Microsoft | PGCONF.EU 2024

Logical replication - getting in & out of the cloud

Alexander Kukushkin & Silvano Coriani

Tue 22 Oct | 9:00 EEST

Logical replication

Getting in&out of the Cloud

Agenda

- Intro
- Fundamentals of Logical Replication in PostgreSQL
  - Understand logical decoding
  - Replication slots and output plugins
  - Built-in core logical replication
  - Failover slots
- Practical use cases and solution patterns
  - Distributed data apps
  - Real-time change streaming
  - Online and offline migrations
- Monitoring and alerting
- Best Practices

09:00

10:30

Break

11:00

12:30

About us

- Alexander Kukushkin is a Principal Engineer in Azure Database for PostgreSQL team at Microsoft
- PostgreSQL Contributor
- Maintainer of Patroni – open-source tool for Clustering and High Availability

- Silvano Coriani is a Program Manager in Azure Database for PostgreSQL team at Microsoft.
- He has over 25 years of experience in application development and database design, troubleshooting, and performance tuning.
- He is specialized in data access libraries, query optimization, and distributed system design. He is also an author and speaker in many international industry conferences.

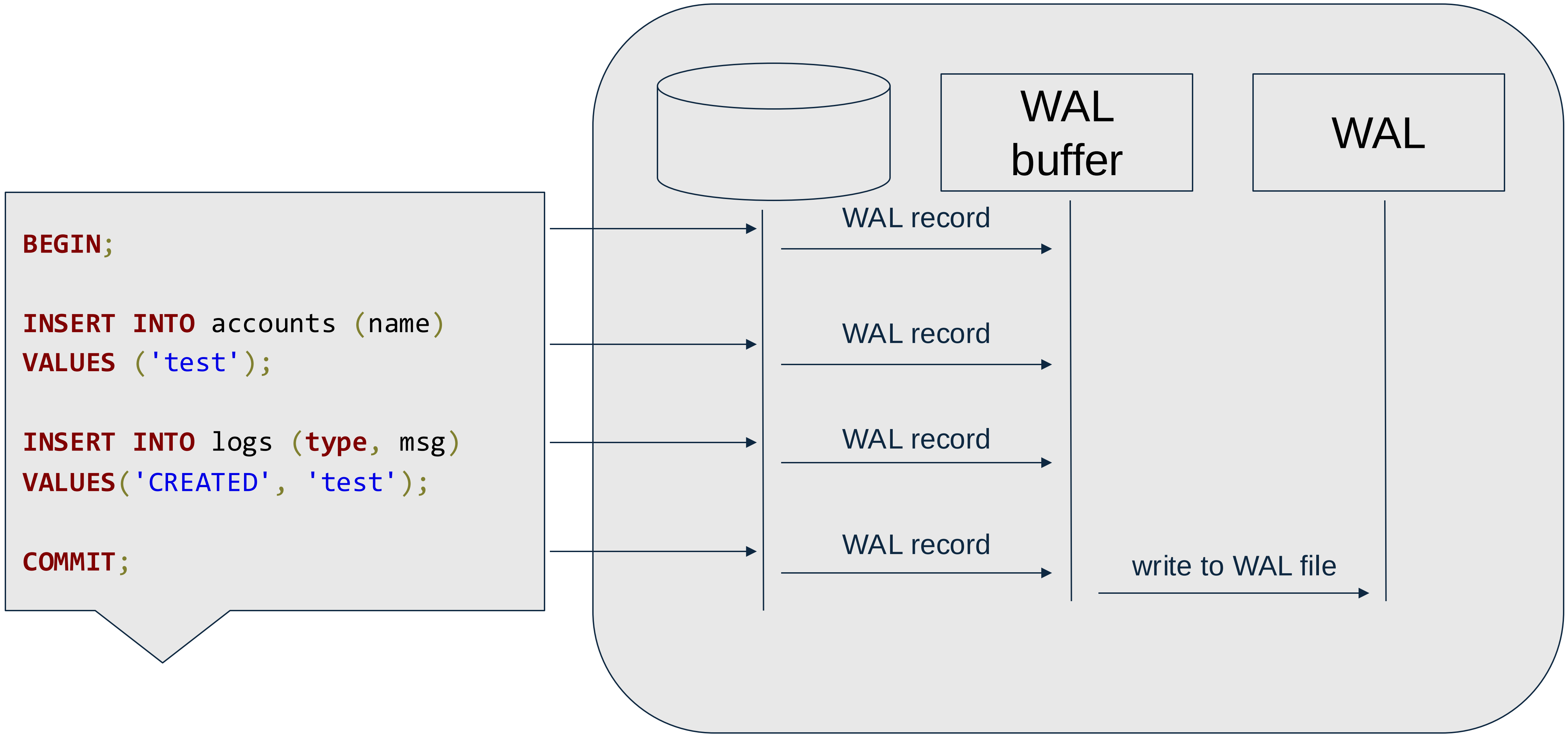
Fundamentals of Logical Replication in PostgreSQL

Introduction to logical decoding

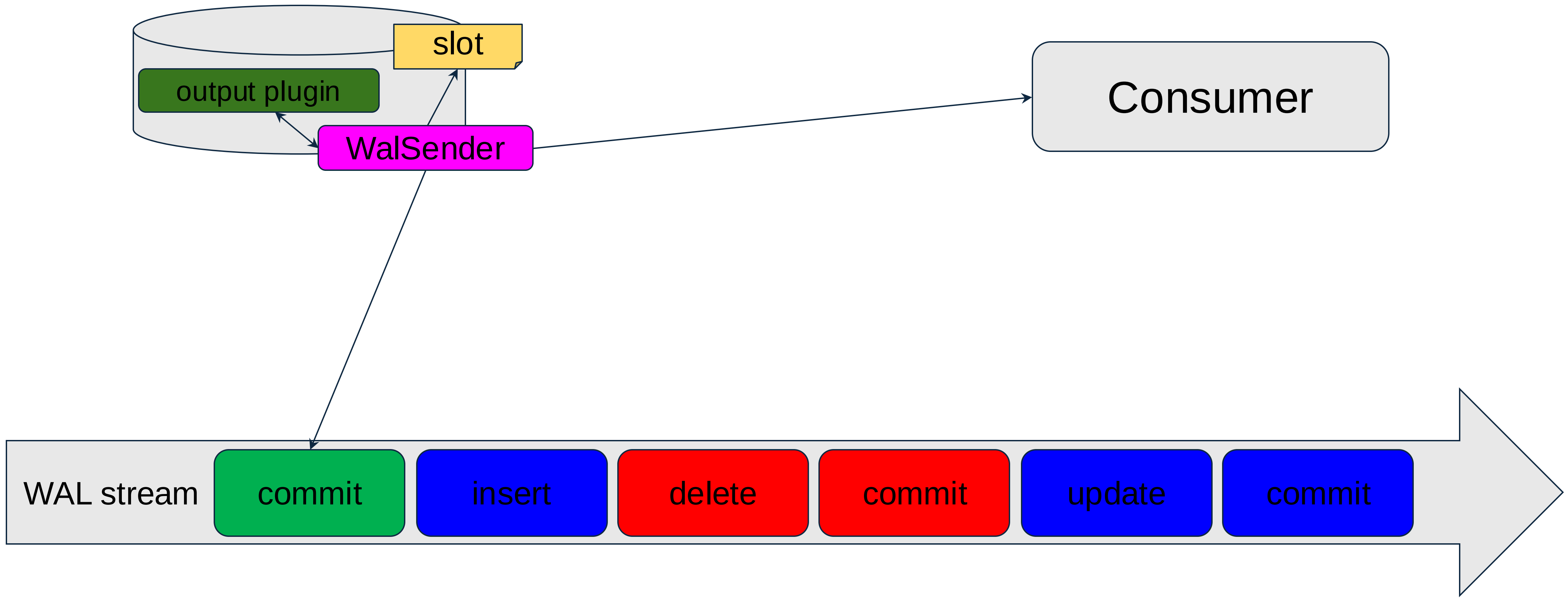
- Extracting data changes from WAL in a "simple" format that could be interpreted by an external tool
  - INSERT, UPDATE, DELETE, and TRUNCATE
- Plugin infrastructure
  - No need to modify PostgreSQL core
- Also known as Change Data Capture (CDC)

