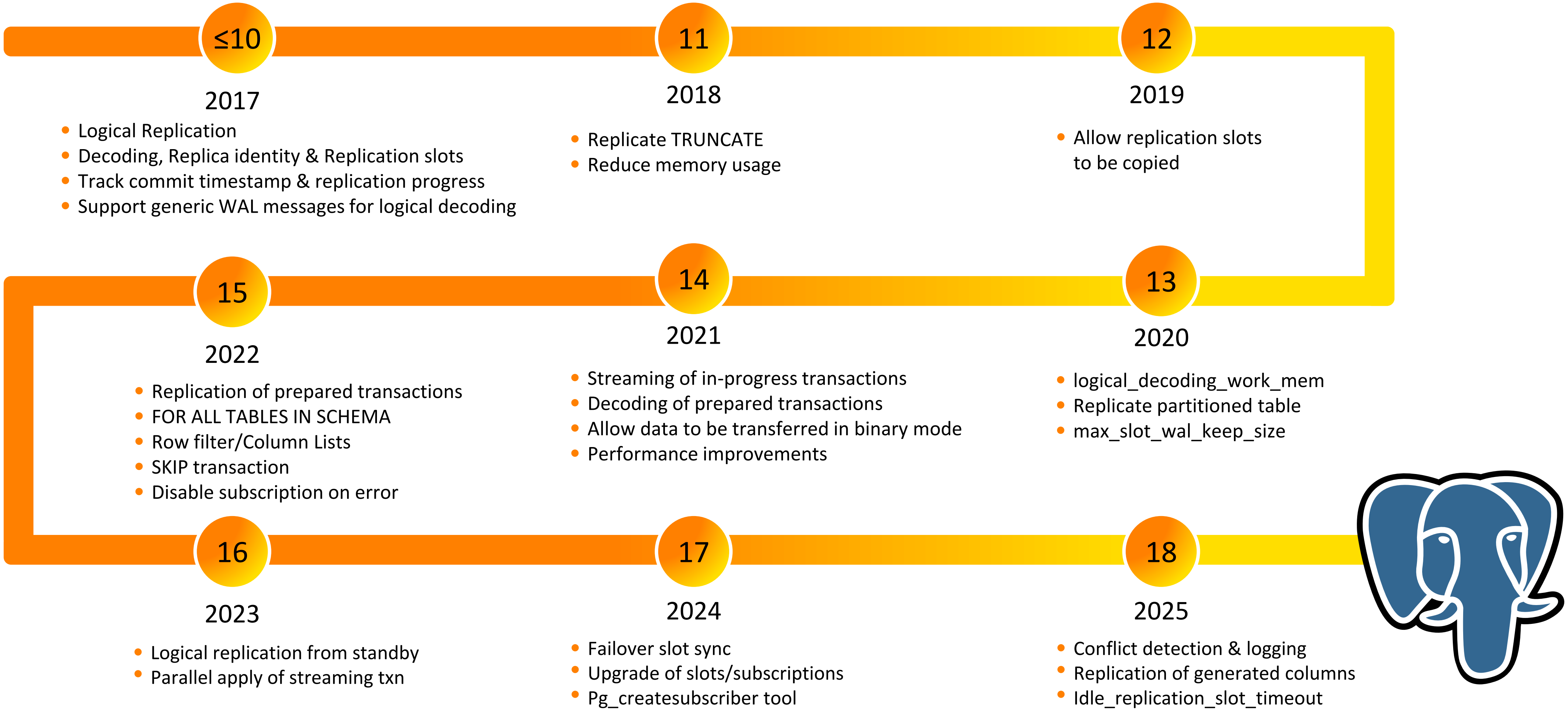
Unlocking new possibilities:

The evolving landscape of PostgreSQL Logical Replication

Amit Kapila
PostgreSQL Committer and Major Contributor
Fujitsu

Evolution of logical replication in PostgreSQL



≤10

2017

- Logical Replication
- Decoding, Replica identity & Replication slots
- Track commit timestamp & replication progress
- Support generic WAL messages for logical decoding

11

2018

- Replicate TRUNCATE
- Reduce memory usage

12

2019

- Allow replication slots to be copied

13

2020

- logical_decoding_work_mem
- Replicate partitioned table
- max_slot_wal_keep_size

