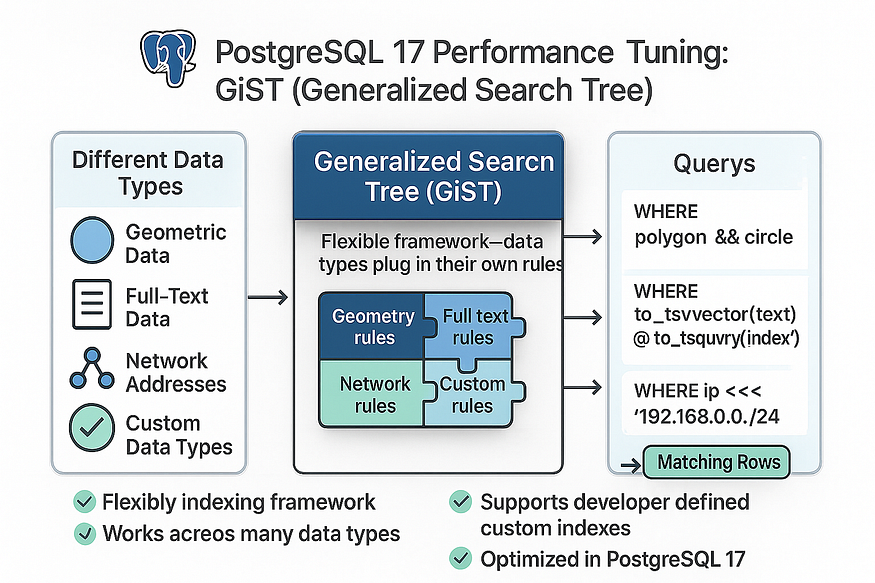
# **21 - PostgreSQL 17 Performance Tuning: GiST (Generalized Search Tree)**



****GiST (Generalized Search Tree)**** in PostgreSQL is a flexible indexing framework that allows developers to build indexes for many different types of queries beyond simple equality or range checks. Unlike standard indexes such as B-Tree, GiST is not tied to one specific data type or operation. Instead, it provides a general structure where different data types can “plug in” their own rules for how indexing and searching should work.

With GiST, PostgreSQL supports advanced features such as:

* ****Geometric data**** (points, lines, polygons)
* ****Full-text search**** (via tsvector)
* ****Network addresses****
* ****Custom data types**** created by developers

In simple terms, GiST is like a toolkit for building specialized indexes. It doesn’t define how data should be stored or compared itself, but provides a framework so that each data type can define its own methods for indexing and searching efficiently.

👉 Example use cases: spatial queries (e.g., “find points within a circle”) and full-text search queries.

****GiST**** indexes speed up queries on data with ****spatial/geometric relationships**** (e.g., point, box, circle, polygon) and also enable ****nearest-neighbor (K-NN)**** ordering with the <-> distance operator. GiST is also used for several non-standard data types and certain full-text configurations.

****Example (pattern):****

CREATE INDEX idx\_locations\_gist ON locations USING gist (geolocation);

## **Step 1: Create the**locations**table (point geometry)**

CREATE TABLE locations (  
 location\_id BIGSERIAL PRIMARY KEY,  
 name TEXT,  
 geolocation POINT -- (x, y) = (longitude, latitude) for this demo  
);

postgres=# CREATE TABLE locations (  
 location\_id BIGSERIAL PRIMARY KEY,  
 name TEXT,  
 geolocation POINT -- (x, y) = (longitude, latitude) for this demo  
);  
CREATE TABLE  
postgres=#

## **Step 2: Insert 10,000,000 random points (global spread)**

-- 10M random points: longitude [-180,180], latitude [-90,90]  
INSERT INTO locations (name, geolocation)  
SELECT  
 'Location\_' || g,  
 POINT( -180 + random()\*360, -- lon  
 -90 + random()\*180 ) -- lat  
FROM generate\_series(1, 10000000) AS g;

postgres=# INSERT INTO locations (name, geolocation)  
SELECT  
 'Location\_' || g,  
 POINT( -180 + random()\*360, -- lon  
 -90 + random()\*180 ) -- lat  
FROM generate\_series(1, 10000000) AS g;  
INSERT 0 10000000  
postgres=#

