



### Copyright

© Postgres Professional, 2019–2024

Authors Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

### Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

### Feedback

Please send your feedback, comments and suggestions to:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

### Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Hash Index

GiST

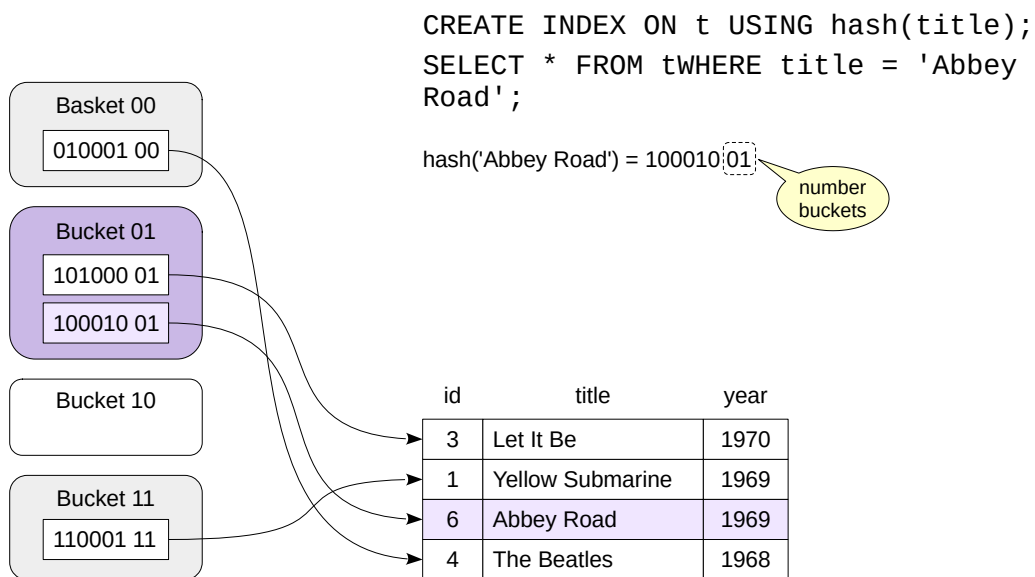
Operator Class

SP-GiST

or generalized inverted index

BRIN

# Concept of Hashing



3

The concept of hashing is that values of any data type are evenly distributed across a limited number of buckets in a hash table using a hash function. If a hash table is large enough to ensure that, on average, each bucket contains only one value (hash code), then searching for a value in the hash table takes constant time. To achieve this:

- 1) The hash function is applied to the given value.
- 2) the bucket number is determined by several bits of the resulting hash code;
- 3) The bucket is scanned for the hash code.

When data is not evenly distributed, a large number of values can end up in a single bucket. In this case, search efficiency will decrease.

Essentially, a hash index is a hash table stored on disk.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

Stores only hash codes, not the original data.

The index size is independent of the indexing key.

Index-only scans are not possible.

The index grows dynamically

spiky growth

Searches only support equality conditions

A hash index stores only hash values and item pointers; the actual indexed value is not stored. Therefore, the hash index size is not affected by the indexing key size, but index-only scans are not possible — the value must be retrieved from the table.

The size of the hash index grows dynamically as new values are added. As the number of buckets doubles with each increase, the size grows in a spiky manner.

Unlike B-trees, hash indexes have several limitations, such as:

- The hash index only supports equality-based searches, as the hash function does not retain the order of the values;
- does not support unique constraints;
- You cannot create a multicolumn index or add additional include columns to an index.

Therefore, hash indexes haven't become widely used. However, the hash index can be faster in some cases due to its smaller size and fixed search time compared to a B-tree index.

## Hash index

Let's examine the query plan for a query that fetches a specific seat in all aircraft cabins:

```
=> EXPLAIN (costs off)
SELECT * FROM seats WHERE seat_no = '31D';
```

QUERY PLAN

```
-----
Seq Scan on seats
  Filter: ((seat_no)::text = '31D'::text)
(2 rows)
```

Since there's no suitable index, a sequential scan is performed. Let's create a hash index on the seat\_no column and re-run the query:

```
=> CREATE INDEX ON seats USING hash(seat_no);
```

CREATE INDEX

```
=> EXPLAIN (costs off)
SELECT * FROM seats WHERE seat_no = '31D';
```

QUERY PLAN

```
-----
Bitmap Heap Scan on seats
  Recheck Cond: ((seat_no)::text = '31D'::text)
    -> Bitmap Index Scan on seats_seat_no_idx
          Index Cond: ((seat_no)::text = '31D'::text)
(4 rows)
```

Now the query planner uses the hash index and generates a bitmap. Let's change the equality condition to "greater than":