14

2021

- Streaming of in-progress transactions
- Decoding of prepared transactions
- Allow data to be transferred in binary mode
- Performance improvements

15

2022

- Replication of prepared transactions
- FOR ALL TABLES IN SCHEMA
- Row filter/Column Lists
- SKIP transaction
- Disable subscription on error

16

2023

- Logical replication from standby
- Parallel apply of streaming txn

17

2024

- Failover slot sync
- Upgrade of slots/subscriptions
- Pg_createsubscriber tool

18

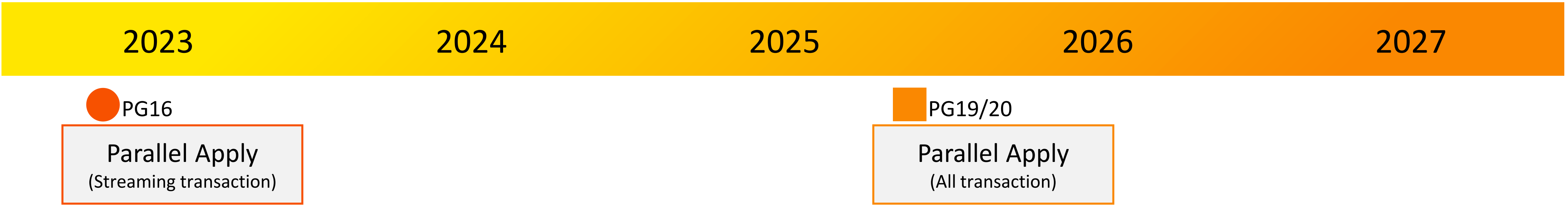
2025

- Conflict detection & logging
- Replication of generated columns
- Idle_replication_slot_timeout

Key empowering features

Parallel Apply (streaming transactions)

Performance



● Released ■ In plan ▲ Future TBD

Key empowering features

	2023	2024	2025	2026	2027
Performance	● PG16 Parallel Apply (Streaming transaction)			■ PG19/20 Parallel Apply (All transaction)	
High Availability		● PG17 Failover Slots ● PG17 Upgrade (Logical Replica)		■ PG19 Sequence Sync ■ PG19 Online wal_level change	

Parallel Apply (streaming transactions)

Failover Logical Slots

● Released ■ In plan ▲ Future TBD

Key empowering features

Parallel Apply (streaming transactions)

