



GIN and GiST

CS541

Shivam Bhat

PUID : 0033760929



GIN

Gin stands for Generalized Inverted Index

Underlying structure is similar to a B-Tree.

Generalized refers to the fact that the index is unaware of the operation it is accelerating. It instead works with custom strategies, defined for specific data types

Hence in the given context, Gin is similar to GiST but differs from B-Trees indices, which have predefined comparison based operations.



An Introduction to Inverted Index?

Inverted Indexes find their popular use in keyword search.

In the context of keyword search, inverted index is an efficient data structure which maps each word from the dataset to the list of document IDs in which the word appears. This allows fast retrieval .

An inverted index for a collection of documents consists of a set of inverted lists also known as posting lists.

Each positions list corresponds to a word, storing all IDs of documents where this word occurs and in ascending order.

Due to large scale nature of datasets keyword search schemes employing Inverted indices often use various compression techniques to reduce cost of storing inverted indexes.

The main purpose of an inverted index hence is to allow fast textual search at a cost of increased processing when a new document is added to Database.

Inverted Index-Simple Example

Generalized inverted index improve on inverted index by encoding consecutive IDs in each inverted listed into intervals, and then uses efficient algorithms to support keyword search using these interval lists

Let $D=\{d_1, d_2 \dots d_N\}$ be a set of documents where in each document in D includes a set of words, and the set of all distinct words in D is denoted by W .

In the inverted index of D , each word $w \in W$ has an inverted list, denoted by I_w , which is an ordered list of IDs of documents that contain the word with all lists sorted in ascending order.

The figure shows collection of titles of 7 papers with its inverted indices

ID	Text
1	Keyword querying and ranking systems in a Database
2	Keyword searching and browsing in databases
3	Keyword search in relational databases
4	Efficient fuzzy type-ahead search
5	Navigation system for product search
6	Keyword search on spatial databases
7	Searching for databases
InvIndex	
Word	IDs
Keyword	1,2,3,6
...	...
Databases	1,2,3,6,7
Searching	2,7
Search	3,4,5,6



Need for GIN

Often an application might need sophisticated database features and data types such as JSONB, array types or full text search in Postgres.

For such a **use case where in one is required to index a JSONB column, a simple B-tree index wont work** and hence a GIN index is preferred.

The term "inverted" is used to describe a particular type of index structure where a table-encompassing tree of all column values is built. This means that a single row can be represented in multiple locations within the tree. **In contrast, a B-tree index typically has only one location where an index entry points to a specific row.**

Created by Teodor Sigaev and Oleg Bartunov on December 5, 2006 and released as part of Postgres 8.2.

Introduction to GIN

The authors Oleg Bartunov and Alexander Korotkov, at their **PGConf.EU 2012** in Prague explained GIN index like table of content in a book, where in the heap pointers act like the page numbers. Multiple entries are combined to get a specific result, like searching for the text “compensation accelerometers”

Report Index	
A	
abrasives, 27	
acceleration measurement, 58	
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74	
actuators, 4, 37, 46, 49	
adaptive Kalman filters, 60, 61	
adhesion, 63, 64	
adhesive bonding, 15	
adsorption, 44	
aerodynamics, 29	
	compensation, 30, 68
	compressive strength, 54
	compressors, 29
	computational fluid dynamics, 23, 29
	computer games, 56
	concurrent engineering, 14
	contact resistance, 47, 66
	convertors, 22
	coplanar waveguide components, 40
	Couette flow, 21
	creep, 17
	crystallisation, 64
	current density, 13, 16
QUERY: compensation accelerometers	
INDEX:	accelerometers
	5,10,25,28,30,36,58,59,61,73,74
	compensation
	30,68
RESULT:	30
	attitude measurement, 59, 61
	automatic test equipment, 71
	automatic testing, 24
B	
backward wave oscillators, 45	
	discrete wavelet transforms, 72
	displacement measurement, 11
	display devices, 56
	distributed feedback lasers, 38
	E



GIN

The primary usage of Gin Indices is to allow highly scalable, full textual search in PostgreSQL.

