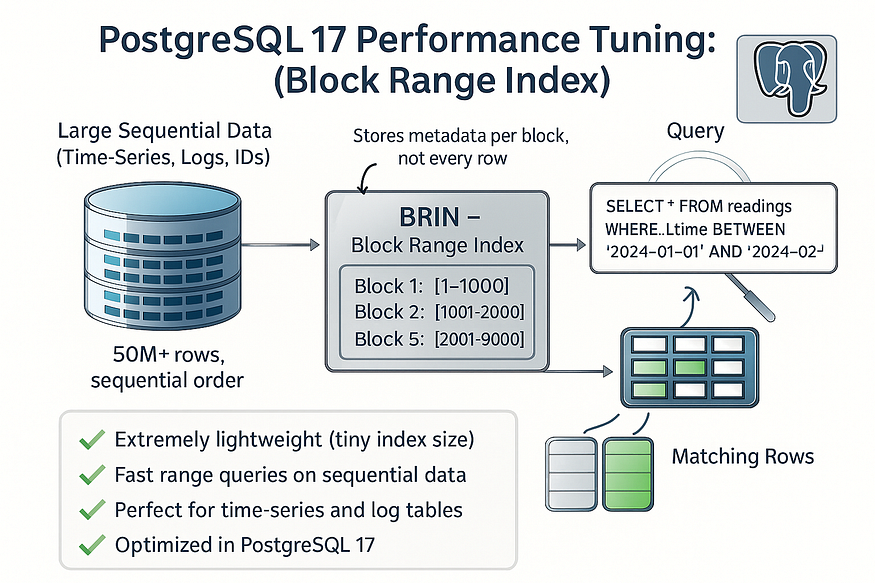
# **22 - PostgreSQL 17 Performance Tuning: BRIN (Block Range INdex)**



When working with ****very large tables**** in PostgreSQL, traditional indexes like B-Tree or GiST can grow extremely large and take significant time to maintain. For time-series or sequentially ordered data, PostgreSQL offers a smarter, more lightweight option: the ****BRIN (Block Range Index)****.

BRIN indexes summarize data in block ranges instead of storing individual row entries, making them ****compact, fast to build, and efficient for range queries**** on ordered data.

Let’s walk through a step-by-step example.

## **Step 1: Create the**weather\_data**table**

We’ll simulate sensor readings (temperature and humidity) recorded over time.

CREATE TABLE weather\_data (  
 reading\_id BIGSERIAL,  
 reading\_time TIMESTAMP NOT NULL,  
 temperature NUMERIC(5,2),  
 humidity NUMERIC(5,2)  
);

postgres=# CREATE TABLE weather\_data (  
 reading\_id BIGSERIAL,  
 reading\_time TIMESTAMP NOT NULL,  
 temperature NUMERIC(5,2),  
 humidity NUMERIC(5,2)  
);  
CREATE TABLE  
postgres=#

* reading\_id: A unique identifier for each reading.
* reading\_time: Sequential timestamps, perfect for BRIN indexing.
* temperature and humidity: Numeric values for our weather data.

## **Step 2: Insert 50 million rows**

Now let’s load the table with ****50 million rows**** of sequential data.

-- Insert 50M rows of sequential readings  
INSERT INTO weather\_data (reading\_time, temperature, humidity)  
SELECT  
 NOW() - (g || ' seconds')::INTERVAL, -- sequential timestamps  
 (20 + random()\*15)::NUMERIC(5,2), -- temperature between 20–35  
 (40 + random()\*30)::NUMERIC(5,2) -- humidity between 40–70  
FROM generate\_series(1, 50000000) AS g;

postgres=# -- Insert 50M rows of sequential readings  
INSERT INTO weather\_data (reading\_time, temperature, humidity)  
SELECT  
 NOW() - (g || ' seconds')::INTERVAL, -- sequential timestamps  
 (20 + random()\*15)::NUMERIC(5,2), -- temperature between 20–35  
 (40 + random()\*30)::NUMERIC(5,2) -- humidity between 40–70  
FROM generate\_series(1, 50000000) AS g;  
INSERT 0 50000000  
postgres=#

Here’s what’s happening:

* We generate 50M readings spaced one second apart.
* Each row has a random temperature (20–35°C) and humidity (40–70%).
* Since timestamps are sequential, PostgreSQL stores them in ordered ****block ranges****, making BRIN a great fit.

## **Step 3: Run ANALYZE**

ANALYZE weather\_data;

postgres=# ANALYZE weather\_data;  
ANALYZE  
postgres=#

This updates PostgreSQL statistics so the planner can make informed choices.