WAL

- Write-Ahead Log
  - A standard method for ensuring data integrity
  - Used for recovery, archives, replication, etc…
  - http://www.postgresql.org/docs/current/static/wal-intro.html
- Log Sequence Number (LSN)
  - Represents the location of WAL record in the WAL file
  - Example: 'A/7F6732B9' -> 45,087,142,585 bytes
    - **SELECT** (0xA << 32) | 0x7F6732B9; or
    - **SELECT** 'A/7F6732B9'::pg_lsn - '0/0'::pg_lsn;



```sql
BEGIN;

INSERT INTO accounts (name)
VALUES ('test');

INSERT INTO logs (type, msg)
VALUES('CREATED', 'test');

COMMIT;
```


slot
output plugin
WalSender
Consumer
WAL stream
commit
insert
delete
commit
update
commit

Replication slots

- Provide guarantees that WAL segments are not removed until consumed
- Provide protection against relevant rows in pg_catalog being removed by (auto)vacuum
- Tightly coupled with a particular database
- Define output plugin

Creating a logical replication slot

- Configuration: max_replication_slots > 0, wal_level = logical
- SQL interface:
  - **SELECT** pg_create_logical_replication_slot(
    '<slot_name>', '<output_plugin_name>');
- Replication protocol:
  - **CREATE_REPLICATION_SLOT** <slot_name>
    LOGICAL <output_plugin_name>

Example

```
localhost/testdb=# SELECT * FROM pg_create_logical_replication_slot('my_slot', 'test_decoding');

 slot_name |    lsn
-----------+-----------
 my_slot   | 0/130694C8

(1 row)

localhost/testdb=# SELECT slot_name, plugin, slot_type, database, confirmed_flush_lsn FROM
pg_replication_slots;

 slot_name |    plugin     | slot_type | database | confirmed_flush_lsn
-----------+---------------+-----------+----------+---------------------
 my_slot   | test_decoding | logical   | testdb   | 0/130694C8

(1 row)
```

Exported snapshots

- Retrieve consistent image of the database
- SQL interface
  - **SELECT** pg_export_snapshot()
  - Available for duration of transaction
  - pg_create_logical_replication_slot() doesn't export snapshot
- Export with replication connection
  - In result of **CREATE_REPLICATION_SLOT**
  - Available for duration of replication connection
- Keep connection/transaction for duration as long as necessary

Example: exporting snapshot

```
$ psql "user=postgres dbname=postgres
replication=database"

postgres=# CREATE_REPLICATION_SLOT my_slot LOGICAL
pgoutput;
-[ RECORD 1 ]----+--------------------
slot_name        | my_slot
consistent_point | 0/155B850
snapshot_name    | 00000003-00000005-1
output_plugin    | pgoutput
```

```
$ psql "user=postgres dbname=postgres"

postgres=# BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN
postgres=*# SET TRANSACTION SNAPSHOT '00000003-
    00000005-1';
SET
/*[do stuff]*/
postgres=*# COMMIT;
COMMIT
```

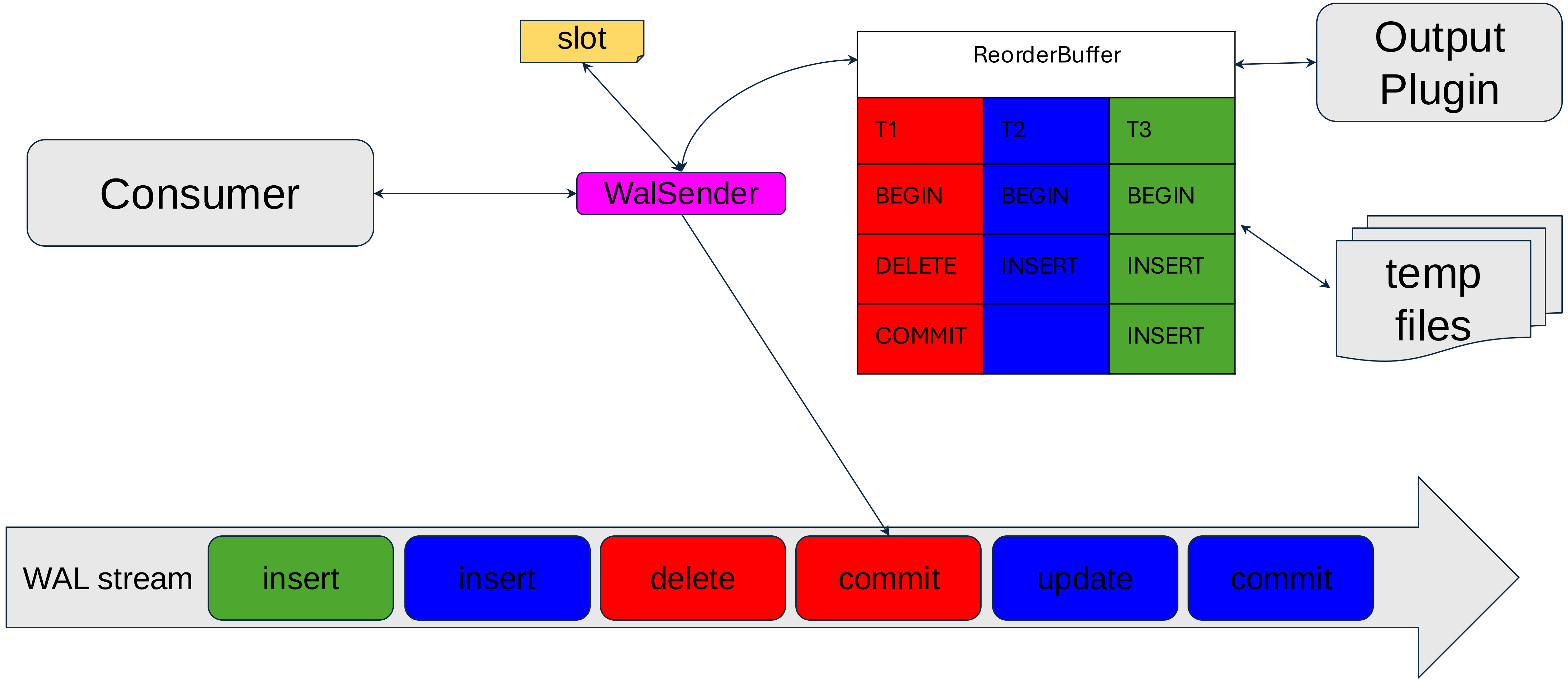
Output plugin

Transforms the data from the write-ahead log's internal representation into the format the consumer of a replication slot desires.

- Built-in core:
  - test_decoding
  - pgoutput
- 3rd party:
  - wal2json
  - pglogical
  - decoder_raw
  - ...

ReorderBuffer

- Reassembles individual pieces of transactions into toplevel transaction sized pieces.
  - In memory; or
  - Spills to disk when exceed **logical_decoding_work_mem = 64MB**
- On commit calls output plugin and sends events to consumer


slot
ReorderBuffer
Output Plugin
Consumer
WalSender
T1
T2
T3
BEGIN
BEGIN
BEGIN
DELETE
INSERT
INSERT
COMMIT
INSERT
temp files
WAL stream
insert
insert
delete
commit
update
commit

ReorderBuffer hands on – prepare sandbox

- **CREATE TABLE** test(
  id int NOT NULL PRIMARY KEY,
  l pg_lsn **NOT NULL DEFAULT** pg_current_wal_lsn());
- pg_recvlogical **--create-slot** \
- **-P test_decoding -S my_slot** \
- -U postgres **-d postgres**
- pg_recvlogical **--start** \
- **-S my_slot** \
- -U postgres **-d postgres** \
- **-f** -
- Open two psql sessions

