Mastering PostgreSQL Performance: Identifying Slow Queries and Leveraging Indexes for Optimization

Efficient database performance is essential for scalable, high-performing applications. PostgreSQL offers a rich set of tools to identify slow queries and optimize them through various techniques, including indexing strategies. Understanding how to capture slow queries, analyze execution plans, and apply appropriate indexes — such as composite, covering, and clustered indexes — can significantly reduce query response times.

Database TableUsed in Examples

In this article, all examples are based on the following table schema:

```
CREATE TABLE customers (
   id SERIAL PRIMARY KEY,
   name TEXT,
   email TEXT,
   city TEXT,
   created_at DATE
);
```

Now, suppose this table contains thousands of rows.

Identifying and Optimizing Slow Queries in PostgreSQL
Step 1 : Enable Query Logging

Before you can fix slow queries, you need to capture them. PostgreSQL allows you to log long-running statements by configuring the postgresql.conf file.

```
-- Log all queries running longer than 100msSET
log_min_duration_statement = 500;

-- Or in postgresql.conf:
```

```
logging_collector = on
log_min_duration_statement = 500  # logs queries longer than 500ms
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
```

This configuration logs any query that takes more than half a second to execute. Once set, restart PostgreSQL to apply the changes.

Step 2 : Analyze Logs with pgBadger

Manually digging through logs is time-consuming. That's where **pgBadger** comes in. It's a powerful log analyzer that transforms raw logs into interactive HTML reports.

```
pgbadger /var/log/postgresql/postgresql-*.log -o report.html
```

In Windows : C:\Program Files\PostgreSQL\<version>\data\log

Key insights from the report :

- Top slowest queries
- Most frequent queries
- Queries causing locks
- Checkpoint and vacuum activities
- Connection stats per user or app

This gives you a high-level overview of where your database is spending time.

Step 3 : Dive Deep with EXPLAIN ANALYZE

Once you've identified a slow query, use PostgreSQL's execution planner to understand **why** it's slow:

```
EXPLAIN (ANALYZE, BUFFERS) <your_query>;
```

- EXPLAIN: Shows the planned execution steps (e.g., scans, joins).
- ANALYZE: Actually executes the query and adds real runtime metrics.
- BUFFERS: Shows buffer usage (i.e., memory/disk reads).

  Note : Even if an index exists on a column, PostgreSQL may choose not to use it — especially when the table is small. In such cases, a sequential scan is often faster and more efficient than using the index, so PostgreSQL deliberately ignores the index to optimize performance.

```
EXPLAIN (ANALYZE, BUFFERS)SELECT * FROM orders WHERE customer_id
= 123;
```

Sample Output :

```
Index Scan using idx_orders_customer_id on orders
(cost=0.42..8.44 rows=3 width=64)
(actual time=0.030..0.060 rows=3 loops=1)
Buffers: shared hit=4
```

- Index Scan using idx_orders_customer_id on orders : PostgreSQL used an index scan
- cost=0.42..8.44 : Estimated cost - 0.42: startup cost - 8.44: total cost ⍰Lower cost = better
- rows=3: PostgreSQL estimated 3 rows
- actual time=0.030..0.060 rows=3: The query actually returned 3 rows in ~0.06 ms
- loops=1: This operation ran once. - In nested loops, this can be higher.
- Buffers: shared hit=4 : All pages were found in memory (shared buffer cache).
- ⍰hit : read from memory
- ⍰read : read from disk
- ⍰written : pages written to disk
- ⍰ Ideally, you want high hit values and low reads to minimize disk I/O.

Step 4: Use pg_stat_statements for Query Profiling

```
-- First enable the extension
CREATE EXTENSION pg_stat_statements;
-- Then query the slowest statements
SELECT query, total_time, calls, mean_timeFROM
pg_stat_statementsORDER BY mean_time DESC
LIMIT 10;
```

It helps find **frequently run or costly queries**.

Step 5: Keep Statistics Up to Date

```
ANALYZE;
```

PostgreSQL keeps some **summary information** about the data in your tables — like how many rows there are, how the values in a column are spread out, and which values are most common. This is called **statistics**.

The database uses these statistics to **guess how many rows** a query will return before running it.

What does ANALYZE do ? is a command that tells PostgreSQL to **look at the current data** in a table and **update these statistics**.

What is EXPLAIN ANALYZE ? runs your query and tells you two things :

- How many rows PostgreSQL expected the query to return (using its statistics)
- How many rows the query actually returned

Simple example : Imagine a table with 1 million rows.

- The statistics say: The query will return **10 rows**.
- But in reality, the query returns **10,000 rows**.

