

Open in app ↗

Sign up

Sign in

Medium

🔍 Search



Mastering PostgreSQL Logical Replication: Your Definitive Guide for Seamless Upgrades and Zero-Downtime Migrations

12 min read · Jun 5, 2025



Fahad Khalid

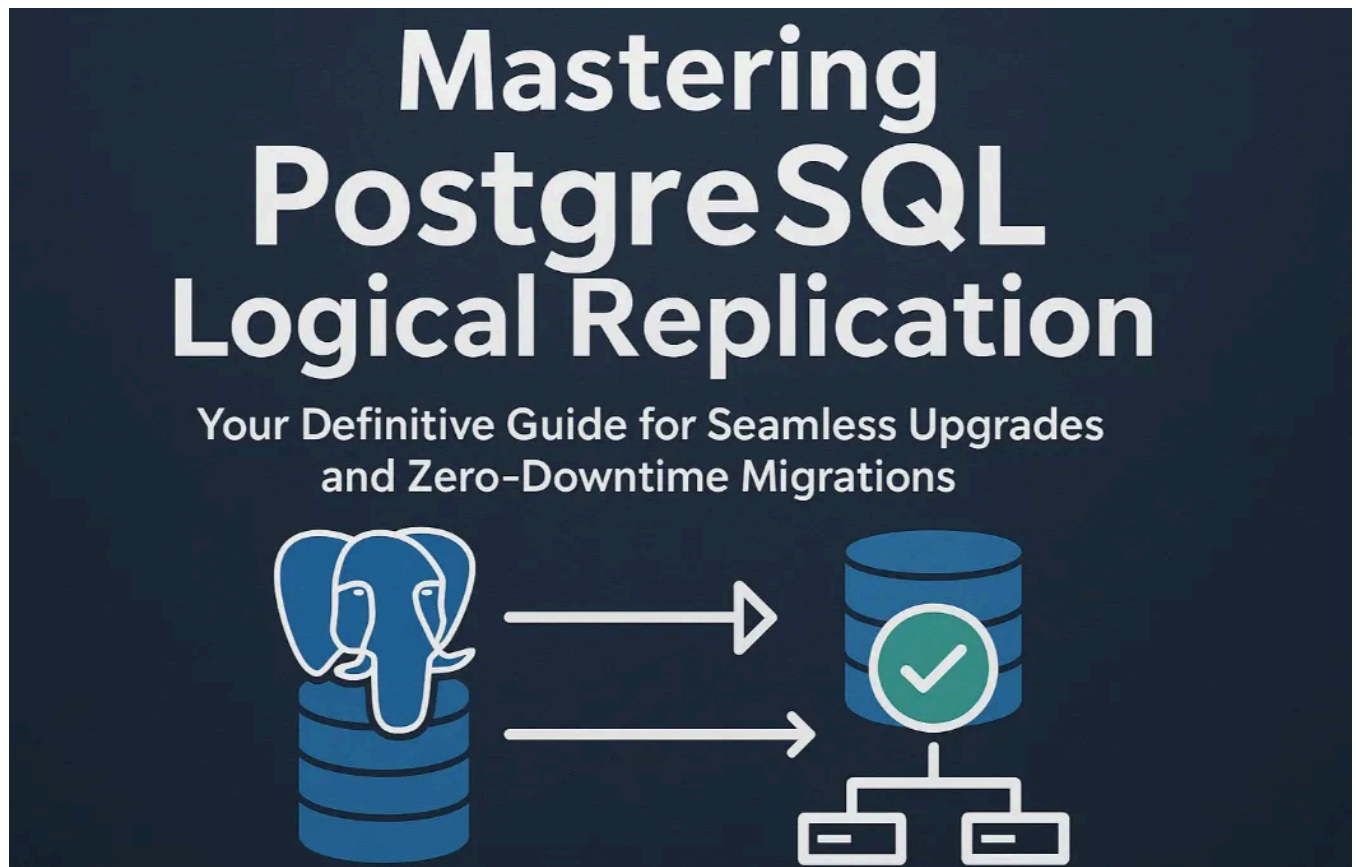
Follow



Listen



Share



As a Database Administrator (DBA) or Database Reliability Engineer (SRE), you're constantly seeking robust strategies to manage PostgreSQL instances. From critical version upgrades to intricate schema changes, the goal is often to perform these operations on a replica, rigorously test, and then seamlessly promote it to become

the new primary. This approach minimizes downtime and risk, making replication a cornerstone of modern database management.