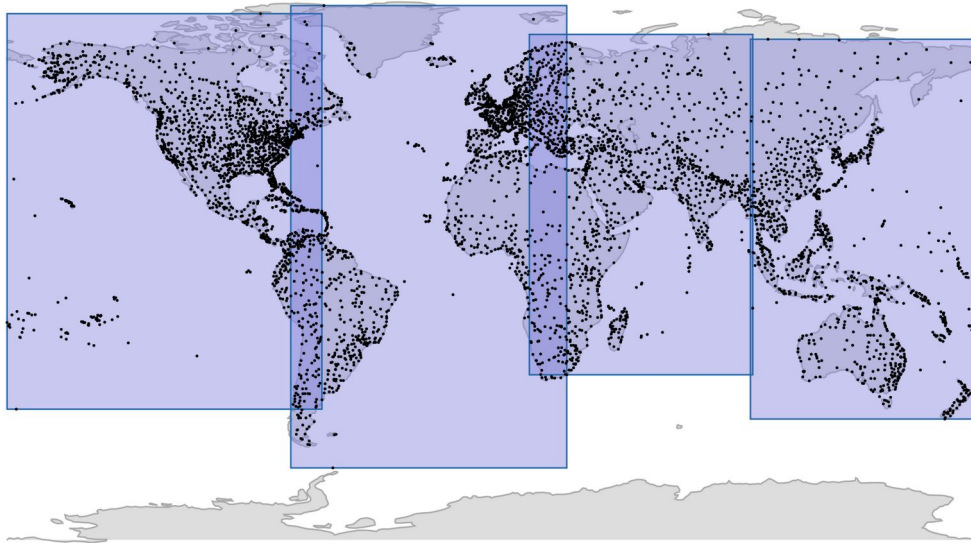
```
=> EXPLAIN (costs off)
SELECT * FROM seats WHERE seat_no > '31D';
```

QUERY PLAN

```
-----
Seq Scan on seats
  Filter: ((seat_no)::text > '31D'::text)
(2 rows)
```

Hash indexes cannot be used with inequalities.

# GiST Index Example



6

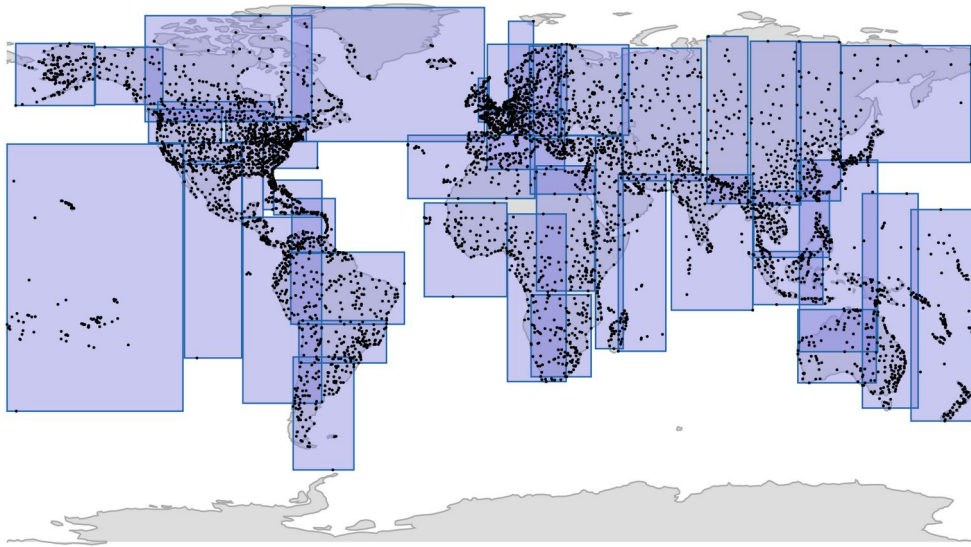
GiST stands for Generalized Search Tree, also known as a generalized search tree.

We'll explore how GiST indexes work using an example of points on a plane.

The plane is divided into several rectangles that together cover all the indexed points. These rectangles make up the top level of the tree.