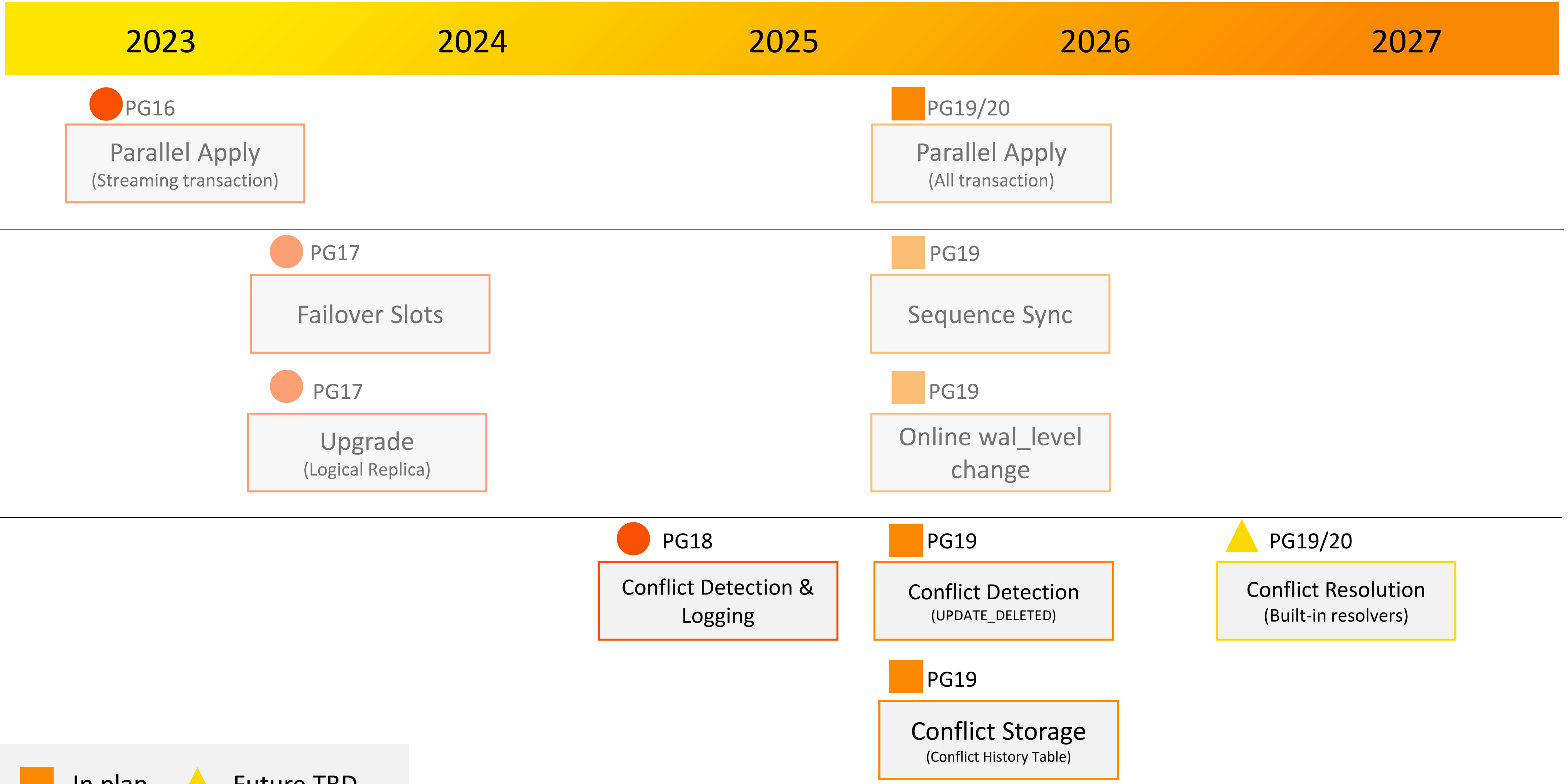
Performance

Failover Logical Slots

High Availability

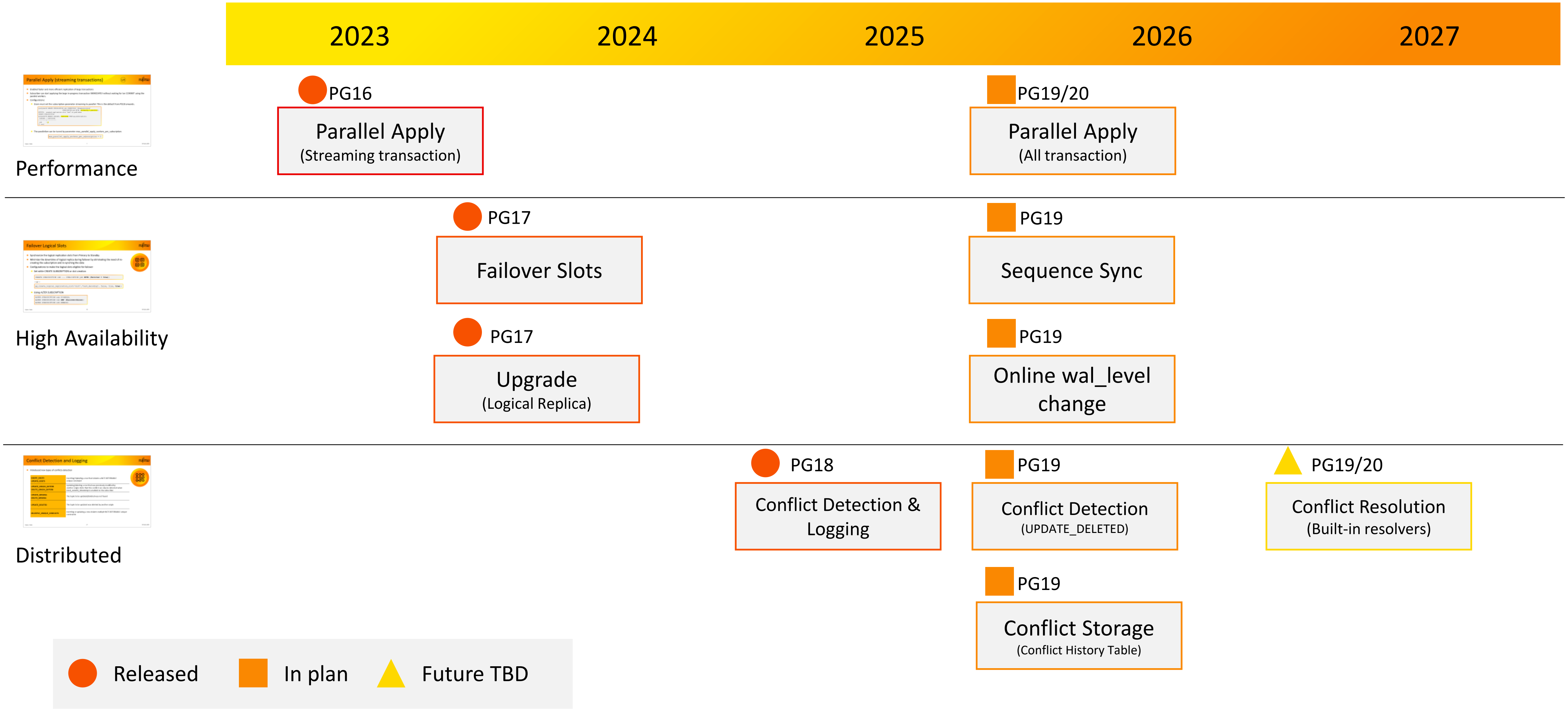
Conflict Detection and Logging

Distributed



● Released ■ In plan ▲ Future TBD

Key empowering features



2023 | 2024 | 2025 | 2026 | 2027

Performance

- ● PG16 – Parallel Apply (Streaming transaction)
- ■ PG19/20 – Parallel Apply (All transaction)

High Availability

- ● PG17 – Failover Slots
- ● PG17 – Upgrade (Logical Replica)
- ■ PG19 – Sequence Sync
- ■ PG19 – Online wal_level change

Distributed

- ● PG18 – Conflict Detection & Logging
- ■ PG19 – Conflict Detection (UPDATE_DELETED)
- ■ PG19 – Conflict Storage (Conflict History Table)
- ▲ PG19/20 – Conflict Resolution (Built-in resolvers)

● Released ■ In plan ▲ Future TBD

Parallel Apply (streaming transactions)
Failover Logical Slots
Conflict Detection and Logging

Parallel Apply (streaming transactions)

1/4

- Enabled faster and more efficient replication of large transactions
- Subscriber can start applying the large in-progress transaction IMMEDIATLY without waiting for txn COMMIT using the parallel workers.
- Configurations:
  - Users must set the subscription parameter *streaming* to *parallel*. This is the default from PG18 onwards.