PostgreSQL offers two primary replication methods: physical and logical. While physical replication provides a byte-for-byte, block-level copy of your entire database cluster — ideal for disaster recovery and hot standby scenarios — **logical replication** offers unparalleled flexibility. It allows for selective replication of data, enabling use cases like multi-version upgrades, partial database migrations, and even replicating data between different major PostgreSQL versions or entirely different database systems (via extensions).

In this comprehensive guide, we'll peel back the layers of PostgreSQL logical replication, walking you through a meticulous step-by-step setup process. We'll also arm you with critical insights, common pitfalls, and powerful monitoring techniques to ensure your replication journey is a resounding success.

When Logical Replication Shines: Use Cases & Why It Matters

Logical replication, despite its nuances, often emerges as the superior choice for scenarios such as:

- **Zero-Downtime Major Version Upgrades:** This is a killer feature. Replicate data from an older PostgreSQL version to a newer one, then switch applications to the new instance with minimal interruption.
- **Selective Data Migration:** Need to move only a few tables or schemas to a new environment? Logical replication offers precise control.
- **Consolidation or Sharding:** Replicate data from multiple sources into a central data warehouse or distribute data to different shards.
- **Cross-Database Replication:** While core PostgreSQL logical replication works within PostgreSQL, it lays the groundwork for tools and extensions that can replicate to other database types.

Understanding the Nuances: Unpacking Logical Replication Limitations

Before you dive in, it's paramount to grasp the inherent limitations of logical replication. Being fully aware of these will guide your planning and help you implement necessary workarounds.

Here's a detailed breakdown:

- **No DDL Replication (Schema Changes):** This is the most significant limitation. `CREATE TABLE`, `ALTER TABLE` (adding/dropping columns, changing data types), `CREATE SCHEMA`, `CREATE INDEX`, `DROP TABLE`, `TRUNCATE` (unless explicitly published) – these **Data Definition Language (DDL)** statements are **not** replicated. You must manually apply schema changes on both the publisher and subscriber databases in a synchronized manner.
- **Sequence Inconsistencies:** `NEXTVAL` operations on sequence objects are not synchronized. If your application heavily relies on sequences for unique identifiers, you'll need a separate strategy (e.g., manual synchronization, adjusting `SETVAL` after cutover) to maintain consistency.
- **Large Object (BLOB/CLOB) Handling:** Creation or modification of large objects are not replicated. If your application stores large binary data directly within PostgreSQL, this is a critical consideration.
- **Materialized View Refresh:** Materialized views are not automatically refreshed on the subscriber. You'll need to set up a mechanism (e.g., scheduled refresh scripts) to keep them up-to-date.
- **Primary Key/Replica Identity Requirement for DML:** This is non-negotiable for `UPDATE` and `DELETE` operations. Tables being replicated logically **must** have a `PRIMARY KEY` or a `REPLICA IDENTITY` set. Without it, PostgreSQL cannot reliably identify which rows to update or delete on the subscriber, leading to replication failures.
- **No Other Database Object Replication:** Beyond table data, logical replication does not replicate other database objects such as roles, users, functions, stored procedures, triggers, views, or tablespaces. These must be managed and synchronized independently.
- **No Conflict Resolution:** Logical replication operates on a “last-writer-wins” or “no-conflict-expected” model. It does not inherently resolve conflicts that may arise due to concurrent writes on both the primary and the replica (e.g., if applications write to the subscriber before it becomes primary). Conflict management has to be handled externally, either through careful application design or custom conflict resolution logic.
- **Increased Primary Load:** Logical replication can introduce additional load on the primary database because it needs to transform WAL (Write-Ahead Log)

records into a logical change format, which can be resource-intensive.

