EDB
Postgres® for the AI Generation

Beyond PostgreSQL 17: 7 DBA Workarounds for Enhanced Management

Vibhor Kumar,
Customer Experience Technical Fellow
10/01/2024

EDB
Postgres for the AI Generation

Vibhor Kumar

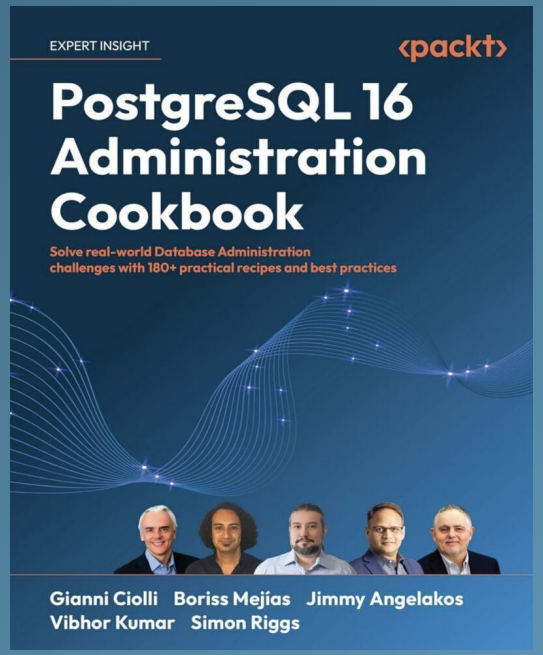
Customer Experience
Technical Fellow/Advisor

Authored tools -

- edb-ansible, postgres-deployment, pg_background, efm_extension, edb_user_login, edb-cloneschema, edb_block_commands…many more

Expertise:

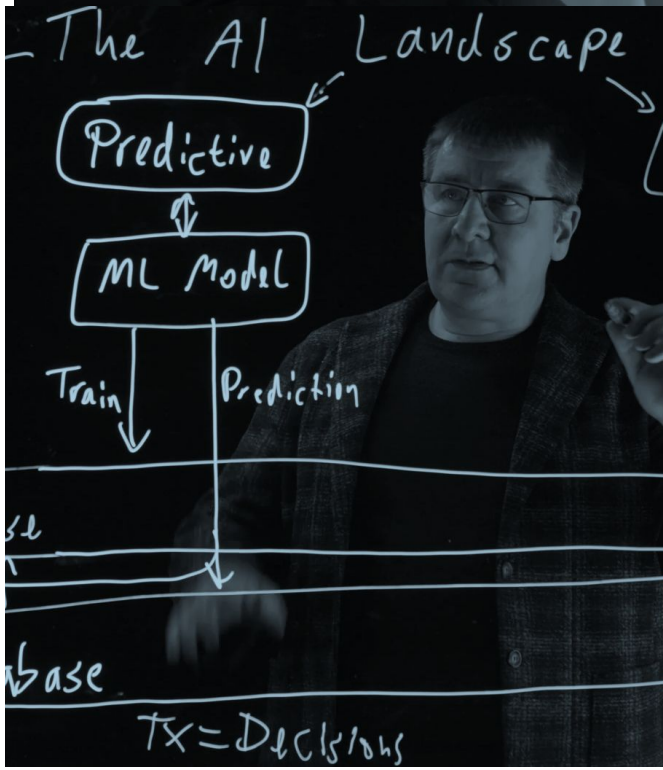
- Enterprise architecture, cloud technology, microservices, database technologies (Oracle, MySQL, PostgreSQL, DB2, EDB Postgres Advanced Server, MongoDB)
- Security best practices, DevOps, Oracle migration and transformation, database and platform performance, and data security and governance, and building team.



EDB
Postgres® for the AI Generation

Pride in Postgres

- Number one contributor to Postgres, the fastest-growing and most loved database in the world
  - 3 Core Team members, 7 Committers, 9 Major Contributors, 20 Contributors, #1 site for desktop downloads
- Nearly 800 employees
  - Over 50,000 Oracle Schema migrations done by our customers (including Financial, Defence, Health Care, etc.)
- EDB Postgres AI
  - The industry's first platform that can be deployed as cloud, software, or physical appliance
  - Secure, compliant, and enterprise-grade performance guaranteed

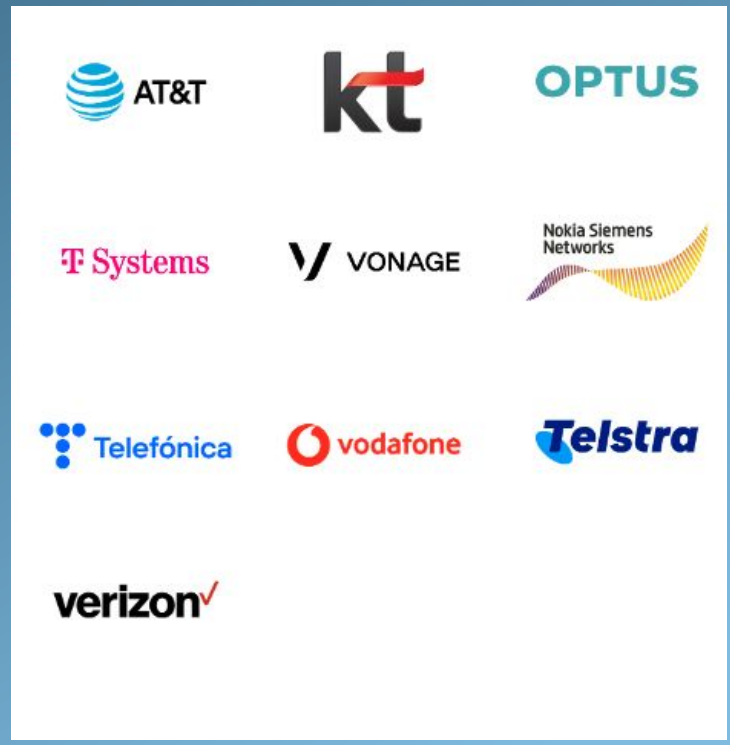


LEADING ENTERPRISES WITH COMPLEX, COMPLIANT, AND SCALING DATA NEEDS TRUST EDB

BFSI (BANKING, FINANCIAL SERVICES, AND INSURANCE)

TECHNOLOGY VENDORS

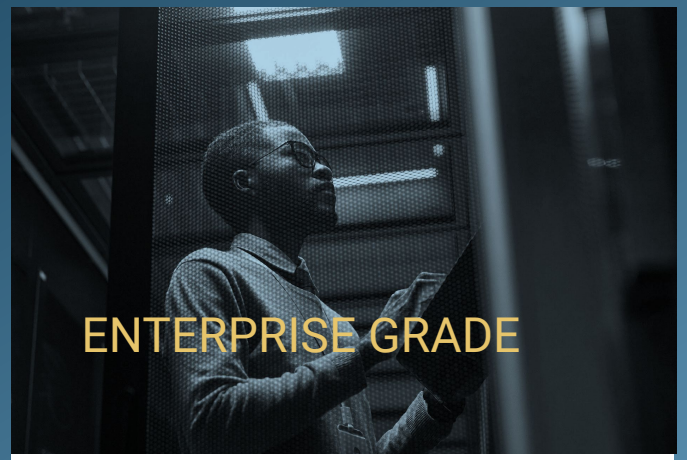
TELCO CSP



(60% of our Top 50)

What EDB Offer that is important

- Observability across your entire hybrid estate, whether is on premise, private or public cloud
- Create a **single source of truth in an organization across applications and data**



- Enterprise-grade reliability (five 9s), performance, security, and scalability
- Agility to develop new applications, including migrating from legacy infrastructure

- Delivery of innovation with AI and analytics while maintaining your enterprise-grade governance, reliability, security, and scalability requirements
- Around 600 AI use cases and initiatives in queue

EDB Postgres AI: the solution to modernize, optimize, and evolve

DO MORE WITH YOUR POSTGRES — WHEREVER YOUR DATA IS OR NEEDS TO BE

SOVEREIGN DATA AND AI PLATFORM

UNIFIED WORKLOAD MANAGEMENT

- TRANSACTIONAL
- ANALYTICAL
- ARTIFICIAL INTELLIGENCE

SINGLE-PANE-OF-GLASS ADMINISTRATION

- HYBRID DATA ESTATE
- INTELLIGENT OBSERVABILITY
- ENTERPRISE SECURITY

HYBRID AND MULTI-CLOUD DEPLOYMENT

- PUBLIC CLOUD (MANAGED)
- PRIVATE CLOUD (SOFTWARE)
- ON PREMISES (APPLIANCE)

PLATFORM TOOLS AND SERVICES

- MIGRATION PORTAL
- CONTINUOUS HIGH AVAILABILITY
- BACKUP AND RECOVERY

EXTENSIBILITY

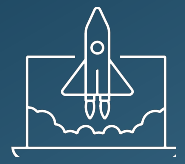
- CSP INTEGRATIONS
- DEVOPS TOOLING
- KUBERNETES TOOLING
- GENAI & LLM INTEGRATIONS
- LAKEHOUSE INTEGRATIONS

EDB Postgres AI

aws · Google Cloud · Azure · Alibaba Cloud

Red Hat · IBM · carahsoft · Supermicro · NUTANIX

EDB POSTGRES AI SOLVES DATA AND AI CHALLENGES AT SCALE FOR COMPLEX AND COMPLIANT INDUSTRIES



MODERNIZE

Modernize your data estate onto Postgres, enabling access to modern developers, features, and functionality all backed with the power of open source and enhanced by EDB's enterprise-grade capabilities

SUPPORT SCALE

High performance and reliability combined with observability and tuning combine to provide a data platform that seamlessly scales to meet the enterprise needs of your entire organization while maintaining full control of security, governance, and compliance

ENABLE INNOVATION

Integrated support across your entire Postgres estate for modern analytics and AI capabilities that enable your developers to deliver the next wave of innovation while maintaining enterprise-grade performance, reliability, security, governance, and compliance

EDB contributions to PG 17 (Released Last Thursday!)

Backup and Recovery

Faster Recovery Times with Incremental Backup

- Quickly recover from disasters with reduced downtime by only backing up what's changed, enabling more frequent backups of large databases, reducing recovery times in the event of a disaster.

