

[Open in Google Cache](#)

# Navigating PostgreSQL Join Strategies

[Open in Read-Medium](#)

[Open in Freedium](#)



Aditya Agrawal

[Follow](#)

5 min read · Mar 11, 2020

[Open in Archive.today](#)

[Open in app](#) ↗



Medium



Search



Write



## Nested Loops

Iframe/gist/embeds are not loaded in the Google Cache proxy. For those, please use the Read-Medium/Archive.today links.

Nested loops join is one of the simplest join strategies that PostgreSQL uses.

It works by taking each row from one table (outer table) and finding matching rows in another table (inner table) by scanning or using an index.

## How Nested Loops Join Works

When PostgreSQL performs a nested loops join:

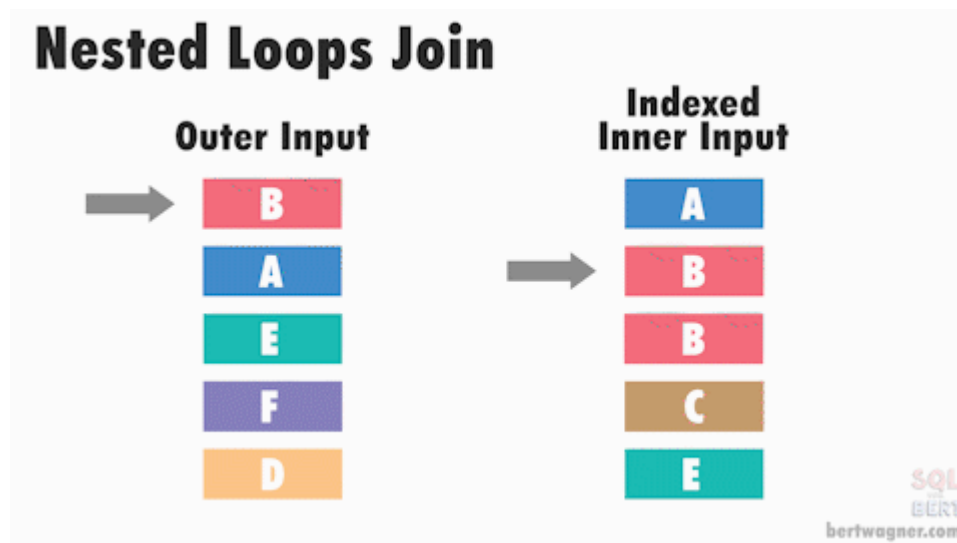
1. It first gets a row from the outer table (usually the table with fewer matching rows after filters)
2. For each outer row, it scans the inner table (either sequentially or using an index) to find matching rows
3. When matches are found, it combines the rows according to the join condition
4. This process repeats for every row from the outer table

Nested loops are most efficient when:

- The outer table has very few rows after filtering
- The inner table has an index on the join columns
- The join needs to return only a small number of rows

think of this like a double for loop.

```
for (order in orders) {
  for (product in products) {
    if (order.product_id == product.id) {
      // join the order and product
    }
  }
}
```



For example, in this query plan:

```
postgres=# explain select * from orders o join products p on p.id=o.product_id w
          QUERY PLAN
```

```
Nested Loop  (cost=0.70..14.81 rows=11 width=50)
->  Index Scan using orders_pkey on orders o  (cost=0.42..2.62 rows=11 width=
      Index Cond: (id < 107219)
->  Index Scan using products_pkey on products p  (cost=0.27..1.11 rows=1 wid
```

Index Cond: (id = o.product\_id)

(5 rows)

**Note:** order is the outer table and product is the inner table.

## Understanding the Query Plan

- The index scan on orders returns **11 rows**.
- For each of these rows, the query performs an index scan on products.
- The index scan on products returns **1 row** for each of the **11 orders**.

## Hash Join / Hash

Hash join is another join strategy that PostgreSQL uses. It works by creating a hash table from one of the tables and then using that hash table to find matching rows in another table.

