

[Open in Google Cache](#)

PostgreSQL Joins Explained - A Backend Developer's Guide

[Open in Read-Medium](#)

[Open in Freedium](#)



Rizqi Mulki

Following ▾

9 min read · May 7, 2020

[Open in Archive.today](#)



[Open in Archive.is](#)



[Open in Proxy API](#)

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

POSTGRESQL JOINS EXPLAINED



A BACKEND DEVELOPER'S GUIDE

As a backend developer, mastering SQL joins is crucial for efficient data retrieval from relational databases. PostgreSQL, with its powerful query engine, offers several join types that can dramatically affect both performance and results. In this comprehensive guide, we'll explore each join type with practical examples, optimization techniques, and common pitfalls to avoid.

Understanding the Basics of Joins

Joins allow you to combine rows from two or more tables based on related columns. Before diving into specific join types, let's establish a foundational understanding with two sample tables we'll use throughout this guide:

```
-- Sample tables for our examples
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    city VARCHAR(100)
);
```

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INTEGER REFERENCES customers(customer_id),
    order_date DATE,
    total_amount DECIMAL(10, 2)
);

-- Add some sample data
INSERT INTO customers (name, email, city) VALUES
('John Smith', 'john@example.com', 'New York'),
('Sarah Johnson', 'sarah@example.com', 'Los Angeles'),
('Michael Brown', 'michael@example.com', 'Chicago'),
('Emma Wilson', 'emma@example.com', 'Houston'),
('David Lee', NULL, 'Seattle');

INSERT INTO orders (customer_id, order_date, total_amount) VALUES
(1, '2023-01-15', 150.75),
(1, '2023-02-20', 89.32),
(2, '2023-01-10', 210.50),
(3, '2023-03-05', 45.00),
(NULL, '2023-03-15', 125.20); -- Order with unknown customer
```

Types of PostgreSQL Joins

1. INNER JOIN

The INNER JOIN returns rows when there is a match in both tables. This is the most common join type.

```
SELECT c.name, o.order_id, o.order_date, o.total_amount
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;
```

Result:

name	order_id	order_date	total_amount
John Smith	1	2023-01-15	150.75
John Smith	2	2023-02-20	89.32
Sarah Johnson	3	2023-01-10	210.50
Michael Brown	4	2023-03-05	45.00

Key characteristics:

- Returns only matching rows
- Excludes rows with NULL values in the join columns
- Often the most efficient join type

2. LEFT JOIN (or LEFT OUTER JOIN)

The LEFT JOIN returns all rows from the left table and the matched rows from the right table. Rows in the left table with no match in the right table will have NULL values for the right table's columns.

```
SELECT c.name, o.order_id, o.order_date, o.total_amount
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

Result:

name	order_id	order_date	total_amount
John Smith	1	2023-01-15	150.75
John Smith	2	2023-02-20	89.32
Sarah Johnson	3	2023-01-10	210.50
Michael Brown	4	2023-03-05	45.00
Emma Wilson	NULL	NULL	NULL
David Lee	NULL	NULL	NULL

Key characteristics:

- Preserves all rows from the left table
- Good for finding “orphaned” records (e.g., customers with no orders)
- Slightly less efficient than INNER JOIN

3. RIGHT JOIN (or RIGHT OUTER JOIN)

The RIGHT JOIN returns all rows from the right table and the matched rows from the left table. It’s the mirror of LEFT JOIN.

```
SELECT c.name, o.order_id, o.order_date, o.total_amount
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;
```

Result:

name	order_id	order_date	total_amount
John Smith	1	2023-01-15	150.75
John Smith	2	2023-02-20	89.32
Sarah Johnson	3	2023-01-10	210.50
Michael Brown	4	2023-03-05	45.00
NULL	5	2023-03-15	125.20

Key characteristics:

- Preserves all rows from the right table
- Less commonly used than LEFT JOIN (developers often prefer to rewrite as LEFT JOIN)
- Shows “orphaned” records on the right side (e.g., orders with no customer)