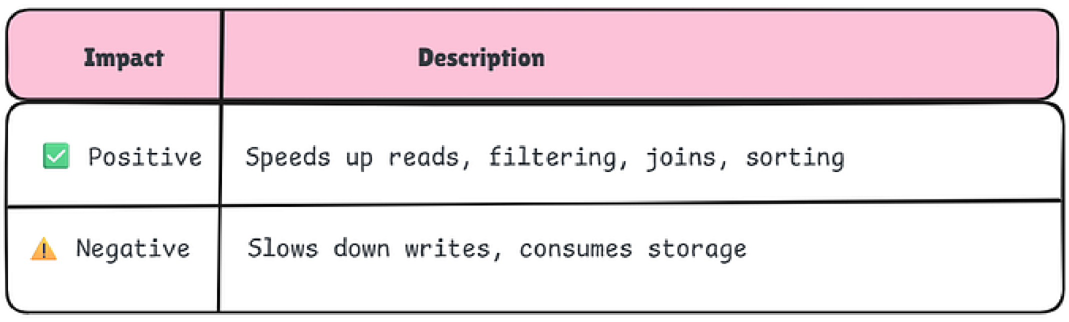
Because of this wrong guess, PostgreSQL might choose a **slow plan**.

□Running ANALYZE updates the statistics to better reflect reality, so PostgreSQL makes better decisions next time.

Step 7 : Use Materialized Views or Caching (When Applicable)

If a query involves heavy aggregation or joins over large datasets and doesn't need real-time data, a **materialized view** might help.

Index



Impact	Description
✅ Positive	Speeds up reads, filtering, joins, sorting
⚠️ Negative	Slows down writes, consumes storage

Clustered vs. Non-Clustered Index
Non-Clustered Index (default in PostgreSQL)

When you create a regular index :

```
CREATE INDEX idx_customers_city ON customers(city);
```

- PostgreSQL creates a **B-tree index** on the city column. **B-tree** stands for **Balanced Tree**. It's a **tree-like data structure** that keeps data **sorted** and allows **fast search, insert, update, and delete** operations. It is the default index type in PostgreSQL

```
CREATE INDEX idx_customers_city ON customers USING BTREE (city);
```

- But the **actual data in the table is not reordered**.
- The index simply contains : The value of city A pointer (row reference) to the corresponding row in the customerstable

So when you run:

```
SELECT * FROM customers WHERE city = 'Brussels';
```

PostgreSQL may do an **Index Scan** on idx_customers_city, like:

```
Index Scan using idx_customers_city on customers
  Index Cond: (dcity = 'Brussels')
```

- PostgreSQL navigates the **B-tree index** on customers.
- Finds matching values (Brussels).
- Follows pointers to the actual rows in the table.
- Returns them — without scanning the whole table.

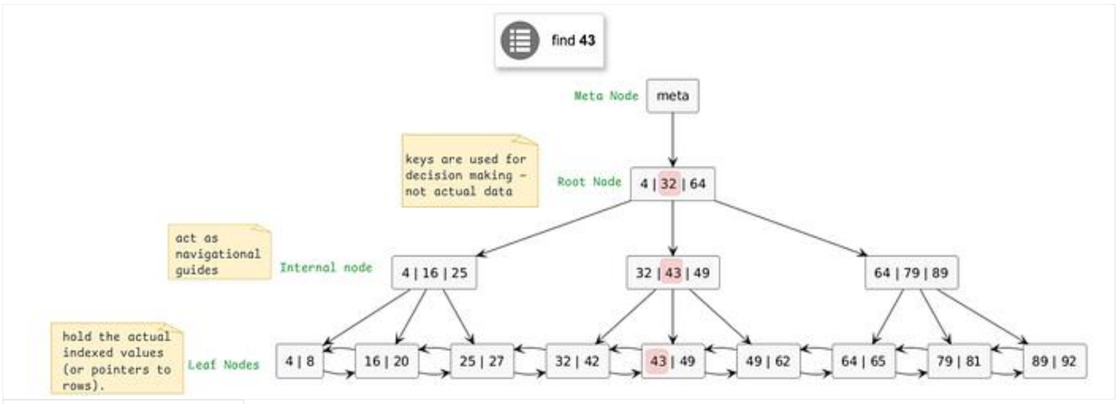
Example of a B-tree structure :

Supposons que l'image représente un index B-tree sur la colonne id de la table customers.