Developer Productivity/Flexible PG

Finished SQL:2023 SQL/JSON

- Support for latest SQL/JSON Standards

JSON_TABLE

- Easily work with JSON data using a table like interface

Performance Enhancements

Reduced Memory for Part-wise JOINs

- Efficiently join large tables using less memory

NULL Constraint Improvements

- Better execution plans with NULL constraint handling

Business Logic and Replication

Helping EDB customers with Complex Business Logic

- Simplify complex logic with improved subtraction support

Convert Physical Replica to Logical Replica

- Easier to initialize logical replication for large datasets with pg_create_subscriber

7 DBA Workarounds for Enhanced Management

I. Zero-Downtime Rolling Major Upgrades

Using Logical Replication

Zero-Downtime Upgrade Essentials

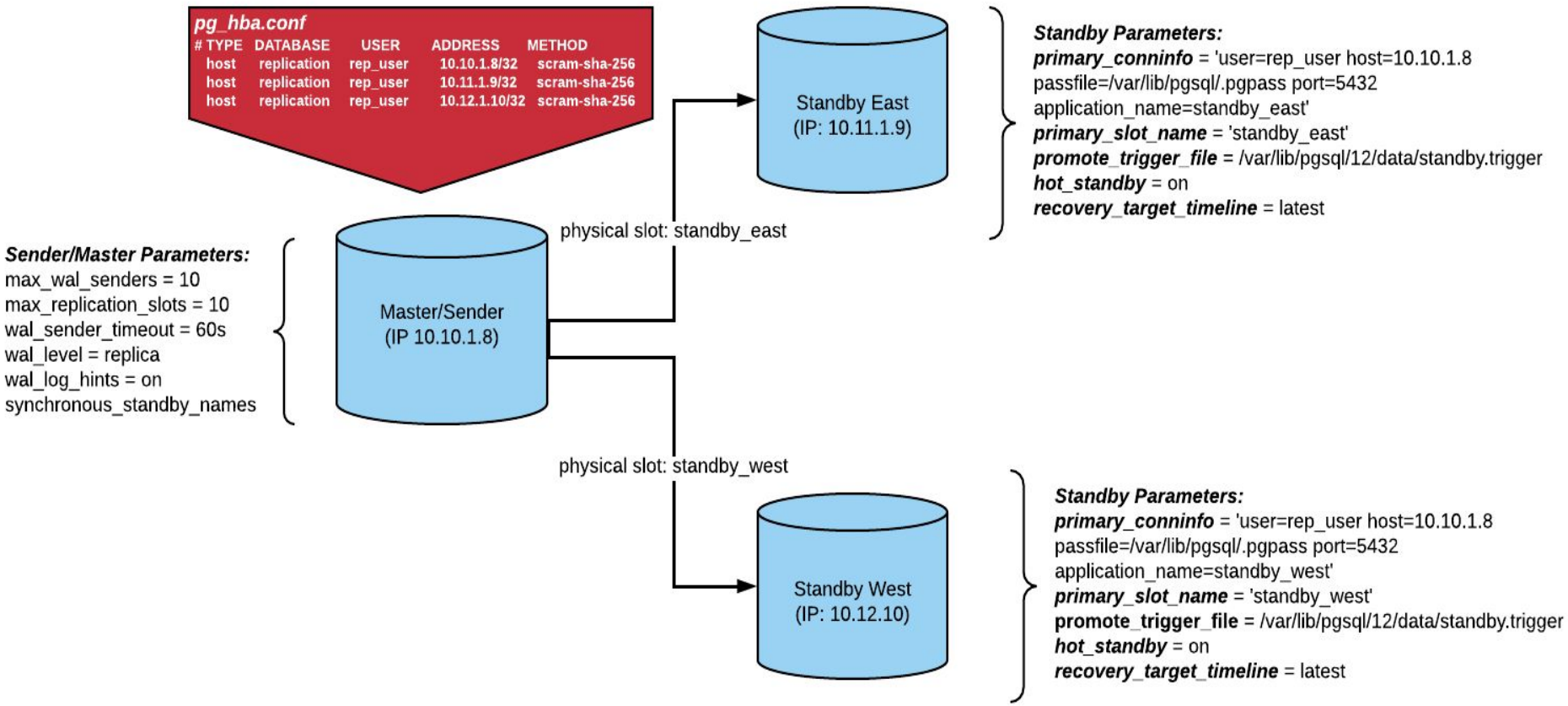
Key Requirements:

- **Near-zero downtime:** Measured in milliseconds or seconds.
- **No data loss:** Maintain complete data integrity and consistency.
- **Seamless transition:** Support schema changes and ongoing transactions.

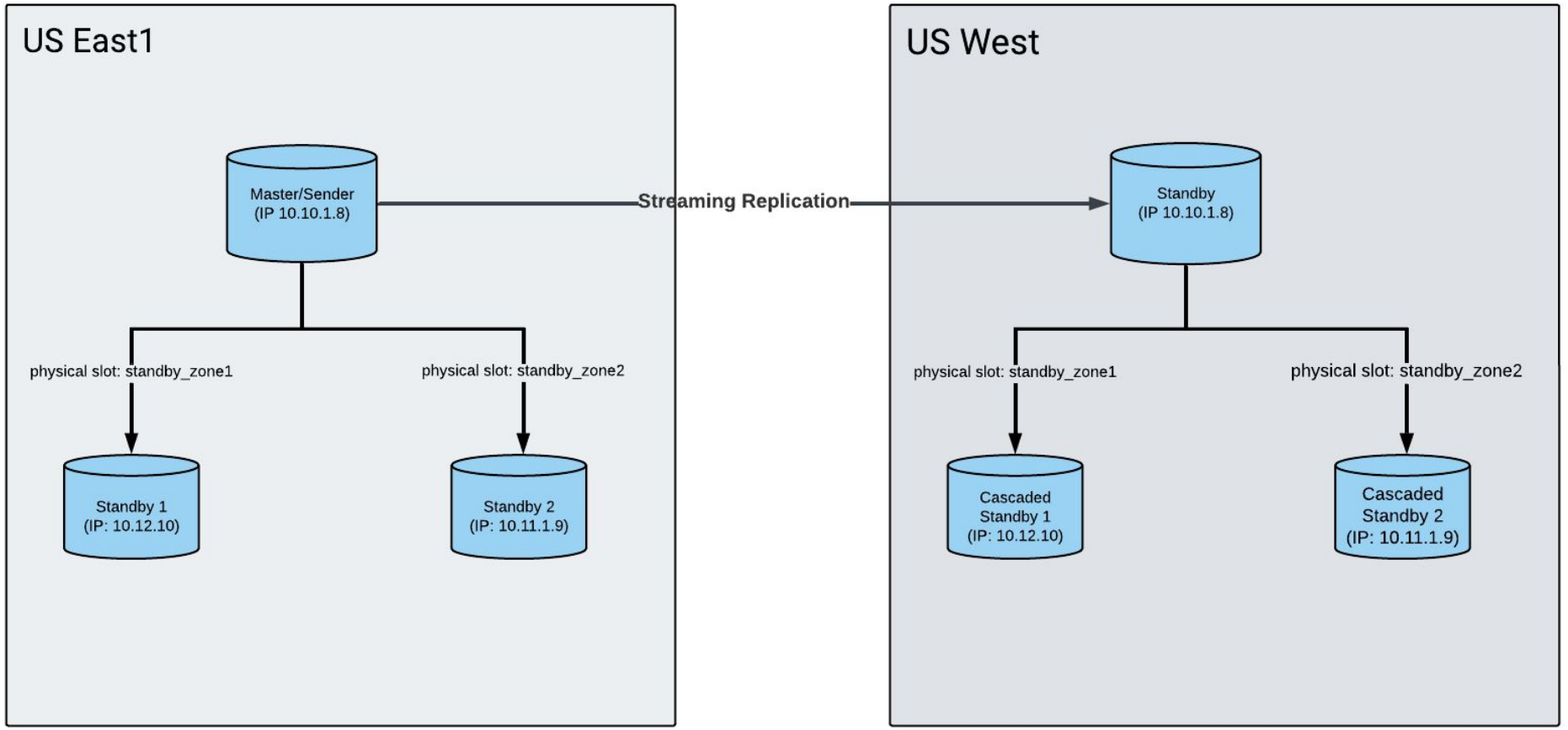
Post-Upgrade Essentials:

- **Rollback plan:** Ensure the ability to revert to the previous version if needed.
- **Continuous availability:** Minimize downtime during rollback.

Popular Production HA Architectures



Other Production HA Architectures



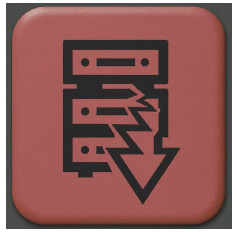
PostgreSQL Upgrade Options

pg_(dump|restore)

- Requires significant downtime.
- Time-consuming, especially for large databases.
- Rollback can be complex and risky.

pg_upgrade with Standby resync

- Reduces downtime compared to pg_dump/pg_restore.
- Still requires some downtime.
- Upgrade time varies depending on database size.
- Rollback complexity (even with disk snapshots) and potential data loss are concerns.




SMALL TITLE: LIGHTBOX FORMAT WITH NON-BINARY HERO AND COPY

Native Logical Replication - An Option For Major Upgrade

PostgreSQL 17 - Native Logical Replication

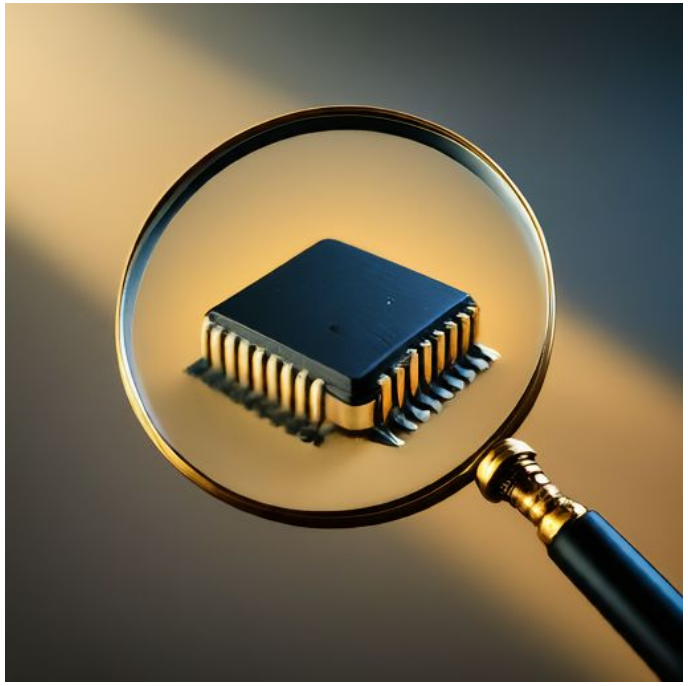
Key improvements in Native Logical replication

- **pg_upgrade preserves replication_slot :** You no longer need to drop and recreate logical replication slots when upgrading to a new major version (from 17 onwards). This significantly reduces downtime and simplifies the upgrade process.
- **Failover Control using failover slots:** Enhanced failover capabilities make logical replication more resilient in high-availability environments.
- **pg_createsubscriber Command-line Tool:** This new tool streamlines the process of converting a physical replica into a logical replica, making it easier to set up and manage logical replication

