Photo by Lauren Mancke on Unsplash

Member-only story

10 PostgreSQL Queries/Functions I Find The Most Useful

Diwash Tamang · Following

Published in Dev Genius

9 min read · Sep 11, 2024

I have been working with PostgreSQL for quite some time now. Although I wouldn't call myself an expert, I can say I am a person who has been writing SQL queries **on and off** for 4-5 years, usually to monitor different processes or fix some incorrect data in the tables.

Here's a list of queries, and SQL functions that have been the most helpful for me.

For non-members, click here to read the full blog.

1. REPLACE & ARRAY_REPLACE

The `REPLACE` function is great for modifying parts of a string, such as correcting typos or standardizing values. `ARRAY_REPLACE` is a similar function used for replacing elements in an array. This is particularly useful when dealing with `ARRAY` type fields such as an array of strings or ints.

Syntax:

```
REPLACE(input_string, string_to_replace, replacement_string)
ARRAY_REPLACE(input_array, element_to_replace, replacement_element)
```

Example:

```
SELECT REPLACE('Bartolomeo Messi', 'Bartolomeo', 'Lionel') AS greeting;
-- OUTPUT
| greeting     |
| ------------ |
| Lionel Messi |

SELECT REPLACE(ARRAY[1,2,3,4,5], 3, 4) AS new_array;
-- OUTPUT
| new_array   |
```

```
|-------------|
| {1,2,4,4,5} |
```

2. CTE

Common Table Expression (CTE) is a temporary result set that can be referenced further down in the query. They allow you to temporarily store query results to be used & referenced in another part of the query. They are especially useful for making complex queries more readable.

Syntax:

```
WITH cte_name AS ( -- defining cte
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition
)
SELECT column1, column2, ...
FROM cte_name; -- referencing cte here
```

Here's an example of how to select the top 10 defenders from FC Barcelona using CTEs. Even when I wrote this example, I noticed how much easier it is to write down the logic with CTEs compared to trying to do it all in a single query.

Example: Getting the top 10 Barcelona defenders of all time

```
-- QUERY: top 10 defenders with CTEs
WITH barca_player_ids AS (
-- get the barca palyer ids first
  SELECT player_id FROM player_club_mapping
  WHERE club_name = 'FC_Barcelona'
),
top_barca_defs AS (
```

```
-- getting top 10 barca defs using game stats table
  SELECT player_id, AVG(attribute_value) def_stats FROM game_stats
  WHERE
    player_id IN (SELECT player_id FROM barca_player_ids)
    AND attribute_name IN ('Defense', 'Pace', 'Tackle', 'Physicality')
  GROUP BY player_id
  ORDER BY AVG(attribute_value) DESC
  LIMIT 10
)
-- joining the top 5 defenders with their names to produce the final result
SELECT name, contract_duration FROM players p
INNER JOIN top_barca_defs tbd
  ON tbd.player_id = p.id
ORDER BY tbd.def_stats DESC;

-- OUTPUT:
| name                                | contract duration |
|-------------------------------------|-------------------|
| Gerard Pique                        |  2008-now         |
| Dani Alves                          |  2008-now         |
| Rafa Marquez                        |  2003-2010        |
| Carles Puyol                        |  1999-now         |
| Michael Reiziger                    |  1997-2004        |
| Ronald "Tintin" Roeman              |  1989-1995        |
| Miguel "Migueli" Bernardo Bianquetti |  1973-1988        |
| Antoni Torres                       |  1965-1976        |
| Sigfrid Gracia                      |  1952-1966        |
| Joan Segarra                        |  1949-1964        |
```

Example: First we get the player_ids of all barca players using `barca_player_ids` CTE. Then, we get the top barca defenders using the average defence-related attributes of barca players using `top_barca_defs` CTE. Finally, we select the player name and contract duration using those top_barca_def IDs and the `players` table.

3. VALUES JOIN

The `VALUES` clause is commonly used to insert multiple rows into a table. However,

its capabilities extend beyond insert statements. I recently discovered that they can be treated as static tables and joined with other tables. This blew my mind.

The following example shows how to add external columns to your fetch query. Suppose you have a table `players` which holds `player_id`, `name`, and `player position`. You have the date of birth (DOB) for these players in an external file and want to view it alongside the DB columns. Here's how you can do it using `VALUES` clause.

