PostgreSQL 18 RETURNING Enhancements: A Game Changer for Modern Applications

Ahsan Hadi | January 6, 2026

PostgreSQL 18 has arrived with some fantastic improvements, and among them, the RETURNING clause enhancements stand out as a feature that every PostgreSQL developer and DBA should be excited about. In this blog, I'll explore these enhancements, with particular focus on the MERGE RETURNING clause enhancement, and demonstrate how they can simplify your application architecture and improve data tracking capabilities.

Background: The RETURNING Clause Evolution

The RETURNING clause has been a staple of Postgres for years, allowing `INSERT`, `UPDATE`, and `DELETE` operations to return data about the affected rows. This capability eliminates the need for follow-up `SELECT` queries, reducing round trips to the database and improving performance. However, before Postgres 18, the RETURNING clause had significant limitations that forced developers into workarounds and compromises.

In Postgres 17, the community introduced RETURNING support for MERGE statements (commit c649fa24a), which was already a major step forward. MERGE itself had been introduced back in Postgres 15, providing a powerful way to perform conditional `INSERT`, `UPDATE`, or `DELETE` operations in a single statement, but `MERGE` without RETURNING didn't provide an easy way to see what you'd accomplished.

What's New in PostgreSQL 18?

Postgres 18 takes the RETURNING clause to the next level by introducing OLD and NEW aliases (commit 80feb727c8), authored by Dean Rasheed and reviewed by Jian He and Jeff Davis. This enhancement fundamentally changes how you can capture data during DML operations.

The Problem Before PostgreSQL 18

Previously, the RETURNING clause had these limitations; despite being syntactically similar, when applied to different query types:

- **INSERT and UPDATE** could only return new/current values.
- **DELETE** could only return old values
- **MERGE** would return values based on the internal action executed (`INSERT`, `UPDATE`, or `DELETE`).

If you needed to compare before-and-after values or track what actually changed during an update, you had limited options; you could:

- run separate `SELECT` queries before the modification.
- implement complex trigger functions.
- use application-level logic to track changes.
- resort to workarounds like checking system columns (e.g., xmax).

These approaches added complexity, increased latency, and made your code harder to maintain.

The Solution: OLD and NEW Aliases

Postgres 18 introduces the special aliases `old` and `new`, which allow you to explicitly access both the previous state and the current state of data within a single statement. This works across all DML operations: `INSERT`, `UPDATE`, `DELETE`, and `MERGE`.

The syntax is straightforward:

```
UPDATE table_name
SET column = new_value
```

```
WHERE condition
RETURNING old.column AS old_value, new.column AS new_value;
```

You can also rename these aliases to avoid conflicts with existing column names or when working within trigger functions:

```
UPDATE accounts
SET balance = balance - 50
WHERE account_id = 123
RETURNING WITH (OLD AS previous, NEW AS current)
    previous.balance AS old_balance,
    current.balance AS new_balance;
```

MERGE RETURNING: The Complete Picture

The combination of MERGE and RETURNING in Postgres 18 creates an incredibly powerful tool for upsert operations - let me walk you through a practical example that demonstrates the full capabilities.

Practical Example: Product Inventory System

Consider a product inventory management system where you need to sync data from external sources. You want to insert new products, update existing products, and track exactly what happened to each row.

First, let's set up our tables:

```
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_code VARCHAR(50) UNIQUE NOT NULL,
    product_name VARCHAR(200) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    stock_quantity INTEGER NOT NULL DEFAULT 0,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE product_staging (
    product_code VARCHAR(50),
    product_name VARCHAR(200),
    price DECIMAL(10, 2),
    stock_quantity INTEGER
);
```

Now, let's populate the tables with some initial data:

```
INSERT INTO products (product_code, product_name, price, stock_quantity)
VALUES
    ('LAPTOP-001', 'Premium Laptop', 999.99, 50),
    ('MOUSE-001', 'Wireless Mouse', 29.99, 200),
    ('KEYBOARD-001', 'Mechanical Keyboard', 79.99, 150);

INSERT INTO product_staging (product_code, product_name, price, stock_quantity
VALUES
    ('LAPTOP-001', 'Premium Laptop Pro', 1099.99, 45),  -- Update existing
    ('MONITOR-001', '4K Monitor', 399.99, 75),          -- New product
    ('MOUSE-001', 'Wireless Mouse', 29.99, 200);        -- No actual change
```