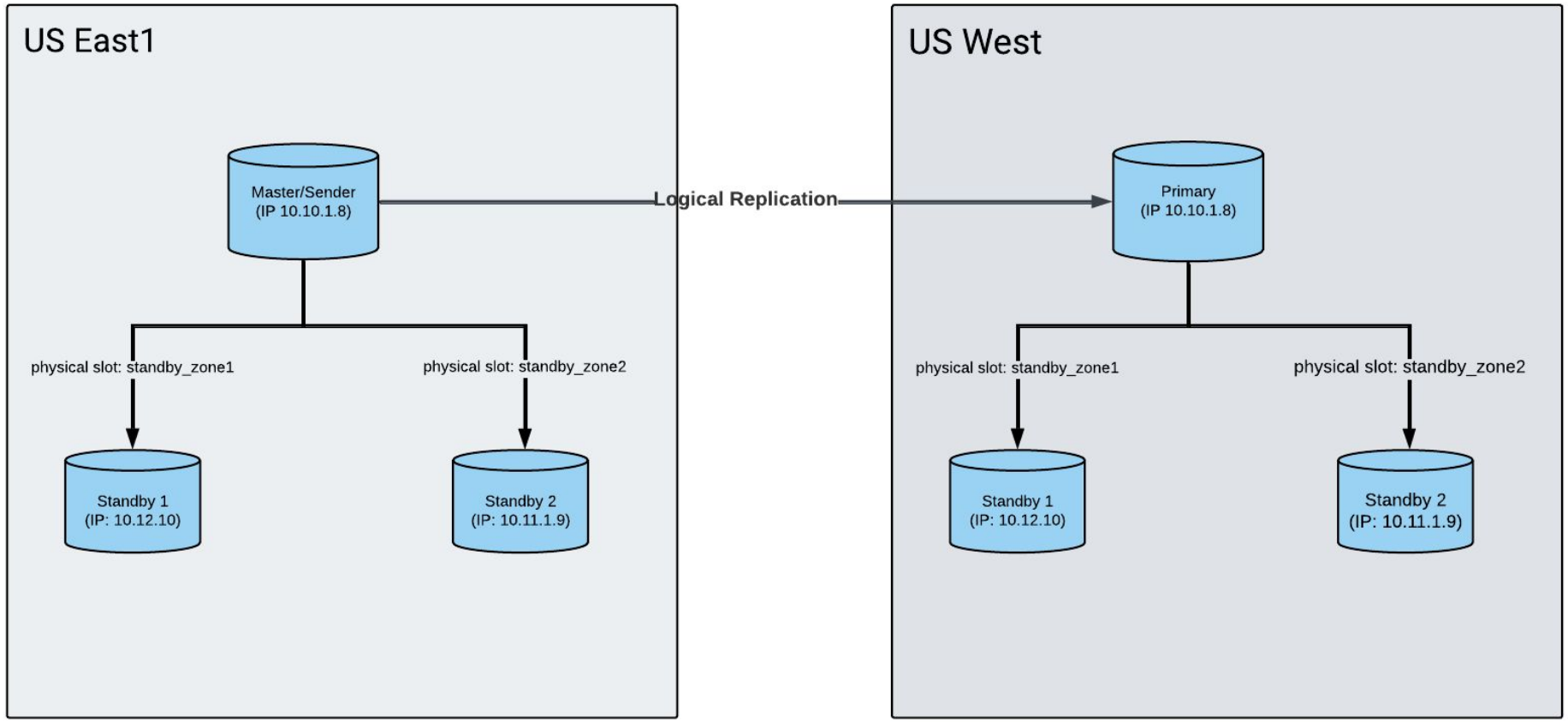
Native Logical Replication - Use Cases

Major use cases

- **Selective replication:** Microservices, data warehousing, mobile apps
- **Cross-platform flexibility:** Hybrid cloud, version upgrades, different OSs
- **Data integration:** Multi-tenant apps, data sharing, custom solutions
- **Real-time pipelines:** Change data capture, event-driven systems, analytics



Major Upgrades - Logical Replication Perspective



Logical Replication: Not a Silver Bullet

Schema Changes:

- New tables are not automatically replicated.
- DDL (Data Definition Language) commands and sequences are not replicated.
- DDL changes on the subscriber can disrupt replication.

Replication Management:

- Setting up and managing multiple publications and subscriptions for optimal performance can be complex.

Replication Direction:

- Logical replication is unidirectional, making rollback more challenging.

Post-Upgrade Reversal:

- Reversing replication after the upgrade and ensuring no data loss adds another layer of complexity.

Logical Replication: Not a Silver Bullet

Node Consistency:

- Requires careful monitoring of individual node states.
- Manual failover decisions based on node health and data consistency.

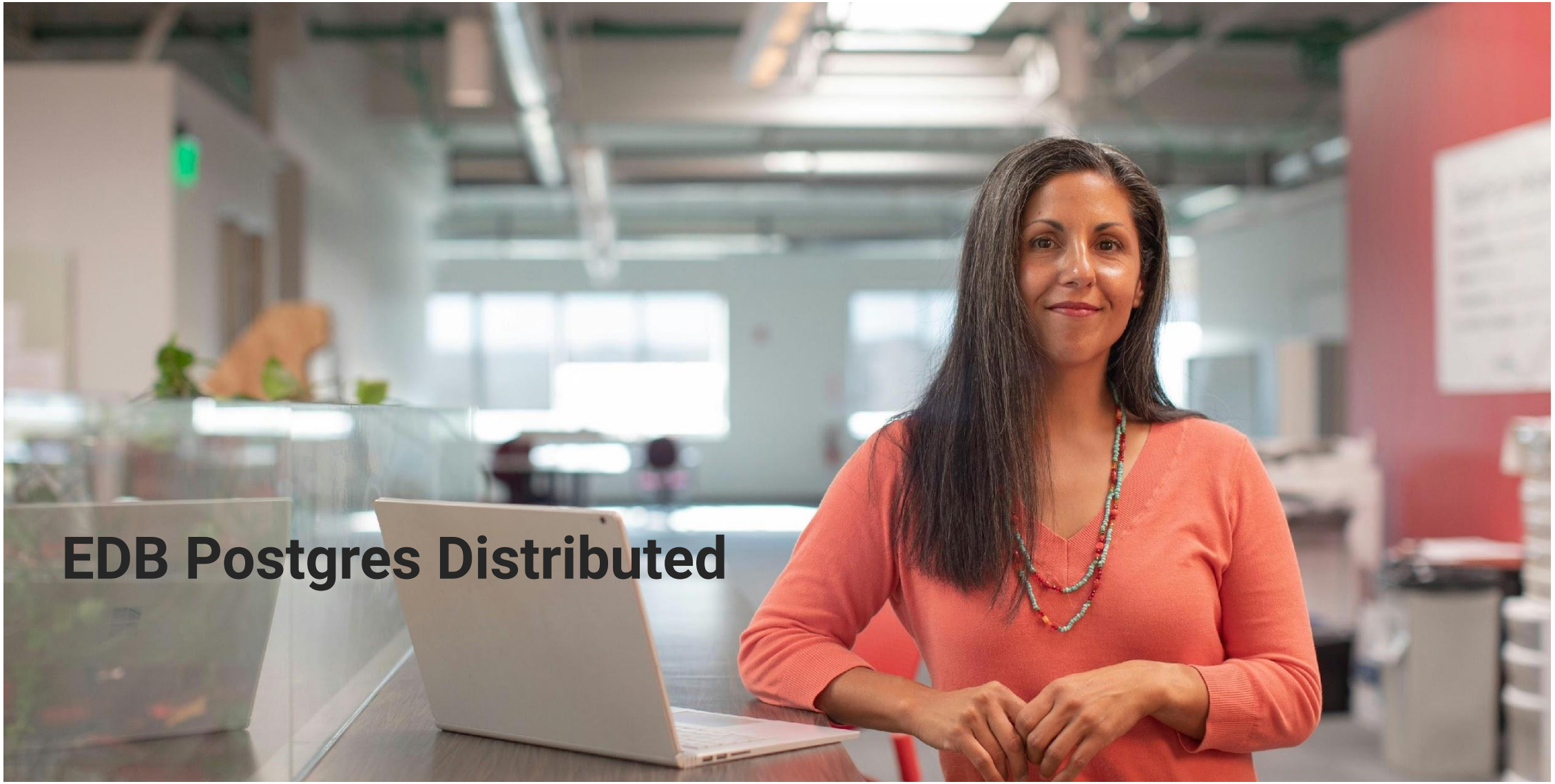
Connection Routing:

- Lack of automated connection routing based on a consensus layer.
- Potential for connection issues during the transition.

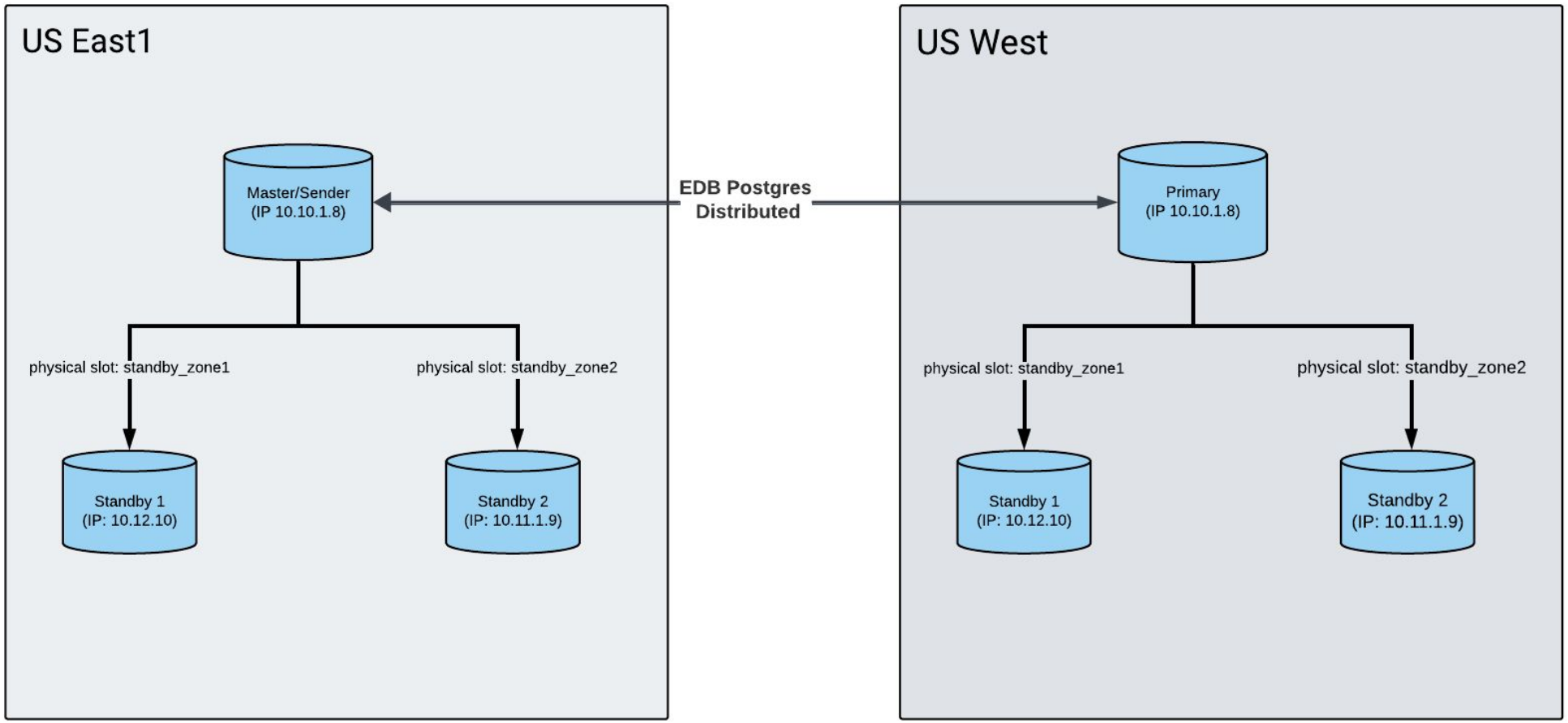
Conflict Resolution:

- Increased risk of data conflicts due to human error and open connections on the old system.
- Manual intervention needed to resolve conflicts.

SMALL TITLE: LIGHTBOX FORMAT WITH FEMALE HERO AND COPY



Major Upgrades - A PGD Perspective



EDB Postgres Distributed: A Silver Bullet

Schema Changes:

- New tables get automatically replicated.
- DDL (Data Definition Language) commands and sequences get replicated.
- DDL changes on the subscriber can get synchronized.

Replication Management:

- No need for setting up and managing multiple publications and subscriptions for optimal performance can be complex.

Replication Direction:

- Logical replication is bi-direction, making rollback more clean and easy.

Post-Upgrade Reversal:

- Simplified reversing replication after the upgrade and ensuring no data loss.

EDB Postgres Distributed: Silver Bullet

Node Consistency:

- PGD Proxy routes connections based on the consistency
- No manual failover decisions based on node health and data consistency. PGD Proxy takes care of it

Connection Routing:

- Automated connection routing based on a consensus layer using PGD Proxy
- Potential for connection issues during the transition.