Search Query Types:

- Phrase Search: Retrieve record containing words in a given order
- Wildcard Search: Retrieve record matching a given pattern
- Proximity Search: Retrieve record where two words are within a specified proximity.

It does so by maintaining an inverted index structure consisting of pairs of form (key, “posting list”) wherein “posting list” is a set of heap rows in which the key occurs.



GIN

Its suited for multi valued data such as:

- jsonb
- tsvector
- srrays
- range type

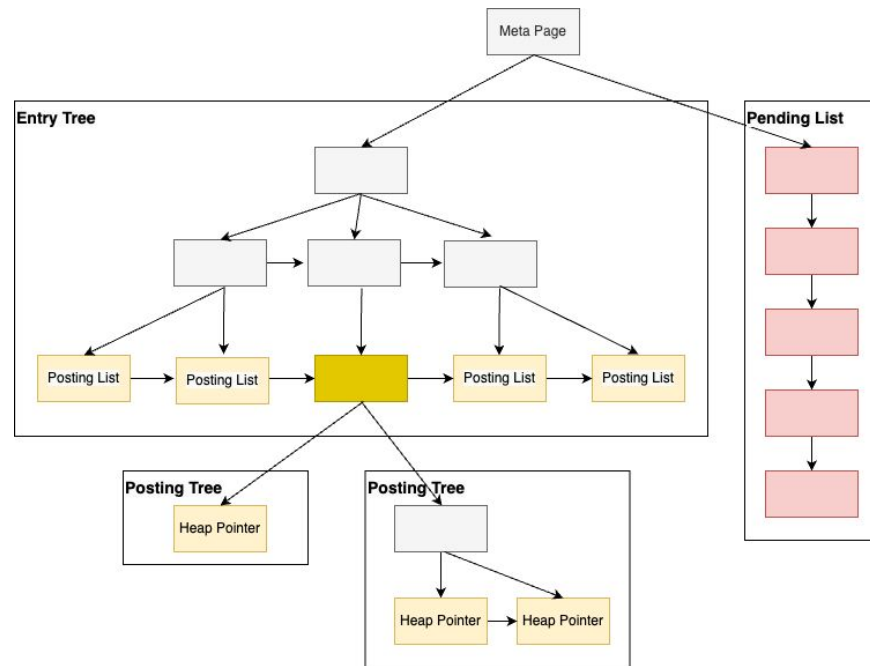
Index entry exist for each value hence can efficiently search for specific value.

Slower to built but fast to use

Since it store each individual possible value it often takes more space

GIN-Architecture

Overview





GIN-Architecture

Gin's Architecture is close to that of a B-Tree index.

Metapage: A BTree starts with a metapage containing information on the root and fast root block numbers and levels. The metapage is always the first page of a postgres index and contains the following:

- Level of root
- Block number of root
- Block number for fast root
- Level of fast root

When a B-tree is created, the metapage is typically the first page allocated and initialized. It serves as the starting point for traversing the tree and accessing its nodes.



GIN-Architecture

Metapage

Just like in BTree Index, **the first page of a GIN index is also its metapage** which contains information about the index.

However there is a slight difference in terms of what information is stored. In a Gin Index's metapage , the fast root is absent.

Information regarding the metapage can be see via “gin_metapage_info” [2] command

It consists of:

- n_total_pages: Total number of pages in the index
- n_entry_page: Number of entry pages
- n_entries: Number of entries in the index
- n_data_pages: Number of data pages



GIN-Architecture

Types of Pages

Gin index has following types of pages

- Entry Pages: Contain the values in the the index
- Data Pages: These are the pages inside a posting tree

Both of these page types have opaque data containing the following:

- **Flag** : describing the type (meta, leaf,data, compressed)
- **Right Sibling**
- **Maxoff: is a** parameter in a GIN index refers to the maximum offset value that can be stored in a posting list of the index. The maxoff value determines the maximum number of positions that can be stored in a single posting list.



