PostgreSQL Query Plan: A Guide for Understanding Execution

InterviewBuddies · Follow
Published in Tech Factor
5 min read · Oct 3, 2024

https://interviewbuddies.com/

The PostgreSQL query plan is a detailed blueprint outlining how the database intends to execute a specific query. Understanding this plan is crucial for identifying potential bottlenecks and optimizing query performance.

1. Obtaining the Query Plan

- **`EXPLAIN`**: The `EXPLAIN` command provides a textual representation of the query plan.
- **`EXPLAIN ANALYZE`**: This command executes the query and adds timing information to the plan, allowing you to analyze performance and resource consumption.

2. Key Components of the Query Plan

- **Node Type**: Each line in the query plan represents a "node," indicating the operation being performed (e.g., Seq Scan, Index Scan, Hash Join, etc.).
- **Relation Name:** The table or relation involved in the operation.
- **Alias:** If an alias is used in the query, it is displayed here.
- **Actual Rows:** The number of rows processed by the node during execution (in `EXPLAIN ANALYZE`).
- **Actual Loops:** The number of times the node was executed (for nested loop joins).
- **Cost:** The estimated cost of the operation in terms of disk blocks and CPU cycles, represented as two numbers: startup cost and total cost.
- **Start Time:** The time taken for the node to start execution (in `EXPLAIN ANALYZE`).
- **End Time:** The time taken for the node to complete execution (in `EXPLAIN ANALYZE`).
- **Buffers:** Indicates the memory consumed by the query, detailing shared buffers hit, read, and temporary buffers used.

3. Sample Query Plan

To illustrate the components of a query plan, let's consider the following SQL query:

```sql
SELECT o.order_id, c.customer_name, SUM(oi.quantity)
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date > '2023-01-01'
GROUP BY o.order_id, c.customer_name
ORDER BY SUM(oi.quantity) DESC;
```

Running `EXPLAIN ANALYZE` for this query might yield an output like the following:

```
                                                      QUERY PLAN
    ------------------------------------------------------------------------------------
     Sort  (cost=234.50..235.50 rows=1000 width=64) (actual time=10.235..10.735 row
       Sort Key: sum(oi.quantity) DESC
       Sort Method: quicksort  Memory: 1200kB
       Buffers: shared hit=100 read=50, temp written=0
       ->  Hash GroupAgg  (cost=234.50..235.00 rows=1000 width=64) (actual time=10.
             Group Key: o.order_id, c.customer_name
             Buffers: shared hit=100 read=50
             ->  Hash Join  (cost=233.00..234.00 rows=1000 width=64) (actual time=9
                   Hash Cond: (oi.order_id = o.order_id)
                   Buffers: shared hit=90 read=50
                   ->  Seq Scan on order_items oi  (cost=0.00..50.00 rows=5000 widt
                         Buffers: shared hit=50
                   ->  Hash  (cost=100.00..100.00 rows=1000 width=32) (actual time=
                         Buckets: 1024  Batches: 1  Memory Usage: 1500kB
                         Buffers: shared hit=40 read=0
                         ->  Seq Scan on orders o  (cost=0.00..100.00 rows=1000 wic
                               Buffers: shared hit=40
                   ->  Hash Motion  (cost=100.00..100.00 rows=1000 width=32) (actua
                         Hash Key: c.customer_id
                         Buffers: shared hit=50
                         ->  Seq Scan on customers c  (cost=0.00..100.00 rows=1000
                               Buffers: shared hit=50
     Planning time: 0.120 ms
     Execution time: 11.235 ms
```

Analyzing the Sample Plan

Let’s break down the output step by step:

- **Sort Operation:**
  - **Node Type:** `Sort` indicates that the results are being sorted by the `SUM(oi.quantity)` in descending order.
  - **Memory Usage:** `Memory: 1200kB` shows how much memory is being consumed during the sort operation.
  - **Buffers:** It shows the buffers accessed, including shared hits and reads from disk.

- **Hash GroupAgg:**
  - This node indicates a hash-based aggregation operation is being performed on the grouped results.