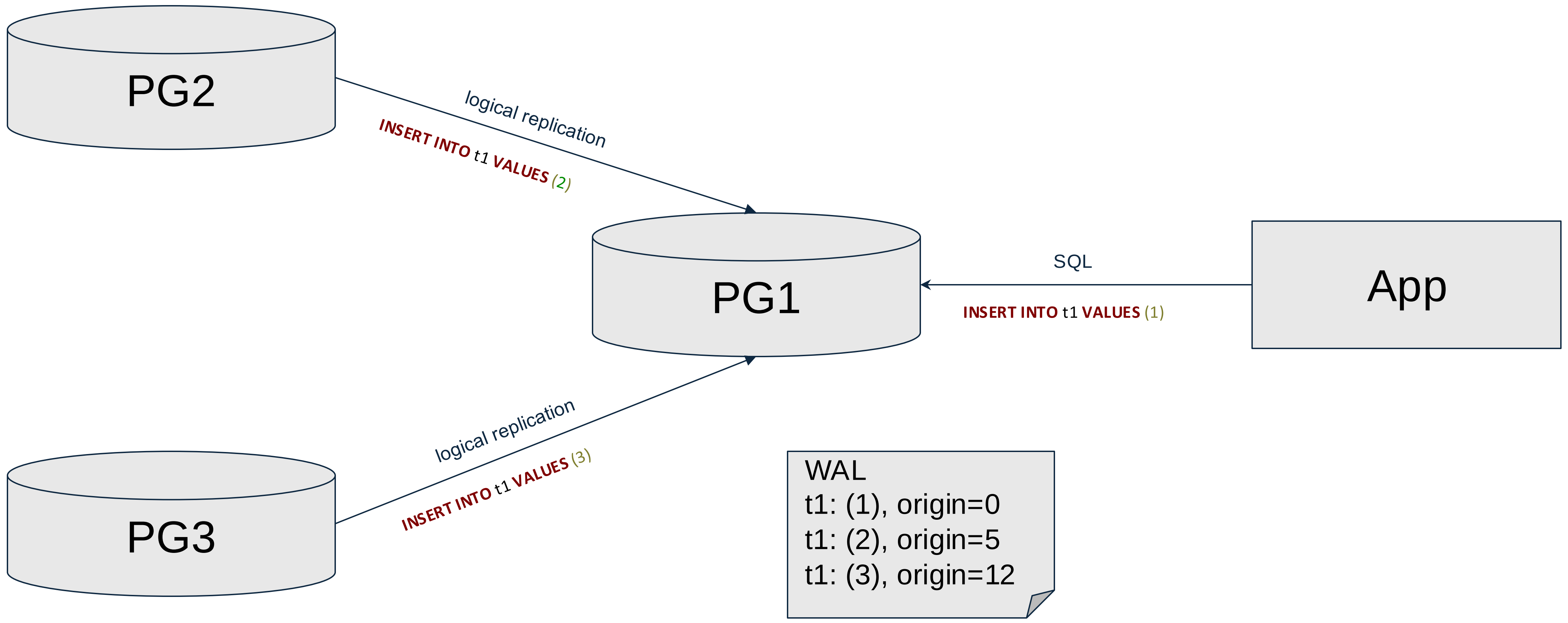
ReorderBuffer hands on

N	Session 1	Session 2	pg_recvlogical output
1	BEGIN;		
2	SELECT txid_current(); => 795	BEGIN;	
3	INSERT INTO test VALUES(1);	SELECT txid_current(); => 796	
4	INSERT INTO test VALUES(3);	INSERT INTO test VALUES(2);	
5	INSERT INTO test VALUES(5);	COMMIT;	BEGIN 796 table public.test: INSERT: id[integer]:2 l[pg_lsn]:'0/15B0B28' COMMIT 796
6	COMMIT;		BEGIN 795 table public.test: INSERT: id[integer]:1 l[pg_lsn]:'0/15B0A00' table public.test: INSERT: id[integer]:3 l[pg_lsn]:'0/15B0BF0' table public.test: INSERT: id[integer]:5 l[pg_lsn]:'0/15B0CE0' COMMIT 795

Monitoring

```
postgres=# SELECT * from pg_replication_slots;

-[ RECORD 1 ]-------+---------------
slot_name           | my_slot
plugin              | test_decoding
slot_type           | logical
datoid              | 5
database            | postgres
temporary           | f
active              | t
active_pid          | 1675174
xmin                |
catalog_xmin        | 771
restart_lsn         | 0/15B0E80
confirmed_flush_lsn | 0/15B0EB8
wal_status          | reserved
safe_wal_size       |
two_phase           | f
conflicting         | f
```

Replication origin


PG2
logical replication
INSERT INTO t1 VALUES (2)
PG1
SQL
App
INSERT INTO t1 VALUES (1)
PG3
logical replication
INSERT INTO t1 VALUES (3)

WAL
t1: (1), origin=0
t1: (2), origin=5
t1: (3), origin=12

Replication origin

- Progress tracking of logical replication (on subscriber side)
- Filtering based on the row origin
    - For example, to prevent loops
- Monitoring: pg_replication_origin_status view

Built-in core logical replication

- Available since PostgreSQL v10
- **CREATE PUBLICATION**
  - Enumerates tables to replicate from
  - However, changes to ALL databases and tables are published to WAL!
- **CREATE SUBSCRIPTION**
  - Connects to one or more publications in a remote database
  - Creates a logical replication slot with **pgoutput** decoding plugin
  - Maintains replication origins

CREATE PUBLICATION

```
CREATE PUBLICATION name
   [ FOR ALL TABLES
     | FOR publication_object [, ... ] ]
   [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

where publication_object is one of:

```
 TABLE [ ONLY ] table_name [ * ] [ ( column_name [, ... ] ) ] [ WHERE ( expression ) ] [, ... ]
 TABLE IN SCHEMA { schema_name | CURRENT_SCHEMA } [, ... ]
```

CREATE PUBLICATION quirks

- **CREATE PUBLICATION** - the invoking user must:
    - have the **CREATE** privilege for the **current database**
    - be an owner of tables
- **FOR TABLES IN SCHEMA** and **WHERE (expression)** - available since PG15
- **FOR ALL TABLES** and **FOR TABLES IN SCHEMA** - requires superuser access
    - Works for members of **azure_pg_admin** on Flexible Server
    - Otherwise, all tables must be explicitly enumerated
- **WITH** (**publish** = **'insert, update, delete, truncate'**) - default
    - truncate available since PG11
- **WITH** (**publish_via_partition_root** = **false**) - default
    - available since P13
- (**column_name [, ...]**) - available since PG15

CREATE PUBLICATION example

```
CREATE PUBLICATION test
FOR TABLE
 companies(id, name),
 campaigns(id, company_id, name),
 ads(id, campaign_id, name),
 clicks,
 geo_ips,
 impressions
WITH (publish = 'insert, truncate');
```

* companies, campains, ads – replicate only subset of columns

Check publications

```
source=> SELECT * FROM pg_publication_tables;
 pubname | schemaname |  tablename  |                    attnames                        | rowfilter
---------+------------+-------------+----------------------------------------------------+-----------
 test    | public     | ads         | {id,campaign_id,name}                              |
 test    | public     | campaigns   | {id,company_id,name}                               |
 test    | public     | clicks      | {id,ad_id,clicked_at,site_url,cost_per_click_usd,user_ip,user_data}   |
 test    | public     | companies   | {id,name}                                          |
 test    | public     | geo_ips     | {addrs,latlon}                                     |
 test    | public     | impressions | {id,ad_id,seen_at,site_url,cost_per_impression_usd,user_ip,user_data} |
(6 rows)

source=> \dRp+
                            Publication test
 Owner  | All tables | Inserts | Updates | Deletes | Truncates | Via root
--------+------------+---------+---------+---------+-----------+----------
 source | f          | t       | f       | f       | t         | f
Tables:
   "public.ads" (id, campaign_id, name)
   "public.campaigns" (id, company_id, name)
   "public.clicks"
   "public.companies" (id, name)
   "public.geo_ips"
   "public.impressions"
```

ALTER PUBLICATION

ALTER PUBLICATION *name* **ADD** *publication_object* [, ...]

ALTER PUBLICATION *name* **SET** *publication_object* [, ...]

ALTER PUBLICATION *name* **DROP** *publication_object* [, ...]

ALTER PUBLICATION *name* **SET** (*publication_parameter* [= *value*] [, ...])

ALTER PUBLICATION .. ADD/DROP

- ADD or DROP objects to/from PUBLICATION

- `DROP TABLES IN SCHEMA` **will not drop any schema tables** that were specified using `FOR TABLE`/ `ADD TABLE`
- adding tables/schemas to a publication that is already subscribed to will require an **`ALTER SUBSCRIPTION ... REFRESH PUBLICATION`** action on the subscribing side in order to become effective

ALTER PUBLICATION ... SET

1. Set publication parameter
   - **ALTER PUBLICATION** noinsert **SET** (publish = 'update, delete');

2. Set publication object
   - **ALTER PUBLICATION** test **SET TABLE** impressions
     WHERE (**NOT** user_ip << '10.0.0.0/8');
   - will **replace** the list of tables/schemas in the publication with the specified list; **the existing tables/schemas** that were present in the publication **will be removed**.

ALTER PUBLICATION … SET TABLE

```
ALTER PUBLICATION test SET TABLE
 campaigns(id, company_id, name),
 ads(id, campaign_id, name),
 clicks,
 companies(id, name),
 geo_ips,
 impressions WHERE (NOT user_ip << '10.0.0.0/8');
```

- tables without changes (campaigns, ads, clicks, companies, geo_ips) remain as before
- for "impressions" table - it will change oid (requires SUBSCRIPTION REFRESH)
    - Just do DROP and ADD in a transaction if you want to "change" only one table in publication

CREATE SUBSCRIPTION

```
CREATE SUBSCRIPTION subscription_name
    CONNECTION 'conninfo'
    PUBLICATION publication_name [, ...]
    [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

- the invoking user must have the **CREATE** privilege for the **current database**
- also, **pg_create_subscription** role starting from v16.