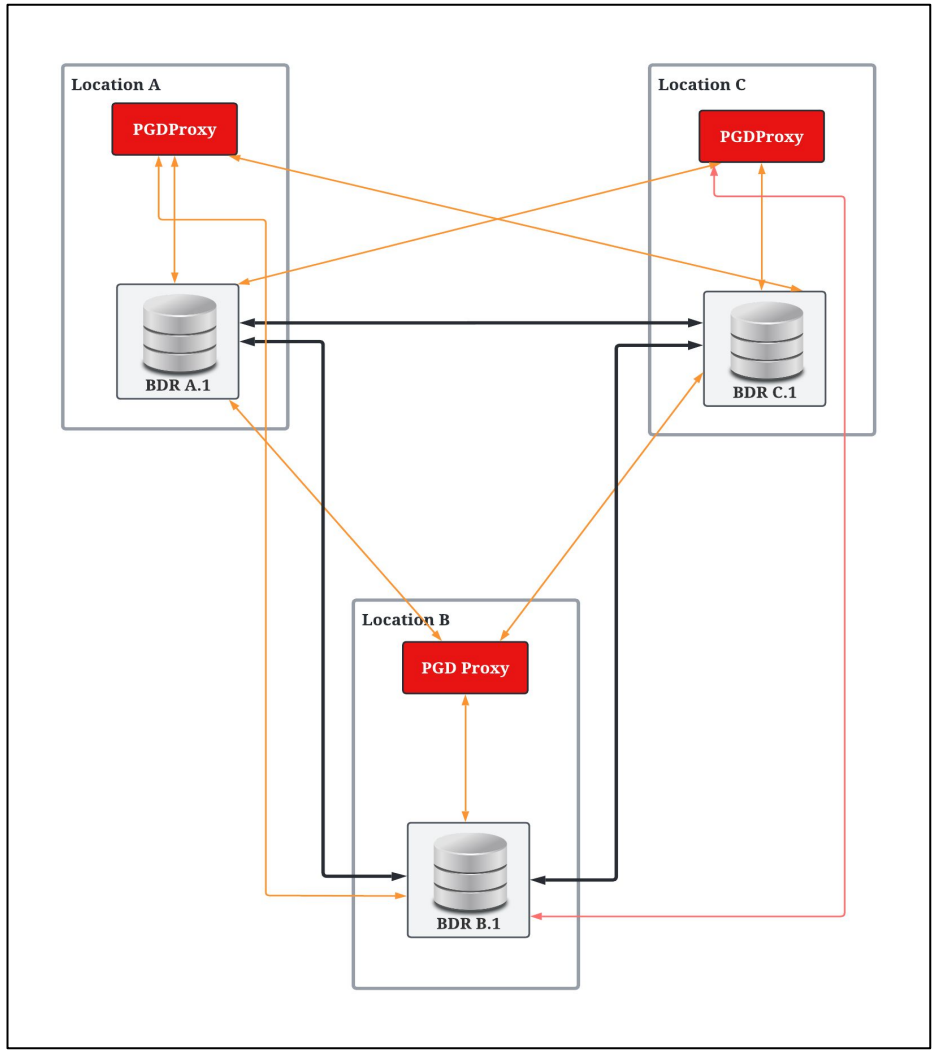
Conflict Resolution:

- Many methods are available for conflict resolution.
- Custom methods are allowed

EDB Postgres Distributed Architecture

Operational Advantages:

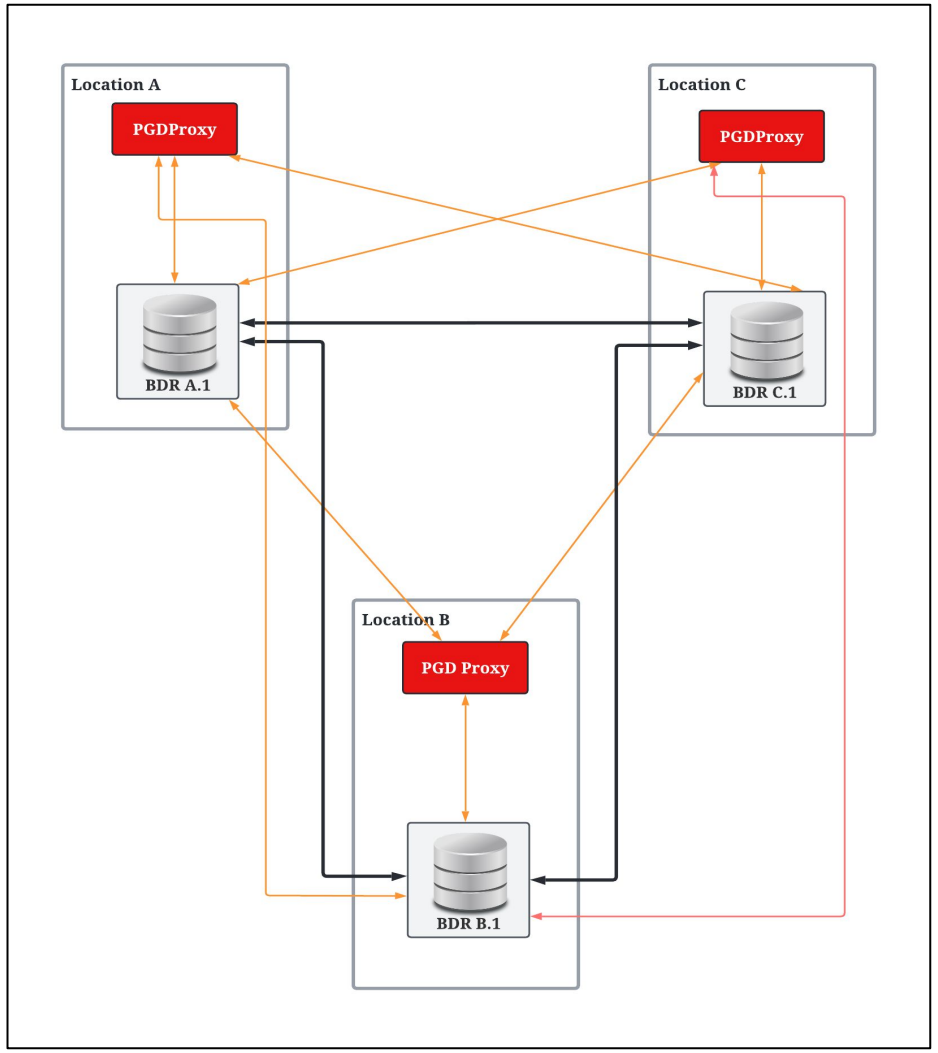
- **Rolling Upgrades/Patches:**
  - Upgrade or patch one node at a time for minimal downtime.
  - Built-in rollback options for easy recovery.
- **Simplified Maintenance:**
  - Perform VACUUM FULL and REINDEX operations on one node at a time without affecting overall availability.
- **Flexible Resource Management:**
  - Seamlessly move data to optimized storage with different IOPS using tablespaces.
  - Easily increase server resources (RAM/CPU) as needed.
  - Rebuild partitioned tables with zero downtime.



EDB Postgres Distributed Architecture

Advanced Capabilities:

- **Connection Routing:** Automatic and seamless connection routing for uninterrupted application access.
- **Load Balancing:** Distribute read traffic across nodes for improved performance.
- **Selective Data Replication:** Replicate only the necessary data to specific nodes.
- **Blue/Green Deployments:** Support for blue/green deployments for risk-free upgrades and testing.



EDB Postgres Distributed

Key Benefits

Eliminate downtime for tier 1 and global apps

Run Postgres in active/active geo-distributed clusters and provide up to 99.999% HA, with 5X throughput efficiency for apps that can't go down.

Deploy continuous HA Postgres with flexibility

Support geo-distributed apps regardless of where data is hosted: on-premises, on any cloud, on Kubernetes, and with hybrid and multi-cloud support.

Build robust, globally distributed Postgres apps

Process thousands of transactions per second, without a hitch, to over-deliver on your user expectations. Meet regional data sovereignty requirements.

EDB Postgres Distributed

Key Features

Hybrid, multi-cloud flexibility

PGD is available for PostgreSQL, EDB Postgres Advanced Server (EPAS), and EDB Postgres Extended (PGE), for both self-managed and DBaaS deployments, and for Kubernetes environments.

Unplanned outages protection

PGD uses an active/active architecture, conflict resolution via Raft-based consensus, and data loss protection to ensure apps and data are available, where and when they're needed. Avoid application impact even during maintenance windows and version upgrades. Achieve high resilience architectures with automated failover across sites and regions.

High availability distributed clusters in the cloud

Running PGD on EDB Postgres AI Cloud Service supports high availability active/active geo-distributed deployments. Leverage PGD to build robust, globally distributed applications that process thousands of transactions per second, with up to 99.999% availability and 5X throughput support versus native logical replication.

Simplified regulatory compliance

PGD allows you to implement controls in multi-region clusters to replicate data selectively where necessary, easing compliance with regulations including SOC2, GDPR, and PCI DSS, and helping to meet regional data sovereignty requirements.

II. Listing Objects Dependent on Procedures/Functions

Using plpgsql_check

Why DBAs Need to Understand Dependencies

Validate Code Integrity:

- Analyze dependency lists to verify the validity and completeness of functions/procedures.
- Identify potential issues caused by missing or invalid dependencies.

Assess Change Impact:

- Measure the impact of dropping objects on dependent code.
- Evaluate the effects of modifications to existing objects and their dependencies.

Understand Relationships:

- Visualize the interdependencies between procedures and functions.
- Identify potential cascading failures or performance bottlenecks.

Proactive Risk Mitigation:

- Pinpoint potential failure points in the application logic.
- Implement safeguards to prevent disruptions caused by object changes.

plpgsql_check: Ensuring Code Integrity

Installation:

- Install the `plpgsql_check` extension:
  ```
  sudo dnf install plpgsql_check_16
  ```
- Enable the extension in your database:
  ```
  /usr/pgsql-16/bin/psql -c "CREATE EXTENSION plpgsql_check;" -d postgres
  ```

Key Function: `plpgsql_show_dependency_tb`

- Lists dependencies for functions/procedures, including:
  - Other functions/procedures
  - Tables
  - Sequences
- Shows the immediate calling function.
- **Note:** Does not currently show recursive dependencies.

plpgsql_check: Ensuring Code Integrity

Example

```
edb=# SELECT * FROM plpgsql_show_dependency_tb('public.test_numeric2'::regproc);
   type    |  oid  |    schema    |     name       |      params
-----------+-------+--------------+----------------+------------------
 FUNCTION  | 18309 | public       | test_numeric1  | (numeric,numeric)
 FUNCTION  | 18306 | test_package | test_function  | (numeric)
 FUNCTION  | 16535 | dbms_output  | put_line       | (text)
(3 rows)
```

plpgsql_check: Building recursive dependency tree

Introducing `get_dependency_tree()`:

- A custom function built upon plpgsql_show_dependency_tb.
- Provides a hierarchical view of object dependencies.
- Source code available on GitHub: https://github.com/vibhorkum/EDB-SPL-SQL/blob/main/dependency_tree.sql

Example Usage:

```
SELECT * FROM get_dependency_tree('your_function_name')
```

Output:

- A tree-like structure showing the function/procedure and its dependencies.