GIN-Architecture

Entry Pages

The keys/Entries in a GIN Index are stored inside entry pages in form of a binary tree.

The way GIN Index's underlying B-Tree like structure differs from an actual B-Tree would be as follows:

- If one was to index an array in a BTree index then the values would directly be the array whereas in case of GIN index the array would be split and each value would be an entry
- Unlike a B-Tree, the pages in same level have only a right link.
- In a GIN index values are unique whereas in a B-tree, we could have several items with same value.



GIN-Architecture

Leaf Pages

In terms of internal pages, there's no meaningful difference between B-tree and GIN. What makes the difference are its *leaf pages*.

When using a GIN index in PostgreSQL, each entry in the index is unique. This means that at the leaf level of the index, there is either a **list** or a **tree of pointers** to the rows in the table that contain the indexed values.

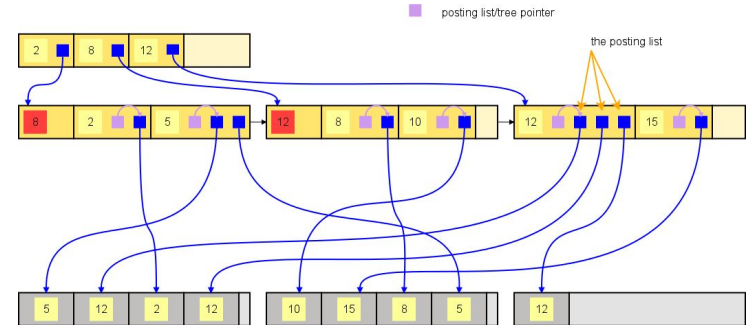
Posting list and Posting tree

- At the leaf level of a GIN index in PostgreSQL, each page contains a list of item pointers in a compressed format, known as a posting list.
- If the posting list grows too large to fit in a single index page, it is split into multiple pages organized as a B-tree.
- This structure of pages that store the posting lists split into multiple pages is called a posting tree.
- **Leaf item** would then store a **pointer to this tree** instead of the posting list.

GIN-Architecture

Posting list

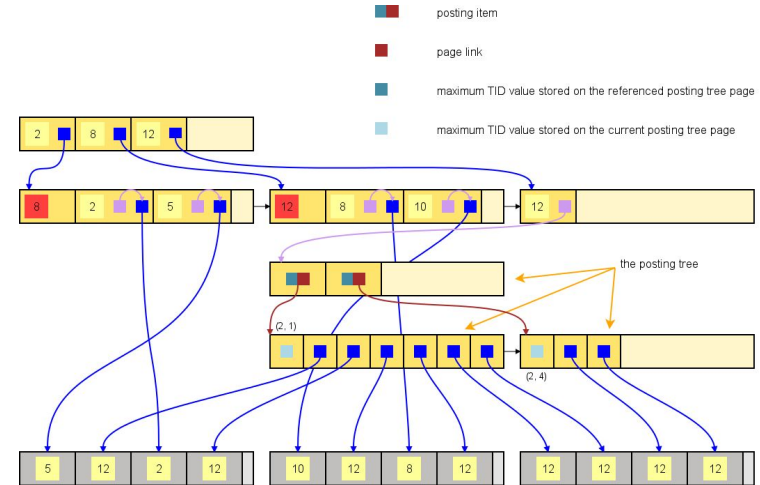
- In a GIN index in PostgreSQL, a single index tuple exists for each key entry on the index leaf page.
- However, the index tuple does not directly point to the corresponding rows..
- Instead, it points to a posting list, which is a list of pointers (known as TIDs) that indicate the location of all rows that contain the key value.
- Therefore to index a new row that already has key entry in the tree for the indexed column, we only need to find the entry and add pointer of the new row to the posting list. This does not trigger any page split in the index tree.