- The `Group Key` reveals the keys by which the results are grouped (i.e., `o.order_id` and `c.customer_name`).

- **Hash Join**:
  - **Hash Condition**: This node joins `order_items` and `orders` on `order_id`. Hash joins are typically efficient for larger datasets, especially when the hash table can fit in memory.
  - **Buffers**: The `Buffers` indicate the shared buffers accessed during this join operation.

- **Sequential Scans**:
  `Seq Scan on order_items` and `Seq Scan on orders` indicate that the entire tables are being scanned. This may suggest missing indices on these tables, which could be improved by adding appropriate indices.

- **Hash Motion**:
  This node indicates that data is being redistributed based on the hash of `customer_id`, which is necessary for ensuring that all rows sharing the same key are processed together in a parallel execution context.

- **Memory Consumption**:
  The overall query execution time is shown at the end, with specific details about planning time and execution time.

4. Understanding Row Estimates and Consumption

- **Estimating Rows**: PostgreSQL uses statistics gathered by the `ANALYZE` command to estimate how many rows each node will process. If your estimates are significantly off, you might need to run `ANALYZE` on your tables to update statistics or adjust your query for better estimates.

- **Memory Consumption**: Use the `Buffers` section in the query output to understand memory usage. The breakdown will typically show:
  - **Shared Buffers Hit:** Number of buffers read from shared memory.
  - **Buffers Read:** Number of buffers read from disk.
  - **Temporary Buffers**: Memory allocated for intermediate results.
  This information helps you gauge if your query is consuming excessive memory or causing I/O bottlenecks.

- **Redirect Motion:**
  In cases where a query involves a large number of rows, you may encounter

"redirect motion," where the plan indicates that data is being moved between different nodes. This can be seen in operations like `Gather` or `Gather Merge`, which collect results from multiple parallel workers. Monitoring this can reveal inefficiencies in how data is processed and transmitted.

5. Common Bottlenecks

- **Missing or Inefficient Indices:** If your query relies on `Seq Scan` or `Bitmap Heap Scan` operations, a missing or inefficient index is likely the culprit.
- **Inefficient Join Methods:** Nested loop joins with large datasets can be very slow. Consider alternative join methods like Hash Join or Merge Join.
- **Large Sort and Group Operations:** Sorting or grouping large datasets can be expensive. Indices or materialized views can sometimes help.
- **Slow Disk I/O:** Excessive disk I/O can impact performance. Consider improving disk hardware or caching frequently accessed data.

6. Optimization Techniques.

Here are several optimization techniques to enhance query performance:

- **Add Indices:** Create appropriate indices for frequently accessed columns in `WHERE`, `JOIN`, and `ORDER BY` clauses.

```
CREATE INDEX idx_column_name ON your_table (column_name);
```

- **Improve Query Structure:** Break down complex queries into smaller, more manageable ones. This can help PostgreSQL optimize execution.
- **Tune Database Settings:** Optimize PostgreSQL parameters related to memory and caching in your `postgresql.conf` file to improve overall performance.
- **Use Materialized Views:** Materialized views can speed up retrieval of complex query results. Assess their effectiveness based on your specific use case — if the underlying data changes infrequently and you query the results often, they may provide a benefit. However, if your data is highly dynamic, the overhead may outweigh the advantages.

- **Analyze and Vacuum:** Regularly use the `VACUUM` and `ANALYZE` commands to maintain optimal performance, especially after significant data changes.

By understanding how to read and analyze query plans in PostgreSQL, you can identify performance bottlenecks and take actionable steps to optimize your queries. Regularly monitoring performance and applying these optimization techniques will help ensure your PostgreSQL database operates efficiently.

Postgresql Query Understanding Execution Plan Database Optimization

Follow

Published in Tech Factor

11 Followers · Last published Feb 1, 2025

We deliver concise and actionable insights across software development, AI, data science, and cloud computing. From deep dives into coding and databases to practical career advice, we empower readers to stay ahead in the fast-paced tech world.

Follow

Written by InterviewBuddies

45 Followers · 2 Following

https://interviewbuddies.com : Mock Interviews and Resume reviews for Software Engineers, Product managers, Engineering managers and New Grads.