CREATE SUBSCRIPTION example

```
CREATE SUBSCRIPTION test
CONNECTION 'host=localhost dbname=source user=log_rep_user
  password=pwd'
PUBLICATION test;
```

- log_rep_user – must have replication privilege
  - initial sync requires SELECT privileges on tables
- password – required if log_rep_user is not a superuser!

CREATE SUBSCRIPTION example

```
CREATE SUBSCRIPTION test
  CONNECTION 'host=localhost dbname=source user=log_rep_user
  password=pwd'
  PUBLICATION test;


ERROR:  relation "public.campaigns" does not exist
```

* Schema must exist on the target database!

Copying schema

- TL;DR
  - `$ pg_dump --schema-only | psql`
- However, maybe it is better to grab only tables you are interested in (from subscription)
  - Mind tables that replicate only subset of columns
- Replica identity columns should have indexes.
  - Significantly slows down initial data sync!

Subscription parameters

- connect(boolean) - default true, when false, you must manually create the replication slot, enable and refresh the subscription
- create_slot(boolean) - default true
- slot_name(text) – by default subscription name
- enabled(boolean) - default true
- binary(boolean) - copy data in binary format, default false
    - faster, but there are dragons!
- copy_data(boolean) - copy preexisting data, default true
    - ROW filtering rules are applied!

Subscription parameters

- streaming(boolean) - stream in-progress transactions, default false
- synchronous_commit – overrides corresponding GUC on the subscription apply worker, default off.
- two_phase(boolean) - send prepared transaction at PREPARE time
- disable_on_error(boolean) - disable subscription on error, default false
- origin – default "any", set to "none" if you don't want to receive changes produced by other subscriptions
- failover (boolean)- synchronized replication slot to standby nodes
    - New in PostgreSQL 17!

CREATE SUBSCRIPTION complex example

```
source=> SELECT pg_create_logical_replication_slot('test', 'pgoutput');
pg_create_logical_replication_slot
-----------------------------------
(test,0/1E628270)

(1 row)

dest=> CREATE SUBSCRIPTION test
CONNECTION 'host=localhost dbname=source user=log_rep_user password=pwd'
PUBLICATION test WITH (connect = false);

dest=> ALTER SUBSCRIPTION test ENABLE;
dest=> ALTER SUBSCRIPTION test REFRESH PUBLICATION;
```

Monitoring (on subscriber)

```
dest=> SELECT * FROM pg_stat_subscription;
-[ RECORD 1 ]---------+------------------------------
subid                 | 177014
subname               | test
pid                   | 2341248
leader_pid            |
relid                 |
received_lsn          | 0/2667E8E0
last_msg_send_time    | 2024-10-01 15:13:15.219776+02
last_msg_receipt_time | 2024-10-01 15:13:15.220005+02
latest_end_lsn        | 0/2667E8E0
latest_end_time       | 2024-10-01 15:13:15.219776+02

dest=# SELECT * FROM pg_stat_subscription_stats;
 subid  | subname | apply_error_count | sync_error_count | stats_reset
--------+---------+-------------------+------------------+-------------
 177014 | test    |                 0 |                0 |
(1 row)

dest=# SELECT * FROM pg_replication_origin_status;
 local_id | external_id | remote_lsn | local_lsn
----------+-------------+------------+------------
        1 | pg_177014   | 0/27009F68 | 0/275CF3E0
(1 row)
```

Monitoring (on publisher)

```
source=> SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-------+-----------
slot_name           | test
plugin              | pgoutput
slot_type           | logical
datoid              | 176515
database            | source
temporary           | f
active              | t
active_pid          | 2386804
xmin                |
catalog_xmin        | 2321
restart_lsn         | 0/2667E990
confirmed_flush_lsn | 0/2667E9C8
wal_status          | reserved
safe_wal_size       |
two_phase           | f
conflicting         | f
```

Monitoring (on publisher)

```
source=> SELECT slot_name,
pg_current_wal_insert_lsn() - restart_lsn as lag
FROM pg_replication_slots
WHERE slot_type = 'logical';

 slot_name | lag
-----------+-----
 test      |  56

(1 row)
```

```
source=> SELECT *
FROM pg_stat_replication_slots;


-[ RECORD 1 ]+--------
slot_name     | test
spill_txns    | 0
spill_count   | 0
spill_bytes   | 0
stream_txns   | 0
stream_count  | 0
stream_bytes  | 0
total_txns    | 7
total_bytes   | 3706989
```

Speeding up initial sync

```
source=> CREATE PUBLICATION test
FOR TABLES campaigns, ads, clicks, companies, geo_ips, impressions;
CREATE PUBLICATION
```

Speeding up initial sync

```
$ psql "host=localhost user=log_rep_user
password=pwd dbname=source replication=database"

source=> CREATE_REPLICATION_SLOT test
LOGICAL pgoutput;

-[ RECORD 1 ]----+--------------------
slot_name        | test
consistent_point | 0/3026ED70
snapshot_name    | 00000005-00000D1A-1
output_plugin    | pgoutput
```

And keep the connection alive!

```
$ pg_dump "host=localhost user=log_rep_user
password=pwd dbname=source --no-publication \
--snapshot=00000005-00000D1A-1 \
| psql "host=localhost user=dest dbname=dest"

$ psql "host=localhost user=dest dbname=dest"

dest=> CREATE SUBSCRIPTION test
CONNECTION 'host=localhost dbname=source
user=log_rep_user password=pwd'
PUBLICATION test WITH (connect = false);
WARNING:  subscription was created, but is not
connected
HINT:  To initiate replication, you must manually
create the replication slot, enable the subscription,
and refresh the subscription.

dest=> ALTER SUBSCRIPTION test ENABLE;
ALTER SUBSCRIPTION

dest=> ALTER SUBSCRIPTION test REFRESH PUBLICATION
WITH (COPY_DATA = false);
```

Logical failover slots

- By default, logical replication slot is lost on failover!
  - Except RDS, which implements block-level replication
  - Requires recreating replication slot and resyncing the entire subscription
    - Could be veeeeery costly.
- PostgreSQL v17:
  - **CREATE SUBSCRIPTION** … **WITH (failover)**
  - **SELECT** pg_create_logical_replication_slot(…, **failover => true**);
    - Enabling sync of logical slots to standby nodes
- Older versions:
  - 3rd party extension by EDB: pg_failover_slots
- If destination is a Patroni cluster – configure permanent slots.

Logical failover slots

- Require **hot_standby_feedback = on** on standby nodes!
  - Long-running transactions on standby nodes affect primary!
- Standby nodes must use physical replication slots on primary!
- PostgreSQL v17
  - **sync_replication_slots = on** on standby nodes
  - **primary_conninfo** should contain dbname
  - **synchronized_standby_slots –** list of physical replication slot names that logical WAL sender processes will wait for
    - Subscriber shouldn't see events that weren't replicated to standby nodes yet!

Limitations of logical replication

- DDL is not replicated
    - However, can be solved by applying in a specific order to publisher/subscriber
- Sequences are not replicated
    - Must be adjusted after switch!
- Large Objects are not replicated
- Requires REPLICA INDENTITY on replicated tables
    - Primary/Unique keys on NOT NULL columns; or
    - REPLICA IDENTITY FULL