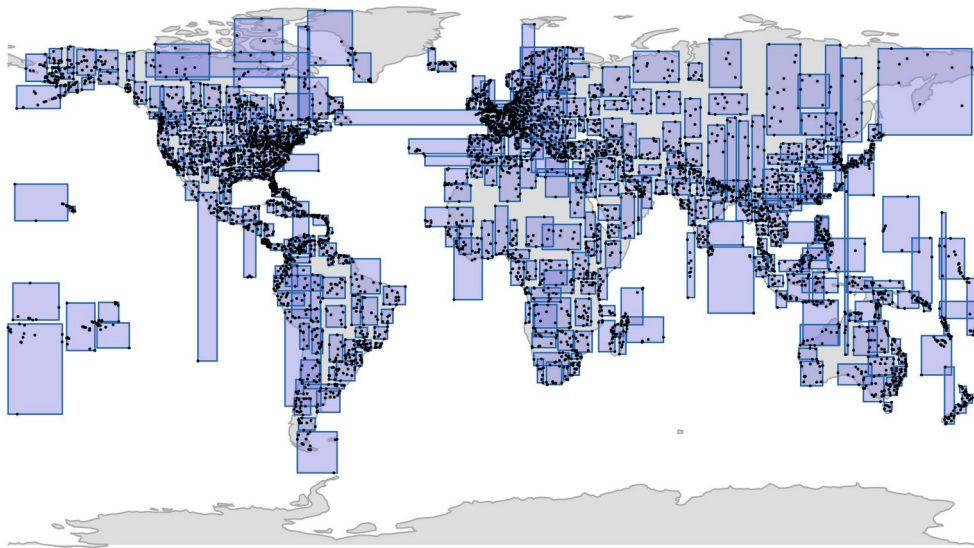
As shown in the figure, rectangles may overlap (although this may reduce search efficiency).

# GiST Index Example



On the next level of the tree, each large rectangle splits into smaller rectangles.

# GiST Index Example



8

On the last level of the tree, each bounding rectangle holds as many points as can fit on one index page.

The basic splitting condition is that the parent node's rectangle encloses all rectangles within the corresponding subtree. This enables, for instance, efficient retrieval of points located within a specific region:

- 1) Locate rectangles intersecting the specified area, at the topmost level of the index;
- 2) We move down into the selected subtrees and repeat the search.

This indexing method is referred to as an R-tree.



## Balanced search tree

- Supports arbitrary data types
- Ordering is not required

## Commonly supported operations

- Inclusion within the area
- Determining left, right, top, and bottom positions relative to the area

## Nearest Neighbor Search

- The first k values nearest to the specified

A B-tree is a balanced tree where the values are ordered according to the 'greater than' and 'less than' operations. The GiST Index also forms a balanced tree, but the values are organized differently, such as based on the relative positioning of points on a plane.

This makes GiST suitable for data types where 'greater than' and 'less than' operations lack inherent meaning, while enabling optimization of other crucial operations for these types. For example, the GiST index can speed up searching for a value that falls within a specific area or finding values located on a particular side of a specified area.

Another important feature of the GiST index — is its support for nearest neighbor searches. The index enables quick retrieval of several values closest to a given one.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

## GiST index

To demonstrate how a GiST index works, we'll use the `airports_data` table (which has a view named `airports`). The table includes a `coordinates` column of type `point`, which we'll use to create the GiST index.

But first, let's run the following query: find all airports near Moscow:

```
=> EXPLAIN (costs off)
SELECT airport_code
FROM airports_data
WHERE coordinates <@ '<(37.622513,55.753220),1.0>'::circle;
```

### QUERY PLAN

```
-----
Seq Scan on airports_data
  Filter: (coordinates <@ '<(37.622513,55.75322),1>'::circle)
(2 rows)
```

Without an index, the query planner scans the entire table. Let's create a GiST index:

```
=> CREATE INDEX airports_gist_idx ON airports_data
USING gist(coordinates);
```

CREATE INDEX

Since the `airports_data` table is small, the planner will still use sequential scanning. Let's temporarily disable this access method:

```
=> SET enable_seqscan = off;
```

SET

Run the query again:

```
=> EXPLAIN (costs off)
SELECT airport_code
FROM airports_data
WHERE coordinates <@ '<(37.622513,55.753220),1.0>'::circle;
```

### QUERY PLAN

```
-----
Index Scan using airports_gist_idx on airports_data
  Index Cond: (coordinates <@ '<(37.622513,55.75322),1>'::circle)
(2 rows)
```

Now the query planner retrieves the relevant rows by using the `airports_gist_idx` index.