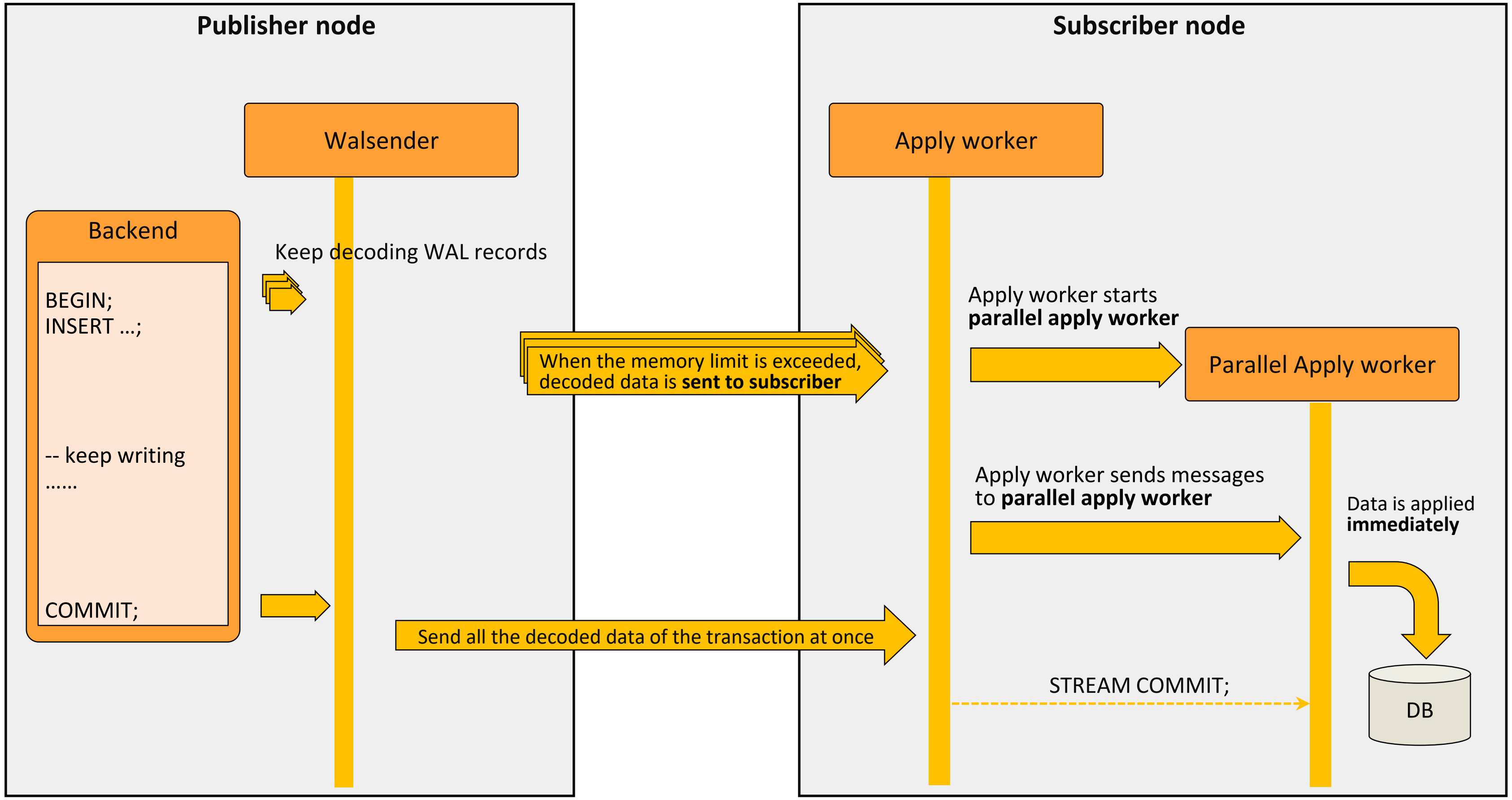
```
postgres=# CREATE SUBSCRIPTION sub CONNECTION 'dbname=postgres’
                            PUBLICATION pub WITH (streaming = parallel);
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
postgres=# SELECT subname, substream FROM pg_subscription;
 subname | substream
---------+-----------
 sub     | p
(1 row)
```

  - The parallelism can be tuned by parameter *max_parallel_apply_workers_per_subscription.*

```
max_parallel_apply_workers_per_subscription = 5
```

Parallel Apply (streaming transactions)

2/4


Publisher node
Walsender
Backend
BEGIN;
INSERT …;
-- keep writing
……
COMMIT;
Keep decoding WAL records
When the memory limit is exceeded, decoded data is **sent to subscriber**
Send all the decoded data of the transaction at once
Subscriber node
Apply worker
Apply worker starts **parallel apply worker**
Parallel Apply worker
Apply worker sends messages to **parallel apply worker**
Data is applied **immediately**
STREAM COMMIT;
DB