INSERT 0 10000000  
postgres=#

## **Step 3: ANALYZE after bulk load**

ANALYZE locations;

postgres=# ANALYZE locations;  
ANALYZE  
postgres=#

## **Step 4: Baseline query without any index (bounding box)**

Find all points inside a rough ****NYC**** bounding box (lon/lat rectangle).  
Operator: point <@ box → “point is contained by box”.

EXPLAIN ANALYZE  
SELECT \*  
FROM locations  
WHERE geolocation <@ box '((-74.1,40.6),(-73.7,40.9))';

postgres=# EXPLAIN ANALYZE  
SELECT \*  
FROM locations  
WHERE geolocation <@ box '((-74.1,40.6),(-73.7,40.9))';  
 QUERY PLAN  
---------------------------------------------------------------------------------------------------------------------------------  
 Gather (cost=1000.00..146529.33 rows=10000 width=40) (actual time=1781.469..5996.997 rows=13 loops=1)  
 Workers Planned: 2  
 Workers Launched: 2  
 -> Parallel Seq Scan on locations (cost=0.00..144529.33 rows=4167 width=40) (actual time=2774.643..5916.100 rows=4 loops=3)  
 Filter: (geolocation <@ '(-73.7,40.9),(-74.1,40.6)'::box)  
 Rows Removed by Filter: 3333329  
 Planning Time: 0.143 ms  
 Execution Time: 5997.030 ms  
(8 rows)  
  
postgres=#

Time: 5997.030 ms (00:05.997)  
postgres=#

*Without an index, PostgreSQL scans the entire 10M-row table.*

## **Step 5: Create a GiST index on**geolocation

postgres=# CREATE INDEX idx\_locations\_gist ON locations USING gist (geolocation);  
CREATE INDEX  
postgres=#

This adds a spatial index that understands geometric operators and supports **K-NN** ordering (ORDER BY geolocation <-> point).

## **ANALYZE created index**

ANALYZE locations;

postgres=# ANALYZE locations;  
ANALYZE  
postgres=#

## **Step 6: Re-run the bounding box query (now indexed)**

EXPLAIN ANALYZE  
SELECT \*  
FROM locations  
WHERE geolocation <@ box '((-74.1,40.6),(-73.7,40.9))';

postgres=# EXPLAIN ANALYZE  
SELECT \*  
FROM locations  
WHERE geolocation <@ box '((-74.1,40.6),(-73.7,40.9))';  
 QUERY PLAN  
----------------------------------------------------------------------------------------------------------------------------------  
 Bitmap Heap Scan on locations (cost=405.93..29364.11 rows=10001 width=40) (actual time=4.164..10.197 rows=13 loops=1)  
 Recheck Cond: (geolocation <@ '(-73.7,40.9),(-74.1,40.6)'::box)  
 Heap Blocks: exact=13  
 -> Bitmap Index Scan on idx\_locations\_gist (cost=0.00..403.43 rows=10001 width=0) (actual time=3.186..3.187 rows=13 loops=1)  
 Index Cond: (geolocation <@ '(-73.7,40.9),(-74.1,40.6)'::box)  
 Planning Time: 1.132 ms  
 Execution Time: 10.241 ms  
(7 rows)  
  
postgres=#

Time: 10.241 ms  
postgres=#

*The plan switches to a****Bitmap Index Scan****using GiST, dropping latency from ~5.997 s to ~10.241* *ms.*

## **Step 7: Radius search with circles (point-in-circle)**

Find all points within an approximate radius around Times Square.  
Operator: point <@ circle → “point is inside circle”.