GIN-Architecture

Posting Tree

- When many rows with the same indexed attribute value are added, multiple pages are assigned to the TIDs (tuple identifiers) and this creates a posting tree.
- The posting tree is a distinct B-tree organization that resides on the pages of the GIN index.
- It uses the TID of the indexed row as its key, enabling the sorted maintenance of TIDs, similar to how we manage them in a posting list.





GIN-Architecture

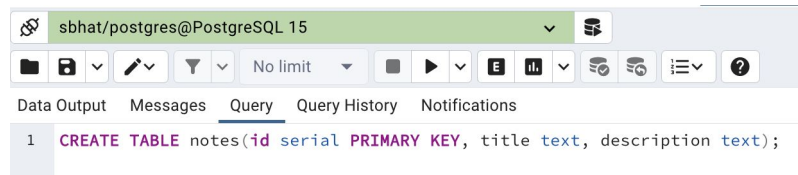
Pending List

- Inserting new rows in a GIN index can quite slow hence in order to optimize inserts, we store the new entries in a simple linear list of pages called the pending list.
- Hence It holds "pending" key entries that haven't yet been merged into the main btree.
- The pending list has a benefit in that that bulk insertion of a few thousand entries can be quicker than retail insertion i.e adding them one by one.
- The disadvantage of using the pending list is that it requires scanning both the BTree and the pending list when searching within a GIN index.

GIN Indices - Example

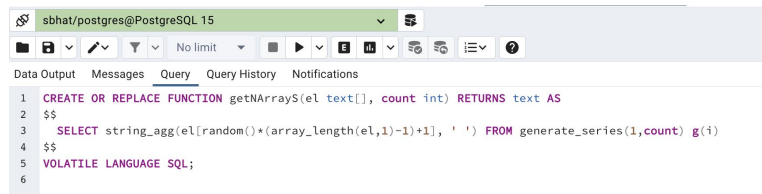
Creating Simple Baseline Indices

1. Create a Table 'notes' consisting of id, title, description

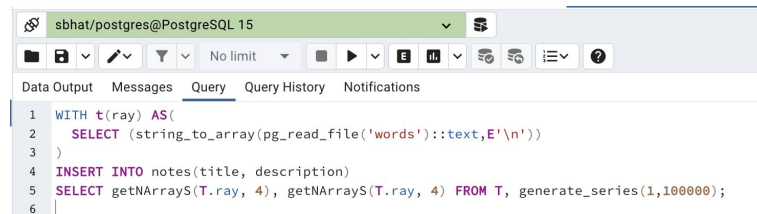


The screenshot shows a PostgreSQL query editor interface. At the top, the connection is 'sbhat/postgres@PostgreSQL 15'. Below the toolbar, the 'Query' tab is selected. The query text is: `1 CREATE TABLE notes(id serial PRIMARY KEY, title text, description text);`

2. Populate the data using macOS default word dictionary present at '`/usr/share/dict/`'



The screenshot shows a PostgreSQL query editor interface. At the top, the connection is 'sbhat/postgres@PostgreSQL 15'. Below the toolbar, the 'Query' tab is selected. The query text is: `1 CREATE OR REPLACE FUNCTION getNArrayS(el text[], count int) RETURNS text AS
2 $$
3 SELECT string_agg(el[random()*(array_length(el,1)-1)+1], ' ') FROM generate_series(1,count) g(i)
4 $$
5 VOLATILE LANGUAGE SQL;
6`



The screenshot shows a PostgreSQL query editor interface. At the top, the connection is 'sbhat/postgres@PostgreSQL 15'. Below the toolbar, the 'Query' tab is selected. The query text is: `1 WITH t(ray) AS(
2 SELECT (string_to_array(pg_read_file('words'))::text,E'\n')
3)
4 INSERT INTO notes(title, description)
5 SELECT getNArrayS(T.ray, 4), getNArrayS(T.ray, 4) FROM T, generate_series(1,100000);
6`

GIN Indices - Example

3. Q1 : Running a simple 'like' text search query

Total query runtime: **137 msec. 8598 rows affected**

4/21/2023 1:30:00 AM	8598	137 msec
Date	Rows affected	Duration

[Copy](#) [Copy to Query Editor](#)

```
SELECT * FROM notes WHERE title ilike '%per%';
```

Messages

Successfully run. Total query runtime: 137 msec. 8598 rows affected.