Basic MERGE with RETURNING

This is a MERGE operation that shows what action was performed:

```
MERGE INTO products p
USING product_staging s ON p.product_code = s.product_code
WHEN MATCHED THEN
    UPDATE SET
        product_name = s.product_name,
        price = s.price,
        stock_quantity = s.stock_quantity,
        last_updated = CURRENT_TIMESTAMP
WHEN NOT MATCHED THEN
    INSERT (product_code, product_name, price, stock_quantity)
    VALUES (s.product_code, s.product_name, s.price, s.stock_quantity)
RETURNING
    p.product_code,
    p.product_name,
    merge_action() AS action_performed;
```

This query returns:

```
 product_code  |    product_name     | action_performed
---------------+---------------------+------------------
 LAPTOP-001    | Premium Laptop Pro  | UPDATE
 MONITOR-001   | 4K Monitor          | INSERT
 MOUSE-001     | Wireless Mouse      | UPDATE
```

Advanced MERGE with OLD and NEW

Now let's leverage the OLD and NEW aliases to track detailed changes:

This query below retrieves both the old (before) and new (after) values for product_name and price columns from the affected rows. By aliasing them (old_name, new_name, old_price, new_price), you can easily compare what the values were before the MERGE operation versus what they are after, enabling change tracking and audit logging.

```
MERGE INTO products p
USING product_staging s ON p.product_code = s.product_code
WHEN MATCHED THEN
    UPDATE SET
        product_name = s.product_name,
        price = s.price,
        stock_quantity = s.stock_quantity,
        last_updated = CURRENT_TIMESTAMP
WHEN NOT MATCHED THEN
    INSERT (product_code, product_name, price, stock_quantity)
    VALUES (s.product_code, s.product_name, s.price, s.stock_quantity)
RETURNING
    p.product_code,
    merge_action() AS action,
    old.product_name AS old_name,
    new.product_name AS new_name,
    old.price AS old_price,
    new.price AS new_price,
    old.stock_quantity AS old_stock,
    new.stock_quantity AS new_stock,
    (old.price IS DISTINCT FROM new.price) AS price_changed,
    (old.stock_quantity IS DISTINCT FROM new.stock_quantity) AS stock_changed;
```

This comprehensive query returns:

```
 product_code  | action | old_name         | new_name            | old_price
---------------+--------+------------------+---------------------+-----------
 LAPTOP-001    | UPDATE | Premium Laptop   | Premium Laptop Pro  | 999.99
 MONITOR-001   | INSERT | NULL             | 4K Monitor          | NULL
 MOUSE-001     | UPDATE | Wireless Mouse   | Wireless Mouse      | 29.99
```

Notice how for the INSERT operation, old values are NULL, while for UPDATE operations, you get complete visibility into what changed.

Building an Audit Trail

One of the most powerful use cases is building comprehensive audit trails without using triggers:

-- Create audit table

```
CREATE TABLE product_audit (
    audit_id SERIAL PRIMARY KEY,
    product_code VARCHAR(50),
    action VARCHAR(10),
    old_values JSONB,
    new_values JSONB,
    changes JSONB,
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

-- Perform MERGE with detailed audit trail

```
WITH merge_results AS (
    MERGE INTO products p
    USING product_staging s ON p.product_code = s.product_code
    WHEN MATCHED THEN
        UPDATE SET
            product_name = s.product_name,
            price = s.price,
            stock_quantity = s.stock_quantity,
            last_updated = CURRENT_TIMESTAMP
    WHEN NOT MATCHED THEN
        INSERT (product_code, product_name, price, stock_quantity)
        VALUES (s.product_code, s.product_name, s.price, s.stock_quantity)
    RETURNING
```

