level key exists
 jsonb_ops      | ?&(jsonb,text[]) |  11  all top-level keys exist
 jsonb_ops      | @>(jsonb,jsonb)  |   7  JSON value is at top level
 jsonb_path_ops | @>(jsonb,jsonb)  |   7
(5 rows)
```

As we can see, two operator classes are available: "jsonb_ops" and "jsonb_path_ops".

The first operator class "jsonb_ops" is used by default. All keys, values, and array elements get to the index as elements of the initial JSON document. An attribute is added to each of these elements, which indicates whether this element is a key (this is needed for "exists" strategies, which distinguish between keys and values).

For example, let's represent a few rows from "routes" as JSON as follows:

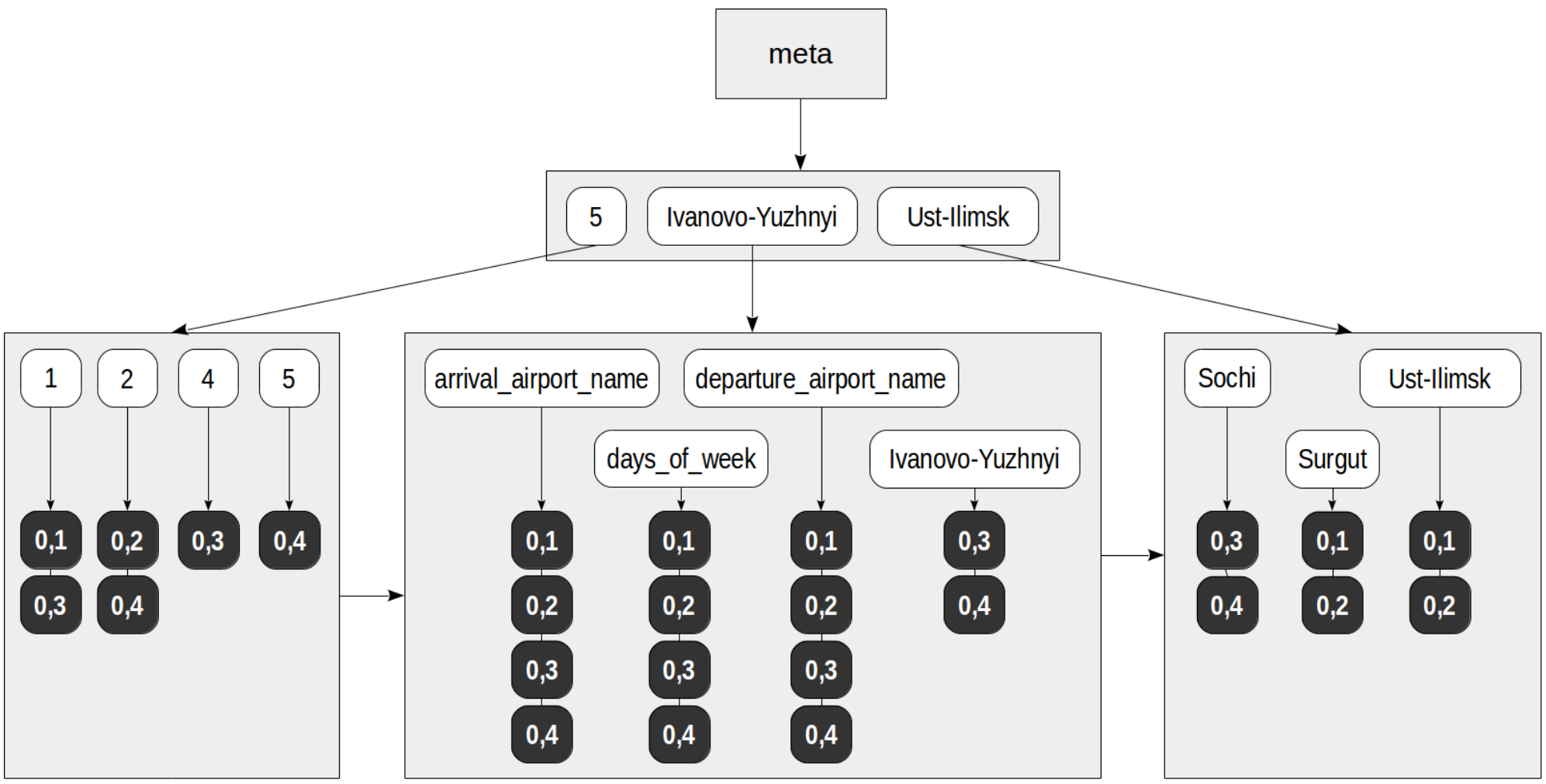
```
demo=# create table routes_jsonb as
  select to_jsonb(t) route
  from (
      select departure_airport_name, arrival_airport_name, days_of_week
      from routes
      order by flight_no limit 4
  ) t;