No responses yet

Gvadakte

What are your thoughts?

More from InterviewBuddies and Tech Factor

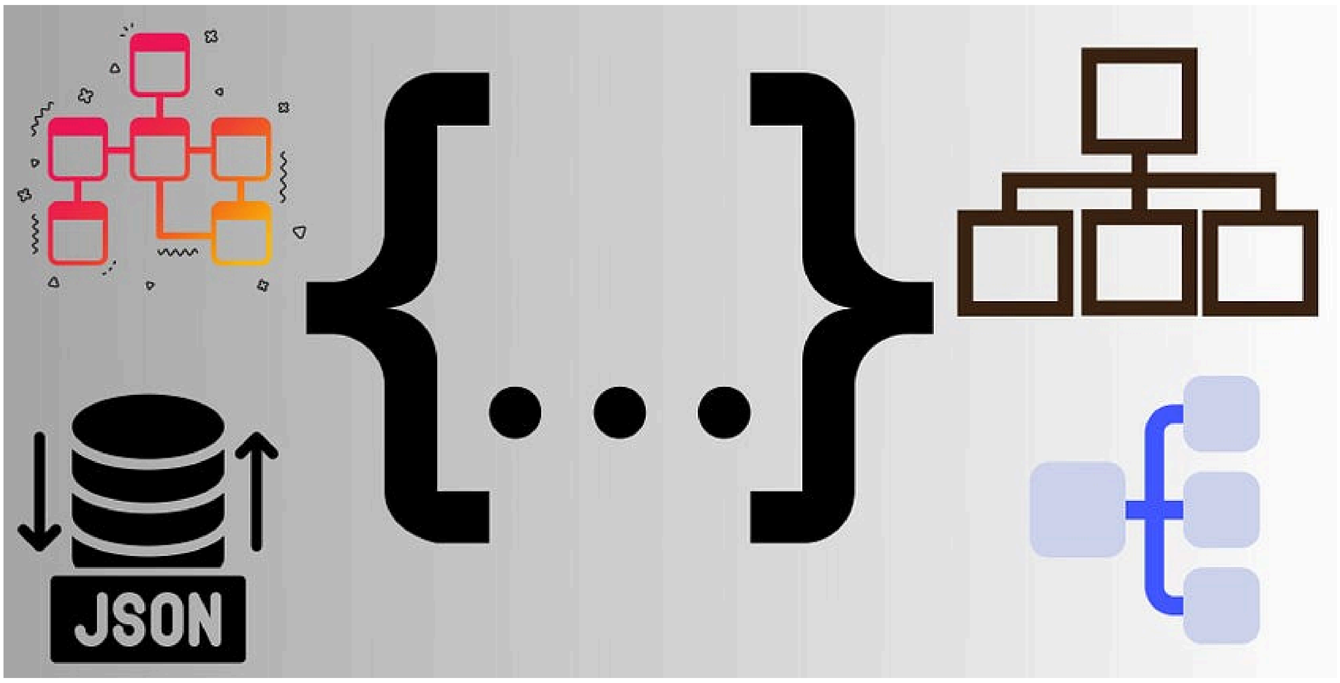
InterviewBuddies

Reverse Polish Notation

Reverse Polish notation (RPN) or simply postfix notation, is a mathematical notation in which

Open in app ↗

Medium Search



In Tech Factor by InterviewBuddies

JSON Schemas: The Importance of Standardized LLM Responses

Large Language Models (LLMs) are transforming how we interact with information, but unlocking their true potential requires more than just...