4. FULL JOIN (or FULL OUTER JOIN)

The FULL JOIN returns all rows when there is a match in either the left or right table. It's a combination of LEFT and RIGHT joins.

```
SELECT c.name, o.order_id, o.order_date, o.total_amount
FROM customers c
FULL JOIN orders o ON c.customer_id = o.customer_id;
```

Result:

name	order_id	order_date	total_amount
John Smith	1	2023-01-15	150.75
John Smith	2	2023-02-20	89.32
Sarah Johnson	3	2023-01-10	210.50
Michael Brown	4	2023-03-05	45.00
Emma Wilson	NULL	NULL	NULL
David Lee	NULL	NULL	NULL
NULL	5	2023-03-15	125.20

Key characteristics:

- Shows all data from both tables
- Helpful for data integrity checks
- Generally less efficient than other join types
- Good for finding “orphaned” records on both sides

5. CROSS JOIN

The CROSS JOIN returns the Cartesian product of both tables (every row in the first table paired with every row in the second table).

```
SELECT c.name, o.order_id
FROM customers c
CROSS JOIN orders;
```

Result (truncated):

name	order_id
John Smith	1
John Smith	2
John Smith	3
...	
David Lee	3
David Lee	4
David Lee	5

Key characteristics:

- Results in rows = (rows in table1) × (rows in table2)
- No join condition needed
- Rarely used in production but useful for generating test data
- Can cause performance issues with large tables

6. SELF JOIN

A SELF JOIN is not a distinct join type but rather a regular join (inner, left, etc.) where a table is joined with itself.

```
-- Create an employees table for the self-join example
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
```

```
manager_id INTEGER REFERENCES employees(employee_id)
);
```

```
INSERT INTO employees (name, manager_id) VALUES
('Alice Smith', NULL),      -- CEO, no manager
('Bob Johnson', 1),        -- Reports to Alice
('Carol Williams', 1),     -- Reports to Alice
('Dave Brown', 2);        -- Reports to Bob

-- Self-join to find employees and their managers
SELECT e.name AS employee, m.name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id;
```

Result:

employee		manager
Alice Smith		NULL
Bob Johnson		Alice Smith
Carol Williams		Alice Smith
Dave Brown		Bob Johnson

Key characteristics:

- Useful for hierarchical or recursive data
- Allows comparison of rows within the same table
- Requires table aliases to distinguish between instances of the same table

Advanced Join Techniques

Using USING Instead of ON

When join columns have the same name, the USING clause provides a more concise syntax:


```
SELECT c.name, o.order_id, o.order_date
FROM customers c
JOIN orders o USING (customer_id);
```

This is equivalent to `ON c.customer_id = o.customer_id` but more concise. Additionally, the column appears only once in the result.

Natural Joins

A NATURAL JOIN automatically joins tables on columns with matching names:

```
SELECT c.name, o.order_id, o.order_date
FROM customers c
NATURAL JOIN orders;
```

Word of caution: While convenient, NATURAL JOINS are considered risky in production code because they depend on column naming conventions that might change. They also don't make the join conditions explicit, reducing code readability.

Joining Multiple Tables

Complex queries often require joining more than two tables:

```
-- Add a products table
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    price DECIMAL(10, 2)
);
```

```
-- Add an order_items table
CREATE TABLE order_items (
```

```

        order_id INTEGER REFERENCES orders(order_id),
        product_id INTEGER REFERENCES products(product_id),
        quantity INTEGER,
        PRIMARY KEY (order_id, product_id)
    );

-- Query joining three tables
SELECT c.name AS customer, o.order_id, p.name AS product, oi.quantity
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id;