- **Disk Space Usage on Primary During Downtime:** In cases of subscriber downtime or significant replication lag, the primary server uses **replication slots** to keep subscribers in sync. This means the primary needs to retain WAL logs until they are confirmed to be received and processed by all subscribers. If a subscriber is down for an extended period, this can lead to increased disk space usage on the primary, potentially filling up the disk and halting writes.
- **No Replication of TRUNCATE ALL TABLES:** A `TRUNCATE` command on a table is replicated, but `TRUNCATE TABLE tab1, tab2;` OR `TRUNCATE TABLE tab1, tab2 CASCADE;` is not. You'd need to truncate tables individually if this is part of your workflow.

Preprocessing for a Flawless Logical Replication Setup

A successful logical replication setup hinges on meticulous preparation. Don't skip these crucial preprocessing steps on your **publisher (source) database**.

1. Verifying and Setting Replica Identity for Every Table

For logical replication to reliably identify `UPDATE` and `DELETE` operations, every table on the publisher database targeted for replication **must** have a `REPLICA IDENTITY`.

- **Primary Key:** The gold standard. If a table has a primary key, its `REPLICA IDENTITY` is automatically set to `DEFAULT`.
- **Unique Constraint:** A unique key constraint can also serve as a replica identity.
- **FULL :** All columns are considered for identifying rows. This is generally inefficient due to the overhead of comparing all columns and should be a last resort.
- **USING INDEX :** A specific unique index can be designated.

Strong Recommendation: If a table lacks a primary key, it is highly advisable to add one or at least a unique key constraint. This significantly improves the efficiency and reliability of logical replication's ability to track changes.

Use the following SQL query on your **publisher database** to audit your tables:

```

SELECT
    n.nspname AS schema_name,
    c.relname AS table_name,
    CASE c.relreplident
        WHEN 'd' THEN 'default (primary key or unique index, if any)'
        WHEN 'n' THEN 'nothing'
        WHEN 'f' THEN 'full (all columns)'
        WHEN 'i' THEN 'index (based on a user-chosen index)'
        ELSE 'unknown'
    END AS replica_identity,
    CASE
        WHEN EXISTS (
            SELECT 1
            FROM pg_constraint AS con
            WHERE con.conrelid = c.oid
            AND con.contype IN ('p', 'u')
        ) THEN 'Yes'
        ELSE 'No'
    END AS has_pk_or_uk
FROM
    pg_class AS c
JOIN
    pg_namespace AS n ON n.oid = c.relnamespace
WHERE
    c.relkind = 'r' -- 'r' for regular tables
    AND n.nspname NOT IN ('pg_catalog', 'information_schema', 'pg_toast') -- Ex
ORDER BY
    n.nspname, c.relname;

```

Address any tables showing `replica_identity = 'nothing'` or `has_pk_or_uk = 'No'` if they are critical for replication.

2. Tuning for Initial Data Copy Speed (Subscriber Side)

When dealing with large databases, the initial data copy (table synchronization) can be time-consuming. PostgreSQL provides GUC (Grand Unified Configuration) parameters primarily on the **subscriber side** to accelerate this process:

- **max_logical_replication_workers** : Specifies the maximum number of logical replication workers. This includes both *apply workers* (which apply changes on the subscriber) and *table synchronization workers*.
- **max_sync_workers_per_subscription** : This parameter specifically controls the number of tables that are synchronized in parallel per subscription. Increasing

it affects parallel table synchronization, not the number of workers per individual table.

To enhance the initial synchronization speed of tables, increase these values on the subscriber side.

Important constraints:

- `max_logical_replication_workers` should not exceed `max_worker_processes`.
- `max_sync_workers_per_subscription` should be less than or equal to `max_logical_replication_workers`.