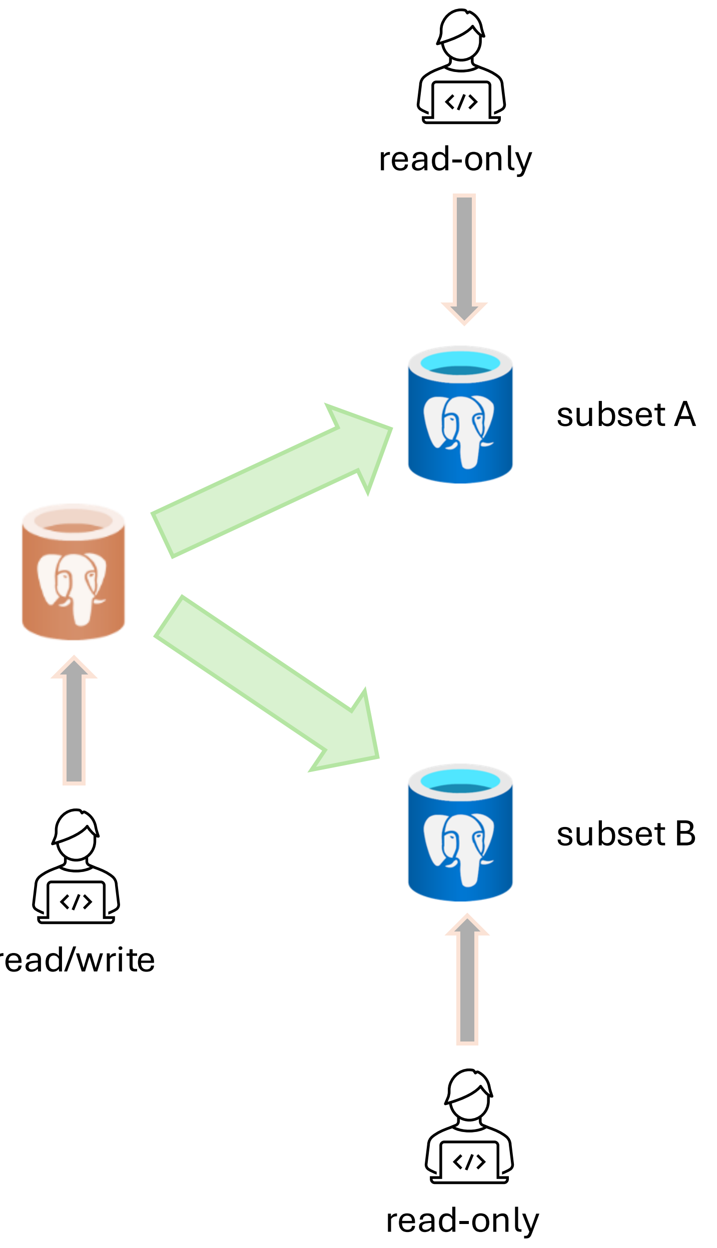
Practical Use Cases and Solution Patterns

Samples and slides will be available here:

https://github.com/scoriani/logicalreplication

Filtered read scale-out

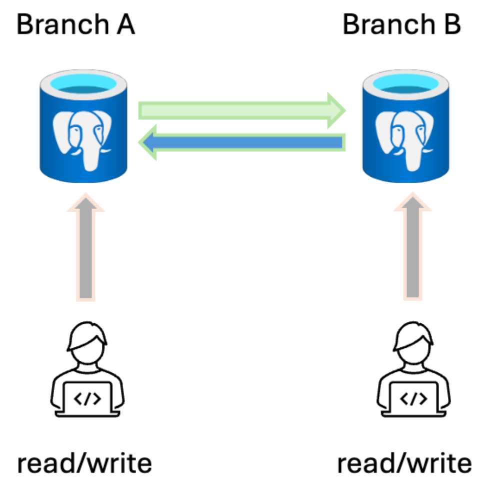
- Why logical replication?
  - Do we really need to scale out all our database content or just a subset of tables?
- Selective Replication
  - Granular control over the replicated data.
  - Choose specific tables, databases, or even filter rows based on defined replication rules
  - Scale different subsets in your database with different scale units (up, down) or isolation levels (e.g. regional-level reporting, etc.)
- Consider performance impact of logical decoding on source system

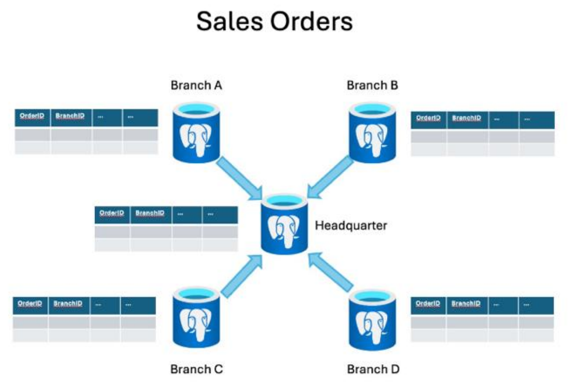


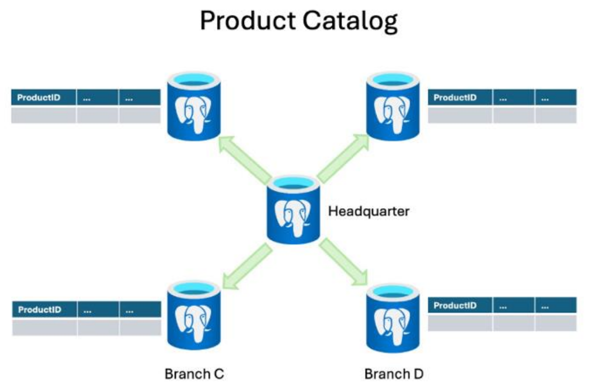
Distributed data apps

Multi-master

Data hub / star topology

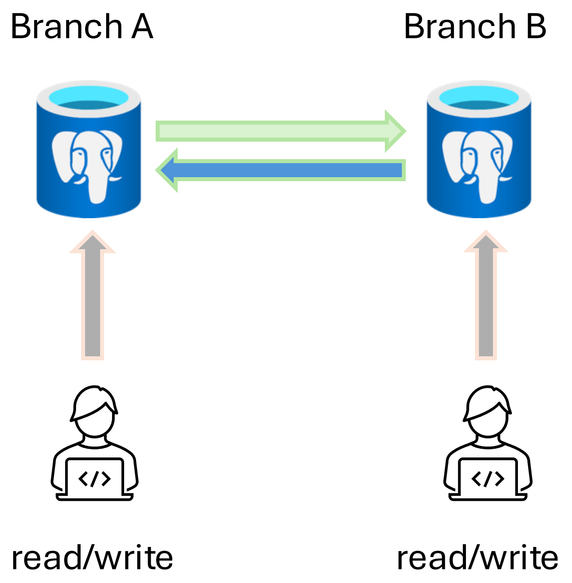






Multi-master (peer to peer replication)

- Typical use cases
  - Multi-site or distributed apps, local writers / global readers, etc.
  - Blue/green deployments
- Relatively simple to implement (from PG16)
  - Bi-directional logical replication
  - Leverage “origin filtering” to identify where a change originated
  - Subscribers only request changes with no origin set to prevent loopback
- Complexity moved at the app level, need to carefully consider topics like
  - Conflicts
  - Sequence management



Multi-master simplified

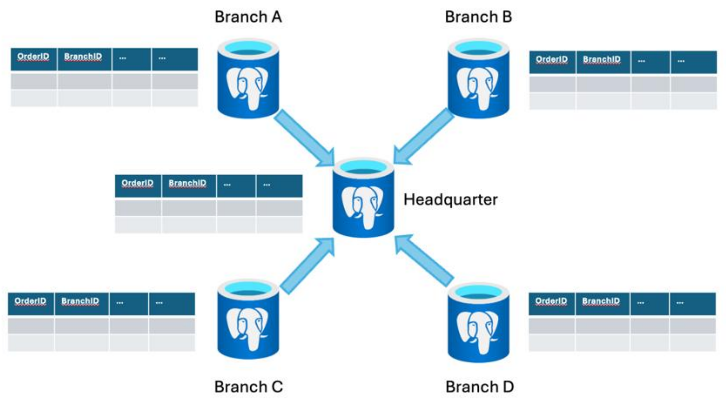
Demo

```
CREATE PUBLICATION pub_headquarter FOR ALL TABLES -- on server 1

CREATE PUBLICATION pub_branch_a FOR ALL TABLES – on server 2


CREATE SUBSCRIPTION sub_to_headquarter -- on server 2

  CONNECTION 'host=<ip_server_1> port=5432 user=<usr> password=<pwd> dbname=<dbname>
connect_timeout=10 sslmode=prefer'

  PUBLICATION pub_headquarter

WITH (connect = true, enabled = true, copy_data=true, create_slot = true, slot_name =
sl_branch_a, origin = 'none');


CREATE SUBSCRIPTION sub_to_branch_a -- on server 1

    CONNECTION 'host=<ip_server_2> port=5432 user=postgres password=Mcp334264! dbname=test
connect_timeout=10 sslmode=prefer'

    PUBLICATION pub_branch_a

    WITH (connect = true, enabled = true, copy_data=false, create_slot = true, slot_name =
sl_headquarter, origin = 'none');
```

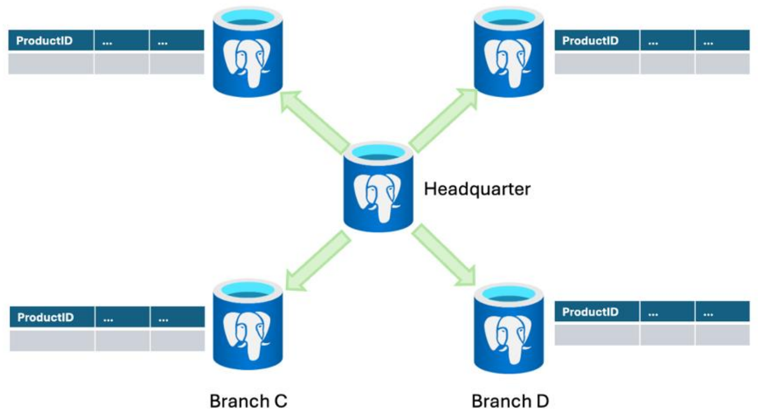
Data hub / star topology

- Typical scenarios
  - Multi-branch solution
  - Data aggregation (e.g. orders, events, etc.)
  - Data distribution (e.g. product catalogs, configs, etc.)
- Clearly define various replication flows
  - Sources
  - Destinations
  - Filters/transformations (not everything, everywhere)
- Minimize conflicts through proper data modeling and application logic (constraints, checks, etc.)