Parallel Apply (streaming transactions) 3/4

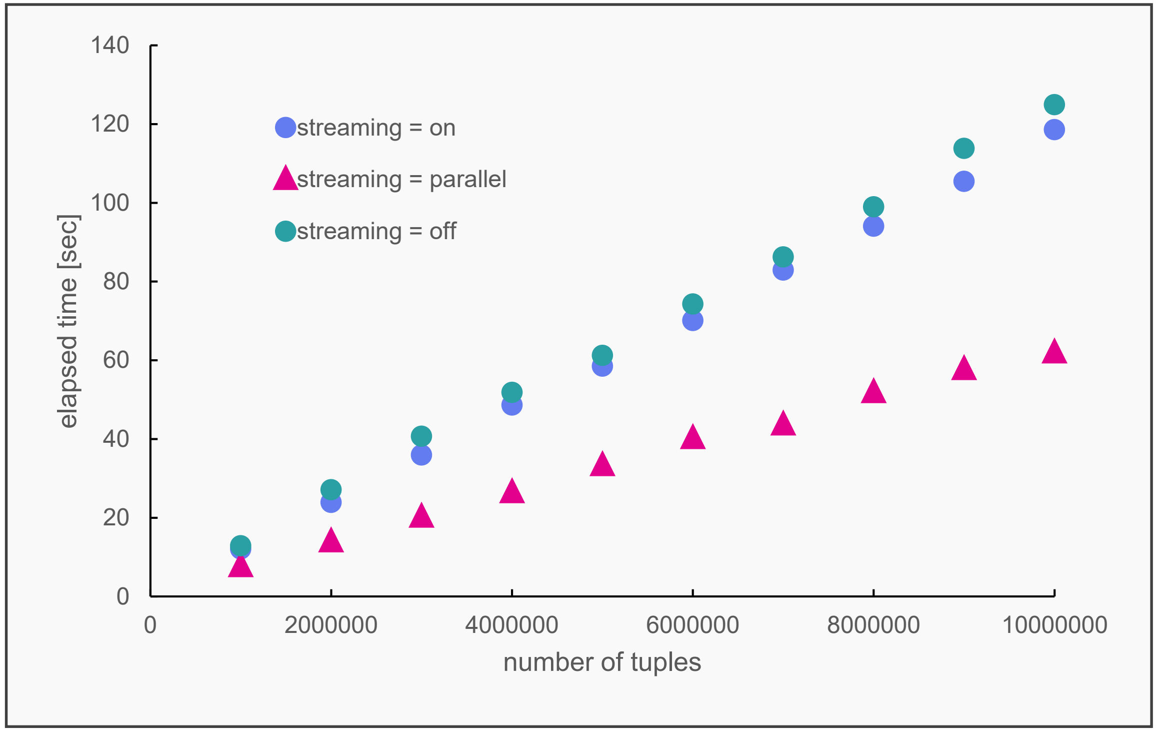
FUJITSU

Performance improvements (data vs time)

- Synchronous logical replication system
- **Setup:** Publisher → Subscriber on the same machine with:

```
shared_buffers = 100GB
checkpoint_timeout = 30min
max_wal_size = 20GB
min_wal_size = 10GB
autovacuum = off
synchronous_commit = remote_apply
logical_decoding_work_mem = 30MB
```

- **Workload:** 1 million to 10 million inserts on a single table
- **Time measured:** Time taken from insert to commit
- **Results:**
  - Elapsed time improved by **~2X** with parallel apply.
  - As the number of tuples increases, parallel apply provides greater benefit since transactions are applied immediately, significantly reducing replication lag between publisher and subscriber.



Machine details

Intel(R) Xeon(R) CPU E7-4890 v2 @ 2.80GHz
CPU(s) :88 cores, - 503 GiB RAM

Parallel Apply (streaming transactions)