Example `postgresql.conf` on Subscriber:

```
# postgresql.conf (Subscriber)
max_logical_replication_workers = 10 # Or more, <= max_worker_processes
max_sync_workers_per_subscription = 5 # Or more, <= max_logical_replication_workers
max_worker_processes = 20           # Ensure this is sufficient for all workers
```

Strategy for Very Large Tables: For massive tables, consider segmenting them into separate publications. For example, one publication for extremely large tables and another for smaller, more numerous tables. This allows for more granular control and potentially different worker allocations if needed.

3. Managing WAL Retention with `max_slot_wal_keep_size`

On the **publisher**, configure `max_slot_wal_keep_size`. This parameter dictates the maximum amount of WAL (Write-Ahead Log) files to retain for replication slots, even if subscribers fall behind.

- A value of `-1` (default) means WAL files are kept indefinitely as long as replication slots are active, which can lead to disk exhaustion if a subscriber is down for too long.
- Set a reasonable, finite value (e.g., `2GB`, `10GB`) to prevent your publisher's disk from filling up during prolonged subscriber downtime. This acts as a safety net,

though it means if a subscriber falls behind by more than this limit, its replication slot will become invalid, requiring a full resynchronization.

Example `postgresql.conf` on Publisher:

```
# postgresql.conf (Publisher)
max_slot_wal_keep_size = 5GB # Adjust based on expected downtime and WAL genera
```

4. Critical `postgresql.conf` Settings for Publisher

Ensure these parameters are correctly configured on your **publisher database**.

- **`wal_level = logical`** : Absolutely essential. This enables the generation of logical decoding information in the WAL, which is what logical replication uses.
- **`max_worker_processes`** : Set high enough to accommodate all background workers, including replication. A good rule of thumb: `max_worker_processes >= max_logical_replication_workers (on subscriber) + 1` (or more, depending on other background tasks).
- **`max_wal_senders`** : The maximum number of concurrent connections from standby servers or logical replication subscribers. It should be greater than or equal to `max_replication_slots` plus any physical replicas you might have.
- **`max_replication_slots`** : The maximum number of replication slots that can be active at the same time. This should be at least equal to the number of subscriptions you plan to have, plus any additional slots for table synchronization workers if you divide publications.

Example `postgresql.conf` (Publisher):

```
# postgresql.conf (Publisher)
wal_level = logical
max_worker_processes = 25           # Example: > max_logical_replication_worker
max_wal_senders = 13                # Example: > max_replication_slots (10) + phy
max_replication_slots = 30          # Example: #subscriptions + potential table
```

After changing any `postgresql.conf` parameters, remember to restart the PostgreSQL service on both publisher and subscriber for changes to take effect.

5. Identifying Invalid Indexes and Unique Constraints

Invalid indexes or unique constraints on the publisher can cause issues, especially if they are used as `REPLICA IDENTITY`. While data might copy initially, refreshes or specific DML operations on tables with invalid replica identity indexes can lead to errors or unexpected full table copies.

Run this query on your **publisher database** to identify invalid indexes:

```
SELECT
    n.nspname AS schema_name,
    c.relname AS index_name,
    t.relname AS table_name
FROM
    pg_index i
JOIN
    pg_class c ON c.oid = i.indexrelid
JOIN
    pg_class t ON t.oid = i.indrelid
JOIN
    pg_namespace n ON n.oid = c.relnamespace
WHERE
    NOT i.indisvalid
    AND n.nspname NOT IN ('pg_catalog', 'information_schema', 'pg_toast'); -- E
```

You should either fix these invalid indexes (e.g., `REINDEX CONCURRENTLY`) or consider setting the `REPLICA IDENTITY` to `FULL` for the affected tables as a temporary workaround, understanding the performance implications.

6. Creating Your Publication(s)

A **publication** on the publisher defines the set of data changes you want to replicate.

- You can create publications for all tables, specific tables, or even tables within certain schemas.
- For version upgrades, replicating the entire database (excluding system schemas) is common.