Example: Showing DOBs of players alongside

```
-- QUERY
SELECT p.player_id, v.name, p.position, v.dob, v.gender FROM players p
INNER JOIN ( VALUES
  ('Ronaldo', '1985-02-05', 'M'),
  ('Messi', '1987-06-24', 'M')
) AS v(name, dob, gender)
ON p.name = v.name;
-- using it add columns that doesn't exists in the players table
-- in the select output

-- OUTPUT
| player_id | name    | position              | dob        | gender |
|-----------|---------|-----------------------|------------|--------|
| 1         | Messi   | Attacking Midfielder |1985-02-05 | M      |
| 2         | Ronaldo | Winger                |1987-06-24 | M      |
```

4. BATCH UPDATE

For some reason, batch updates with raw SQL are always something that I struggle with. Although I don't use it all the time, it is a really useful query when needed.

Example: Update player salary in `players` table using another table `salary_updates`

```
-- QUERY: batch update using another table
UPDATE players p
SET salary = su.new_salary
FROM salary_updates su
WHERE p.player_id = su.player_id;
-- where condition is used to join the two tables
```

Explanation: The above query updates the `salary` column in the `players` table based on the data from the `salary_updates` table. It joins the two tables on `player_id` to apply the new salary values to the corresponding players.

Example: Updating player's club in batch using VALUES clause

```
-- QUERY: batch update using the values clause
UPDATE players
SET club = new_values.club
( VALUES
  ('Lionel Messi', 'Inter Miami FC'),
  ('Cristiano Ronaldo', 'Al-Nassr FC')
) AS new_values(player_name, club)
WHERE players.name = new_values.player_name;
```

Explanation: You can also use `VALUES` clause to do a similar thing. The query above updates the `club` column in the `players` table using static data provided in the `VALUES` clause. It joins the static values with the `players` table on `name` to apply the new club values to the corresponding players.

5. Window Functions

Window functions are powerful tools that allow you to perform calculations across a set of table rows based on the value of a particular column. However, unlike group

by, they do not collapse rows into a single result.

Example: Comparing the average salary of a club vs player's individual salary

```
-- QUERY: window function (partition by)
SELECT
    id,
    name,
    salary,
    AVG(salary) OVER (PARTITION BY club) AS avg_salary_by_club,
    club
FROM players;
-- OUTPUT
| id | name     | salary   | av_salary_by_club | club    |
|----|----------|----------|-------------------|---------|
| 1  | Alvarez  | 20,000   | 100,000           | BFC     |
| 2  | Swarez   | 180,000  | 100,000           | BFC     |
| 3  | Rodrique | 20,000   | 25,000            | RFC     |
| 4  | Modrique | 30,000   | 25,000            | RFC     |

-- as you can see, avg salary by
-- club (the partition by column) is calculated
-- then it is added to all players that belong
-- to the club & not grouped by the club.

-- contrast with group by query
SELECT club, AVG(salary) FROM players GROUP BY club;
-- OUTPUT
| av_salary_by_club | club    |
|-------------------|---------|
| 100,000           | BFC     |
| 25,000            | RFC     |
```

Explanation: This query calculates the average salary for each club while still returning individual player records, allowing you to compare each employee's salary to the club's average.

6. Delete Duplicates

All developers come across a time when they need to clean up duplicates in a table. Here's how you can clean up duplicate records. In the first scenario, we have multiple records in the `players` table with the same `name`, `jersey_number` and `club`. We want to resolve this by keeping the latest inserted data and removing the old ones.

Example:

```
-- QUERY: deleting duplicate players only keeping the latest records
WITH duplicates AS (
    SELECT
       id,
      ROW_NUMBER() OVER (
        PARTITION BY name, jersey_number, club
        ORDER BY insert_date DESC
      ) AS rn
    FROM active_players
)
DELETE FROM players
WHERE id IN (
    SELECT id FROM duplicates
    WHERE rn > 1
);
```

What if even the ID column is duplicated? Well, there's a system column that PostgreSQL uses to keep track of records called **ctid**. This column is always unique. We can use this to delete the duplicates as shown below.

```
-- QUERY: identifies duplicates and removes them while keeping the first occurr
WITH duplicates AS (
    SELECT
      ctid,
      ROW_NUMBER() OVER (
        PARTITION BY id, name, jersery_number, club
        ORDER BY ctid DESC
      ) AS rn
    FROM players
)
```

```
DELETE FROM players
WHERE ctid IN (
    SELECT ctid FROM duplicates
    WHERE rn > 1
);
```

7. Using RETURNING in Update and Delete Queries

The `RETURNING` clause allows you to immediately retrieve the affected rows after an `UPDATE` or `DELETE` operation. This is useful for logging or further processing the modified data. I use it to make sure I correctly changed the tables after running delete and update queries.