```

Lateral Joins

PostgreSQL's LATERAL join is a powerful feature that allows the right-hand side of the join to reference columns from the left-hand side:

```

-- Find each customer's three most recent orders
SELECT c.name, recent_orders.*
FROM customers c
CROSS JOIN LATERAL (
    SELECT order_id, order_date, total_amount
    FROM orders o
    WHERE o.customer_id = c.customer_id
    ORDER BY order_date DESC
    LIMIT 3
) AS recent_orders;

```

LATERAL joins are particularly useful for:

- Applying limits per group
- Correlated subqueries in the FROM clause
- Complex analytics that depend on outer table values

Optimization Techniques for Joins

1. Use Appropriate Indexes

Indexes on join columns are crucial for performance:

```
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
```

Remember to:

- Index foreign key columns (usually the “many” side of relationships)
- Consider composite indexes for multi-column joins
- Avoid over-indexing as it impacts write performance

2. Choose the Right Join Order

PostgreSQL’s query planner generally does a good job, but for complex queries with many tables, the join order matters. Place smaller tables first and tables with more restrictive filters earlier in the query when possible.

3. Use EXPLAIN ANALYZE

Always profile your joins to identify bottlenecks:

```
EXPLAIN ANALYZE
SELECT c.name, o.order_id, o.order_date
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE c.city = 'New York';
```

Look for:

- Sequential scans on large tables (may indicate missing indexes)
- Nested loop joins with high row counts (might benefit from hash joins)
- Hash joins with excessive memory usage

4. Consider Join Types and Algorithms

PostgreSQL supports three physical join algorithms:

- **Nested Loop:** Good for small tables or when one table is small and the other has an index on the join column
- **Hash Join:** Efficient for large tables without appropriate indexes
- **Merge Join:** Works well when both tables are sorted on the join columns

You can force a specific join method for testing:

```
SET enable_nestloop = off;  
SET enable_hashjoin = on;  
SET enable_mergejoin = off;
```

Remember to reset these after testing:

```
RESET enable_nestloop;  
RESET enable_hashjoin;  
RESET enable_mergejoin;
```

5. Materialized Views for Complex Joins

For frequently executed complex joins, consider a materialized view:

```
CREATE MATERIALIZED VIEW customer_order_summary AS  
SELECT c.customer_id, c.name, COUNT(o.order_id) AS order_count,  
       SUM(o.total_amount) AS total_spent  
FROM customers c  
LEFT JOIN orders o ON c.customer_id = o.customer_id  
GROUP BY c.customer_id, c.name;
```

```
-- Create an index on the materialized view  
CREATE INDEX idx_customer_order_summary_id ON
```

```
customer_order_summary(customer_id);  
  
-- Refresh when needed  
REFRESH MATERIALIZED VIEW customer_order_summary;
```

Common Join Pitfalls and How to Avoid Them

1. Cartesian Products

Unintended CROSS JOINS can happen when you forget join conditions:

```
-- Bad: Missing join condition  
SELECT c.name, o.order_id FROM customers c, orders o;
```

```
-- Good: Proper join condition  
SELECT c.name, o.order_id FROM customers c JOIN orders o ON  
c.customer_id = o.customer_id;
```

2. Ambiguous Column References

When tables share column names, always qualify them with table aliases:

```
-- Bad: Ambiguous 'id' column  
SELECT id, name, order_date FROM customers JOIN orders ON customer_id = customer
```

```
-- Good: Qualified column names  
SELECT c.customer_id, c.name, o.order_date
```

[Open in app](#) ↗



Medium



Search



Write



NULL values in join columns

Remember that NULL values don't match in joins:

```
-- This won't match rows where customer_id is NULL
SELECT * FROM orders o JOIN customers c ON o.customer_id = c.customer_id;
```

```
-- To include NULL matches, use a special condition
SELECT * FROM orders o LEFT JOIN customers c
ON (o.customer_id = c.customer_id OR (o.customer_id IS NULL AND
c.customer_id IS NULL));
```

4. Join Selectivity Issues

Low-selectivity joins (where the join matches many rows) can cause performance problems:

```
-- Potentially problematic with many statuses
SELECT * FROM orders o JOIN order_status s ON o.status = s.status;
```