Example Publication Creation (on Publisher):


```
-- Replicate all tables in specific schemas
CREATE PUBLICATION my_app_data_pub FOR TABLES IN SCHEMA public, audit, booking,
```

```
-- OR, to replicate selected tables from anywhere
-- CREATE PUBLICATION my_selected_tables_pub FOR TABLE
my_schema.table1, another_schema.table2;

-- OR, for specific DML operations (less common for full migration)
-- CREATE PUBLICATION my_inserts_only_pub FOR TABLE my_table WITH
(publish = 'insert');
```

Avoid replicating system schemas like `pg_catalog`, `information_schema`, `pg_toast`, `pg_statistic`, `pg_replication_origin`, etc. They contain internal database metadata and are not meant for logical replication.

7. Creating a Dedicated Replication User

On the **publisher**, create a dedicated PostgreSQL user for replication. This user will be used by the subscriber to connect and fetch data. Grant it the `REPLICATION` privilege.

```
CREATE USER rpl_user WITH REPLICATION ENCRYPTED PASSWORD 'your_very_strong_pass
-- Optionally, grant read access to the specific tables if not using FOR ALL TA
-- GRANT SELECT ON ALL TABLES IN SCHEMA public TO rpl_user;
-- GRANT SELECT ON ALL TABLES IN SCHEMA audit TO rpl_user;
-- ... and so on for other schemas in the publication
```

Then, configure `pg_hba.conf` on the publisher to allow this user to connect from the subscriber's IP address:

```
# pg_hba.conf (Publisher)
host    replication    rpl_user    <subscriber_ip_address>/32    md5
```

Replace `<subscriber_ip_address>` with the actual IP address of your subscriber server. Reload `pg_hba.conf` (e.g., `pg_ctl reload` or `SELECT pg_reload_conf();`).

The Grand Setup: Establishing Logical Replication

Assuming you've already provisioned a new PostgreSQL instance for your subscriber with the same PostgreSQL version and similar configurations (hardware, OS, etc.) as your publisher, we can now proceed with the actual logical replication setup.

Step 1: Create the Exact Database on Subscriber

The database on the subscriber must have the exact same name as the publisher database.

```
CREATE DATABASE parkdepot; -- Use your actual database name
```

Step 2: Extract Global Objects (Roles, Tablespaces) from Publisher

Global objects like roles and tablespaces are not replicated by logical replication. You must dump and restore them separately.

```
pg_dumpall -h <publisher_endpoint> -p 5432 -U <admin_user> --globals-only > "gl
```

Step 3: Extract Only the Schema (Structure) from Publisher

Logical replication only copies data. The schema (tables, indexes, constraints, functions, views, etc.) must exist on the subscriber beforehand.

```
pg_dump -h <publisher_IP> -U <admin_user> -d <database_name> --schema-only > sc
```

Step 4: Migrate/Install Extensions (If Applicable)

If your database uses extensions (e.g., `pg_partman`, `PostGIS`, `timescaledb`), ensure they are installed and configured on the subscriber. For version-sensitive extensions like `pg_partman`, you might need to copy specific SQL files from the publisher's extension directory (`/usr/share/postgresql/<version>/extension/`) to the subscriber's, then modify the `pg_partman.control` file on the subscriber to match the correct default version.

Step 5: Import Global Objects and Schema to Subscriber

Now, restore the extracted global objects and schema to your subscriber instance.

```
psql -U <admin_user> -h <subscriber_IP> -p 5432 -d postgres -f globals-roles-tables.sql
psql -h <subscriber_IP> -U <admin_user> -d <database_name> -f schema.sql
```

Step 6: Configure Extensions on Subscriber (e.g., pg_partman)

If you have extensions that require per-table or per-database configuration (like pg_partman's create_parent()), you'll need to apply these configurations on the subscriber.

```
-- Example for pg_partman
SELECT partman.create_parent('public.my_partitioned_table', 'column_name', 'range', 'auto', '1000', '1000');
-- Repeat for all partitioned tables
```

Step 7: Handle Sequences (Crucial for Application Cutover)

As NEXTVAL operations are not replicated, sequences on the subscriber will have their last_value at zero (or their start_value). Before cutting over application traffic, you **must** synchronize sequence values. A common strategy is to set them significantly higher than the current last_value from the publisher to avoid conflicts.