plpgsql_check: Building recursive dependency tree

```
edb=# SELECT caller_procedure,
       calling_procedure,
       procedure_type,
       nested_level
FROM get_dependency_tree(schema_name := 'public',
                         procedure_name := 'test_numeric3',
                         depth_level := 30000);
NOTICE:  table "temp_dep_tree" does not exist, skipping
NOTICE:  Level 1
NOTICE:  LEVEL 1 => dbms_output.put_line
NOTICE:  LEVEL 2 => public.test_numeric2
NOTICE:  LEVEL 3 => test_package.test_procedure
NOTICE:  LEVEL 4 => dbms_output.put_line
NOTICE:  Level 1
NOTICE:  LEVEL 1 => dbms_output.put_line
NOTICE:  LEVEL 2 => public.test_numeric2
NOTICE:  LEVEL 3 => test_package.test_procedure
NOTICE:  LEVEL 4 => dbms_output.put_line
                              caller_procedure                                |                  calling_procedure                   | procedure_type | nested_level
------------------------------------------------------------------------------+------------------------------------------------------+----------------+--------------
 public.test_numeric3(IN p_abc integer, IN p_def integer)                     | sys.dbms_output.put_line(IN item text)               | PROCEDURE      |            0
 public.test_numeric3(IN p_abc integer, IN p_def integer)                     | public.test_numeric2(IN p_abc numeric, IN p_def numeric) | PROCEDURE      |            0
 public.test_numeric3(IN p_abc integer, IN p_def integer)                     | public.test_package.test_procedure(IN p_param numeric) | PROCEDURE      |            0
 sys.dbms_output.put_line(IN item text)                                       | sys._put_line(item text)                             | FUNCTION       |            1
 public.test_numeric2(IN p_abc numeric, IN p_def numeric)                     | public.test_numeric1(p_abc numeric, p_def numeric)   | FUNCTION       |            2
 public.test_numeric2(IN p_abc numeric, IN p_def numeric)                     | public.test_package.test_function(p_param numeric)   | FUNCTION       |            2
 public.test_numeric2(IN p_abc numeric, IN p_def numeric)                     | sys.dbms_output.put_line(IN item text)               | PROCEDURE      |            2
 public.test_package.test_procedure(IN p_param numeric)                       | sys.dbms_output.put_line(IN item text)               | PROCEDURE      |            3
 sys.dbms_output.put_line(IN item text)                                       | sys._put_line(item text)                             | FUNCTION       |            4
 public.test_numeric3(IN p_abc character varying, IN p_def character varying) | sys.dbms_output.put_line(IN item text)               | PROCEDURE      |            0
 public.test_numeric3(IN p_abc character varying, IN p_def character varying) | public.test_numeric2(IN p_abc numeric, IN p_def numeric) | PROCEDURE      |            0
 public.test_numeric3(IN p_abc character varying, IN p_def character varying) | public.test_package.test_procedure(IN p_param numeric) | PROCEDURE      |            0
 sys.dbms_output.put_line(IN item text)                                       | sys._put_line(item text)                             | FUNCTION       |            1
 public.test_numeric2(IN p_abc numeric, IN p_def numeric)                     | public.test_numeric1(p_abc numeric, p_def numeric)   | FUNCTION       |            2
 public.test_numeric2(IN p_abc numeric, IN p_def numeric)                     | public.test_package.test_function(p_param numeric)   | FUNCTION       |            2
 public.test_numeric2(IN p_abc numeric, IN p_def numeric)                     | sys.dbms_output.put_line(IN item text)               | PROCEDURE      |            2
 public.test_package.test_procedure(IN p_param numeric)                       | sys.dbms_output.put_line(IN item text)               | PROCEDURE      |            3
 sys.dbms_output.put_line(IN item text)                                       | sys._put_line(item text)                             | FUNCTION       |            4
(18 rows)
```

plpgsql_check: Beyond Dependency Analysis

Code Quality & Performance:

- **Identify and fix compilation errors:** `plpgsql_check` can detect syntax errors and other code issues before runtime.

```
postgres=# SELECT * FROM plpgsql_check_function('example01', fatal_errors=>false);
                  plpgsql_check_function
------------------------------------------------------------
 error:42703:8:assignment:record "r" has no field "k"
 Context: PL/pgSQL assignment "s := s + r.k"
 error:2F005:control reached end of function without RETURN
 warning extra:00000:3:DECLARE:never read variable "r"
 warning extra:00000:4:DECLARE:never read variable "s"
(5 rows)
```

plpgsql_check: Beyond Dependency Analysis

Code Quality & Performance:

- **Profile function/procedure performance:** Analyze execution time and identify bottlenecks.

```
postgres=# SELECT * FROM plpgsql_profiler_function_tb('example01');
 lineno | stmt_lineno | queryids | cmds_on_row | exec_stmts | exec_stmts_err | total_time | avg_time | max_time  | processed_rows |                    source
--------+-------------+----------+-------------+------------+----------------+------------+----------+-----------+----------------+-----------------------------------------------
      1 |             |          |             |            |                |            |          |           |                |
      2 |             |          |             |            |                |            |          |           |                | DECLARE
      3 |             |          |             |            |                |            |          |           |                |   r record;
      4 |             |          |             |            |                |            |          |           |                |   s numeric DEFAULT 0;
      5 |           5 |          |           1 |          1 |              0 |      0.044 |    0.044 | {144.924} | {0}            | BEGIN
      6 |           6 |          |           1 |          1 |              0 |    144.879 |  144.879 | {144.879} | {0}            |   FOR r IN SELECT * FROM bigtable WHERE id = _id
      7 |             |          |             |            |                |            |          |           |                |   LOOP
      8 |           8 |          |           1 |          2 |              0 |      0.088 |    0.044 | {0.086}   | {0}            |     s := s + r.v;
      9 |             |          |             |            |                |            |          |           |                |   END LOOP;
     10 |          10 |          |           1 |          1 |              0 |      0.001 |    0.001 | {0.001}   | {0}            |   RETURN s;
     11 |             |          |             |            |                |            |          |           |                | END;
(11 rows)
```

plpgsql_check: Beyond Dependency Analysis

Code Quality & Performance:

- **Get performance improvement tips:** Receive warnings and suggestions for optimizing your code.

```
postgres=# SELECT * FROM plpgsql_check_function('example01', performance_warnings => true);
                                                plpgsql_check_function
-------------------------------------------------------------------------------------------------------------------------------
 performance:42804:5:statement block:target type is different type than source type
 Detail: cast "integer" value to "numeric" type
 Hint: Hidden casting can be a performance issue.
 Context: during statement block local variable "s" initialization on line 4
 performance:42804:6:FOR over SELECT rows:implicit cast of attribute caused by different PLpgSQL variable type in WHERE clause
 Query: SELECT * FROM bigtable WHERE id = _id
 --                                     ^
 Detail: An index of some attribute cannot be used, when variable, used in predicate, has not right type like a attribute
 Hint: Check a variable type - int versus numeric
 performance:00000:routine is marked as VOLATILE, should be STABLE
 Hint: When you fix this issue, please, recheck other functions that uses this function.
(11 rows)
```

```
postgres=#  SELECT * FROM plpgsql_check_function('example01', performance_warnings => true);
                                plpgsql_check_function
-------------------------------------------------------------------------------------
 performance:00000:routine is marked as VOLATILE, should be STABLE
 Hint: When you fix this issue, please, recheck other functions that uses this function.
(2 rows)
```

plpgsql_check: Beyond Dependency Analysis

Security Enhancement:

- **Detect potential SQL injection vulnerabilities:** `plpgsql_check` can help you identify areas in your code that might be susceptible to SQL injection attacks.

```
postgres=# SELECT count_rows('bigtable');
 count_rows
------------
 1000000
(1 row)

postgres=# SELECT * FROM plpgsql_check_function('count_rows', security_warnings => true);
                          plpgsql_check_function
-------------------------------------------------------------------------
 security:00000:4:EXECUTE:text type variable is not sanitized
 Query: 'SELECT count(*) FROM ' || tablename
 --                              ^
 Detail: The EXECUTE expression is SQL injection vulnerable.
 Hint: Use quote_ident, quote_literal or format function to secure variable.
(5 rows)
```

III. Automated PostgreSQL Tuning

Using EDB Ansible Or edb_pg_tuner

Why Automate PostgreSQL Tuning?

Challenges of Manual Tuning:

- **Dynamic Workloads:** Difficult to keep up with constantly changing workloads and adjust tuning parameters accordingly.
- **Scale:** Manually tuning numerous databases is time-consuming and inefficient.
- **Deployment Consistency:** Ensuring optimal performance for every deployment requires significant effort.

Benefits of Automation:

- **Reduced Manual Effort:** Free up DBAs from tedious tuning tasks.
- **Continuous Optimization:** Maintain peak performance even as workloads evolve.
- **Reduced Human Error:** Eliminate the risk of misconfigurations and incorrect tuning decisions.
- **Improved Performance:** Achieve consistent and optimal performance across all deployments.

Automated PostgreSQL Tuning with TPA

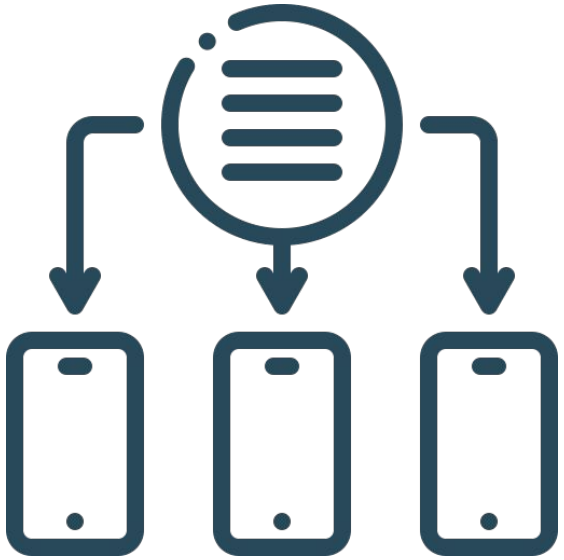
- TPA is an orchestration tool that uses Ansible to deploy Postgres clusters according to EDB's recommendations.
- TPA embodies the best practices followed by EDB, informed by many years of hard-earned experience with deploying and supporting Postgres.
- These recommendations are as applicable to quick testbed setups as to production environments.

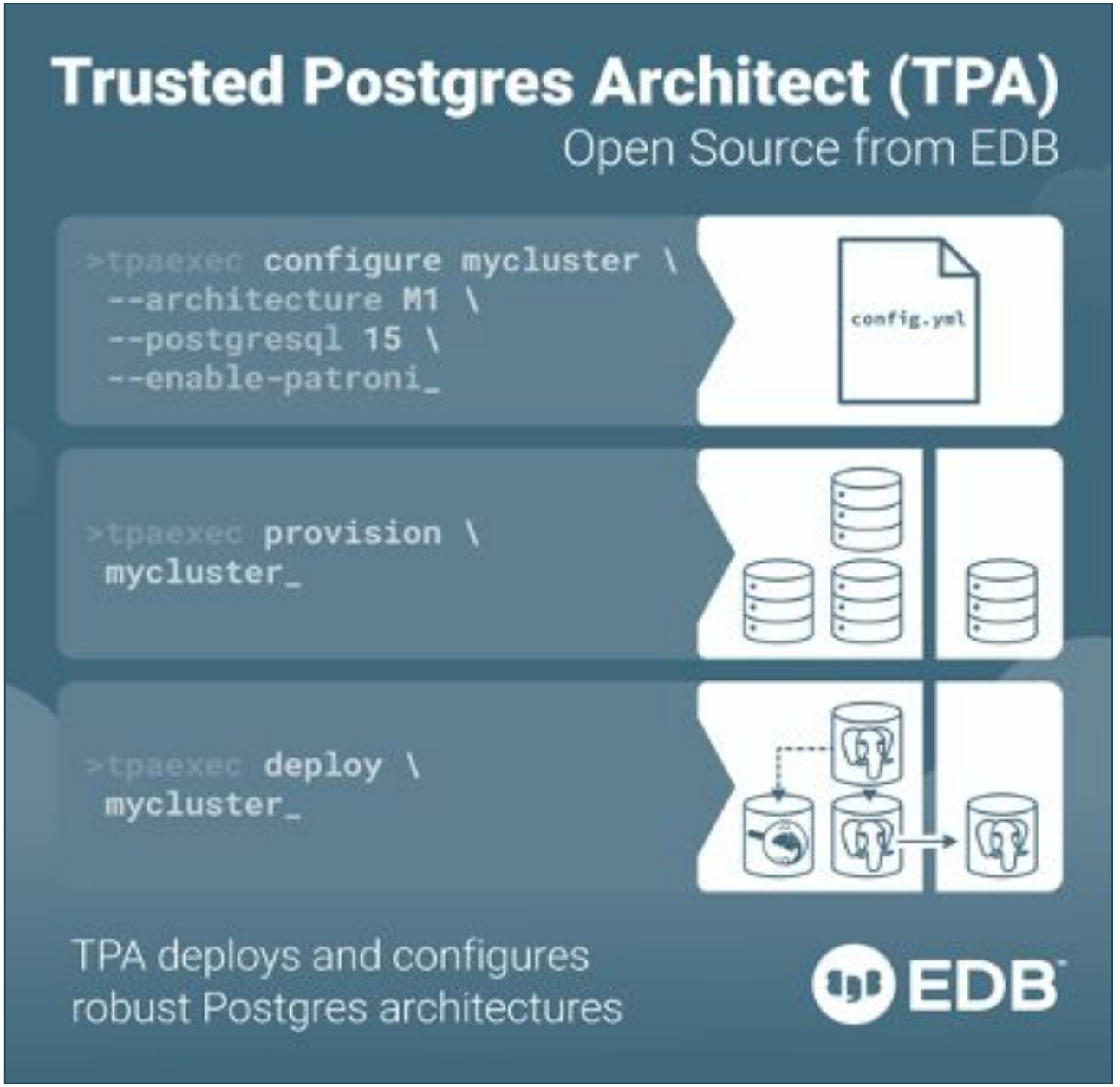
What can TPA do?