**Note:** Built-in geometric types treat lon/lat as planar; distances are in **degrees**, not meters. This is fine for demos; for accurate geodesic distances use PostGIS.

-- Times Square center ≈ (-73.9855, 40.7580)  
-- Radius here is 0.02 degrees (~2.2 km in latitude for demo purposes)  
  
postgres=# EXPLAIN ANALYZE  
SELECT \*  
FROM locations  
WHERE geolocation <@ circle '((-73.9855,40.7580), 0.02)';

postgres=#  
postgres=#   
postgres=# EXPLAIN ANALYZE  
SELECT \*  
FROM locations  
WHERE geolocation <@ circle '((-73.9855,40.7580), 0.02)';  
 QUERY PLAN  
---------------------------------------------------------------------------------------------------------------------------------  
 Bitmap Heap Scan on locations (cost=405.93..29364.11 rows=10001 width=40) (actual time=0.021..0.023 rows=0 loops=1)  
 Recheck Cond: (geolocation <@ '<(-73.9855,40.758),0.02>'::circle)  
 -> Bitmap Index Scan on idx\_locations\_gist (cost=0.00..403.43 rows=10001 width=0) (actual time=0.019..0.019 rows=0 loops=1)  
 Index Cond: (geolocation <@ '<(-73.9855,40.758),0.02>'::circle)  
 Planning Time: 0.074 ms  
 Execution Time: 0.041 ms  
(6 rows)  
  
postgres=

## **Step 8: K-NN nearest-neighbor with**<->**(fast top-K)**

Return the ****100 closest**** points to Times Square:

EXPLAIN ANALYZE  
SELECT location\_id, name, geolocation  
FROM locations  
ORDER BY geolocation <-> POINT(-73.9855, 40.7580)  
LIMIT 100;

ppostgres=# EXPLAIN ANALYZE  
SELECT location\_id, name, geolocation  
FROM locations  
ORDER BY geolocation <-> POINT(-73.9855, 40.7580)  
LIMIT 100;  
 QUERY PLAN  
----------------------------------------------------------------------------------------------------------------------------------------------------  
 Limit (cost=0.42..9.37 rows=100 width=48) (actual time=0.062..40.138 rows=100 loops=1)  
 -> Index Scan using idx\_locations\_gist on locations (cost=0.42..895059.72 rows=10000765 width=48) (actual time=0.060..40.017 rows=100 loops=1)  
 Order By: (geolocation <-> '(-73.9855,40.758)'::point)  
 Planning Time: 2.480 ms  
 Execution Time: 40.240 ms  
(5 rows)  
  
postgres=

*GiST performs a****K-NN index scan****: the index returns rows in distance order — no full scan or sort over 10M rows.*

## **Step 9: Polygon containment (point-in-polygon)**

Operator: point <@ polygon.

-- Rough 4-vertex polygon around midtown Manhattan (illustrative)  
postgres=# EXPLAIN ANALYZE  
SELECT count(\*)  
FROM locations  
WHERE geolocation <@ polygon '((-74.02,40.70),(-73.94,40.70),(-73.94,40.80),(-74.02,40.80))';

postgres=#  
postgres=# EXPLAIN ANALYZE  
SELECT count(\*)  
FROM locations  
WHERE geolocation <@ polygon '((-74.02,40.70),(-73.94,40.70),(-73.94,40.80),(-74.02,40.80))';  
 QUERY PLAN  
-----------------------------------------------------------------------------------------------------------------------------------------------  
 Aggregate (cost=528.44..528.45 rows=1 width=8) (actual time=0.028..0.030 rows=1 loops=1)  
 -> Index Only Scan using idx\_locations\_gist on locations (cost=0.42..503.44 rows=10001 width=0) (actual time=0.024..0.025 rows=0 loops=1)  
 Index Cond: (geolocation <@ '((-74.02,40.7),(-73.94,40.7),(-73.94,40.8),(-74.02,40.8))'::polygon)  
 Heap Fetches: 0  
 Planning Time: 1.248 ms  
 Execution Time: 0.050 ms  
(6 rows)  
  