Example:

```
-- QUERY: returning the updated records
UPDATE players
SET age += 1
WHERE id = 1001
RETURNING id, name, age;
-- OUTPUT
| id | name    | age |
|----|---------|-----|
| 1  | Ronaldo | 50  |

-- QUERY: returning deleted records
DELETE FROM employees
WHERE id = 5
RETURNING id, name;
-- OUTPUT
| id | name    |
|----|---------|
| 5  | Diwash  |
```

You can also use returning in combination with CTEs. This can be useful when you want to update some records and based on the ids / records were updated you want

to update some other records. Or maybe to delete some values based on another updated or deleted ids.

```
-- updating records in one table and using the ids to update another table as w
WITH updated_ids AS (
  UPDATE table1
  SET column='some_value'
  WHERE some_condition
  RETURNING id
)
UPDATE some_other_table
SET column = 'some_other_value'
WHERE table1_id IN (SELECT id FROM updated_ids);

-- deleting some records based on some updated records
WITH delete_ids AS (
  UPDATE table1
  SET column='some_value'
  WHERE some_condition
  RETURNING id
)
DELETE FROM some_other_table
WHERE table1_id IN (SELECT id FROM updated_ids);
```

8. Adding a Serial Number Using ROW_NUMBER

On my first ever job, I needed to add a serial number in a `SELECT` query. I was trying to do some calculations, though I can't remember exactly what they were. I tried for like 30 minutes and couldn't figure it out. A senior developer eventually helped me solve the issue without needing the serial number at all.

But the curiosity stayed with me. I still wanted to try it after having a better understanding of how window functions work. And that is the only reason this is here on the list. Turns out it wasn't that difficult after all.

Example:

```
-- QUERY
SELECT *, ROW_NUMBER() OVER (PARTITION BY 1) AS rn
FROM diwash_test;
-- OR more simply
SELECT *, ROW_NUMBER() OVER () AS rn
FROM diwash_test;
-- OUTPUT
| id | name        | rn |
|----|-------------|----|
| 1  | Messi       | 1  |
| 2  | Ronaldo     | 2  |
| 3  | Dembele     | 3  |
| 5  | Bellingham  | 4  |
| 7  | Mbappe      | 5  |
```

Explanation: This query is useful when you need to assign a serial row number to each row in a table or result set without any specific partitioning. The use of `PARTITION BY 1` ensures that the entire result set is treated as a single group, and thus each row is assigned a unique sequential number.

9. ARRAY_AGG & ARRAY CONTAINS(@>)

The `ARRAY_AGG` function is useful for aggregating values into an array. This is definitely the one aggregation function that I use the most.

Example: Find the players who have played for both Real Madrid and Barcelona, along with a list of all the clubs they have played for.

```
-- QUERY
SELECT player_name, ARRAY_AGG(club) clubs FROM player_club_mappings
GROUP BY player_name
HAVING ARRAY_AGG(club) @> ARRAY['Barcelona', 'Real Madrid']
LIMIT 3
```

```
-- @> Operator: This is the "contains" operator for arrays
-- in PostgreSQL. It verifies if the aggregated array
-- includes all elements of the specified array
-- (ARRAY['Barcelona', 'Real Madrid']).

-- OUTPUT
| player_name     | clubs
|-----------------|-------------------------------------------------------------
| Ronaldo Nazario | {Corinthians, Cruzeiro, Palmeiras, PSV Eindhoven, Barcelona
| Samuel Eto''o   | {Kadji Sports, Union Douala, Canon Yaoundé, Real Madrid, Ma
| Bernd Schuster  | {1. FC Köln, Hamburger SV, Real Madrid, Barcelona, Paris Sa
```

Explanation: This query aggregates all the clubs that players have played in using `ARRAY_AGG`. Then, it filters out only the players that have played with both Barcelona and Real Madrid using `@>` (array contains).