Let's drop the created index:

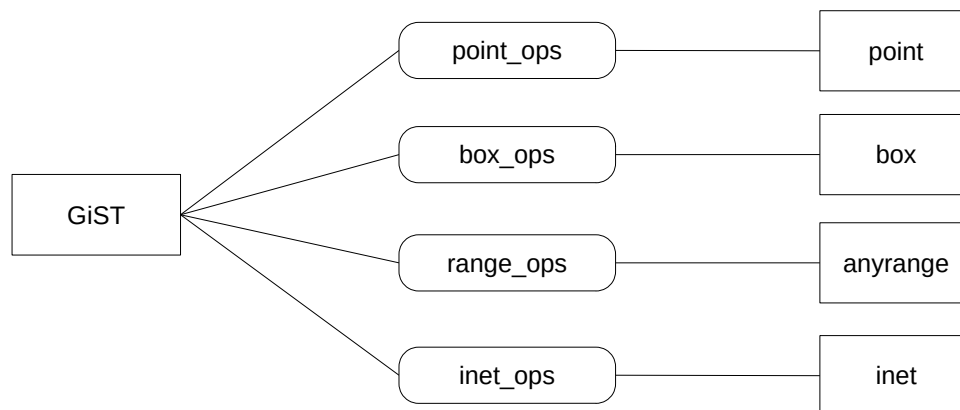
```
=> DROP INDEX airports_gist_idx;
```

DROP INDEX

# Operator Class

Bridge between the index method and data type

Can encompass a substantial portion of the indexing logic



11

To enable access methods to work with various data types (which can be dynamically loaded in PostgreSQL), there's an intermediary called an operator class that contains the required operators and support functions.

B-trees and hash indexes, like other index types, rely on operator classes, though these classes are relatively simple, containing basic operators like "equal," "greater than," and "less than."

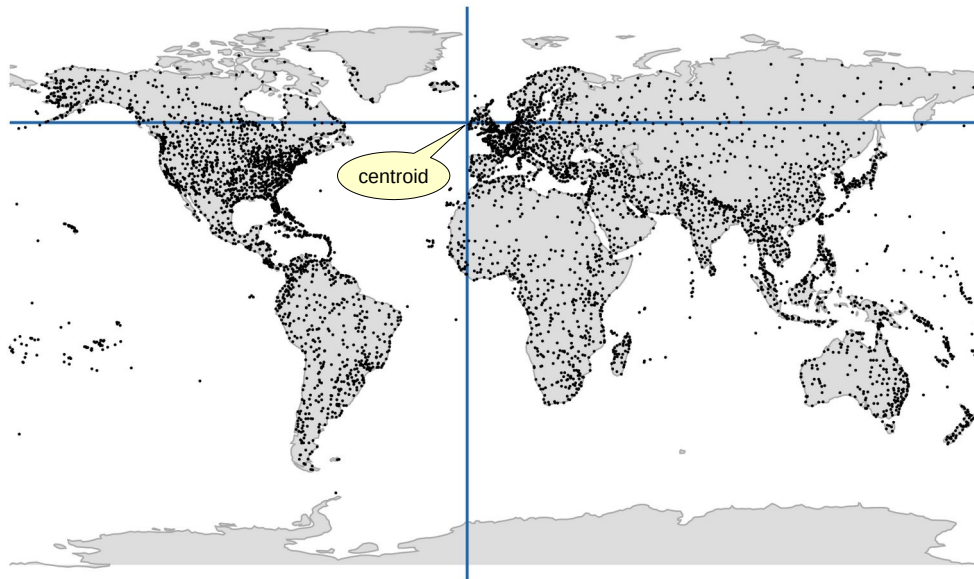
Operator classes for GiST indexes contain a significant portion of the indexing logic and define the rules for adding values to the index and searching them. Therefore, GiST can accelerate different operations for various data types.

For instance, GiST can be used to index values of range types, such as `int4range` or `tstzrange`. This index enables you to find ranges contained within, overlapping, or adjacent to the specified range, and so on.

GiST can be viewed as a framework upon which custom indexing schemes (not just R-trees) are built, allowing for the implementation of operator classes as needed. This approach is far simpler than building a new index type from the ground up, a process that is highly complex and requires substantial developer expertise.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