Sales Orders



Product Catalog



Data hub / star topology

Demo

```
CREATE PUBLICATION hq_pub FOR TABLE products; -- on headquarter

SELECT * FROM pg_create_logical_replication_slot('branch_a_slot', 'pgoutput');-- on branch A

SELECT * FROM pg_create_logical_replication_slot('branch_b_slot', 'pgoutput');-- on branch B


CREATE SUBSCRIPTION branch_a_sub CONNECTION 'host=<hq> dbname=sales user=<user>
password=<pwd>' PUBLICATION hq_pub WITH (origin=‘none’); -- on branch A

CREATE SUBSCRIPTION branch_b_sub CONNECTION 'host=<hq> dbname=sales user=<user>
password=<pwd>' PUBLICATION hq_pub WITH (origin=‘none’); -- on branch B


CREATE PUBLICATION branch_a_pub FOR TABLE orders; -- on branch A

CREATE PUBLICATION branch_b_pub FOR TABLE orders; -- on branch B


CREATE SUBSCRIPTION hq_sub_branch_a CONNECTION 'host=<branch1_ip> dbname=sales user=<user>
password=<pwd>' PUBLICATION branch_a_pub WITH (origin=‘none’); -- on headquarter

CREATE SUBSCRIPTION hq_sub_branch_b CONNECTION 'host=<branch1_ip> dbname=sales user=<user>
password=<pwd>'' PUBLICATION branch_b_pub WITH (origin=‘none’); -- on headquarter
```

Real-time change streaming and process integration

Turning data changes to events

Debezium + event stores

Custom consumers (python, .net)

Turning data changes to events

- **Change Data Capture (CDC):** Logical replication allows PostgreSQL to track changes (inserts, updates, deletes) to rows in specific tables and capture these as events. These changes are streamed as logical replication messages.
- **Publish-Subscribe Model:** PostgreSQL logical replication uses a publish-subscribe model where you can define publications (sets of tables to track changes) and subscriptions (receivers of those changes). Subscribed applications can then act on these changes, treating them as events.
- **Custom Event Handling:** Once data changes are captured via logical replication, external services can process them. For example, you can forward these changes to message brokers like Kafka or RabbitMQ, effectively turning database changes into asynchronous events.

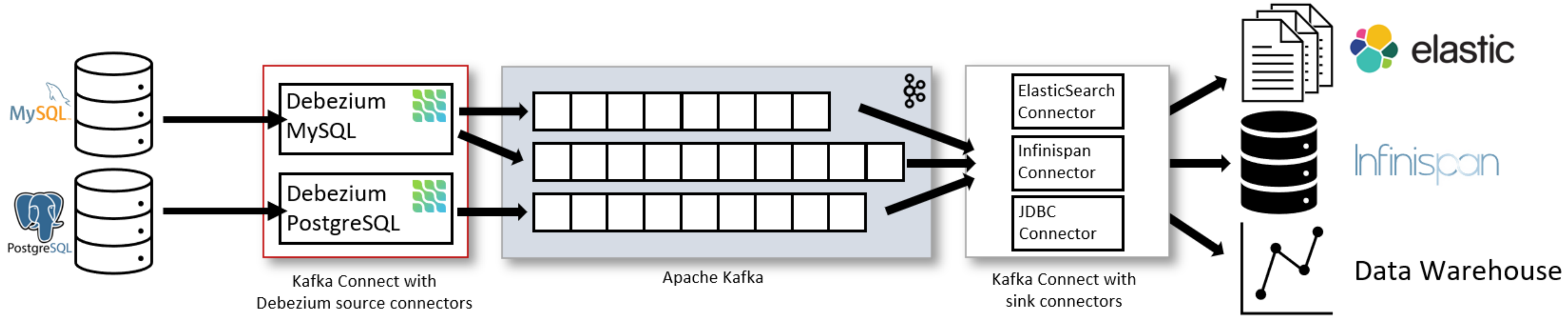
Turning data changes to events

- **Granular Control:** You can choose to replicate only certain changes (e.g., only inserts, or only changes to certain columns) which allows you to trigger specific event types based on granular database activity.
- **Real-time Event Streaming:** Logical replication streams changes in near real-time, allowing external systems to process these events promptly, creating a responsive event-driven architecture.
- **Decoupled Architecture:** Logical replication supports decoupling the database from other systems by making it possible to react to data changes in applications or services without directly querying the database.
- **Integration with Event-Driven Systems:** Changes replicated through logical replication can be easily integrated into event-driven systems, microservices, or CQRS patterns, where database updates are turned into domain events.

An example: Debezium & PostgreSQL

Demo

- Debezium is an open-source distributed platform for implementing heterogenous change data capture patterns.
- Debezium captures row-level changes in a database and streams them to external systems like Apache Kafka or Azure Event Hub.
- Change events can be serialized to different formats like JSON or Apache Avro and then will be sent to one of a variety of messaging infrastructures
- This can be used to trigger real-time events in applications when data changes occur in PostgreSQL.



https://debezium.io/documentation/reference/3.0/architecture.html

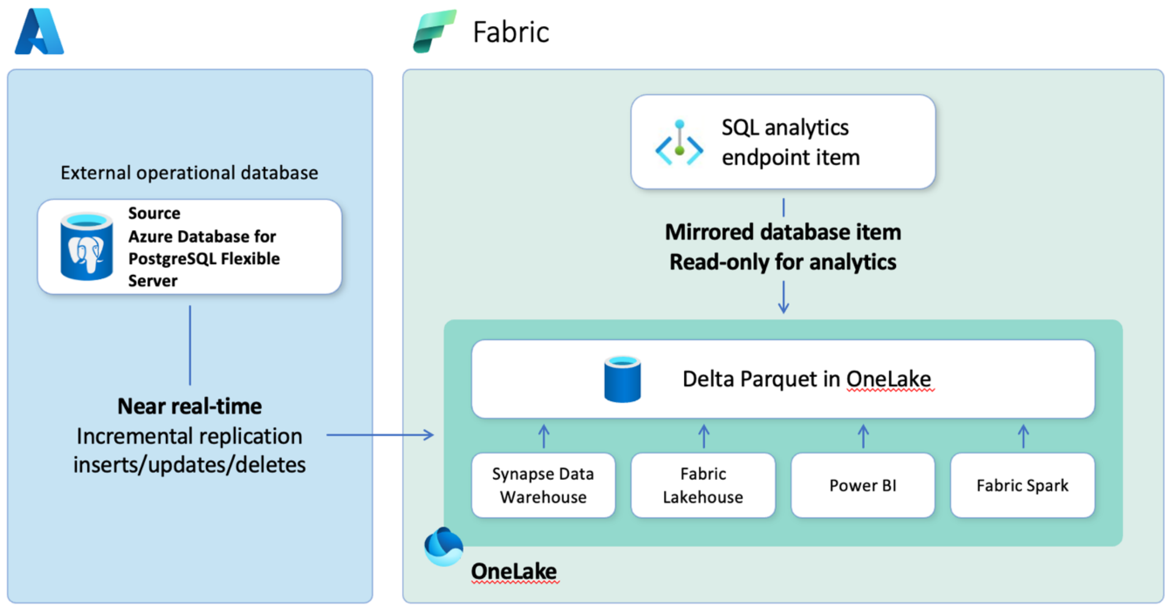
Custom logical replication consumers

Demo

- Custom applications listening to changes in the PostgreSQL database and processing data change events
- **Manage feedback** to ensure WAL logs are not growing excessively.
- **Process changes consistently** in order, and handle `INSERT`, `UPDATE`, and `DELETE` appropriately.
- **Optimize performance** by using batching, asynchronous processing, and monitoring replication lag.
- **Implement fault tolerance** and error handling, ensuring the consumer can restart from where it left off.
- **Ensure security** by using proper authentication, role permissions, and secure connections.
- **Monitor the replication process** using PostgreSQL views like `pg_stat_replication` and `pg_replication_slots`.
- **Plan for schema changes** since logical replication does not replicate schema modifications.