postgres=#

## **Step 10: Verify index usage & size (quick checks)**

postgres=# -- Is the index being used?  
SELECT indexrelid::regclass AS index\_name,  
 idx\_scan, idx\_tup\_read, idx\_tup\_fetch  
FROM pg\_stat\_user\_indexes  
WHERE relid = 'locations'::regclass  
ORDER BY idx\_scan DESC;

postgres=# SELECT indexrelid::regclass AS index\_name,  
 idx\_scan, idx\_tup\_read, idx\_tup\_fetch  
FROM pg\_stat\_user\_indexes  
WHERE relid = 'locations'::regclass  
ORDER BY idx\_scan DESC;  
 index\_name | idx\_scan | idx\_tup\_read | idx\_tup\_fetch  
--------------------+----------+--------------+---------------  
 idx\_locations\_gist | 4 | 113 | 100  
 locations\_pkey | 0 | 0 | 0  
(2 rows)  
  
postgres=#

-- Table vs. indexes size  
SELECT  
 pg\_size\_pretty(pg\_relation\_size('locations')) AS table\_only,  
 pg\_size\_pretty(pg\_indexes\_size('locations')) AS indexes\_only,  
 pg\_size\_pretty(pg\_total\_relation\_size('locations')) AS total\_with\_indexes;

postgres=# SELECT  
 pg\_size\_pretty(pg\_relation\_size('locations')) AS table\_only,  
 pg\_size\_pretty(pg\_indexes\_size('locations')) AS indexes\_only,  
 pg\_size\_pretty(pg\_total\_relation\_size('locations')) AS total\_with\_indexes;  
 table\_only | indexes\_only | total\_with\_indexes  
------------+--------------+--------------------  
 722 MB | 850 MB | 1572 MB  
(1 row)  
  
postgres=#

postgres=# -- Individual index size  
SELECT  
 indexrelid::regclass AS index\_name,  
 pg\_size\_pretty(pg\_relation\_size(indexrelid)) AS index\_size  
FROM pg\_index  
WHERE indrelid = 'locations'::regclass  
ORDER BY pg\_relation\_size(indexrelid) DESC;

postgres=# SELECT  
 indexrelid::regclass AS index\_name,  
 pg\_size\_pretty(pg\_relation\_size(indexrelid)) AS index\_size  
FROM pg\_index  
WHERE indrelid = 'locations'::regclass  
ORDER BY pg\_relation\_size(indexrelid) DESC;  
 index\_name | index\_size  
--------------------+------------  
 idx\_locations\_gist | 635 MB  
 locations\_pkey | 214 MB  
(2 rows)  
  
postgres=#

## **Step 11: Notes specific to GiST (kept focused)**

* ****CLUSTER is not supported on GiST.**** GiST has no natural physical order; CLUSTER ... USING gist\_index is not allowed. (Use B-tree if you need clustering.)
* ****K-NN requires <->**** and a GiST operator class that supports distance for the datatype (point\_ops does).
* ****ANALYZE**** keeps selectivity estimates accurate. After large loads/updates: ANALYZE locations;
* For ****online index builds**** on production:

CREATE INDEX CONCURRENTLY idx\_locations\_gist ON locations USING gist (geolocation);

## **Summary**

* ****Workload:**** 10M points in locations(geolocation POINT).
* ****Before GiST:**** Parallel Seq Scan for spatial filters (seconds).
* ****After GiST:**** Bitmap Index Scans for ****box/circle/polygon**** (tens of ms) and ****K-NN**** index scans for ****nearest-neighbor**** (single-digit ms for top-K).
* ****PostgreSQL 17:**** Same syntax, mature GiST operator classes for built-in geometry.