## How Hash Join Works

When PostgreSQL performs a hash join:

1. It creates a hash table from one of the tables (usually the smaller one)
2. It then uses that hash table to find matching rows in another table
3. When matches are found, it combines the rows according to the join condition

Hash join is most efficient when:

- The join needs to return only a small number of rows
- The inner table has an index on the join columns

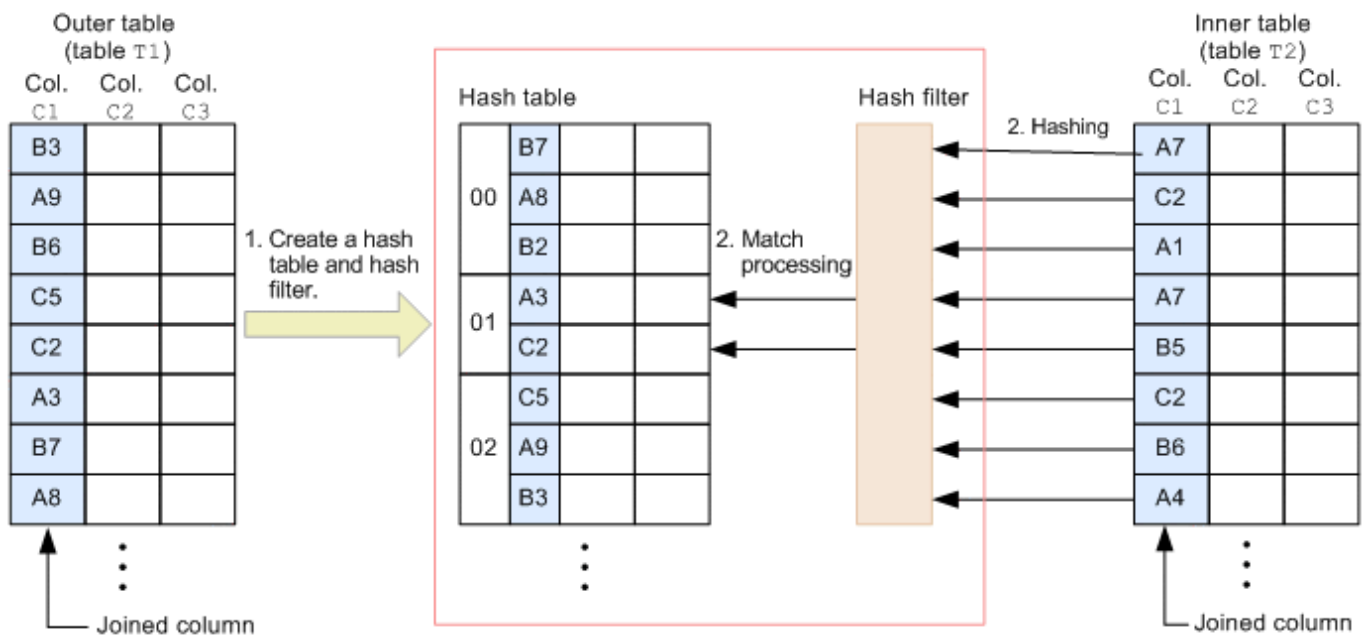
think of this like a hash map.

```

hash_map = {}
for (product in products) {
    hash_map[product.id] = product
}

for (order in orders) {
    if (hash_map[order.product_id]) {
        // join the order and product
    }
}

```



```

postgres=# explain select * from orders o join products p on p.id=o.product_id w
          QUERY PLAN

```

```

Hash Join (cost=15.68..339.95 rows=10536 width=50)

```

```

  Hash Cond: (o.product_id = p.id)

```

```

    -> Index Scan using orders_pkey on orders o (cost=0.42..296.81 rows=10536 w
          Index Cond: (id < 117219)

```

```

    -> Hash (cost=9.00..9.00 rows=500 width=23)

```

```

          -> Seq Scan on products p (cost=0.00..9.00 rows=500 width=23)

```

```

(6 rows)

```

**Note:** orders is the outer table and products is the inner table.

## Understanding the Query Plan

- The index scan on orders returns **10,536 rows**.
- The hash join creates a hash table from the products table (smaller table).
- For each row in the orders table, the query uses the hash table to find matching rows in the products table.
- The hash join returns **10,536 rows**.