4. Q2 : Running a **AND** 'like' text search query

Total query runtime : **174 msec. 78 rows affected.**

4/21/2023 1:34:05 AM	78	174 msec
Date	Rows affected	Duration

[Copy](#) [Copy to Query Editor](#)

```
SELECT * from notes where title ilike '%per%' and description ilike '%gar%';
```

Messages

Successfully run. Total query runtime: 174 msec. 78 rows affected.

GIN Indices - Example

5. Q3 : Running a OR 'like' text search query

Total query runtime : **207 msec. 9482 rows affected.**

4/21/2023 1:39:34 AM

Date

9482

Rows affected

207 msec

Duration

Copy Copy to Query Editor

```
SELECT * from notes where title ilike '%per%' or description ilike '%gar%';
```

Messages

Successfully run. Total query runtime: 207 msec. 9482 rows affected.

6. Q4 : Running a Concatenation text search query

Total query runtime : **148 msec. 16402 rows affected.**

sbhat/postgres@PostgreSQL 15

No limit

Data Output Messages **Query** Query History Notifications

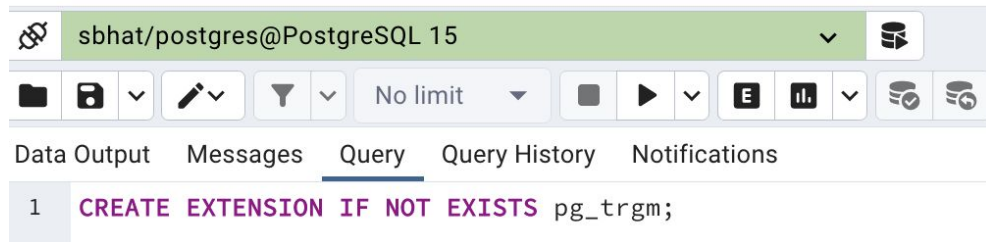
1 select * from notes where coalesce(title, '') || ' ' || coalesce(description, '') ilike '%per%';

GIN Indices - Example

Creating GIN Indices NOW

Creating Trigram extension: Creates a index with the first 3 letters of the words in the field

Creating an index just on the title field using the trigrams extension



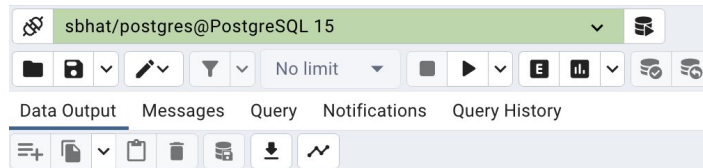
GIN Indices - Example

Creating GIN Indices

Running Q1 query again with GIN index

Total query runtime : 102 msec. 4 rows affected.

The query is using our GIN index '*notes_title_idx*' and takes only 102 msec instead of 137 msec which is **~25% improvement**.



The screenshot shows a PostgreSQL client interface with a green header bar displaying 'sbhat/postgres@PostgreSQL 15'. Below the header is a toolbar with icons for file operations, query execution, and other database functions. A tab labeled 'Data Output' is active, showing a table with 4 rows and 2 columns. The first row is 'QUERY PLAN' and the second row is 'text'. The table contains the following data:

	QUERY PLAN
1	Bitmap Heap Scan on notes (cost=106.28..1713.55 rows=10101 width=88)
2	Recheck Cond: (title ~* '%per%':text)
3	-> Bitmap Index Scan on notes_title_idx (cost=0.00..103.76 rows=10101 width=0)
4	Index Cond: (title ~* '%per%':text)

4/21/2023 1:55:18 AM	4	102 msec
Date	Rows affected	Duration

Copy Copy to Query Editor

```
EXPLAIN SELECT * FROM notes WHERE title ilike '%per%';
```

Messages

Successfully run. Total query runtime: 102 msec. 4 rows affected.