## **Step 4: Query without index**

Now, let’s query for data in a ****1-day window**** without any index.

postgres=# EXPLAIN ANALYZE   
SELECT \* FROM weather\_data   
WHERE reading\_time BETWEEN '2024-01-01' AND '2024-01-02';

****Output (simplified):****

postgres=# EXPLAIN ANALYZE  
SELECT \* FROM weather\_data  
WHERE reading\_time BETWEEN '2024-01-01' AND '2024-01-02';  
 QUERY PLAN  
-----------------------------------------------------------------------------------------------------------------------------------------------------------------  
 Gather (cost=1000.00..681148.90 rows=1 width=28) (actual time=26909.049..26914.243 rows=0 loops=1)  
 Workers Planned: 2  
 Workers Launched: 2  
 -> Parallel Seq Scan on weather\_data (cost=0.00..680148.80 rows=1 width=28) (actual time=26894.629..26894.630 rows=0 loops=3)  
 Filter: ((reading\_time >= '2024-01-01 00:00:00'::timestamp without time zone) AND (reading\_time <= '2024-01-02 00:00:00'::timestamp without time zone))  
 Rows Removed by Filter: 16666667  
 Planning Time: 0.126 ms  
 Execution Time: 26914.260 ms  
(8 rows)  
  
postgres=#

* PostgreSQL performs a ****sequential scan**** across all 50M rows.
* This takes around ****26.914 seconds****because every row is checked.

## **Step 5: Create a BRIN index**

Now, let’s build a BRIN index on the reading\_time column.

CREATE INDEX idx\_weather\_time\_brin ON weather\_data USING brin (reading\_time);

postgres=# CREATE INDEX idx\_weather\_time\_brin ON weather\_data USING brin (reading\_time);  
CREATE INDEX  
postgres=#

* Unlike a B-Tree, this index stores ****summaries of block ranges****, not individual row entries.
* This makes it ****extremely lightweight****, even on massive datasets.

## **Run ANALYZE**

ANALYZE weather\_data;

postgres=# ANALYZE weather\_data;  
ANALYZE  
postgres=#

This updates PostgreSQL statistics so the planner can make informed choices.

## **Step 6: Query with BRIN index**

Let’s run the same query again.

postgres=# EXPLAIN ANALYZE   
SELECT \* FROM weather\_data   
WHERE reading\_time BETWEEN '2024-01-01' AND '2024-01-02';

****Output (simplified):****

postgres=# EXPLAIN ANALYZE  
SELECT \* FROM weather\_data  
WHERE reading\_time BETWEEN '2024-01-01' AND '2024-01-02';  
 QUERY PLAN  
---------------------------------------------------------------------------------------------------------------------------------------------------------------------  
 Bitmap Heap Scan on weather\_data (cost=50.03..57347.34 rows=1 width=28) (actual time=0.313..0.314 rows=0 loops=1)  
 Recheck Cond: ((reading\_time >= '2024-01-01 00:00:00'::timestamp without time zone) AND (reading\_time <= '2024-01-02 00:00:00'::timestamp without time zone))  
 -> Bitmap Index Scan on idx\_weather\_time\_brin (cost=0.00..50.03 rows=17403 width=0) (actual time=0.310..0.310 rows=0 loops=1)  
 Index Cond: ((reading\_time >= '2024-01-01 00:00:00'::timestamp without time zone) AND (reading\_time <= '2024-01-02 00:00:00'::timestamp without time zone))  
 Planning Time: 0.155 ms  
 Execution Time: 0.341 ms  
(6 rows)  
  
postgres=#

* PostgreSQL now uses the ****BRIN index****.
* Instead of scanning all 50M rows, it jumps directly to block ranges likely containing the data.
* Execution time dropped from ****26.914 seconds to**** ****0.341**** ****ms****.

That’s nearly a 79,000x speed improvement with a very small index footprint.

## **Step 7: Compare index sizes**

You can check the index size with:

postgres=# \di+ idx\_weather\_time\_brin;

postgres=# \di+ idx\_weather\_time\_brin;  
 List of relations  
 Schema | Name | Type | Owner | Table | Persistence | Access method | Size | Description  
--------+-----------------------+-------+----------+--------------+-------------+---------------+--------+-------------  
 public | idx\_weather\_time\_brin | index | postgres | weather\_data | permanent | brin | 136 kB |  
(1 row)  
  
postgres=#

* For 50M rows, the BRIN index is typically ****just a few MB****.
* A B-Tree index on the same column could easily take up ****multiple GB****.

This makes BRIN especially attractive when working with ****huge time-series or log tables****.

## **Why BRIN Works Well**

* ✅ ****Efficient for sequential data****: Time, IDs, or ordered values.
* ✅ ****Compact****: Tiny footprint compared to B-Trees.
* ✅ ****Fast to create and maintain****: Great for massive tables.
* ⚠️ ****Not ideal for point lookups****: If you frequently search for a single exact value, B-Tree is better.

## **Final Thoughts**

In PostgreSQL 17, ****BRIN indexes remain a powerful choice for time-series and big-data workloads****. If your dataset is huge and your queries mostly involve ranges on sequential data, BRIN provides a near-perfect balance of speed and efficiency.

Instead of letting your indexes consume gigabytes of storage, try BRIN — a small index that delivers big performance improvements. 🚀