```
        p.product_code,
        merge_action() AS action,
        jsonb_build_object(
            'name', old.product_name,
            'price', old.price,
            'stock', old.stock_quantity
        ) AS old_values,
        jsonb_build_object(
            'name', new.product_name,
            'price', new.price,
            'stock', new.stock_quantity
        ) AS new_values
)
INSERT INTO product_audit (product_code, action, old_values, new_values, chang
SELECT
    product_code,
    action,
    old_values,
    new_values,
    CASE
        WHEN action = 'INSERT' THEN new_values
        WHEN action = 'DELETE' THEN old_values
        ELSE (
            SELECT jsonb_object_agg(key, value)
            FROM jsonb_each(new_values)
            WHERE value IS DISTINCT FROM old_values->key
        )
    END AS changes
FROM merge_results;
```

-- Results from audit trail

```
select * from product_audit;
 audit_id | product_code | action |                          old_values
             new_values                           | changes |         changed_
----------+--------------+--------+------------------------------------------
--------------------------------------------------+---------+------------------
        1 | LAPTOP-001   | UPDATE | {"name": "Premium Laptop Pro", "price": 10
emium Laptop Pro", "price": 1099.99, "stock": 45} |         | 2025-12-12 16:27
        2 | MONITOR-001  | UPDATE | {"name": "4K Monitor", "price": 399.99, "s
 Monitor", "price": 399.99, "stock": 75}          |         | 2025-12-12 16:27
        3 | MOUSE-001    | UPDATE | {"name": "Wireless Mouse", "price": 29.99,
reless Mouse", "price": 29.99, "stock": 200}      |         | 2025-12-12 16:27
(3 rows)
```

This creates a complete audit trail showing exactly what changed, all in a single atomic operation.

Looking Forward

The RETURNING enhancements in Postgres 18 represent a significant step forward in making Postgres more developer-friendly and reducing the need for complex workarounds. The ability to access both old and new values in a single atomic operation simplifies many common patterns in application development.

Some areas where this feature could evolve in future releases:

- Extended MERGE capabilities that provide more sophisticated MERGE operations with additional WHEN clauses.
- Aggregate support that offers the ability to aggregate RETURNING results directly.
- Cross-table returns that enable returning data from related tables in a single operation.

Technical Details and Commit References

For those interested in the technical implementation details:

- **MERGE RETURNING** (PostgreSQL 17): Commit `c649fa24a` by Dean Rasheed
- **OLD/NEW Support** (PostgreSQL 18): Commit `80feb727c8` by Dean Rasheed, reviewed by Jian He and Jeff Davis
- **Discussion Thread**: https://postgr.es/m/CAEZATCWx0J0-v=Qjc6gXzR=KtsdvAE7Ow=D=mu50AgOe+pvisQ@mail.gmail.com

The implementation involved changes across multiple components:

- Executor (execExpr.c, execExprInterp.c, nodeModifyTable.c)
- Parser (parse_target.c)
- Optimizer (createplan.c, setrefs.c, subselect.c)
- Nodes (makefuncs.c, nodeFuncs.c)

Conclusion

Postgres 18's RETURNING enhancements, particularly the OLD and NEW aliases for tracking changes across `INSERT`, `UPDATE`, `DELETE`, and `MERGE` operations, represent a significant improvement for application developers. These features eliminate the need for many workarounds and external audit trail implementations, allowing you to build more maintainable and performant applications.