Database Mirroring in Microsoft Fabric

- Based on PostgreSQL logical replication
- Automates server configuration parameters
  - WAL level, max workers, etc.
- Separate transactional and analytical workflows with minimal impact on source systems
- Supports filtered publications



Database online and offline migrations

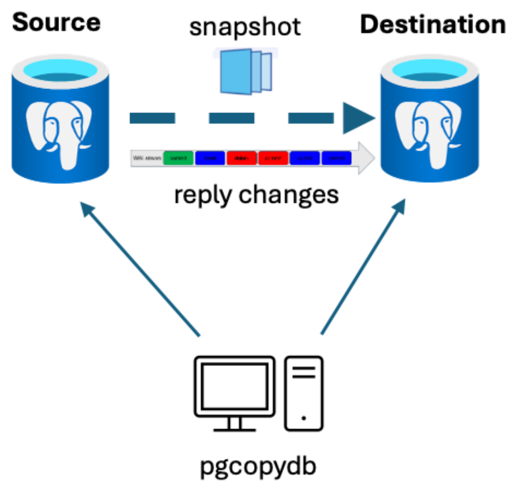
pgcopydb

Managed migration services

Monitoring and alerting

pgcopydb

- pgcopydb is a tool that automates copying a PostgreSQL database to another server
  - Set of commands (clone, follow, compare, cleanup, etc.) modeled around required steps to support major data migration scenarios
  - One-off schema and data transfer
  - Offline/Online migration
  - Schema and data validation
- Incorporates many best practices for copying and moving a database from source to destination



pgcopydb

- pg_dump -jN | pg_restore -jN on steroids
  - Without temporary files
  - Parallel data copy
    - Including same table concurrency!
  - COPY FREEZE (unless same table concurrency)
  - Parallel index creation
  - Change Data Capture (CDC)
    - Enables (almost) zero downtime migration
- Compare schema and data

One-off migrations

Demo

```
$ pgcopydb clone --table-jobs 4 --index-jobs 8
```

- pg_dump and pg_restore for schema
- Concurrent copy for table data
- Concurrent CREATE INDEX
- Recreate PRIMARY KEYS
- Execute VACUUM ANALYZE for target tables
- Sync sequences
- Restore post-data schema (views, functions, etc.)

One-off migrations

- Resume operations if interrupted
- Support for roles with or without passwords
- Can skip extensions and collations
- Supports partial copies

Online migrations

Demo

```
$ pgcopydb clone --follow
```

- Automate Change Data Capture pattern
- Initial snapshot creation
- Setup logical decoding replication slot and origin
- Copy the snapshot like described in previous scenario
- Apply changes to destination database (control with sentinel table)
- Resync sequences
- Once catch-up with data chances can execute the cutover and switch to the new database
- pgcopydb compare schema (and pgcopydb compare data)
- Clean-up CDC and release resources

Limitations and challenges

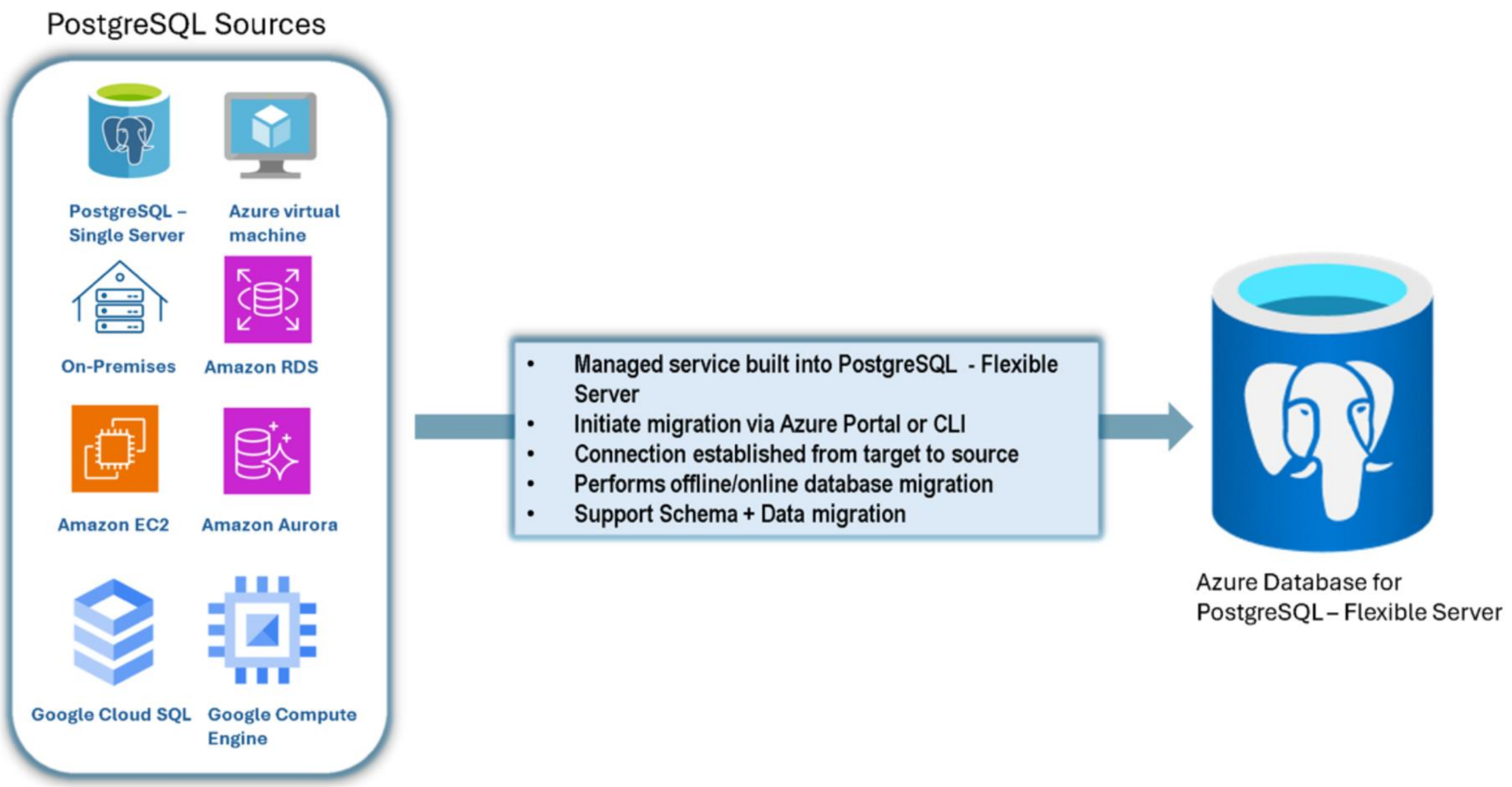
- Replica identity
- DDL not replicated
- Sequence data changes are not replicated
- Unsupported extensions and large object (LOB) handling may need custom solutions.
- Downtime and sync consistency challenges during the migration.
- Large databases may cause performance or locking issues.
- Network latency and throughput impact migration speed.
- Schema changes during migration need manual handling.
- WAL and replication slot management is critical to avoid disk space issues.
- High availability and failover setup needs additional configuration post-migration.

Migrate to/from cloud managed services

- You can use a cloud database role with elevated permissions that includes replication capabilities without full superuser access. Make sure this role has:
  - REPLICATION permission to create logical replication slots.
  - LOGIN rights to authenticate.
  - Access to the relevant tables in the source database to copy data.
- In services like Azure Database for PostgreSQL or Amazon RDS ensure the user has been assigned proper role (azure_pg_admin or equivalent).
  - This is not a full superuser role, but it grants additional privileges, including the ability to create replication slots.