Dans PostgreSQL, un **index** (comme un B-tree) ne contient pas **les données complètes** de la table. À la place, il contient :

- Les **valeurs indexées** (ex : 43)

- Un **pointeur** vers l'emplacement exact de la ligne dans la table (le **TID**). ➞Le **TID (Tuple ID)** indique : □RLe numéro de bloc (page dans le fichier) □RL'offset (position de la ligne dans cette page).



Clustered Index

- A **clustered index defines the physical order** of the rows in the table.
- There can be **only one clustered index** per table, because the data rows can only be stored in one order.
- The table **is** the index.

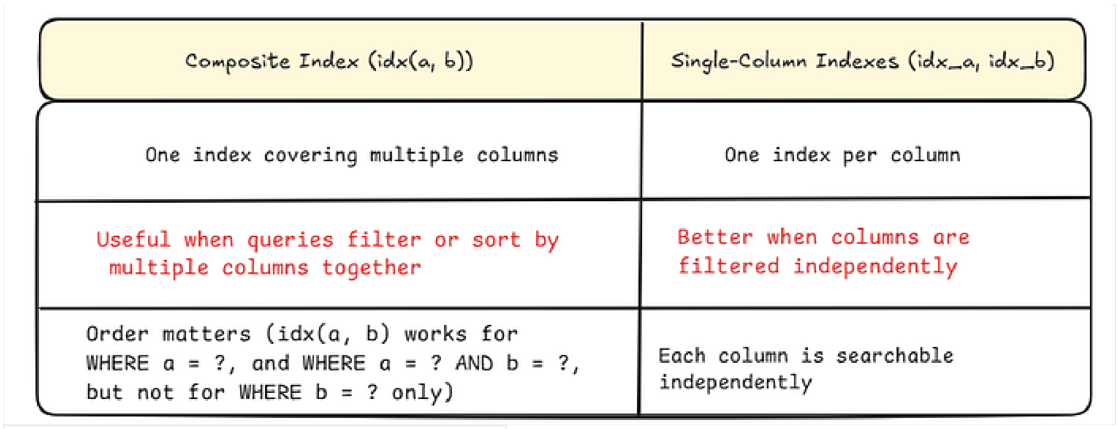
If you create a clustered index on id , the table's rows will be stored on disk **sorted by id**.

□PostgreSQL **does not** support clustered indexes in the same automatic way as SQL Server. But it allows you to **manually cluster** a table:

```
CLUSTER orders USING idx_orders_created_at;
```

□You can **manually cluster** a table, but it doesn't stay clustered automatically.

Composite Index vs. Single-Column Index



Composite Index (idx(a, b))	Single-Column Indexes (idx_a, idx_b)
One index covering multiple columns	One index per column
Useful when queries filter or sort by multiple columns together	Better when columns are filtered independently
Order matters (idx(a, b) works for WHERE a = ?, and WHERE a = ? AND b = ?, but not for WHERE b = ? only)	Each column is searchable independently

Covering Index / Index-only scan

A **covering index** is an index that contains **all the columns needed** to satisfy a query — **so PostgreSQL doesn't need to read the actual table (heap)**.

```
CopySELECT name,email FROM customers WHERE city = 'Brussels';
```

If you create this covering index

```
CopyCREATE INDEX idx_name_email_city ON customers(name, email,
city);
```

PostgreSQL can :

- Use **only the index** to get everything it needs.
- **Skip table access entirely.**

This is called an **index-only scan**.

Index-only scans are **faster** because PostgreSQL doesn't need to go to the base table (heap). Saves I/O, especially on large tables.

Full table scan

Seq Scan in PostgreSQL, means PostgreSQL reads **every row** in a table to find matches for your query — even if you only need one or a few rows. This is usually **slower** than using an index, especially for large tables

Common Causes of Full Table Scan

Missing Indexes : If the column used in a WHERE, JOIN, or ORDER BY clause isn't indexed, PostgreSQL has no choice but to scan the whole table.

Using Non-Sargable Expressions : Expressions that prevent the use of indexes, such as :

```
WHERE LOWER(city) = 'brussels'WHERE id + 1 = 5
```

Prevent PostgreSQL from using an index on city, or id. If you **must** use functions, consider **creating an index on the function result** :

```
CREATE INDEX idx_lower_city ON customers(LOWER(city));
```

Small Tables For small tables, PostgreSQL may **prefer a Seq Scan**, because it's faster than using the index and then doing random I/O to fetch rows.

Bad or Outdated Statistics If ANALYZE hasn't been run in a while, PostgreSQL might misestimate row counts and choose a Seq Scan incorrectly.

Functions or Casts on Indexed Columns Even if there's an index on created_at, this won't use it:

```
WHERE DATE(created_at) = '2023-01-01'
```