The combination of `MERGE` with comprehensive RETURNING capabilities gives you unprecedented control over upsert operations, complete visibility into what changed, and the ability to build sophisticated audit trails without trigger overhead.

If you're working with Postgres and dealing with data synchronization, change tracking, or audit requirements, Postgres 18's enhanced RETURNING clause should be at the top of your list of features to explore. It's one of those features that, once you start using it, you'll wonder how you ever lived without it.

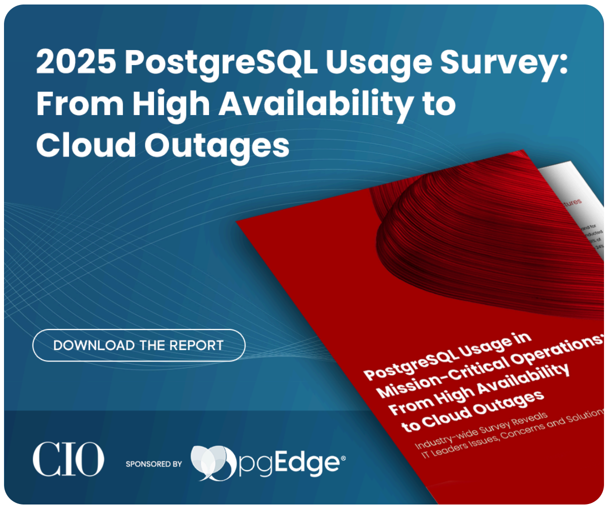
As the Postgres community continues to evolve the database with practical, developer-friendly features, enhancements like these reinforce why PostgreSQL remains the world's most advanced open source database. I encourage you to upgrade to Postgres 18 and start experimenting with these powerful new capabilities in your applications!

About the Author

Ahsan Hadi

Ahsan is a database evangelist with over 20 years of development and management experience. He is passionate about databases, and has worked with Postgres and Oracle extensively throughout his career. With over 15 years of working with PostgreSQL, he has worked with companies like EDB as a Senior

Director of Product Development, HighGo Software as VP of Product Development, and as a Programmer Analyst with British Telecom.


2025 PostgreSQL Usage Survey: From High Availability to Cloud Outages
DOWNLOAD THE REPORT
PostgreSQL Usage in Mission-Critical Operations: From High Availability to Cloud Outages
Industry-wide Survey Reveals IT Leaders Issues, Concerns and Solutions
CIO
SPONSORED BY
pgEdge

Distributed PostgreSQL Buyers Guide
How to evaluate vendor claims of Postgres compatibility
GET IT NOW
pgEdge
Distributed PostgreSQL Buyers Guide
pgEdge

SUBSCRIBE TO BLOG

Email* SUBMIT

Results of industry-wide survey on Postgres high availability

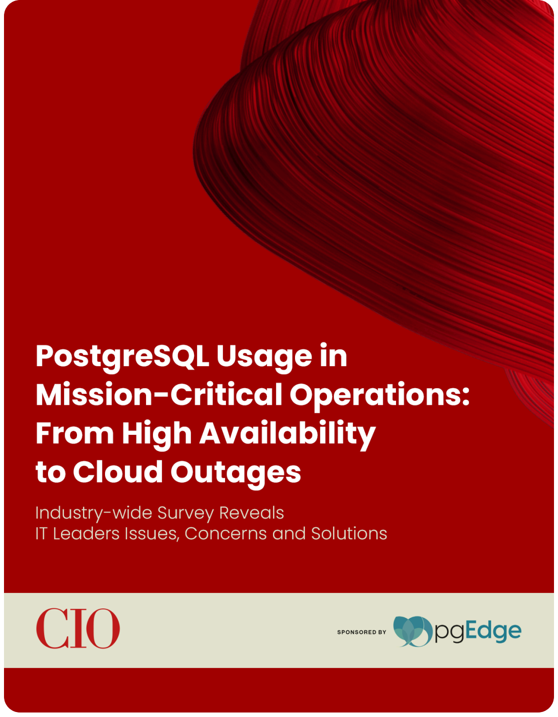
For complete survey details on IT leaders common issues, threats, and solutions please download the white paper:

First name*	Last name*
Email*	Company*
Title	

[✓] Subscribe to blog for technical tips and updates.

DOWNLOAD WHITEPAPER

Your data is always private and never sold. You can unsubscribe from communications any time. For more info view our Privacy Policy.


PostgreSQL Usage in Mission-Critical Operations: From High Availability to Cloud Outages

Industry-wide Survey Reveals IT Leaders Issues, Concerns and Solutions

CIO

SPONSORED BY pgEdge

Get started today.

Experience the magic of pgEdge for non-distributed and distributed Postgres deployments.

SIGN UP FREE

Hot Topics

Distributed Postgres

Enterprise Postgres

Postgres Replication

PostgreSQL High Availability

Multi-master

Conflict resolution and avoidance

Postgres Download

Products

pgEdge Enterprise Postgres

pgEdge Distributed Postgres

pgEdge Agentic AI Toolkit

Postgres Support Services

Feature Comparison

pgEdge Cloud

pgEdge Container on Kubernetes

Company

About Us

Contact

Support

Solutions

Low Data Latency

Data Residency

PostgreSQL High Availability

Multi-master

Developer Hub

Blog

Webinars

Documentation

Videos