demo=# select ctid, jsonb_pretty(route) from routes_jsonb;
```

```
 ctid  |                 jsonb_pretty
-------+-----------------------------------------------
 (0,1) | {                                            +
       |     "days_of_week": [                        +
       |         1                                    +
       |     ],                                       +
       |     "arrival_airport_name": "Surgut",        +
       |     "departure_airport_name": "Ust-Ilimsk"   +
       | }
 (0,2) | {                                            +
       |     "days_of_week": [                        +
       |         2                                    +
       |     ],                                       +
       |     "arrival_airport_name": "Ust-Ilimsk",    +
       |     "departure_airport_name": "Surgut"       +
       | }
 (0,3) | {                                            +
       |     "days_of_week": [                        +
       |         1,                                   +
       |         4                                    +
       |     ],                                       +
       |     "arrival_airport_name": "Sochi",         +
       |     "departure_airport_name": "Ivanovo-Yuzhnyi"+
       | }
 (0,4) | {                                            +
       |     "days_of_week": [                        +
       |         2,                                   +
       |         5                                    +
       |     ],                                       +
       |     "arrival_airport_name": "Ivanovo-Yuzhnyi", +
       |     "departure_airport_name": "Sochi"        +
       | }
(4 rows)
```

```
demo=# create index on routes_jsonb using gin(route);
```

The index may look as follows:



Now, a query like this, for example, may be performed using the index:

```
demo=# explain (costs off)
select jsonb_pretty(route)
from routes_jsonb
where route @> '{"days_of_week": [5]}';
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Bitmap Heap Scan on routes_jsonb
   Recheck Cond: (route @> '{"days_of_week": [5]}'::jsonb)
   ->  Bitmap Index Scan on routes_jsonb_route_idx
         Index Cond: (route @> '{"days_of_week": [5]}'::jsonb)
(4 rows)
```

Starting with the root of the JSON document, `@>` operator checks whether the specified route (`"days_of_week": [5]`) occurs. Here the query will return one row:

```
demo=# select jsonb_pretty(route) from routes_jsonb where route @> '{"days_of_week": [5]}';
```

```
                jsonb_pretty
-----------------------------------------------
 {                                            +
     "days_of_week": [                        +
         2,                                   +
         5                                    +
     ],                                       +
     "arrival_airport_name": "Ivanovo-Yuzhnyi",+
     "departure_airport_name": "Sochi"        +
 }
(1 row)
```

The query is performed as follows:

1. In the search query (`"days_of_week": [5]`) elements (search keys) are extracted: "days_of_week" and "5".
2. In the tree of elements the extracted keys are found, and for each of them the list of TIDs is selected: for "5" – (0,4), and for "days_of_week" – (0,1), (0,2), (0,3), (0,4).
3. Of all TIDs found, the consistency function selects those that match the operator from the query. For `@>` operator, documents that contain not all elements from the search query won't do for sure, so only (0,4) is left. But we still need to recheck the TID left with the table since it is unclear from the index in what order the found elements occur in JSON document.

To discover more details of other operators, you can read the documentation.

In addition to conventional operations for dealing with JSON, "jsquery" extension has long been available, which defines a query language with richer capabilities (and certainly, with support of GIN indexes). Besides, in 2016, a new SQL standard was issued, which defines its own set of operations and query language "SQL/JSON path". An implementation of this standard has already been accomplished, and we believe it will appear in PostgreSQL 11.

The SQL/JSON path patch was finally committed to PostgreSQL 12, while other pieces are still on the way. Hopefully we'll see the fully implemented feature in PostgreSQL 13.

Internals

We can look inside GIN index using "pageinspect" extension.