Consider:

- Adding more restrictive conditions
- Using appropriate indexes
- Breaking the query into smaller parts

5. Multiple Join Paths

When tables have multiple ways to join, be explicit about which path to use:

```
-- Potentially confusing with multiple relationships
SELECT * FROM employees e JOIN departments d ON e.department_id = d.department_id
```

```
-- More explicit approach when tables have multiple relationships
SELECT * FROM employees e
```

```
JOIN departments d ON e.current_department_id = d.department_id;
```

Real-World Join Patterns

Pattern 1: Finding Records That Don't Match

To find customers who haven't placed orders:

```
SELECT c.customer_id, c.name
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```

Pattern 2: Aggregating with Joins

To get customer order statistics:

```
SELECT c.name,
       COUNT(o.order_id) AS order_count,
       COALESCE(SUM(o.total_amount), 0) AS total_spent,
       MAX(o.order_date) AS last_order_date
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name;
```

Pattern 3: Pivoting Data with CROSSTAB

Using the `tablefunc` extension to create pivot tables:

```
-- Enable the extension
CREATE EXTENSION IF NOT EXISTS tablefunc;
```

```
-- Pivot sales data by quarter
SELECT * FROM crosstab(
```

```

'SELECT
    p.name,
    EXTRACT(QUARTER FROM o.order_date) AS quarter,
    SUM(oi.quantity) AS units_sold
FROM products p
JOIN order_items oi ON p.product_id = oi.product_id
JOIN orders o ON oi.order_id = o.order_id
WHERE EXTRACT(YEAR FROM o.order_date) = 2023
GROUP BY p.name, quarter
ORDER BY p.name, quarter',
'SELECT q FROM generate_series(1,4) q'
) AS (
    product VARCHAR,
    "Q1" NUMERIC,
    "Q2" NUMERIC,
    "Q3" NUMERIC,
    "Q4" NUMERIC
);

```

Pattern 4: Hierarchical Data with Recursive CTEs

For handling tree structures:

```

WITH RECURSIVE org_chart AS (
    -- Base case: top-level employees (no manager)
    SELECT employee_id, name, manager_id, 1 AS level, ARRAY[name] AS path
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive case: employees with managers
    SELECT e.employee_id, e.name, e.manager_id, oc.level + 1, oc.path || e.name
    FROM employees e
    JOIN org_chart oc ON e.manager_id = oc.employee_id
)
SELECT level, repeat(' ', level-1) || name AS org_structure
FROM org_chart
ORDER BY path;

```

Conclusion

Mastering PostgreSQL joins is essential for efficient data retrieval and manipulation. By understanding the different join types, optimization

techniques, and common patterns, you can write more efficient queries that perform well even as your data grows.

Remember these key takeaways:

- Choose the right join type based on your specific needs
- Index your join columns appropriately
- Use EXPLAIN ANALYZE to verify query performance
- Watch out for common pitfalls like NULL values and Cartesian products
- Consider advanced techniques like LATERAL joins and materialized views for complex scenarios

With these tools in your arsenal, you'll be well-equipped to tackle even the most complex data relationships in your PostgreSQL databases.

Additional Resources

For further reading on PostgreSQL joins and optimization:

1. [PostgreSQL Official Documentation on Joins](#)
2. [Use the Index, Luke!](#) — A guide to database performance for developers
3. [PostgreSQL: Introduction and Concepts](#) — The definitive PostgreSQL resource
4. [PostgreSQL Query Optimization](#) — Official performance tips

Postgresql

High Performance



Written by Rizqi Mulki

Following ▾

170 followers · 29 following

Backend Developer specializing in PHP, Python & Node.js. Build fast, secure & scalable systems. Email : rizqimulkisrc@gmail.com.

No responses yet



Gvadakte

What are your thoughts?

More from Rizqi Mulki



Rizqi Mulki

PostgreSQL for Real-Time Apps: Performance Techniques

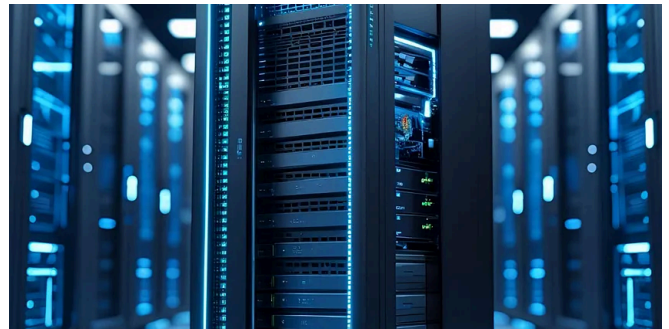
How we built a chat app serving 100,000 concurrent users with PostgreSQL — and w...



Sep 4



8



Rizqi Mulki

Advanced PostgreSQL Partitioning Strategies That Scale to Billions o...

How a single table with 10 billion rows brought our startup to its knees—and the partitionin...



Aug 13

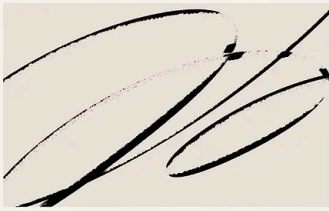


23



1





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 Rizqi Mulki

AWS Lambda Cold Starts in 2025: What Changed and How to...

The serverless landscape just shifted dramatically—here's everything developers...



6d ago

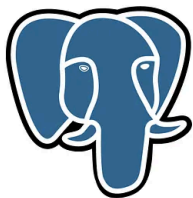


5



See all from Rizqi Mulki

Recommended from Medium



**Beyond Basic
PostgreSQL
Programmable
Objects**



In Stackademic by bektaw

Beyond Basics: PostgreSQL Programmable Objects (Automat...


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



Rizqi Mulki

Why PostgreSQL is Beating MySQL in 2025: The Surprising Truth



 Azlan Jamal

Mastering PostgreSQL Array Types: When, Why, and How to Us...

PostgreSQL's array type is a powerful feature that lets you store multiple values of the sam...

★ Aug 11 🔖 ...

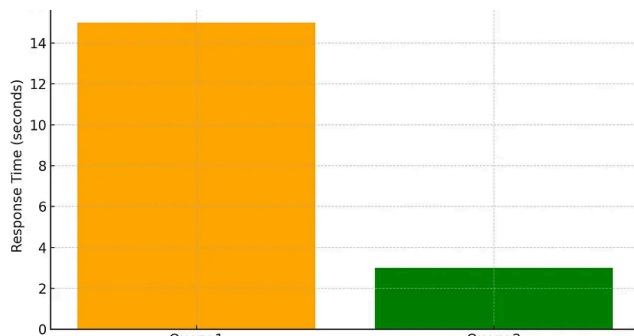


 Saakshi Srivastava

Multi-Column and Single-Column Indexes in PostgreSQL

As the names suggest, a multi-column index is an index on multiple columns (in a...

Apr 8 🔖 ...

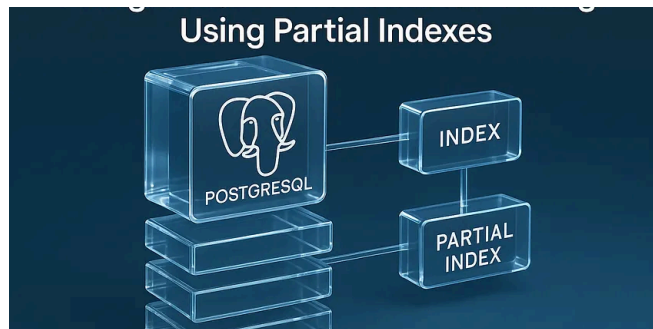


 In Javarevisited by Jitin Kayyala

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...

★ Jul 9 👤 15 🔖 ...



 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