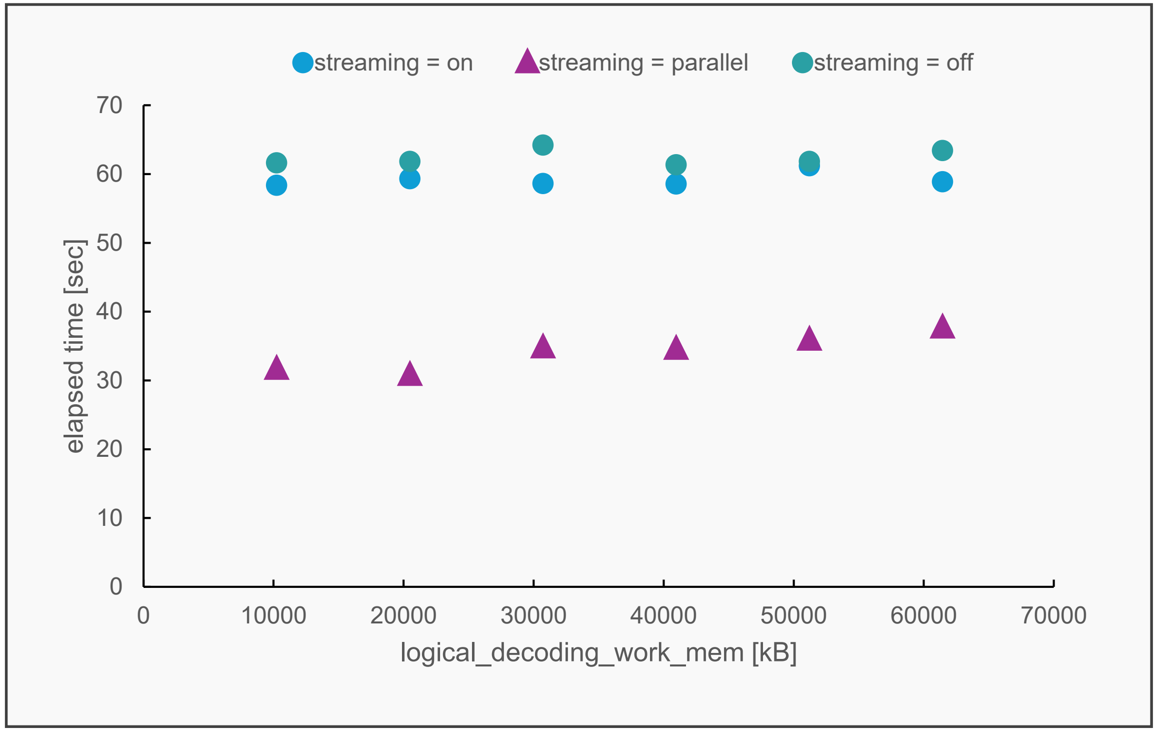
4/4

Performance improvements (memory vs time)

- Synchronous logical replication system
- **Setup:** Publisher → Subscriber on the same machine with:

```
shared_buffers = 100GB
checkpoint_timeout = 30min
max_wal_size = 20GB
min_wal_size = 10GB
autovacuum = off
synchronous_commit=remote_apply
```

- **Workload:** 5 million inserts on a single table with 10 MB to 60MB logical decoding work memory
- **Time measured:** Time taken from insert to commit
- **Results:**
  - Elapsed time improved by **~2X** with parallel apply.
  - As the logical decoding work memory increases, parallel apply provides greater benefit since transactions are applied immediately, significantly reducing replication lag between publisher and subscriber.



Parallel Apply (all transactions) 1/5

Background & design goals

- **Current Limitation**: PostgreSQL's single apply process for a subscription can become a **bottleneck** in high-throughput systems.
- **Existing Parallelism**: Only large streaming transactions (*streaming=parallel*) can use multiple workers.
- **Proposal:** Extend parallelism to non-streaming, frequent small transactions for better replication throughput.

Challenges

- Must respect transaction dependencies to avoid failures or deadlocks.
- Must maintain replication consistency even with out-of-order commits.

Design goals

- Apply independent transactions concurrently.
- Detect & serialize dependent transactions.
- Support safe, restart-consistent replication.

Parallel Apply (all transactions)

2/5

Core mechanisms

Dependency detection

- Hash Table Tracking: (RelationId, ReplicaIdentity, Unique Keys) → track XID ownership.
- Unique Key Conflicts: Prevent incorrect ordering that causes constraint violations.
- Foreign Keys: Check FK references across tables (e.g., owner → car).
- Safe Exclusions: Don’t parallelize xacts on tables with user-defined triggers/complex constraints in v1.

Coordination by the leader Apply Worker

- Detects dependencies.
- Dispatches independent transactions to parallel workers and let them execute.
- Enforces ordering where required.