10. SPLIT_PART

The `SPLIT_PART` function allows you to split a string on a specified delimiter and return the nth substring. This is useful for extracting parts of a string, such as pulling out domain names from email addresses.

Example: Pulling out domain names from email addresses

```
-- QUERY
SELECT name, email, SPLIT_PART(email, '@', 2) email_domain AS domain
FROM players
LIMIT 3;
-- OUTPUT
| name    | email              | email_domain  |
|---------|--------------------|---------------|
| Diwash  | diwash@gmail.com   | gmail.com     |
| Hari    | hari@outlook.com   | outlook.com   |
| Rudra   | rudra@discord.com  | discord.com   |
```

Conclusion

We have covered 10 PostgreSQL functions/queries that I find very useful. Hopefully, this blog was helpful to you & you found something that you can use in your work.

Thank you for reading. If you learnt something from the blog, please leave some claps on the blog. 😄

If you enjoyed this blog, you might also like my blog on, "How I achieved 130x speed optimization using relatively simple techniques".

How I Made A Python Script 130 Times Faster with PostgreSQL Query Optimization

From 14 days to 2.5 hours

blog.devgenius.io

Postgresql Tech Programming Database Sql

Following

Written by Diwash Tamang

47 Followers · Writer for Dev Genius

Geomatics Engineer | Python Developer | Lifelong Learner | Aspiring Blogger | Pursuit of Excellence