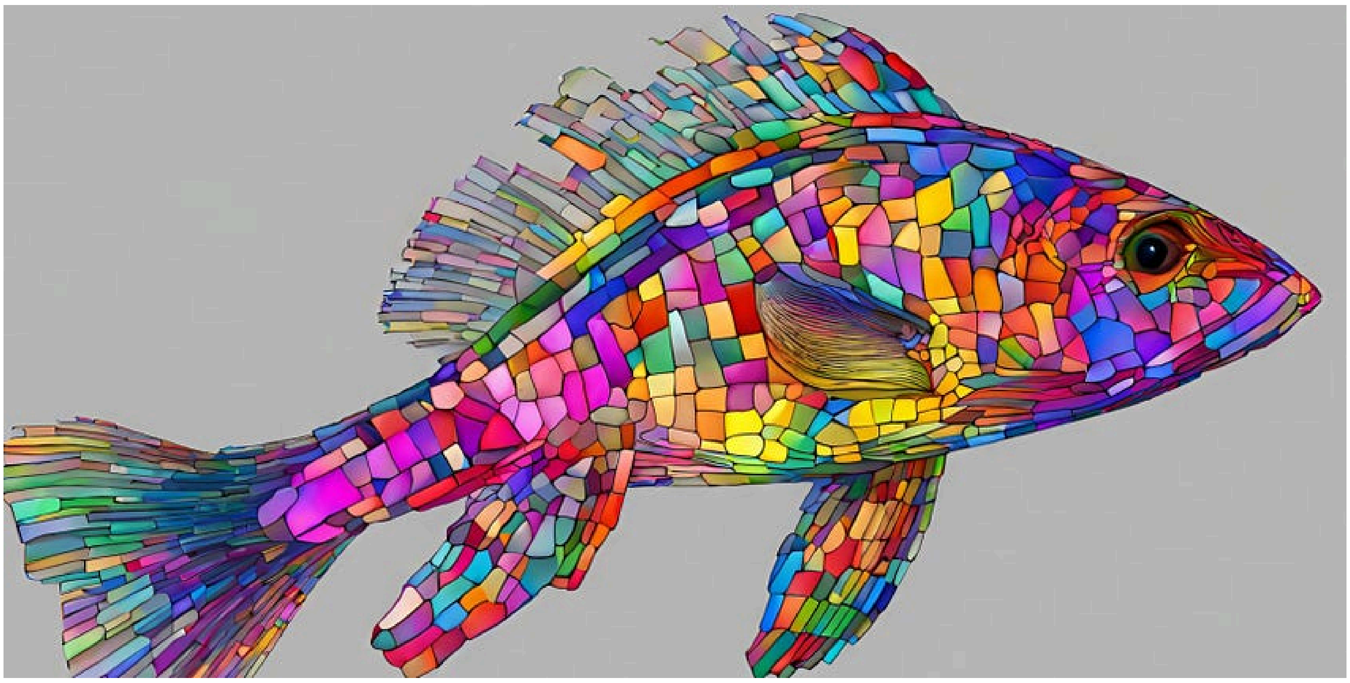
Sep 7, 2024

In Tech Factor by Niharika Pandey

Trending Jobs in 2025: The Roles Defining the Future of Work

As we step into 2025, the employment landscape is transforming with unprecedented velocity. Driven by technological advancements, evolving...