Managed migration services

- Hosted services for migrating from multiple PostgreSQL sources
- Don’t require dedicated resources or complex configurations
- Built on pgcopydb plus cloud-specific integrations



- Managed service built into PostgreSQL - Flexible Server
- Initiate migration via Azure Portal or CLI
- Connection established from target to source
- Performs offline/online database migration
- Support Schema + Data migration

Managed migration services

Demo

- Simplifies the process of moving your PostgreSQL databases to Azure
  - Designed to help you move to Azure Database for PostgreSQL - Flexible Server with ease and confidence
- Use a simple UI Wizard in Azure Portal with minimum migration parameters
- Can be automated via CLI/ARM templates
- Provides a comprehensive Migration experience
  - Includes Schema, Data, Users/Roles, Privileges, etc
- Offline and Online Migration options available
- Validation Feature to Assess Business Rules and Identify Limitations

Monitoring and alerting

Track Replication Lag

- Replication lag indicates how far behind the subscriber is compared to the publisher. Monitoring this helps ensure that the data on the target system is up-to-date.
- **pg_stat_replication**: This view provides information about replication activity, including the lag between the primary (publisher) and standby (subscriber) databases.
- Key columns:
  - `write_lag`, `flush_lag`, and `replay_lag`: These show the lag in different stages of replication.
  - `sent_lsn` and `replay_lsn`: Compare these values to check the amount of data that is sent but not yet replayed on the subscriber.

```
SELECT pid,
application_name,
client_addr,
state,
sent_lsn,
write_lsn,
flush_lsn,
replay_lsn,
(write_lag + flush_lag +
replay_lag) AS total_lag
FROM pg_stat_replication;
```

Subscription Status

- For logical replication, the state of the subscriptions on the subscriber is important to monitor.
- **pg_stat_subscription**: This view shows the status of logical replication subscriptions.
- Key columns:
  - `pid`: The process ID of the subscription worker.
  - `relid`: The ID of the relation (table) being replicated.
  - `received_lsn`: The last received WAL LSN.
  - `last_msg_send_time`, `last_msg_receipt_time`: Timestamps indicating when the last message was sent and received, useful to track replication activity.

```
SELECT
subid,
subname,
pid,
received_lsn,
last_msg_send_time,
last_msg_receipt_time
FROM
pg_stat_subscription;
```

Replication Slots

- Replication slots are used to ensure that WAL segments required for logical replication are retained until they are replicated.
- Monitoring replication slots helps you avoid disk space issues due to accumulating WAL logs.
- **pg_replication_slots**: This view displays information about both physical and logical replication slots.
- Key columns:
  - `slot_name`: The name of the replication slot.
  - `active`: Whether the slot is actively used.
  - `restart_lsn`: The LSN at which WAL files will be retained. Large differences between `restart_lsn` and the current LSN could indicate a lag.
  - `confirmed_flush_lsn`: For logical replication slots, this shows the last WAL record that has been confirmed as flushed by the subscriber.

```
SELECT
  slot_name,
  plugin,
  active,
  restart_lsn,
  confirmed_flush_lsn
FROM
  pg_replication_slots;
```

OS level checks

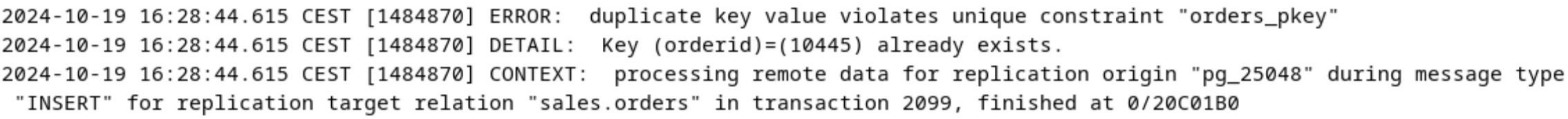
- **Check Errors in Logs**
  - Regularly check PostgreSQL logs for replication-related errors or warnings. Enable detailed logging for replication-related events by adjusting your `log_min_messages` or `log_replication_commands` settings.
- **Monitor Disk Usage**
  - Logical replication may lead to increased disk usage due to WAL retention or large replication slots. Regularly monitor disk space, especially the location where WAL files are stored (`pg_wal` directory).

Monitoring Tools and Dashboards

- **pgwatch2**: another PostgreSQL monitoring tool that provides predefined monitoring dashboards and supports automated alerting and deeper insights into replication activity.
- **Prometheus + Grafana**: Set up custom queries based on PostgreSQL's replication views and display the metrics on Grafana for real-time monitoring with alerting.
  - postgres_exporter
  - set alert thresholds and notification channels in Grafana for alerting
- Many other tools available (e.g. Nagios, custom scripts, commercial tools, etc.)

Monitor replication conflicts

- Data modifications happening when subscriber and publisher are trying to modify the same row
  - Can happen during INSERT/UPDATE/DELETE operations
- PostgreSQL does not automatically log logical replication conflicts by default. However, you can enable conflict detection and logging by adjusting your logging settings.
  - **`log_replication_commands`**: Enable this setting to log every replication command, including conflicts.
  - **`log_error_verbosity`**: Set this to `verbose` to get detailed error messages in the log files, including replication conflicts.
  - **`log_min_messages`**: Set to `WARNING` or lower to ensure that conflict-related warnings are logged.



```
2024-10-19 16:28:44.615 CEST [1484870] ERROR:  duplicate key value violates unique constraint "orders_pkey"
2024-10-19 16:28:44.615 CEST [1484870] DETAIL:  Key (orderid)=(10445) already exists.
2024-10-19 16:28:44.615 CEST [1484870] CONTEXT:  processing remote data for replication origin "pg_25048" during message type
 "INSERT" for replication target relation "sales.orders" in transaction 2099, finished at 0/20C01B0
```

- Monitor delays in replication stats, increase in delays may indicate conflicts
- Automate conflict detection by setting up tools like Promethrus or Nagios to search for conflict related keywords in logs.

Best Practices

Management and maintenance

- **Plan Migrations Carefully**: Define a detailed migration plan that includes replication steps, validation, and rollback strategies. Ensure that both source and target systems are properly synchronized before switching traffic.
- **Minimal Downtime**: Use logical replication for zero-downtime migrations by replicating data continuously until the target is up-to-date, then switch over to the new database.
- **Consistent Schema Changes**: Ensure the schema is consistent between publisher and subscriber databases. Any schema changes (e.g., adding columns) should be carefully managed and applied to both sides to prevent replication issues.
- **Monitor Replication Lag**: Continuously monitor replication lag to detect potential synchronization problems between the source and target databases. Tools like `pg_stat_replication` help track the replication process.
- **Graceful Switchover**: When switching over to the target database, ensure that all in-flight transactions have been replicated to avoid data loss.
- **Disabling Replication After Migration**: Once the migration is complete, ensure logical replication is disabled to prevent unnecessary overhead and to avoid confusion.
- **Manage your sequences:** logical replication doesn't manage sequences so, if you're not using a tool like pgcopydb, you'll need to manually re-sync your sequences.

Performance and scalability

- **Filter Publications**: Only replicate necessary tables or rows by using filtered publications. This reduces the amount of data replicated, lowering the resource overhead on the source database.
- **Tune WAL Parameters**: Logical replication relies on Write-Ahead Logs (WAL). Ensure that the WAL parameters (`max_wal_size`, `wal_level`, etc.) are properly tuned for your workload to avoid performance bottlenecks.
- **Parallelism**: Utilize PostgreSQL's ability to create multiple logical replication subscriptions to scale out large data migrations by splitting the load across multiple workers or nodes.
- **Manage Resource Usage**: Logical replication consumes CPU, memory, and disk resources on both the source and target systems. Monitor resource usage closely and adjust performance parameters (`maintenance_work_mem`, `work_mem`, etc.) to ensure replication doesn't degrade system performance.
- **Batch Data Loading**: For large datasets, consider batching the initial data load separately before enabling real-time logical replication to minimize load on the system during the migration phase.
- **Vacuum and Index Maintenance**: Regularly vacuum both source and target databases to prevent bloat and ensure that indexes are maintained during the replication process.

Security

- **Use Encrypted Connections**: Ensure that replication data is transferred over secure connections using SSL/TLS to protect against eavesdropping or man-in-the-middle attacks.
- **Replication Role with Minimal Privileges**: The replication user should be assigned only the necessary permissions to avoid security risks. Use a dedicated PostgreSQL role with `REPLICATION` privileges for logical replication, but restrict access as much as possible.
- **Data Masking**: If sensitive data is being replicated, consider masking or encrypting sensitive fields before replication to prevent exposure of critical information.
- **Audit and Logging**: Enable auditing and detailed logging on both publisher and subscriber databases to track any unauthorized access or suspicious activity related to replication.
- **Regular Access Reviews**: Periodically review the access rights of the replication user, and revoke any unnecessary privileges after the migration is complete.
- **Network Security**: Ensure that only trusted IP addresses or hosts can access the replication ports on both the source and target databases by configuring firewall rules and network access controls.

Thank you!

Please provide your feedback