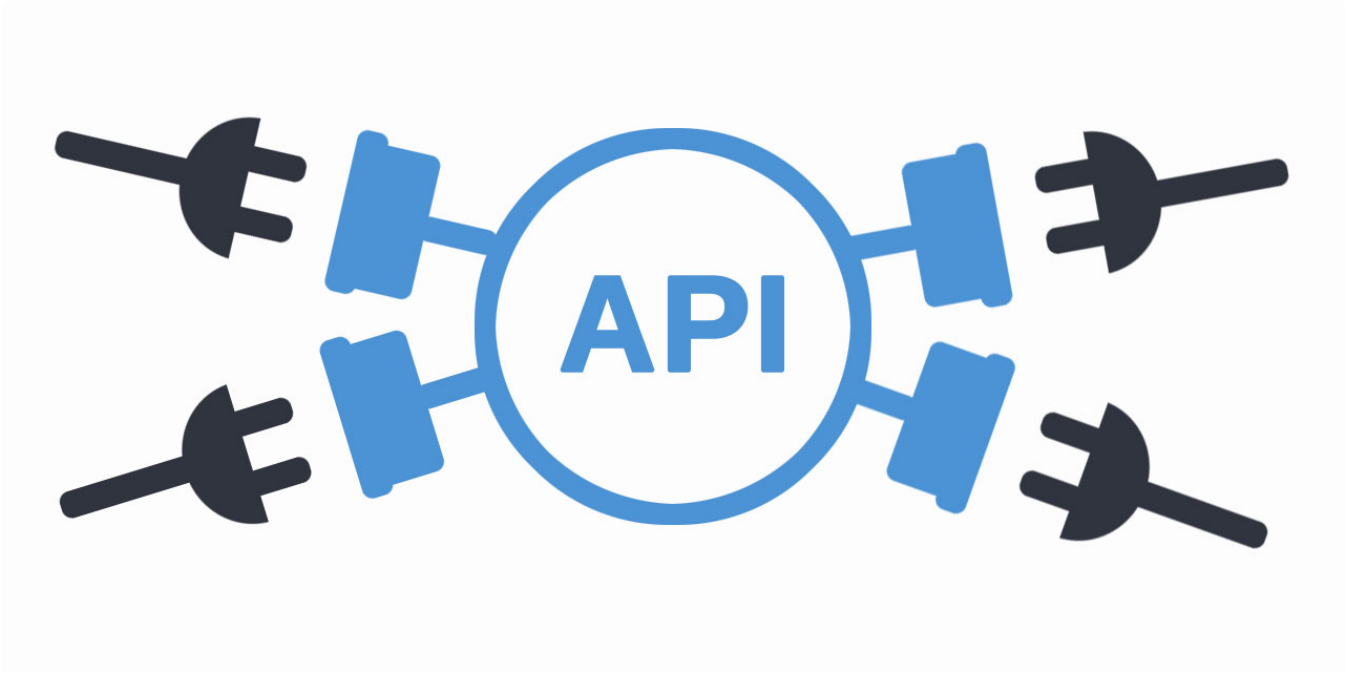
More from Diwash Tamang and Dev Genius

Diwash Tamang in Dev Genius

How I Made A Python Script 130 Times Faster with PostgreSQL Query Optimization

From 14 days to 2.5 hours

Aug 18 364 2



Seliesh Jacob in Dev Genius

API Design: From Basics to Best Practices

Introduction

Jun 3 2.2K 24

Most Asked
Java 8
Interview
Coding Questions
(Part-1)

Anusha SP in Dev Genius

Java 8 Coding and Programming Interview Questions and Answers

It has been 8 years since Java 8 was released. I have already shared the Java 8 Interview Questions and Answers and also Java 8 Stream API...

1.3K 17

Diwash Tamang in Dev Genius

Handling Large Tables With Partitioning in PostgreSQL

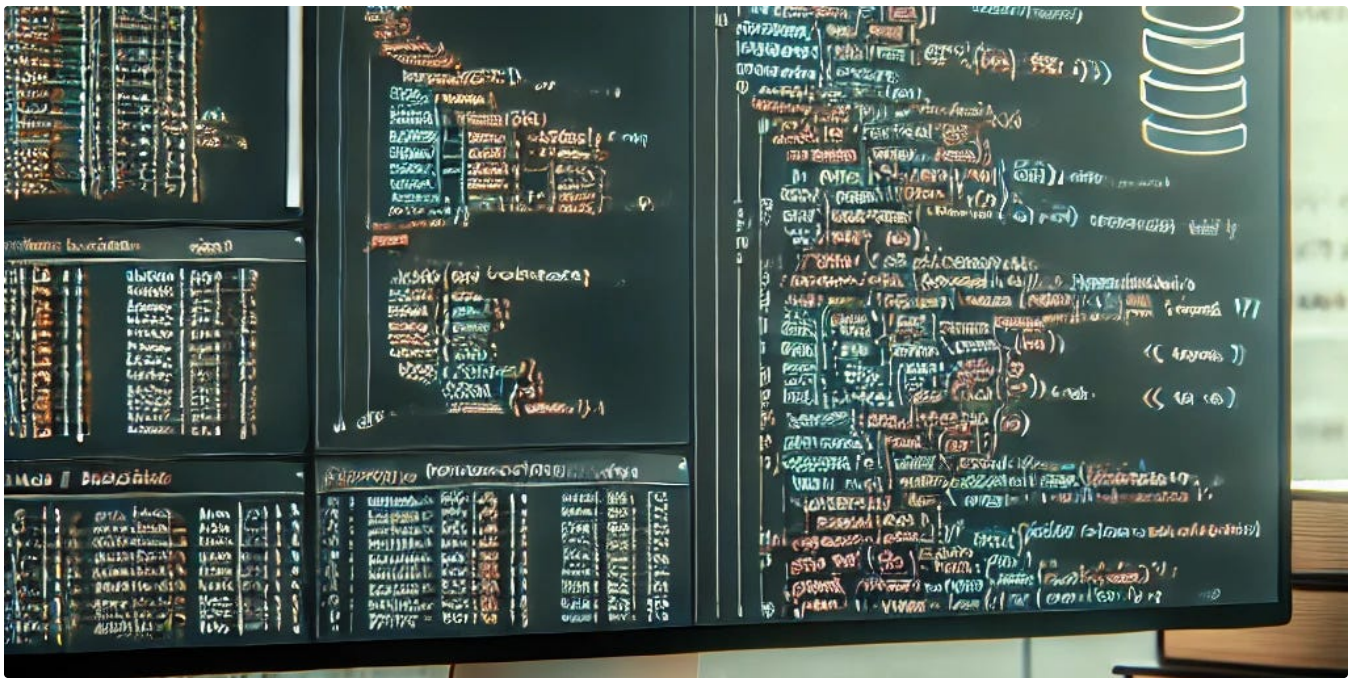
Sometimes we come across tables that grow to millions and billions of rows. One of the ways we can deal with this problem is partitioning...