GIN Indices - Example

Creating GIN Indices

Running Q2 - AND query again with GIN index

Total query runtime : 67 msec. 78 rows
affected

4/21/2023 2:03:36 AM
Date

78
Rows affected

67 msec
Duration

Copy

Copy to Query Editor

```
SELECT * from notes where title ilike '%per%' and description ilike '%gar%';
```

Messages

Successfully run. Total query runtime: 67 msec. 78 rows affected.

The query is using our GIN index '*notes_title_idx*' and takes only 67 msec instead of 174 msec which is **~62% improvement**.

GIN Indices - Example

Creating GIN Indices

Running Q3 - OR query again with GIN index

Total query runtime : 79 msec. 78 rows affected

The query is using our GIN index '*notes_title_idx*' and takes only 79 msec instead of 207 msec which is **~62% improvement**.

4/21/2023 2:07:38 AM	2	79 msec
Date	Rows affected	Duration
Copy	Copy to Query Editor	
EXPLAIN SELECT * from notes where title ilike '%per%' or description ilike '%ge		
Messages		
Successfully run. Total query runtime: 79 msec. 2 rows affected.		



GIN Indices - Example

Creating GIN Indices

Lets Create a GIN description Index

```
CREATE INDEX notes_description_idx ON notes USING gin (description gin_trgm_ops);
```

On Running **Q3 - OR** query again with description GIN index it further speeds up

GIN Indices - Example

Creating GIN Indices

Q4 : Running a **Concatenation** text search query again we see that it doesn't use our indices hence we notice that its better to use OR query with expression index

4/21/2023 2:16:21 AM

Date

16402

Rows affected

181 msec

Duration

Copy

Copy to Query Editor

```
select * from notes where coalesce(title, '') || ' ' || coalesce(description, ''
```

Messages

Successfully run. Total query runtime: 181 msec. 16402 rows affected.

The screenshot shows a database query interface with a dropdown menu set to 'sbhat/postgres@PostgreSQL 15'. Below the menu is a toolbar with icons for file operations, filters, and execution. The main panel displays the 'QUERY PLAN' for the query. The plan consists of two steps: a 'Seq Scan on notes' and a 'Filter' step. The 'Filter' step is highlighted, showing the query condition: 'Filter: (((COALESCE(title, ''::text) || ''::text) || COALESCE(description, ''::text)) ~* '%per%'::te...'. The interface also shows tabs for 'Data Output', 'Messages', 'Query', 'Notifications', and 'Query History'.

Step	Operation
1	Seq Scan on notes (cost=0.00..3231.00 rows=4000 width=88)
2	Filter: (((COALESCE(title, ''::text) ''::text) COALESCE(description, ''::text)) ~* '%per%'::te...)

GIN Indices - Example

Creating GIN Indices

Q4 Query. Using Expression Index, it speeds our query by great amount

Total query runtime : 69 msec. 16402 rows affected

4/21/2023 2:22:51 AM

Date

16402

Rows affected

69 msec

Duration

Copy

Copy to Query Editor

```
select * from notes where coalesce(title, '') || ' ' || coalesce(description,
```

Messages

Successfully run. Total query runtime: 69 msec. 16402 rows affected.

```
select * from notes where coalesce(title, '') || ' ' || coalesce(description, '') ilike '%per%';
```



GiST

GiST is short for Generalized Search Tree and was first presented in the influential research paper "Generalized Search Trees for Database Systems" authored by Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer in 1995.

It was and implemented by J. Hellerstein and P. Aoki in an early version of PostgreSQL.