- TPA is built around a declarative configuration mechanism that you can use to describe a Postgres cluster, from its topology right down to the smallest details of its configuration.
- TPA can:
  - Provision servers (e.g.: AWS EC2 or Docker). Or you can deploy to existing servers)
  - Configure the operating system
  - Install and configure Postgres and associated components (PGD, barman, pgbouncer, repmgr and various Postgres extensions)
  - Run automated tests on the cluster after deployments
  - Deploy future changes to your configuration (e.g., changing Postgres settings, installing and upgrading packages, adding new servers, and so on)

How do I use it?

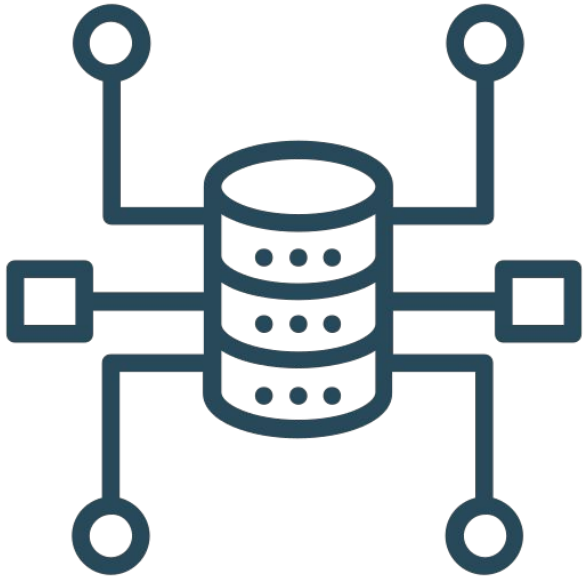
- Configure
- Provision
- Deploy





Configuration

- You must select an architecture and a platform for the cluster.
- An architecture is a recommended layout of servers and software to set up Postgres for a specific purpose.
- Architectures:
  - "M1" (Postgres with a primary and streaming replicas)
  - "PGD-Always-ON" (EDB Postgres Distributed 5 in an Always On configuration).

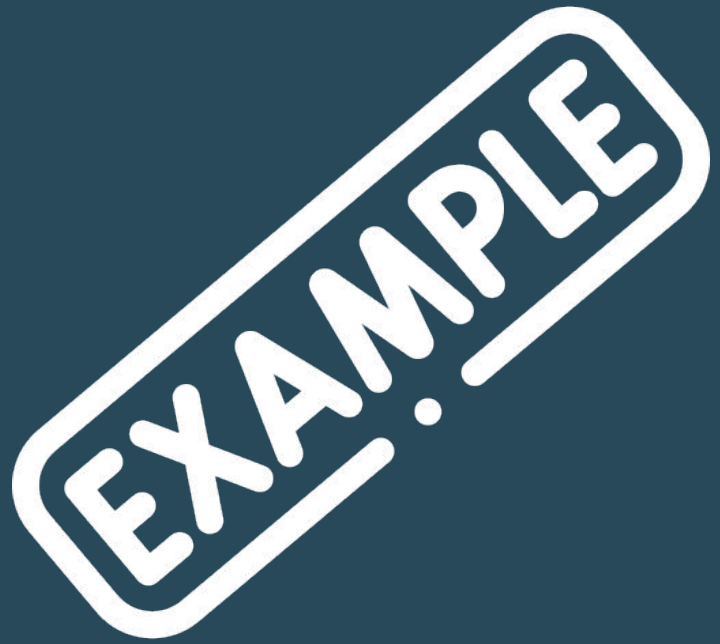


```
# Configure
tpaexec configure ~/clusters/my-cluster \
--architecture M1 \
--postgresql 16 \
--failover-manager efm \
--platform bare \
--hostnames-from ~/clusters/hostnames.txt

# Provision
tpaexec provision ~/clusters/my-cluster

# Deploy
tpaexec deploy ~/clusters/my-cluster

# Test
tpaexec test ~/clusters/my-cluster -v
```



EDB Postgres Tuner Extension

The Challenge: Default PostgreSQL settings are often conservative and don't fully utilize available resources (CPU, memory, storage).

The Solution: EDB Postgres Tuner automatically optimizes 15+ parameters based on your system and workload.

Benefits:

- Maximizes resource utilization for improved performance.
- Provides safe and controlled tuning recommendations.
- Offers both automatic and manual tuning options.
- Makes expert tuning accessible to all.

Installation and Configuration

Install the package:

- For RPM-based systems (e.g., Red Hat, CentOS):
  ```
  sudo dnf -y install edb-pg16-postgres-tuner1
  ```

Configure PostgreSQL:

- Edit postgresql.conf:
  ```
  shared_preload_libraries = 'edb_pg_tuner'
  ```
  (Add to existing libraries if needed)
- Restart Postgres:
  ```
  sudo systemctl restart postgresql-16
  ```
- Enable the extension:
  ```
  CREATE EXTENSION edb_pg_tuner;
  ```

Fine-Tuning EDB Postgres Tuner

Customize Tuner Behavior:

- **edb_pg_tuner.autotune:** Enables automatic application of tuning recommendations. (Default: false)
- **edb_pg_tuner.naptime:** Sets the interval (in seconds) between tuning checks. (Default: 600 seconds / 10 minutes)
- **edb_pg_tuner.max_wal_size_limit:** Sets an upper limit for the max_wal_size recommendation. (Default: 0 / no limit)

How to Apply Changes:

- Edit postgresql.conf to modify these parameters.
- Restart the service

EDB Postgres Tuner: SQL Interface

Get Recommendations:

- Use the edb_pg_tuner_recommendations() function to generate tuning suggestions.

Default (conf) format: Provides recommendations in the format used in postgresql.conf.

```
postgres=# SELECT * FROM edb_pg_tuner_recommendations();
          recommendation
-------------------------------
 effective_cache_size = '12 GB'
 maintenance_work_mem = '1024 MB'
 shared_buffers = '3931 MB'
(3 rows)
```

sql format: Generates ALTER SYSTEM commands that can be directly executed.

```
postgres=# SELECT edb_pg_tuner_recommendations('sql');
            edb_pg_tuner_recommendations
-----------------------------------------------------
 ALTER SYSTEM SET effective_cache_size = '12 GB';
 ALTER SYSTEM SET maintenance_work_mem = '1024 MB';
 ALTER SYSTEM SET shared_buffers = '3931 MB';
(3 rows)
```

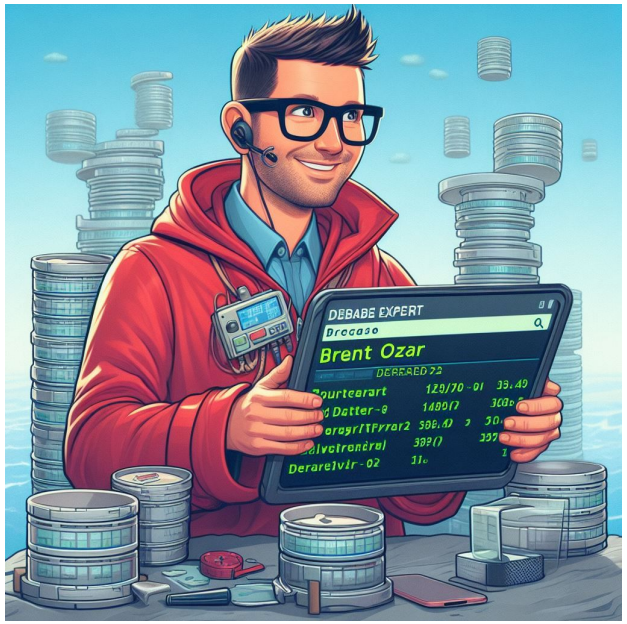
IV. Online VACUUM FULL

Using pg_squeeze/pg_repack

Online VACUUM FULL: A DBA's Dream

The Problem with Traditional `VACUUM FULL`:

- Requires an exclusive lock on the table.
- Leads to significant downtime, especially for large tables.
- Disrupts ongoing operations and affects application availability.



Why DBAs Want Online `VACUUM FULL`:

- **Eliminate Downtime:** Perform `VACUUM FULL` without blocking other operations.
- **Maximize Availability:** Keep applications running smoothly during maintenance.
- **Reduce Maintenance Windows:** Perform table rebuilds during peak hours without disruption.
- **Improve User Experience:** Ensure uninterrupted access to data for users.
- **Simplify Operations:** Reduce the complexity of scheduling maintenance tasks.

pg_squeeze: Online Table Reorganization Made Easy

What it does:

- Removes bloat (unused space) from tables.
- Optionally reorders rows based on an index (like `CLUSTER`, but online).
- A modern alternative to `pg_repack`.

Why it's better:

- **Server-side only:** Simpler to use and configure than `pg_repack`.
- **Background workers:** Enables automated, unattended operation.
- **Leverages PostgreSQL advancements:** Uses logical decoding for efficient change tracking.

pg_squeeze: Online Table Reorganization Made Easy

Install the package:

```
sudo dnf -y install pg_squeeze_16
```

Configure PostgreSQL:

- Edit postgresql.conf:

```
shared_preload_libraries = 'pg_squeeze'
wal_level = logical
max_replication_slots = 1  # ... or add 1 to the current value
(Add pg_squeeze to existing libraries if needed)
```

- **Restart Postgres:**
  `sudo systemctl restart postgresql`
- **Enable the extension:**
  `CREATE EXTENSION pg_squeeze;`

pg_squeeze: Online Table Reorganization Made Easy

How to schedule

```
postgres=# INSERT INTO squeeze.tables (tabschema, tabname, schedule)
           VALUES ('public', 'foo', ('{30}', '{22}', NULL, NULL, '{3, 5}'));
INSERT 0 1
```

Schedule format:

```
CREATE TYPE schedule AS (
    minutes       minute[],
    hours         hour[],
    days_of_month dom[],
    months        month[],
    days_of_week  dow[]
);
```

pg_squeeze: Ad Hoc Table Reorganization

On-Demand Optimization:

- `pg_squeeze` allows you to reorganize tables manually, without prior registration or bloat checks.
- Useful for immediate optimization of specific tables.

`squeeze_table()` Function:

```
squeeze.squeeze_table(
    tabschema name,
    tabname name,
    clustering_index name DEFAULT NULL,
    rel_tablespace name DEFAULT NULL,
    ind_tablespaces name[] DEFAULT NULL
)
```

pg_squeeze: Ad Hoc Table Reorganization

Parameters:

- tabschema, tabname: Specify the schema and name of the table.
- clustering_index: Optionally cluster rows based on this index.
- rel_tablespace: Move the table to a different tablespace.
- ind_tablespaces: Move indexes to specific tablespaces.