# SP-GiST Index Example



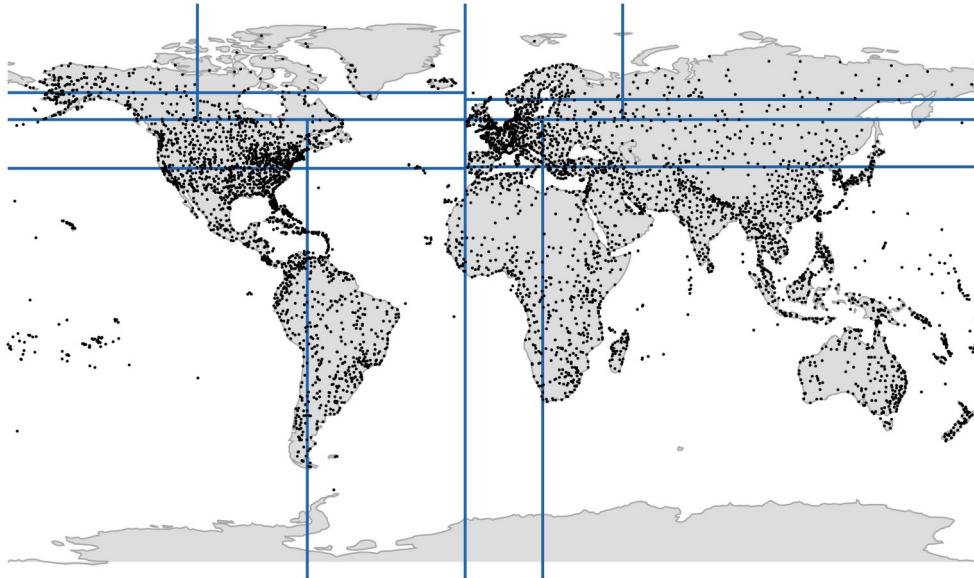
12

SP-GiST is short for space partitioning GiST. This is also a generalized search tree, but it is built by dividing the search space into non-overlapping regions.

Let's look at an example of an SP-GiST index for points on a plane.

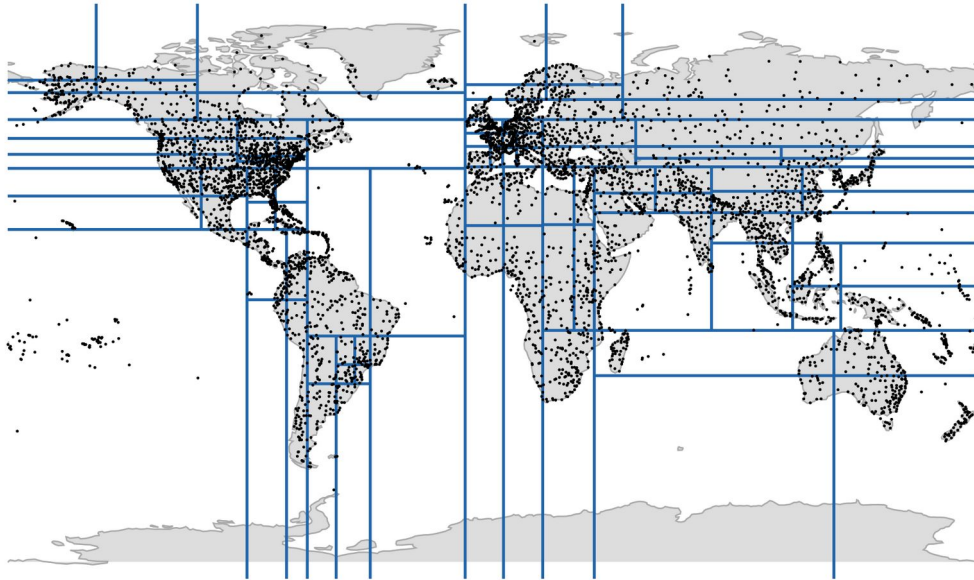
One of the options is a quadrant tree. The root node splits the plane into four quadrants relative to the selected centroid point.

# SP-GiST Index Example



Each of the four quadrants is further divided into sub-quadrants.

# SP-GiST Index Example



14

The splitting will continue until all points in the quadrant are contained within a single index page.

## Unbalanced search tree

- A sparsely branching tree with significant depth
- Supports arbitrary data types

## The operations are similar to those in GiST

- Inclusion within the area
- Determining left, right, top, and bottom positions relative to the area
- Nearest neighbor search

SP-GiST, similar to GiST, serves as a framework for building arbitrary indexing schemes through the implementation of operator classes. For example, the quadrant tree for points is implemented by the `point_ops` operator class. Another approach to dividing the plane is into two parts instead of four. This approach is known as a k-d tree, implemented by a different operator class — `kd_point_ops`.

Dividing the plane into non-overlapping regions results in unbalanced trees that typically exhibit limited branching and possess greater depth.

SP-GiST indexes typically support the same data types and operators as GiST. However, their different index structure can make them either more or less efficient than GiST.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

## SP-GiST Index SP-GiST Index

Let's create an SP-GiST index on the coordinates column in the airports\_data table. There are two operator classes for points. By default, point\_ops (quadrant tree) is used, but for our example, we'll use kd\_point\_ops (k-dimensional tree):

```
=> CREATE INDEX airports_spgist_idx ON airports_data
USING spgist(coordinates kd_point_ops);
```

```
CREATE INDEX
```

Let's try to find all airports located north of Nadym:

```
=> EXPLAIN (costs off)
SELECT airport_code
FROM airports_data
WHERE coordinates >^ '(72.69889831542969,65.48090362548828)::point;
```