GiST

- GiST index is a balanced tree.
- Similar to a BTree, a GiST also uses a key that consists of a tuple of a value and a pointer. However, unlike a BTree, the data within a GiST is not ordered in the pages.
- Additionally, the key ranges of the pages in a GiST can overlap, which implies that a tuple may have the possibility of fitting in various pages.
- Its good for full-text search, used for geometric data types



GiST- Insert a new tuple

- **gistdoinsert()** handles inserts. In order to insert a new tuple, it will start from the root and descend the tree following the smallest “Penalty”.
- Recursively, until a leaf page is reached it performs:
 - Check for page split : In case of page split, the algorithm recurs back to the current page parent.
 - Find the best path: ‘**gistdoinsert()**’ calls the function ‘**gistchoose()**’ which finds the best tree branch to follow. For this it employs ‘**Penalty()**’ class to calculate the penalty associated with items of current page
 - Insert: After locating the correct page, the function ‘**gistinserttuples()**’ is invoked, which will perform either an insertion of the tuple if it is a leaf or update the value if it’s a parent. Internally “**gistplacetopage()**” handles the insert into pages.



GiST

- While calling the '**gistplacetopage()**' the code checks if there is sufficient space on the page to add the tuple. If there is enough space, it adds the tuple to the page, and there is no need to update the parent levels because they were already updated with the union key.
- When there isn't enough space on the page, a page split becomes necessary. However, page splits in a GiST index are different from those in a BTree.
- In a BTree, approximately half of the original page's data is moved to a new page. Since the data is already ordered, it's relatively easy to decide what should be placed on the left and right pages.
- In contrast, a GiST index aims to create groups of items with minimal distance between them to optimize searches, which makes page splits more complex.



GiST

- In order to create groups with minimal distance between items during a page split in a GiST index, the PickSplit method is used. This method is defined for each key class and determines which items will remain on the old page and which ones will be moved to the new page.
- If a page split is required, it is necessary to insert a new item with a pointer to the new page in the parent level by recursively moving up the hierarchy.



GiST - instantiate Index and Examples

I am using [demo database](#) provided by [openflights.org](#)

Since GiST is used for geometric and geospatial data I am creating a table of Points and then seeing how GiST index Speeds up queries

4/21/2023 9:08:51 PM

45 msec

Date

Rows affected

Duration

Copy

Copy to Query Editor

```
create table points(p point)
```

Messages

Query returned successfully in 45 msec.



Insert some points

GiST - instantiate Index and Examples

4/21/2023 9:11:54 PM	6	117 msec
Date	Rows affected	Duration

Copy Copy to Query Editor

```
insert into points(p) values
  (point '(1,1)'), (point '(4,2)'), (point '(6,3)'),
  (point '(5,5)'), (point '(6,6)'), (point '(7,7));
```

Messages

Query returned successfully in 117 msec.



GiST - instantiate Index and Examples

Create a **GiST** Index

4/21/2023 9:12:47 PM

38 msec

Date

Rows affected

Duration

Copy

Copy to Query Editor

```
create index on points using gist(p);
```

Messages

Query returned successfully in 38 msec.




GiST - instantiate Index and Examples

Switching off sequential Scan so it uses our indices. Since my dataset and Queries are not that heavy that it has to employ the index it often choose to use seq scan. Hence to test it I am switching it off

	Data Output	Messages	Query	Notifications	Query History
1			<code>set enable_seqscan = off;</code>		
2					

GiST - instantiate Index and Examples

Data Output	Mess												
 points_p_idx													
<table><thead><tr><th></th><th>p point</th></tr></thead><tbody><tr><td>1</td><td>(4,2)</td></tr><tr><td>2</td><td>(6,3)</td></tr><tr><td>3</td><td>(5,5)</td></tr><tr><td>4</td><td>(6,6)</td></tr><tr><td>5</td><td>(7,7)</td></tr></tbody></table>		p point	1	(4,2)	2	(6,3)	3	(5,5)	4	(6,6)	5	(7,7)	
	p point												
1	(4,2)												
2	(6,3)												
3	(5,5)												
4	(6,6)												
5	(7,7)												

points_p_idx	x
Node Type	Index Only Scan
Parallel Aware	false
Async Capable	false
Scan Direction	NoMovement
Index Name	points_p_idx
Relation Name	points
Alias	points
Index Cond	(p <@ '(8,8),(2,1)'::box)
loops	1

4/21/2023 9:18:39 PM

Date

5

Rows affected

45 msec

Duration

Copy

Copy to Query Editor

```
select * from points where p <@ box '(2,1),(8,8)';
```

Messages

Successfully run. Total query runtime: 45 msec. 5 rows affected.