Example:

```
SELECT squeeze.squeeze_table('public', 'pgbench_accounts');
```

Log table

```
postgres=# select * from squeeze.log ;
 tabschema |    tabname    |            started            |           finished            | ins_initial | ins | upd | del
-----------+---------------+-------------------------------+-------------------------------+-------------+-----+-----+-----
 public    | bigtable_pkey | 2024-09-28 21:48:15.834977+00 | 2024-09-28 21:48:17.766721+00 |     1000000 |   0 |   0 |   0
(1 row)
```

V. Postgres Workload Analysis (AWR-like)

Using edb-pwr

Postgres Performance Insights: Beyond Basic Monitoring

Why Basic Monitoring Is Not Enough:

- Provides a limited view of database activity.
- Doesn't offer deep insights into performance bottlenecks.
- Makes it difficult to diagnose complex performance issues.

What DBAs Need for Effective Performance Management:

- **Detailed Performance Data:** Granular information about wait events, resource consumption, and execution statistics.
- **Historical Trends:** Ability to analyze performance over time to identify patterns and anomalies.
- **Query-Level Insights:** Understanding the performance of individual SQL queries.
- **Proactive Monitoring:** Early detection of potential performance issues.

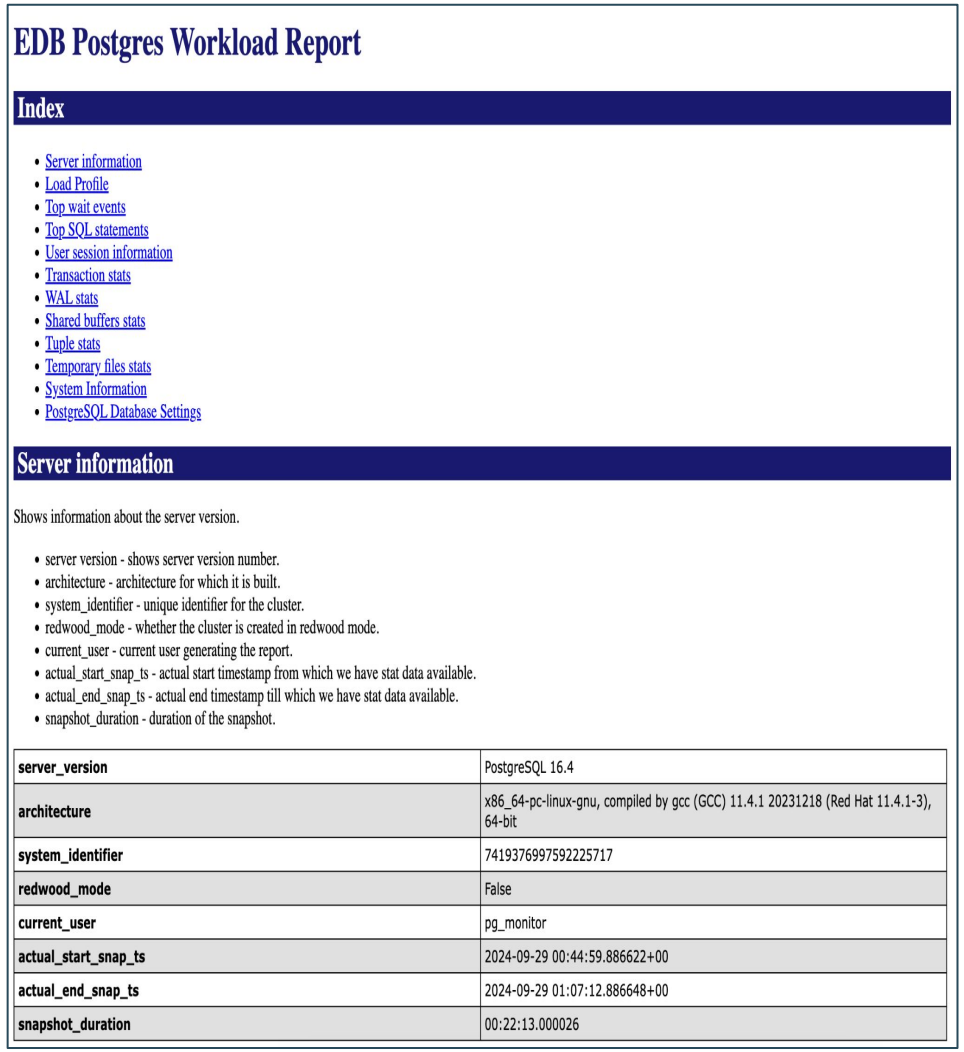
PWR: AWR-like Reporting for PostgreSQL

What is PWR?

- A Python-based tool for generating detailed PostgreSQL workload reports.
- Provides insights similar to Oracle's AWR reports.
- Output formats: HTML, Markdown, DOCX, and PDF.

Key Features:

- **Comprehensive Data:** Captures wait events, SQL performance statistics, and more.
- **Historical Analysis:** Enables analysis of workload trends over time.
- **Flexible Deployment:** Runs on any machine with access to the PostgreSQL server.



EDB Postgres Workload Report

Index

- Server information
- Load Profile
- Top wait events
- Top SQL statements
- User session information
- Transaction stats
- WAL stats
- Shared buffers stats
- Tuple stats
- Temporary files stats
- System Information
- PostgreSQL Database Settings

Server information

Shows information about the server version.

- server version - shows server version number.
- architecture - architecture for which it is built.
- system_identifier - unique identifier for the cluster.
- redwood_mode - whether the cluster is created in redwood mode.
- current_user - current user generating the report.
- actual_start_snap_ts - actual start timestamp from which we have stat data available.
- actual_end_snap_ts - actual end timestamp till which we have stat data available.
- snapshot_duration - duration of the snapshot.

server_version	PostgreSQL 16.4
architecture	x86_64-pc-linux-gnu, compiled by gcc (GCC) 11.4.1 20231218 (Red Hat 11.4.1-3), 64-bit
system_identifier	7419376997592225717
redwood_mode	False
current_user	pg_monitor
actual_start_snap_ts	2024-09-29 00:44:59.886622+00
actual_end_snap_ts	2024-09-29 01:07:12.886648+00
snapshot_duration	00:22:13.000026

PWR: AWR-like Reporting for PostgreSQL

EDB Postgres Workload Report

Index

Server information

Shows information about the server version.

- server version - shows server version number.
- architecture - architecture for which it is built.
- system_identifier - unique identifier for the cluster.
- redwood_mode - whether the cluster is created in redwood mode.
- current_user - current user generating the report.
- actual_start_snap_ts - actual start timestamp from which we have stat data available.
- actual_end_snap_ts - actual end timestamp till which we have stat data available.
- snapshot_duration - duration of the snapshot.

server_version	PostgreSQL 16.4
architecture	x86_64-pc-linux-gnu, compiled by gcc (GCC) 11.4.1 20231218 (Red Hat 11.4.1-3), 64-bit
system_identifier	7419376997592225717
redwood_mode	False
current_user	pg_monitor
actual_start_snap_ts	2024-09-29 00:44:59.886622+00
actual_end_snap_ts	2024-09-29 01:07:12.886648+00
snapshot_duration	00:22:13.000026

Top SQL statements

- dbtime - total dbtime spent by the sql statement.
- waittime - total waittime spent by the sql statement.
- cputime - total cputime spent by the sql statement.
- top_waitevent - waitevent name on which this statement spent maximum time.
- query - actual sql query.

Top 10 sql statements sorted by dbtime (high to low)

dbtime	waittime	cputime	top_waitevent	query
4	2	2	LogicalRewriteSync	vacuum full

Top 10 sql statements sorted by cputime (high to low)

cputime	dbtime	waittime	top_waitevent	query
2	4	2	LogicalRewriteSync	vacuum full

Top 10 sql statements sorted by wait time (high to low)

waittime	dbtime	cputime	top_waitevent	query
2	4	2	LogicalRewriteSync	vacuum full

Top wait events

Show total wait time on top wait events in seconds.

See the Wait Event Table Information here: 28.2. The Cumulative Statistics System.

- waitevent - waitevent name (waitevent name CPU means time spent working on CPU or a non-wait time).
- wait_class - waitevent type.
- waittime - waiting time in seconds spent on this waitevent.
- pct_dbtime - %dbtime spent waiting on this waitevent type.

waitevent	wait_class	waittime	pct_dbtime
CPU	N/A	3	60.0
LogicalRewriteSync	IO	2	40.0

For more information -
https://www.enterprisedb.com/docs/pwr/latest/

VI. Job Scheduler

Using pg_cron

Why DBAs Love In-Database Job Schedulers

Centralized Management:

- Schedule and manage all database tasks in one place.
- No need to rely on external tools or cron jobs.
- Easier to track and monitor scheduled jobs.

Increased Reliability:

- Jobs run even if the database server is restarted.
- Ensures tasks are executed on time, regardless of external factors.
- Built-in error handling and logging for improved reliability.

Enhanced Security:

- Jobs run with database user privileges for better security control.
- No need to grant OS-level access for scheduled tasks.

Why DBAs Love In-Database Job Schedulers

Improved Performance:

- Jobs execute within the database environment, reducing overhead.
- Direct access to database objects for efficient task execution.

Simplified Maintenance:

- Easier to manage and update scheduled tasks within the database.
- Streamlined deployment of database changes and updates.

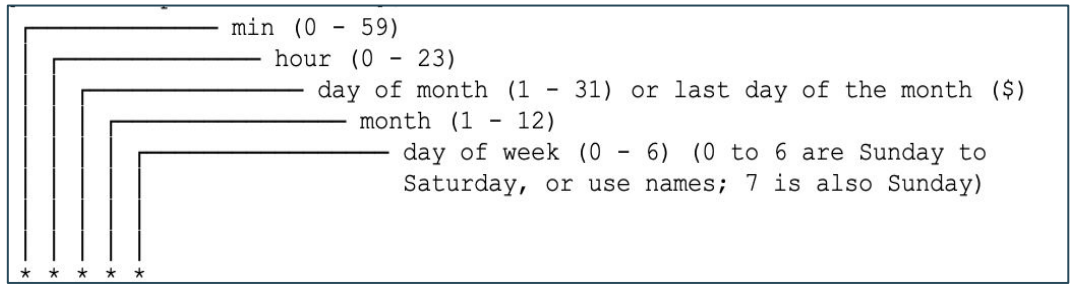
pg_cron: Effortless Job Scheduling Within PostgreSQL

What is pg_cron?

- An extension that brings cron-like job scheduling to PostgreSQL (10+).
- Schedule SQL commands directly within the database.

Key Features:

- **Familiar Syntax:** Uses standard cron expressions for easy scheduling.
- **Second-Level Precision:** Schedule jobs down to the second.
- **End-of-Month Scheduling:** Supports '$' to specify the last day of the month.
- **Job Queuing:** Ensures jobs run sequentially, even if delayed.
- **Parallel Execution:** Runs multiple jobs concurrently.