```
fts=# create extension pageinspect;
```

The information from the meta page shows general statistics:

```
fts=# select * from gin_metapage_info(get_raw_page('mail_messages_tsv_idx',0));
```

```
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 22968
n_entry_pages    | 13751
n_data_pages     | 9216
n_entries        | 1423598
version          | 2
```

The page structure provides a special area where access methods store their information; this area is "opaque" for ordinary programs such as vacuum. "gin_page_opaque_info" function shows this data for GIN. For example, we can get to know the set of index pages:

```
fts=# select flags, count(*)
from generate_series(1,22967) as g(id), -- n_total_pages
     gin_page_opaque_info(get_raw_page('mail_messages_tsv_idx',g.id))
group by flags;
```

```
         flags          | count
------------------------+-------
 {meta}                 |     1  meta page
 {}                     |   133  internal page of element B-tree
 {leaf}                 | 13618  leaf page of element B-tree
 {data}                 |  1497  internal page of TID B-tree
 {data,leaf,compressed} |  7719  leaf page of TID B-tree
(5 rows)
```

"gin_leafpage_items" function provides information on TIDs stored on pages {data,leaf,compressed}:

```
fts=# select * from gin_leafpage_items(get_raw_page('mail_messages_tsv_idx',2672));
```

```
-[ RECORD 1 ]--------------------------------------------------------------
first_tid | (239,44)
nbytes    | 248
tids      | {"(239,44)","(239,47)","(239,48)","(239,50)","(239,52)","(240,3)",...
-[ RECORD 2 ]--------------------------------------------------------------
first_tid | (247,40)
nbytes    | 248
tids      | {"(247,40)","(247,41)","(247,44)","(247,45)","(247,46)","(248,2)",...
...
```

Note here that leave pages of the tree of TIDs actually contain small compressed lists of pointers to table rows rather than individual pointers.

Properties

Let's look at the properties of GIN access method (queries have already been provided).

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

Interestingly, GIN supports creation of multicolumn indexes. However, unlike for a regular B-tree, instead of compound keys, a multicolumn index will still store individual elements, and the column number will be indicated for each element.

The following index-layer properties are available:

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

Note that returning results TID by TID (index scan) is not supported; only bitmap scan is possible.

Backward scan is not supported either: this feature is essential for index scan only, but not for bitmap scan.

And the following are 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        | f
```

Nothing is available here: no sorting (which is clear), no use of the index as covering (since the document itself is not stored in the index), no manipulation of NULLs (since it does not make sense for elements of compound type).

Other data types

A few more extensions are available that add support of GIN for some data types.

- "pg_trgm" enables us to determine "likeness" of words by comparing how many equal three-letter sequences (trigrams) are available. Two operator classes are added, "gist_trgm_ops" and "gin_trgm_ops", which support various operators, including comparison by means of LIKE and regular expressions. We can use this extension together with full-text search in order to suggest word options to fix typos.
- "hstore" implements "key-value" storage. For this data type, operator classes for various access methods are available, including GIN. However, with the introduction of "jsonb" data type, there are no special reasons to use "hstore".
- "intarray" extends the functionality of integer arrays. Index support includes GiST, as well as GIN ("gin__int_ops" operator class).

And these two extensions have already been mentioned above:

- "btree_gin" adds GIN support for regular data types in order for them to be used in a multicolumn index along with compound types.
- "jsquery" defines a language for JSON querying and an operator class for index support of this language. This extension is not included in a standard PostgreSQL delivery.

Previous article

Next article

Egor Rogov

← Back to all articles

Egor Rogov

Willing to get notified about the latest Postgres Pro posts? Subscribe to our blog!

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

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