Sep 18 5

See all from Diwash Tamang

See all from Dev Genius

Recommended from Medium

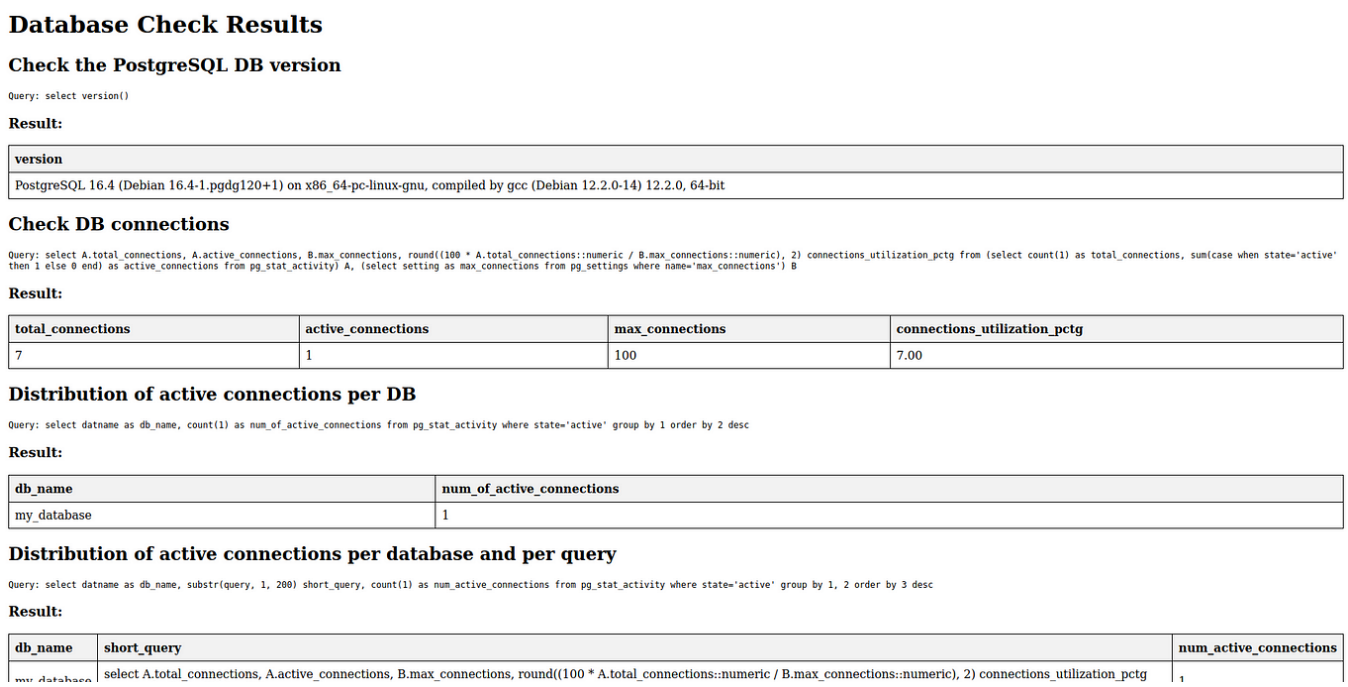


Rengith Manickam(Ranjith)

PostgreSQL writing multiline code

In PostgreSQL, writing multiline code, such as functions or complex queries, can be done using the DO block or by creating a function. You...

Sep 18 1



Database Check Results

Check the PostgreSQL DB version

Query: select version()

Result:

version
PostgreSQL 16.4 (Debian 16.4-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit

Check DB connections

Query: select A.total_connections, A.active_connections, B.max_connections, round((100 * A.total_connections::numeric / B.max_connections::numeric), 2) connections_utilization_pctg from (select count(1) as total_connections, sum(case when state='active' then 1 else 0 end) as active_connections from pg_stat_activity) A, (select setting as max_connections from pg_settings where name='max_connections') B

Result:

total_connections	active_connections	max_connections	connections_utilization_pctg
7	1	100	7.00

Distribution of active connections per DB

Query: select datname as db_name, count(1) as num_of_active_connections from pg_stat_activity where state='active' group by 1 order by 2 desc

Result:

db_name	num_of_active_connections
my_database	1

Distribution of active connections per database and per query

Query: select datname as db_name, substr(query, 1, 200) short_query, count(1) as num_active_connections from pg_stat_activity where state='active' group by 1, 2 order by 3 desc

Result:

db_name	short_query	num_active_connections
my_database	select A.total_connections, A.active_connections, B.max_connections, round((100 * A.total_connections::numeric / B.max_connections::numeric), 2) connections_utilization_pctg	1

Dmitry Romanoff

Automating Database Health Checks with Python: A Step-by-Step Guide

In the world of database management, ensuring the health and performance of your databases is crucial. One effective way to achieve this is...