Jan 1 1



InterviewBuddies

Importing and Exporting JSON Data in PostgreSQL

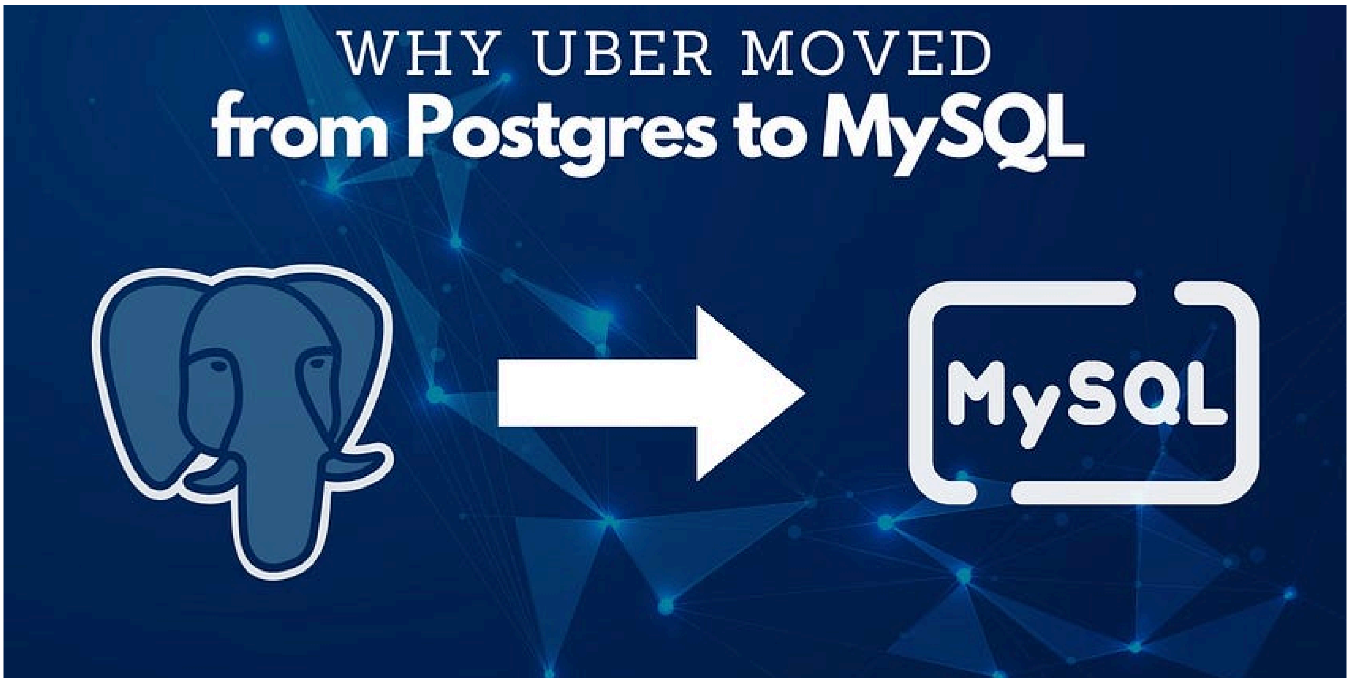
PostgreSQL, known for its flexibility and reliability, provides built-in support for JSON (JavaScript Object Notation) data, allowing...

Apr 26, 2024 2

See all from InterviewBuddies

See all from Tech Factor

Recommended from Medium



In Databases by Sergey Egorenkov

Why Uber Moved from Postgres to MySQL

How PostgreSQL's architecture clashed with Uber's scale—and why MySQL offered a better path forward