Query employing GiST and search operators to search for points in between a given range



GiST - instantiate Index and Examples

Creating tsrange GiST index

Data Output Messages Explain × Query Notifications Query History

```
1 create table orders(during tsrange);
```

Data Output Messages Explain × Query Notifications Query History

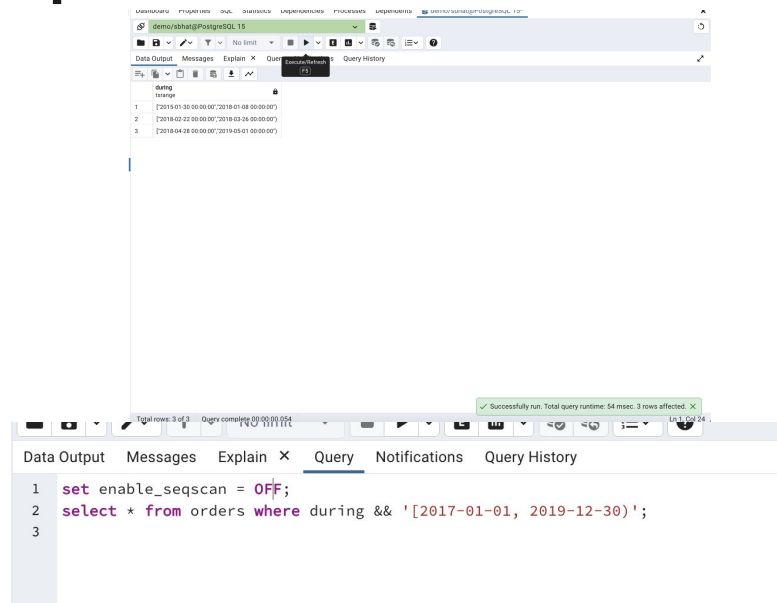
```
1 insert into orders(during) values
2 ('[2015-01-30, 2018-01-08]'),
3 ('[2018-02-22, 2018-03-26]'),
4 ('[2018-04-28, 2019-05-01]');
```

Data Output Messages Explain × Query Notifications Query History

```
1 create index on orders using gist(during);
```

GiST - instantiate Index and Examples

GiST index Speeds Up query for Range Search which took **54 msec**



The screenshot displays a PostgreSQL query editor interface. The top toolbar includes icons for running queries, saving, and other standard database operations. The main query area contains the following SQL code:

```
1 set enable_seqscan = OFF;  
2 select * from orders where during && '[2017-01-01, 2019-12-30)';  
3
```

Below the query, the 'Data Output' tab is active, showing the results of the query. The results are displayed in a table with three rows, each representing a range of dates:

during
[2015-01-30 00:00:00/2018-01-08 00:00:00)
[2018-02-22 00:00:00/2018-03-26 00:00:00)
[2018-04-28 00:00:00/2019-05-01 00:00:00)

At the bottom of the interface, a status bar indicates the query was executed successfully, showing 'Total rows: 3 of 3' and 'Successfully run. Total query runtime: 54 msec: 3 rows affected.' The date '15-12-24' is also visible in the bottom right corner.



GiST - instantiate Index and Examples

Without GiST Range Search scan
took **116 msec** for same query

```
Data Output  Messages  Explain ×  Query  Notifications  Query History
1  set enable_seqscan = ON;
2  select * from orders where during && '[2017-01-01, 2019-12-30)';
3
```




Thank You