You can generate the SETVAL queries from your publisher like this:

```
SELECT
  CONCAT(
    'SELECT pg_catalog.SETVAL(''',
    schemaname,
    '."',
    sequencename,
    ''', ',
    CASE
      WHEN last_value IS NULL THEN
        start_value + 1000 -- Add a buffer if last_value is null
      ELSE
        last_value + 1000 -- Add a buffer (e.g., 1000 or more)
```

```

        END,
        ', true);'
    ) AS setval_query
FROM
    pg_sequences
WHERE
    schemaname NOT IN (
        'pg_catalog',
        'information_schema',
        '_timescaledb_catalog',
        '_timescaledb_config'
    )
ORDER BY
    sequencename ASC;

```

Run the generated `setval_query` results on your **subscriber** database just before the cutover. The `+ 1000` (or whatever buffer you choose) ensures that the subscriber's sequence values are sufficiently ahead, accommodating any inflight transactions during the final switch.

Step 8: Create the Subscription

This is the final step to initiate the logical replication process. On your **subscriber** database, create the subscription.

```

CREATE SUBSCRIPTION my_subscription
    CONNECTION 'host=<PublisherIP> port=5432 dbname=<database> user=rpl_user pa
    PUBLICATION my_app_data_pub;

```

Once executed, PostgreSQL will automatically start the initial data synchronization. For large tables, this might take some time, but logical replication will then seamlessly transition to streaming continuous changes.

Monitoring Your Logical Replication: The Lifeline

Setting up is only half the battle. Robust monitoring is crucial to ensure your replication is healthy and not falling behind.

On the Publisher (`pg_stat_replication_slots`):

```
SELECT slot_name, active, wal_status, wal_delay_lags_bytes, restart_lsn, confir
```

Pay close attention to `wal_delay_lags_bytes` (PostgreSQL 14+) or monitor `restart_lsn` and `confirmed_flush_lsn` to understand how far behind your subscriber is. If `active` is false, your subscriber is disconnected.

On the Subscriber (`pg_stat_subscription` and `pg_stat_subscription_workers`):

```
SELECT subname, subenabled, subconninfo, substate, sublag FROM pg_stat_subscrip
```

`sublag` (PostgreSQL 14+) is a great indicator of how far behind your subscriber is. `sync_state` for `pg_stat_subscription_workers` shows the initial table copy status (e.g., 'syncing', 'ready').

PostgreSQL Logs: Always check both publisher and subscriber logs (`postgresql.log`) for any errors, warnings, or detailed messages related to `logical replication`, `replication slots`, and `apply workers`. These are your first line of defense for troubleshooting.

When Things Go Wrong: Troubleshooting Common Logical Replication Failures

- **“ERROR: relation “tablename” does not have a primary key”:** This is a classic. Revisit “Preprocessing Step 1”. The table on the publisher needs a `PRIMARY KEY` or a `REPLICA IDENTITY` set.
- **Schema Mismatch Errors:** If you alter a table schema on the publisher (e.g., add a column) but forget to apply the same DDL on the subscriber, replication will halt. Always synchronize DDL manually.
- **Publisher Disk Full (`WAL Segment too large`):** If the subscriber is down or severely lagging, the publisher might retain too many WAL files. This points to `max_slot_wal_keep_size` being too high, or the subscriber needing attention.

- **Network Connectivity Issues:** Basic connectivity problems (`connection refused` , `timeout`) between publisher and subscriber. Check firewalls, `pg_hba.conf` , and network routes.
- **Resource Exhaustion (Subscriber):** If the subscriber is struggling to apply changes (e.g., high CPU, I/O, or low `max_logical_replication_workers`), replication lag will increase. Tune `max_logical_replication_workers` and `max_sync_workers_per_subscription` and ensure sufficient hardware.
- **Invalid Index/Constraint Errors:** As noted in “Preprocessing Step 5,” invalid indexes can cause issues during replication or table synchronization.
- **TRUNCATE operations on subscriber not replicating:** Remember `TRUNCATE TABLE tab1, tab2;` is not replicated. Truncate tables individually if this is part of your workflow.