Cron Syntax:

- *: Run every time period (e.g., every minute, every hour).
- Specific number: Run only at that specific time (e.g., at 10 minutes past the hour).
- Example: 0 0 * * * (run every day at midnight)

Installing and Configuring pg_cron

Installation:

```
sudo yum install -y pg_cron_16
```

Configuration (postgresql.conf):

```
shared_preload_libraries = 'pg_cron' # Load pg_cron on startup
cron.database_name = 'postgres'      # Optionally specify the database (default: postgres)
cron.timezone = 'PRC'                # Optionally specify the timezone (default: GMT)
```

Restart Postgres:

```
sudo systemctl restart postgresql
```

Enable the extension:

```
CREATE EXTENSION pg_cron;
GRANT USAGE ON SCHEMA cron TO <user>;  -- Optionally grant usage to other users
```

Installing and Configuring pg_cron

Configure Job Execution:

- **Enable local connections in pg_hba.conf** or use .pgpass for authentication.
- **Alternatively, use background workers:**

```
cron.use_background_workers = on
max_worker_processes = 20   # Adjust as needed
```

Schedule & View Active Jobs:

```
postgres=# SELECT cron.schedule(job_name := 'nightly-vacuum',
                      schedule := '0 3 * * *',
                      command := 'VACUUM');
 schedule
----------
        2
(1 row)

postgres=# SELECT * FROM cron.job;
 jobid | schedule  | command | nodename  | nodeport | database | username | active |     jobname
-------+-----------+---------+-----------+----------+----------+----------+--------+-----------------
     2 | 0 3 * * * | VACUUM  | localhost |     5433 | postgres | postgres | t      | nightly-vacuum
(1 row)

postgres=# SELECT cron.unschedule(job_name := 'nightly-vacuum');
 unschedule
------------
 t
(1 row)

postgres=# SELECT * FROM cron.job;
 jobid | schedule | command | nodename | nodeport | database | username | active | jobname
-------+----------+---------+----------+----------+----------+----------+--------+---------
(0 rows)
```

VII. Cross-Environment Schema/Data Cloning

Using pg_dump/process or EDB Clone Schema

Schema Cloning: Secret Weapon for Efficiency and Agility

What is Schema Cloning?

- Creating an exact copy of a database schema (tables, views, functions, etc.) without the associated data.

Why DBAs Need a Schema Cloning Tool:

- **Multi-tenant use case:**
  - Onboarding new tenants efficiently.
- **Rapid Development and Testing:**
  - Quickly create new environments for developers to test code changes without impacting production.
  - Experiment with schema modifications in a safe environment.
- **Simplified Deployment:**
  - Stage schema changes in a test environment before rolling them out to production.
  - Reduce downtime and risk associated with schema migrations.
- **Training and Education:**
  - Provide trainees with a realistic database environment for practice and learning.
- **Security and Compliance:**
  - Create isolated environments for security testing and vulnerability assessments.
  - Comply with data privacy regulations by removing sensitive data from test environments.

Cross-Environment Schema Cloning: Methods and Tools

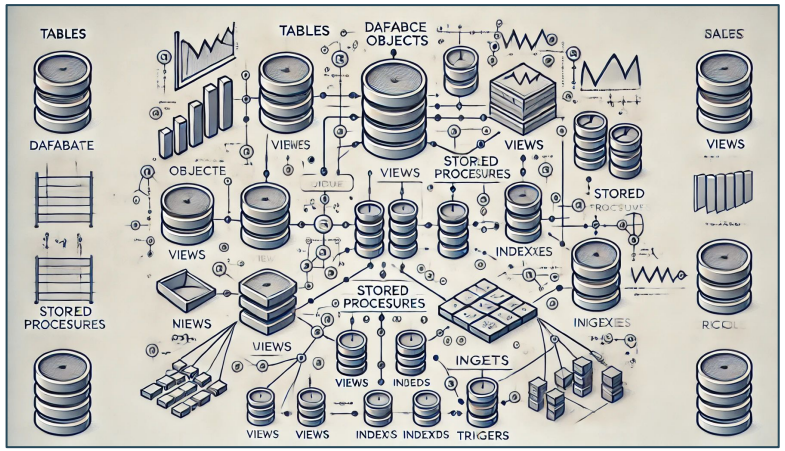
Many Options for data cloning

- **Native Logical Replication:**
  - Replicate data at the logical level (using SQL statements).
  - Fine-grained control over what gets replicated (tables, schemas, etc.).
  - Minimal impact on the source database.
- **pg_dump/pg_restore:**
  - Create a consistent backup of the source database using pg_dump.
  - Restore the backup on the target environment using pg_restore.
  - Suitable for smaller databases or when a full copy is needed.
- **Storage Snapshots:**
  - Create a point-in-time snapshot of the storage volume containing the database.
  - Mount the snapshot on the target environment.
  - Fast and efficient for large databases.
  - May require storage-level support.

Cross-Environment Schema Cloning: Methods and Tools

Why Schema Cloning Can Be Tricky:

- **More Than Just Tables:** Schemas include tables, functions, procedures, triggers, and more.
- **Dependencies:** Objects often depend on each other; cloning must maintain these relationships.
- **Data Integrity:** Parent-child relationships and constraints must be preserved.
- **Cross-Schema Dependencies:** Objects may depend on objects in other schemas.



Limited Options for Cloning to a Different Schema:

- pg_dump -s doesn't support renaming the target schema.
- Manual scripting can be complex and error-prone.
- Third-party tools might be needed for advanced scenarios.

Schema Cloning with pg_dump: Three-Step Approach

Three Steps Without Changing Schema name

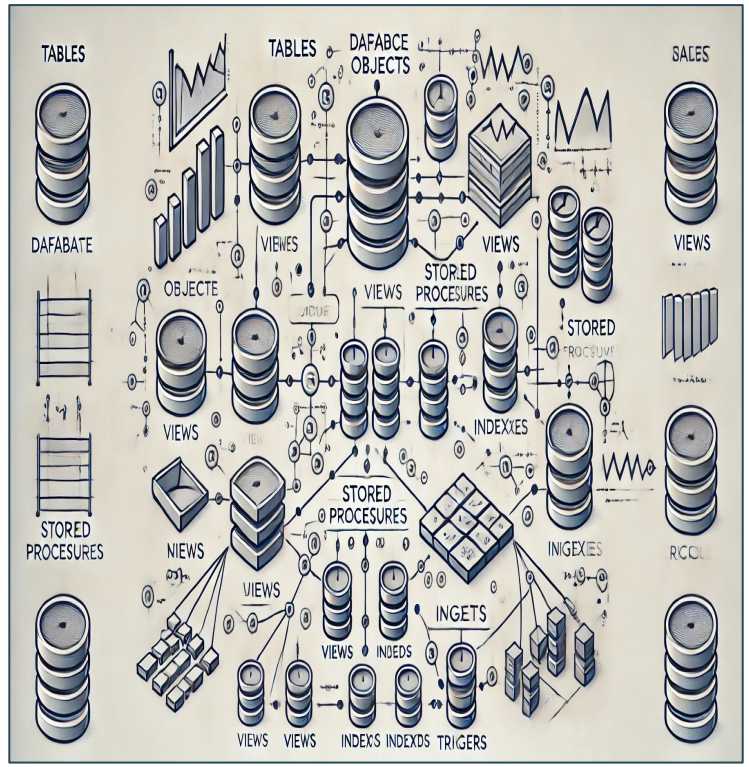
- pg_dump Commands -

```
pg_dump --schema=public --section=pre-data postgres | psql <options>
pg_dump --schema=public --section=data postgres | psql <options>
pg_dump --schema=public --section=post-data postgres | psql <options>
```

- Or Take schema backup in custom/tar/dir backup and use pg_restore

```
pg_dump --schema=public -Fd --section=pre-data --file=<backup_name>
postgres

pg_restore --schema=public --section=pre-data <backup_name> | psql
<options>
pg_restore --schema=public --section=data <backup_name> | psql
<options>
pg_restore --schema=public --section=post-data <backup_name> | psql
<options>
```



Schema Cloning with pg_dump and pg_restore

However for restoring with different schema name

- Requires using string manipulation tools like `sed`, `awk` or custom tools using python or other language

```
pg_dump --schema=public --section=pre-data postgres |sed 's/original_schema_name/new_schema_name/g' |
psql <options>
pg_dump --schema=public --section=data postgres |sed 's/original_schema_name/new_schema_name/g' | psql
<options>
pg_dump --schema=public --section=post-data postgres |sed 's/original_schema_name/new_schema_name/g'|
psql <options>
```

- Or Take schema backup in custom/tar/dir backup and use pg_restore

```
pg_dump --schema=public -Fd --section=pre-data --file=<backup_name> postgres

pg_restore --schema=public --section=pre-data <backup_name> |sed
's/original_schema_name/new_schema_name/g'| psql <options>
pg_restore --schema=public --section=data <backup_name> |sed 's/original_schema_name/new_schema_name/g'|
psql <options>
pg_restore --schema=public --section=post-data <backup_name> |sed
's/original_schema_name/new_schema_name/g'| psql <options>
```

Schema Cloning via Snapshots

Create a Volume Snapshot:

- Take a point-in-time snapshot of the storage volume containing your PostgreSQL database.

  Example using Google Cloud:

```
gcloud compute disks snapshot <disk-name> \
    --snapshot-names=<snapshot-name> \
    --zone=<zone>
```

Restore the Snapshot:

- Create a new Persistent Disk from the snapshot.

  Example using Google Cloud:

```
gcloud compute disks create <new-disk-name> \
    --source-snapshot=<snapshot-name> \
    --zone=<zone>
```

Schema Cloning via Snapshots

Mount and Access:

- Attach the new disk to a VM instance.
- Access the PostgreSQL database on the new disk.

Rename the Schema:

- Use ALTER SCHEMA to rename the schema to the desired target name.

  Example:

  ```
  ALTER SCHEMA <original_schema_name> RENAME TO <new_schema_name>;
  ```

(Optional) Export the Schema:

- Use pg_dump/pg_restore for export and restore of renamed schema in other environments.

EDB Clone Schema: Effortless Schema Cloning

Simplified Schema Cloning:

- Copy schemas within the same database or across different databases.
- Works with local and remote databases, even across clusters.

Flexible Source and Target:

- Clone from and to:
  - The same database
  - Different databases in the same cluster
  - Databases in separate clusters on different hosts

Setting Up EDB Clone Schema

Install Extensions:

```
CREATE EXTENSION postgres_fdw SCHEMA public;
CREATE EXTENSION dblink SCHEMA public;
CREATE EXTENSION adminpack;
```

Modify postgresql.conf:

```
 shared_preload_libraries =
'$libdir/dbms_pipe,$libdir/dbms_aq,$libdir/parallel_clone'
```

Install PL/Perl:

```
CREATE TRUSTED LANGUAGE plperl;
```

Install EDB Clone Schema:

```
CREATE EXTENSION parallel_clone SCHEMA public;
CREATE EXTENSION edb_cloneschema;
```

EDB Clone Schema: In Action

Create a Foreign Data Wrapper:

- Establish a connection to the source or target database.

```
CREATE SERVER local_server FOREIGN DATA WRAPPER
postgres_fdw
  OPTIONS(
    host '/tmp',
    port '5444',
    dbname 'edb'
);
CREATE USER MAPPING FOR enterprisedb SERVER
local_server
  OPTIONS (
    user 'enterprisedb',
    password 'E68123'
);
```

Clone schema within database

```
edb=#  SELECT edb_util.localcopyschema ( source_fdw := 'local_server',
                                         source_schema := 'public',
                                         target_schema := 'public_clone',
                                         log_filename := 'public_clone.log',
                                         verbose_on := true,
                                         worker_count := 2);
 localcopyschema
-----------------
 t
(1 row)
```

EDB Clone Schema: In Action

Clone remote schema

```
edb=# SELECT edb_util.remotecopyschema( source_fdw := 'local_server',
                                        target_fdw := 'local_server',
                                        source_schema := 'public',
                                        target_schema := 'remote_public_clone',
                                        log_filename := 'remote_public_clone.log',
                                        verbose_on := true,
                                        worker_count := 2);
 remotecopyschema
------------------
 t
(1 row)
```

```
edb=# SELECT edb_util.process_status_from_log( log_file := 'remote_public_clone.log');
                                                process_status_from_log
-------------------------------------------------------------------------------------------------------------------------
 (FINISH,"30-SEP-24 01:25:31.28463 +00:00",2445434,INFO,"STAGE: FINAL","successfully clone schema into remote_public_clone")
(1 row)
```

Thank you.

LinkedIN

Blog

Thank you.