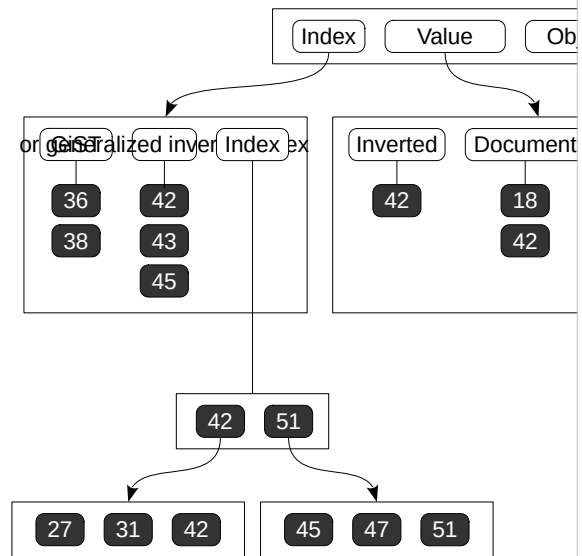
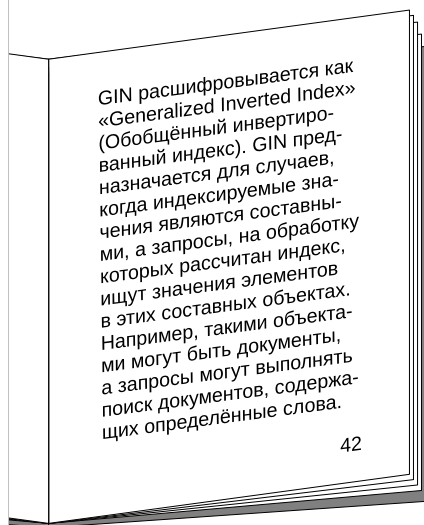
### QUERY PLAN

```
-----
Bitmap Heap Scan on airports_data
  Recheck Cond: (coordinates >^ '(72.69889831542969,65.48090362548828)::point)
    -> Bitmap Index Scan on airports_spgist_idx
          Index Cond: (coordinates >^ '(72.69889831542969,65.48090362548828)::point)
(4 rows)
```

Now the table is scanned using a bitmap generated from the SP-GiST index airports\_spgist\_idx.



# The Concept of the GIN Index



GIN — generalized inverted index, generalized inverted index.

The easiest way to understand this indexing method is by examining a subject index in a regular book. Terms appear on the book's pages, while the subject index lists all terms in alphabetical order, along with the page numbers where they appear.

The GIN index is primarily used for document indexing to speed up full-text searches. Essentially, it's a standard B-tree, but instead of storing the documents themselves, it stores the individual words that compose them. The GIN index is optimized for scenarios where each word may appear in multiple documents. If the "page list" is very large, it is stored in a separate B-tree instead of the index page itself.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

## Inverted List

For data types where the values (documents) are composed of elements  
Elements are indexed, not the documents themselves.

## Commonly Supported Operation

Check if a document matches a search query  
Check if an element is present in an array  
Search JSON documents by keys or values

GIN is designed for data types where values consist of elements rather than being atomic. Rather than indexing the values themselves, the elements are indexed.

Like GiST and SP-GiST, GIN is a framework that can be configured not only for text (consisting of words) but also for other data types, such as arrays (composed of elements) and JSON documents (containing keys and values). To accomplish this, an operator class is created to break down the document into elements and verify if it matches the search query.

## GIN Index

The `days_of_week` column in the `routes` view contains an array of weekday numbers representing the days the flight operates on:

```
=> SELECT flight_no, days_of_week FROM routes LIMIT 5;
```

```
flight_no | days_of_week
-----+-----
PG0001    | {6}
PG0002    | {7}
PG0003    | {2,6}
PG0004    | {3,7}
PG0005    | {2,5,7}
(5 rows)
```

A view cannot have an index built on it, so we'll store its rows in a separate table.

```
=> CREATE TABLE routes_tbl
AS SELECT * FROM routes;
```

```
SELECT 710
```

Now, let's create a GIN index:

```
=> CREATE INDEX routestbl_gin_idx ON routes_tbl USING gin(days_of_week);
```

```
CREATE INDEX
```

For example, a GIN index can be used to filter flights that operate exclusively on Wednesdays and Saturdays:

```
=> EXPLAIN (costs off)
SELECT flight_no, departure_airport_name AS departure,
       arrival_airport_name AS arrival, days_of_week
FROM routes_tbl
WHERE days_of_week = ARRAY[3,6];
```

```
               QUERY PLAN
-----
Bitmap Heap Scan on routes_tbl
  Recheck Cond: (days_of_week = '{3,6}'::integer[])
    -> Bitmap Index Scan on routestbl_gin_idx
          Index Cond: (days_of_week = '{3,6}'::integer[])
(4 rows)
```

The created GIN index contains just seven elements: the integers 1 through 7, representing the days of the week. Each entry in the index holds pointers to flights operating on the corresponding day.