Aug 23 9

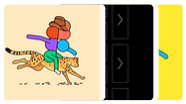
Lists

Apple's Vision Pro

7 stories · 75 saves

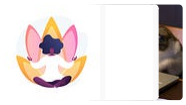
Business 101

25 stories · 1180 saves



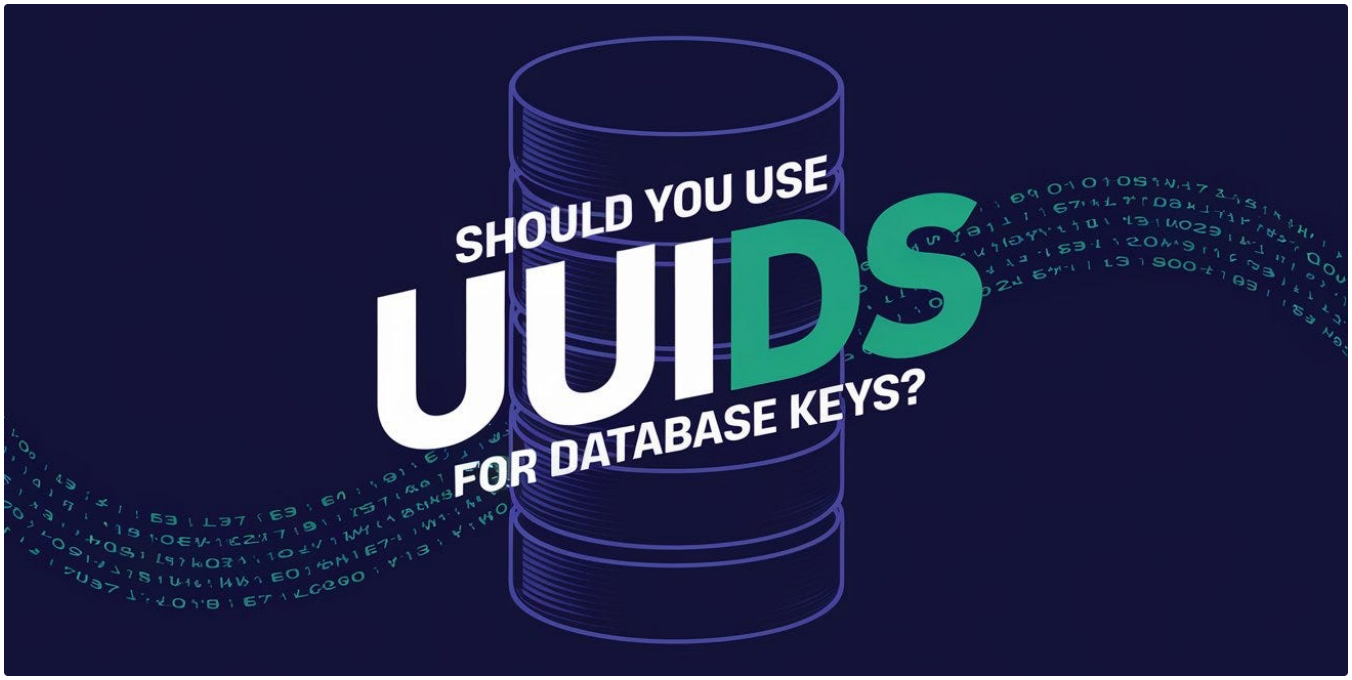
Figma 101

7 stories · 729 saves



Stories to Help You Grow as a Software Developer

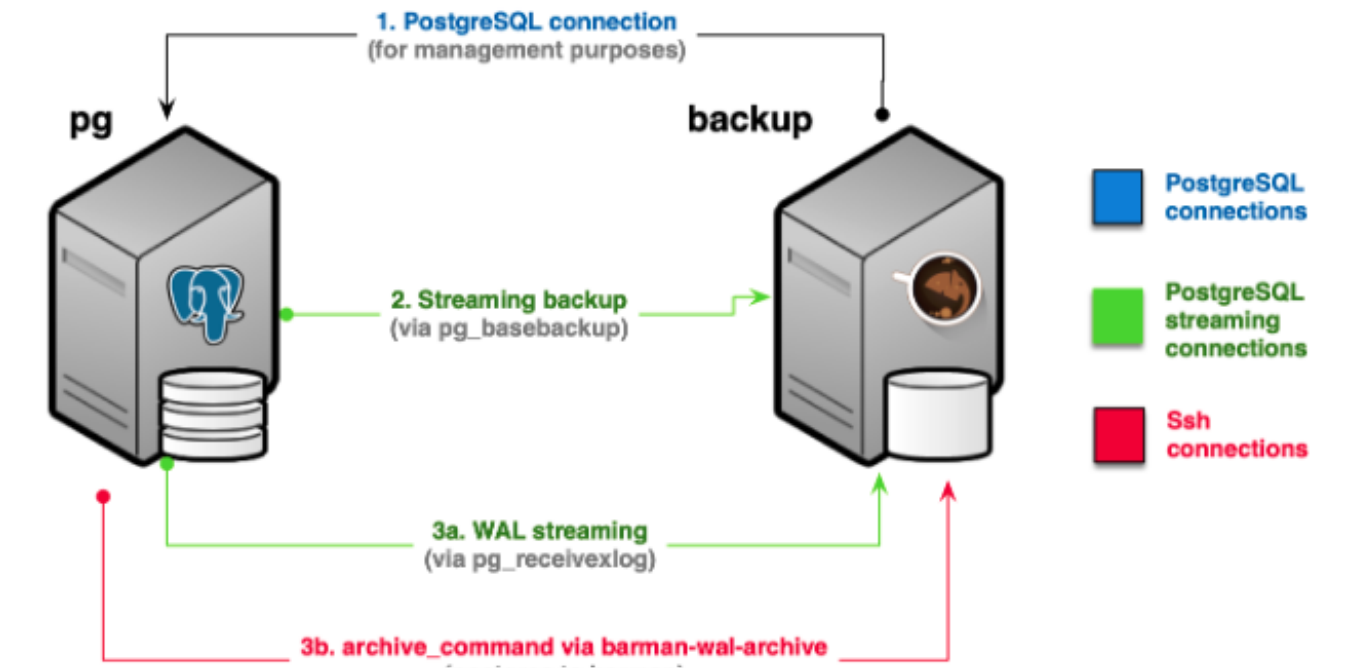
19 stories · 1378 saves



Rabinarayan Patra

Should You Use UUIDs for Database Keys?

UUIDs offer uniqueness in databases but come with performance costs. Learn when to use them, their drawbacks, and alternatives loved by...




Sheikh Wasiu Al Hasib

Understanding WAL Archiving and WAL Streaming in PostgreSQL

WAL (Write-Ahead Logging) is a critical feature in PostgreSQL that ensures data integrity and enables point-in-time recovery (PITR).

Jul 13 53

Server	Role	IP Address
Node1	Primary (Read/Write)	10.5.85.96
Node2	Standby (Read)	10.5.85.97
Node3	Standby (Read)	10.5.85.98
etcd1	etcd Cluster	10.5.85.100
etcd2	etcd Cluster	10.5.85.101
etcd3	etcd Cluster	10.5.85.102
HAProxy	Master	10.5.85.200
HAProxy	Backup	10.5.85.201
PgBackRest	Backup	10.5.85.205
VIP	Virtual IP	10.5.85.80


Yasemin Büşra Karakaş

PostgreSQL with Patroni: Installation and Configuration

PostgreSQL is a powerful and flexible open-source database system widely used around the world. However, to ensure business continuity and...

Aug 26 4

 Rufat Khaslarov in JavaScript in Plain English

9 Naming Tricks Every Developer Should Know to Avoid Headache

Code with poor naming is difficult to read, and hard-to-read code is like a ticking time bomb. Sooner or later, you'll have to revisit it...

Sep 16 1.5K 24

See more recommendations