## Why Did PostgreSQL Choose a Hash Join Instead of a Nested Loop?

Key Differences from Previous Query (`id < 107219`):

- The previous query had only **11 rows** from orders, so a Nested Loop Join was efficient.
- This query has **10,536 rows**, so a Nested Loop would require **10,536 index lookups** in products, which would be slow.
- Instead, PostgreSQL builds a hash table once (cheap) and does fast lookups ( $O(1)$ ) instead of repeated index scans ( $O(\log N)$ ).

## Merge Join

Merge join is another join strategy that PostgreSQL uses. It works by sorting the two tables and then merging them together.

## How Merge Join Works

When PostgreSQL performs a merge join:

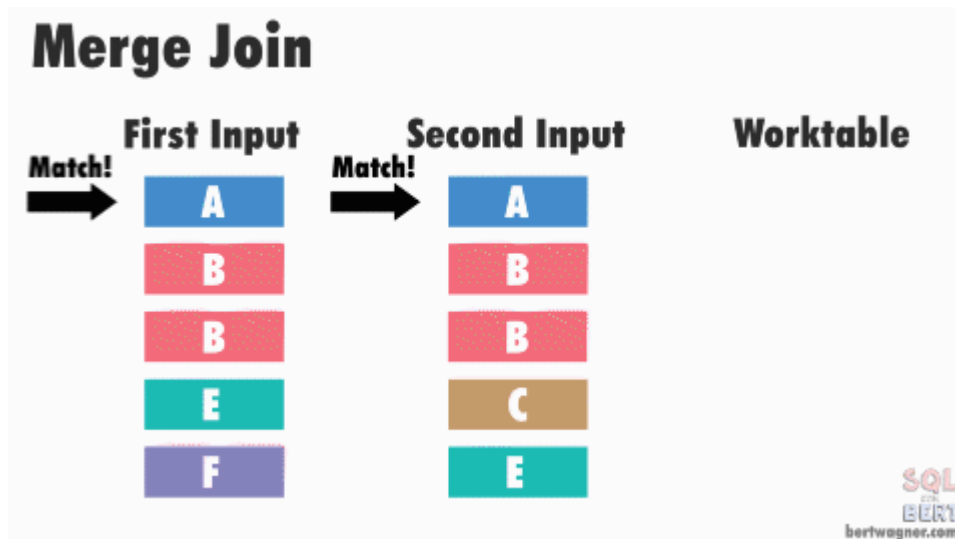
1. It sorts the two tables by the join columns
2. It then merges the two tables together
3. When matches are found, it combines the rows according to the join condition

Merge join is most efficient when:

- Both tables are already sorted (or can be sorted efficiently).
- There's no suitable index for a Nested Loop.
- The tables are large, making Hash Join expensive.

think of this like a merge sort.

```
merge_sort(left, right) {
  if (left.length == 0) return right;
  if (right.length == 0) return left;
}
```



```
explain
SELECT *
FROM employees e
JOIN salary s
  ON e.id = s.employee_id
WHERE e.age < 40;
```

QUERY PLAN

```
-----
Merge Join  (cost=198.11..268.19 rows=10 width=488)
  Merge Cond: (e.id = s.employee_id)
    -> Index Scan using employees_id on employees e  (cost=0.29..656.28 rows=101)
        Filter: (age < 40)
    -> Sort  (cost=197.83..200.33 rows=1000 width=244)
```

Sort Key: s.employee\_id

-> Seq Scan on salary s (cost=0.00..148.00 rows=1000 width=244)

## Understanding the Query Plan

- The index scan on employees returns **101 rows**.
- The sort operation sorts the salary table by the join column (employee\_id).
- The merge join then merges the two sorted tables together.
- The merge join returns **10 rows**.