Conclusion

PostgreSQL logical replication is an indispensable tool for DBAs and SREs. It unlocks capabilities like seamless major version upgrades, highly flexible data migrations, and advanced blue/green deployment strategies. While it requires a deeper understanding of its mechanics and limitations, the control and flexibility it offers are unmatched by physical replication for many modern use cases.

By diligently following the preprocessing steps, understanding the configuration parameters, and establishing robust monitoring, you can build a reliable and high-performance logical replication system. This empowers you to manage your PostgreSQL databases with confidence, ensuring high availability, minimal downtime, and smoother transitions for critical changes.

Embrace logical replication — your database operations will thank you for it!

[Postgresql](#)[Postgres](#)[Logical Replication](#)[Database](#)[Database Administration](#)

[Follow](#)

Written by Fahad Khalid

30 followers · 30 following

Experienced Database Reliability Engineer specializing in high availability, performance tuning, automation and robust backup strategies for seamless operations

Responses (1)



Write a response

What are your thoughts?



Fahad Khalid Author
Jun 5



The Grand Setup: Establishing Logical Replication

Before proceeding with this step, you may need to review the TCP settings: `tcp_keepalives_idle`, `tcp_keepalives_interval`, and `tcp_keepalives_count`.


In my case, I encountered an issue where the initial COPY process on the subscriber took a long time to...
[more](#)



1

[Reply](#)

More from Fahad Khalid


 Fahad Khalid

Tackling a 1.5TB Table: Near Zero-Downtime Partitioning in PostgreSQL Using `pg_partman`

As part of our ongoing mission to improve the reliability and scalability of our PostgreSQL database at Wemolo, I recently faced the...

Apr 14  79  1



 Fahad Khalid

Finding Home Away from Home: My First Steps at Wemolo

Relocating to Germany and starting my career at Wemolo has been a life-changing journey, filled with challenges, growth, and unforgettable...

Jan 16  54  1



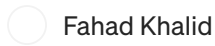
 Fahad Khalid

Mastering AWS Redis Cluster Management: Best Practices for Infrastructure Engineers

Image source: <https://www.betterteam.com/images/infrastructure-engineer-job-description>

Nov 9, 2023





Handling Large Log Streams—CloudWatch Logs Streams and Batch Processing


Ever wondered how to navigate the complexities of handling large streaming logs while seamlessly aggregating them for insightful analytics...

Jul 29, 2023 🖱 1



See all from Fahad Khalid

Recommended from Medium

 Abhinav

Docker Is Dead—And It's About Time

Docker changed the game when it launched in 2013, making containers accessible and turning “Dockerize it” into a developer catchphrase.

✦ Jun 9 🖱 413 💬 11



 Oz

Bash Profile & Aliases for Postgres

Managing PostgreSQL on Linux? You're probably juggling between dozens of commands, logs, and scripts. With a few tweaks to your...

✦ Jun 17 🖱 53



○ Jeyaram Ayyalusamy 🗨

PostgreSQL log_statement Explained: Complete Guide With Real-Time Examples

■ Understanding log_statement in PostgreSQL: Your Key to Query Visibility

Jun 16 🖱 3



○ Sohail Saifi

Postgres Hidden Features That Make MongoDB Completely Obsolete (From an Ex-NoSQL Evangelist)

For six years, I was that developer.

★ May 26 👏 187 💬 10




○ Himanshu Singour

Not kidding, One line of SQL brought down our query time from 3s to 300ms.

Let me share a quick story from our backend team.

Jun 5 👏 858 💬 16



 Ajaymaurya

How to Supercharge Your PostgreSQL with pgvector for Semantic Search

In today's data-driven world, delivering intelligent, context-aware search experiences is more crucial than ever. Traditional keyword-based...

★ May 1 🖱️ 11



See more recommendations