# BRIN Index Example

```
SELECT * FROM t WHERE temperature BETWEEN 20 AND 30;
```

group  
sequentially  
arranged  
pages

range	pages	summary information
1	1128	010001 00
2	129256	010001 00
3	257384	010001 00
4	385512	010001 00
5	513640	010001 00
6	641768	010001 00
7	769896	010001 00
8	8971024	010001 00
9	1025 1152	010001 00

20

BRIN — block range index, «block range index». The table is divided into zones of a defined (configurable) length, each comprising a set of sequentially arranged pages. Each zone contains summary data, including the minimum and maximum values of the indexed column.

During query execution, you can skip all zones that are guaranteed not to satisfy the condition. As shown on the slide, only two zones could contain temperature values satisfying the condition.

Unlike other indexes, the BRIN index does not store row version identifiers, so all row versions in the selected zones must be examined. In a way, BRIN can be considered an accelerator for sequential scans.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

## List of Zones Containing Summary Information

- A zone spans a group of consecutive pages
- summary information: minimum, maximum, and so on
- Correlation with the physical location of rows is required.

## Does not maintain item pointers to tuple versions.

- Only a bit map scan

## Designed for very large tables

- compact size
- configurable size-accuracy ratio

Using operator classes, you can select the summary data stored in the index for each zone. This can range from just minimum and maximum values to multiple value ranges, and for geometric data types, it can store an enclosing rectangle (as in GiST).

Regardless of the case, BRIN requires a correlation between column values and the physical row location to function effectively, ensuring that values with similar summary information are grouped into the same zone. Data updates can disrupt the correlation, potentially impacting the index's efficiency.

Since BRIN does not store row version pointers, it returns an approximate bit map of the zone's pages. Standard index scans (and index-only scans) are not possible.

However, the BRIN index has a very small size and can be adjusted by specifying the zone size. The larger the zone, the smaller the index, but the lower the accuracy. This makes BRIN a perfect fit for massive tables commonly found in data warehouses.

## BRIN Index BRIN Index

As an example, let's create a BRIN index on the largest table:

```
=> CREATE INDEX tflights_brin_idx ON ticket_flights USING brin(flight_id);
```

CREATE INDEX

BRIN indexes are most effective for very large tables. However, the query planner can leverage the built index by pruning regions that don't contain the required values, using the minimum and maximum values from the summary data:

```
=> EXPLAIN (analyze, costs off, timing off)
SELECT *
FROM ticket_flights
WHERE flight_id BETWEEN 3000 AND 4000;
```

### QUERY PLAN

```
-----
Gather (actual rows=46357 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Bitmap Heap Scan on ticket_flights (actual rows=15452 loops=3)
        Recheck Cond: ((flight_id >= 3000) AND (flight_id <= 4000))
        Rows Removed by Index Recheck: 2377352
        Heap Blocks: lossy=19792
        -> Bitmap Index Scan on tflights_brin_idx (actual rows=598210 loops=1)
              Index Cond: ((flight_id >= 3000) AND (flight_id <= 4000))
Planning Time: 1.863 ms
Execution Time: 4503.617 ms
(11 rows)
```

Since BRIN does not store pointers to row versions, the only viable access method is scanning using a lossy (non-precise) bitmap.

Besides B-tree, there are other, more specialized index types.

Hash index for equality queries

GiST and SP-GiST for non-sortable data types

GIN for documents

BRIN for very large tables

1. Compare the size and build time of the hash index on columns with varying sizes (book\_ref and contact\_data) in the tickets table.

Do the same for a B-tree index.

2. Use the GIN index and pg\_trgm extension to find passengers whose phone numbers contain the digit sequence 1234.

Can a B-tree help speed up this query?

1. 1. To estimate the size of an index, use the pg\_total\_relation\_size function with the index name as a parameter.

2. 2. Create a GIN index on the expression contact\_data->>'phone' using the gin\_trgm\_ops operator class provided by the pg\_trgm extension.