## Why Did PostgreSQL Choose a Merge Join Instead of Nested Loop or Hash Join?

Key Differences from Previous Queries:

- The employees table is filtered (age < 40), reducing rows to 101.
- The salary table is not indexed on employee\_id, so a Hash Join would require creating a large hash table.
- The tables can be efficiently sorted and merged, making a Merge Join the best option.

## Summary

<https://gist.githubusercontent.com/aditya999123/11208e19ebf0c19c6db5dd72b9bf2d6f/raw/4c73d835181a742e3806de477ff837677ac1932f/gistfile1.txt>

Join Type	When Used	How It Works
Nested Loop	Small <b>outer</b> tables (<1000) Has indexes <b>on join</b> cols Heavily filtered <b>outer</b>	Iterates <b>outer rows</b> , checks <b>matches in inner</b> via index <b>Like</b> nested <b>for</b> loops
Hash Join	Medium/ <b>large</b> tables <b>No</b> useful indexes	Builds hash <b>table from</b> small <b>table</b> , probes <b>with</b> larger

### Merge Join

Memory available  
Pre-sorted data  
Sort cost acceptable  
Large tables ok

$O(1)$  hash lookups  
Sorts inputs if needed, then walks both in parallel to match rows like merge sort

### Join Type

### Time Complexity

### Space Complexity

#### Nested Loop

$O(n*m)$  no index  
 $O(n*\log m)$  with index  
 $n=\text{outer}$ ,  $m=\text{inner rows}$

$O(1)$  minimal memory

#### Hash Join

$O(n + m)$  build & probe

$O(n)$  for hash table  
 $n = \text{smaller table size}$

#### Merge Join

$O(n \log n + m \log m)$  if sorting  
 $O(n + m)$  if pre-sorted

$O(1)$  if pre-sorted  
 $O(n + m)$  if sorting

## References

- [Bert Wagner 1](#)
- [Bert Wagner 2](#)
- [PostgreSQL Documentation](#)
- [Airbyte](#)

## Next Steps

- Apply all our learnings to a real query and optimize it.

Originally published at <https://www.adiagr.com>.

Database

Postgres

Postgresql

Tech

Webdev





Written by Aditya Agrawal

64 followers · 32 following

Follow

No responses yet



Gvadakte

What are your thoughts?