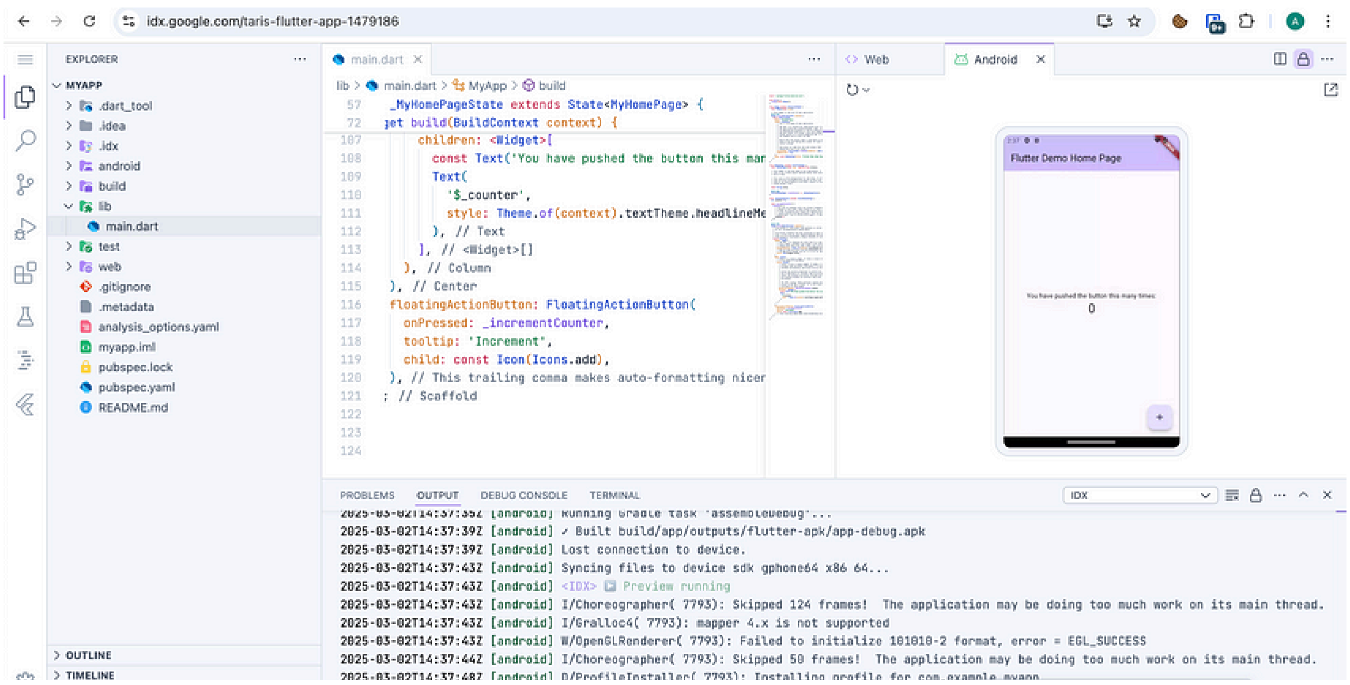
Mar 29 295 10

In Hack the Stack by Coders Stop

9 Database Optimization Tricks SQL Experts Are Hiding From You

Most developers learn enough SQL to get by—SELECT, INSERT, UPDATE, DELETE, and maybe a few JOINs. They might even know how to create…

Mar 27 244 5



```
 _MyHomePageState extends State<MyHomePage> {
jet build(BuildContext context) {
      children: <Widget>[
        const Text('You have pushed the button this man
        Text(
          '$_counter',
          style: Theme.of(context).textTheme.headlineMe
        ), // Text
      ], // <Widget>[]
    ), // Column
  ), // Center
  floatingActionButton: FloatingActionButton(
    onPressed: _incrementCounter,
    tooltip: 'Increment',
    child: const Icon(Icons.add),
  ), // This trailing comma makes auto-formatting nicer
; // Scaffold
```

```
2025-03-02T14:37:35Z [android] Running Gradle task 'assembleDebug'...
2025-03-02T14:37:39Z [android] ✓ Built build/app/outputs/flutter-apk/app-debug.apk
2025-03-02T14:37:39Z [android] Lost connection to device.
2025-03-02T14:37:43Z [android] Syncing files to device sdk gphone64 x86 64...
2025-03-02T14:37:43Z [android] <IDX> Preview running
2025-03-02T14:37:43Z [android] I/Choreographer( 7793): Skipped 124 frames!  The application may be doing too much work on its main thread.
2025-03-02T14:37:43Z [android] I/Gralloc4( 7793): mapper 4.x is not supported
2025-03-02T14:37:43Z [android] W/OpenGLRenderer( 7793): Failed to initialize 101010-2 format, error = EGL_SUCCESS
2025-03-02T14:37:44Z [android] I/Choreographer( 7793): Skipped 50 frames!  The application may be doing too much work on its main thread.
2025-03-02T14:37:48Z [android] D/ProfileInstaller( 7793): Installing profile for com.example.myapp
```

In Coding Beauty by Tari Ibaba

This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.

Mar 12 3.7K 203

In Dev Genius by Doran Gao

Efficient Large Table Cleanup in PostgreSQL

“We can only see a short distance ahead, but we can see plenty there that needs to be done.” — Alan Turing

Oct 31, 2024 1

Tihomir Manushev

Vector Search with pgvector in PostgreSQL

Simple AI-powered similarity search

Mar 9

In Towards Dev by Nakul Mitra

PostgreSQL Performance Optimization — Cleaning Dead Tuples & Reindexing

Performance optimization is crucial in PostgreSQL to ensure efficient query execution and minimal resource consumption.

Mar 28 1

See more recommendations