Documentation page: <https://postgrespro.ru/docs/postgresql/16/pgtrgm>



## 1. Comparing Index Size and Creation Time

Let's take a look at the structure of the tickets table:

```
=> \d tickets
```

```
Table "bookings.tickets"
  Column      |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
ticket_no     | character(13)   |           | not null |
book_ref      | character(6)    |           | not null |
passenger_id  | character varying(20) |         | not null |
passenger_name | text           |           | not null |
contact_data   | jsonb          |           |          |
Indexes:
    "tickets_pkey" PRIMARY KEY, btree (ticket_no)
Foreign-key constraints:
    "tickets_book_ref_fkey" FOREIGN KEY (book_ref) REFERENCES bookings(book_ref)
Referenced by:
    TABLE "ticket_flights" CONSTRAINT "ticket_flights_ticket_no_fkey" FOREIGN KEY (ticket_no) REFERENCES tickets(ticket_no)
```

The book\_ref field is a fixed-size field, whereas the contact\_data field is of type jsonb.

Let's enable tracking of query execution time:

```
=> \timing on
```

Timing is on.

Let's create hash indexes...

```
=> CREATE INDEX tickets_hash_br ON tickets USING hash(book_ref);
```

```
CREATE INDEX
Time: 11264.156 ms (00:11.264)
```

```
=> CREATE INDEX tickets_hash_cd ON tickets USING hash(contact_data);
```

```
CREATE INDEX
Time: 18588.778 ms (00:18.589)
```

...and B-tree indexes:

```
=> CREATE INDEX tickets_btree_br ON tickets(book_ref);
```

```
CREATE INDEX
Time: 22455.442 ms (00:22.455)
```

```
=> CREATE INDEX tickets_btree_cd ON tickets(contact_data);
```

```
CREATE INDEX
Time: 66271.987 ms (01:06.272)
```

The time required to create hash indexes is roughly similar, whereas the time needed for B-tree indexes depends on the size of the indexed field.

```
=> \timing off
```

Timing is off.

Now let's check the sizes of the created indexes:

```
=> SELECT pg_size_pretty(pg_total_relation_size('tickets_hash_br')) "hash book_ref",
       pg_size_pretty(pg_total_relation_size('tickets_hash_cd')) "hash contact_data",
       pg_size_pretty(pg_total_relation_size('tickets_btree_br')) "btree book_ref",
       pg_size_pretty(pg_total_relation_size('tickets_btree_cd')) "btree contact_data" \gx
```

```
-[ RECORD 1 ]-----+-----
hash book_ref      | 81 MB
hash contact_data  | 80 MB
btree book_ref     | 59 MB
btree contact_data | 226 MB
```

Hash indexes also have a similar size. In contrast, B-tree indexes (and the time needed to create them) vary in size based on the indexed field's size, as they store the actual values.

## 2. pg\_trgm Extension

Let's run the query:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM tickets
WHERE contact_data->>'phone' LIKE '%1234%';
```

### QUERY PLAN

```
-----
Gather (actual rows=1801 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  Buffers: shared hit=128 read=49287
  -> Parallel Seq Scan on tickets (actual rows=600 loops=3)
        Filter: ((contact_data ->> 'phone'::text) ~ '%1234%'::text)
        Rows Removed by Filter: 982685
        Buffers: shared hit=128 read=49287
Planning:
  Buffers: shared hit=25 read=10 dirtied=3
Planning Time: 4.771 ms
Execution Time: 2838.301 ms
(12 rows)
```

Note the number of read pages (Buffers) — nearly fifty thousand.

Let's add the pg\_trgm extension:

```
=> CREATE EXTENSION pg_trgm;
```

CREATE EXTENSION

Let's create a GIN index using the gin\_trgm\_ops operator class:

```
=> CREATE INDEX tickets_gin
ON tickets USING GIN ((contact_data->>'phone') gin_trgm_ops);
```

CREATE INDEX

Such an operator class speeds up pattern matching for strings starting with a percent sign, including regular expressions. B-tree indexes cannot perform this.

Let's run the query again:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM tickets
WHERE contact_data->>'phone' LIKE '%1234%';
```

### QUERY PLAN

```
-----
Bitmap Heap Scan on tickets (actual rows=1801 loops=1)
  Recheck Cond: ((contact_data ->> 'phone'::text) ~ '%1234%'::text)
  Rows Removed by Index Recheck: 46
  Heap Blocks: exact=1820
  Buffers: shared hit=23 read=1818
  -> Bitmap Index Scan on tickets_gin (actual rows=1847 loops=1)
        Index Cond: ((contact_data ->> 'phone'::text) ~ '%1234%'::text)
        Buffers: shared hit=21
Planning:
  Buffers: shared hit=26 read=1 dirtied=1
Planning Time: 0.309 ms
Execution Time: 490.929 ms
(12 rows)
```

The number of read pages dropped by more than an order of magnitude, and the query execution time significantly decreased.