## More from Aditya Agrawal

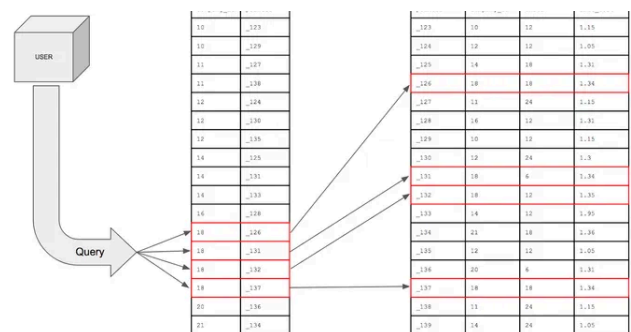


Aditya Agrawal

### OAuth2.0 Auth Tokens: A Comprehensive Guide

OAuth 2.0 in Real Life: The Netflix Streaming Analogy

Apr 1



Aditya Agrawal

### Navigating PostgreSQL — Index, Index Only and Bitmap Index Scan...

Index Scan

Mar 10





 Aditya Agrawal


## Authentication vs Authorization: Understanding the Difference

Securing the Digital World: Authentication vs Authorization

Mar 31



## JSON WEB TOKENS (JWTs)

 Aditya Agrawal

## JWT — Based Auth: A Deep Dive

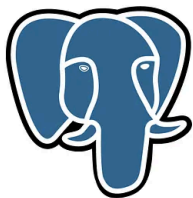
Understanding JWT-Based Security

Mar 31




See all from Aditya Agrawal

## Recommended from Medium

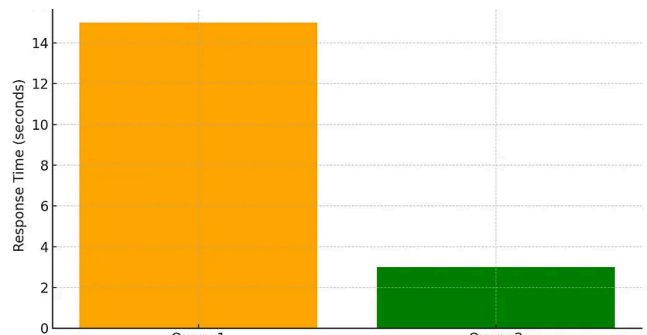


**Beyond Basic**  
PostgreSQL  
Programmable  
Objects

 In Stackademic by bektiaw

## Beyond Basics: PostgreSQL Programmable Objects (Automat...

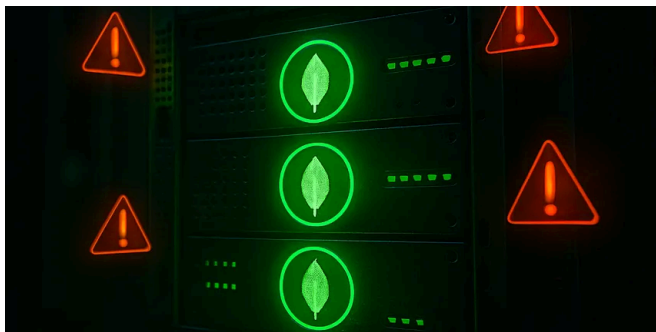
Tired of repeating the same SQL logic? Learn how PostgreSQL can do the work for us.




 In Javarevisited by Jitin Kayyala

## How one character improved Postgres performance from 15 se...

Ok the title was a bit of hyperbole but there is some truth to the title as the performance...




 Prem Chandak

## MongoDB in Production: Avoid These Fatal Deployment Mistakes

Hard-learned lessons, and a war story to save you from sleepless nights.

★ Aug 12 🖱 9 📌 ⋮

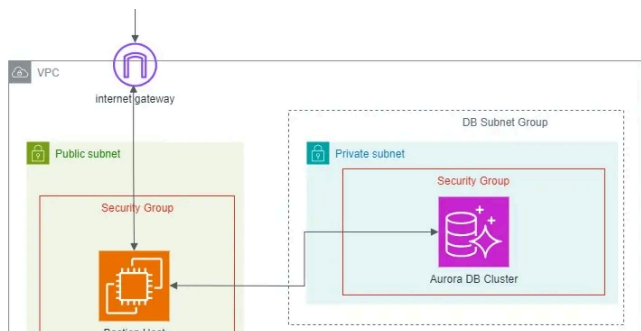


 Rizqi Mulki

## PostgreSQL Caching Strategies That Make a Difference

How Shopify serves 80,000 requests per second with PostgreSQL — and the caching...

★ Jul 19 🖱 12 💬 2 📌 ⋮

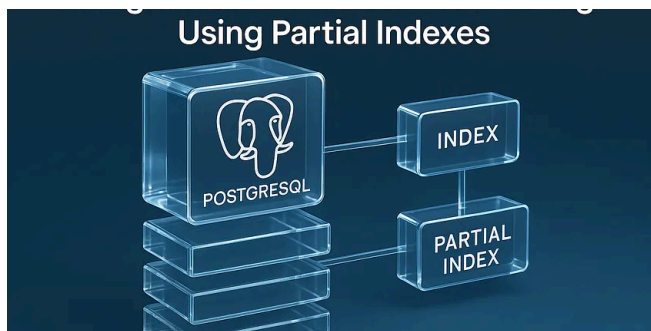


 In AWS in Plain English by Juan Andrés Leiva

## Aurora PostgreSQL Cross-Cluster Migration: Step-by-Step with an...

Guide to migrate schemas and data between Amazon Aurora PostgreSQL clusters using a...

Aug 18 📌 ⋮



 Jeyaram Ayyalusamy

## 17 - PostgreSQL 17 Performance Tuning: Using Partial Indexes

Even though we usually create indexes on all rows of a table, PostgreSQL also allows you t...

Sep 4 🖱 3 📌 ⋮

See more recommendations