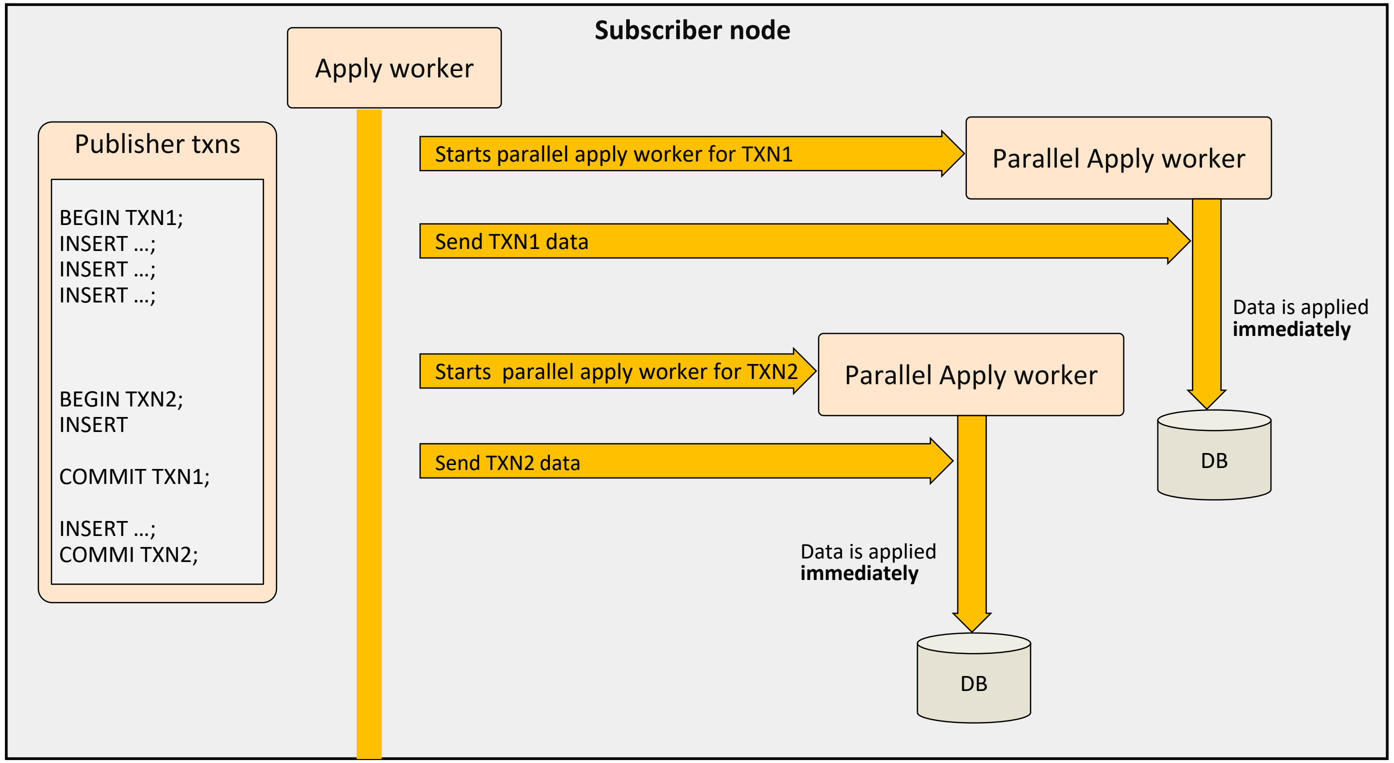
Replication progress tracking

- Track the lowest, highest, and list of commit LSNs to ensure correct recovery after crash/restart:
- During recovery, resume safely from lowest_remote_lsn and skip already applied commits

Optional: Preserve commit order for users relying on app-level integrity (no explicit PK/FK).

Parallel Apply (all transactions)

3/5



- **Improved Throughput:** Multiple apply workers process transactions in parallel, reducing lag between publisher and subscriber.
- **Reduced Latency:** Transactions can commit earlier without waiting for others (unless dependencies exist).
- **Scalability:** System can scale with workload by increasing parallel apply workers.
- **Dependency Management:** Built-in mechanism ensures correctness by delaying only conflicting transactions.
- **Seamless Fallback:** If no parallel worker is available, leader apply worker takes over to maintain progress.

Parallel Apply (all transactions)

4/5

FUJITSU

Replication time improvements

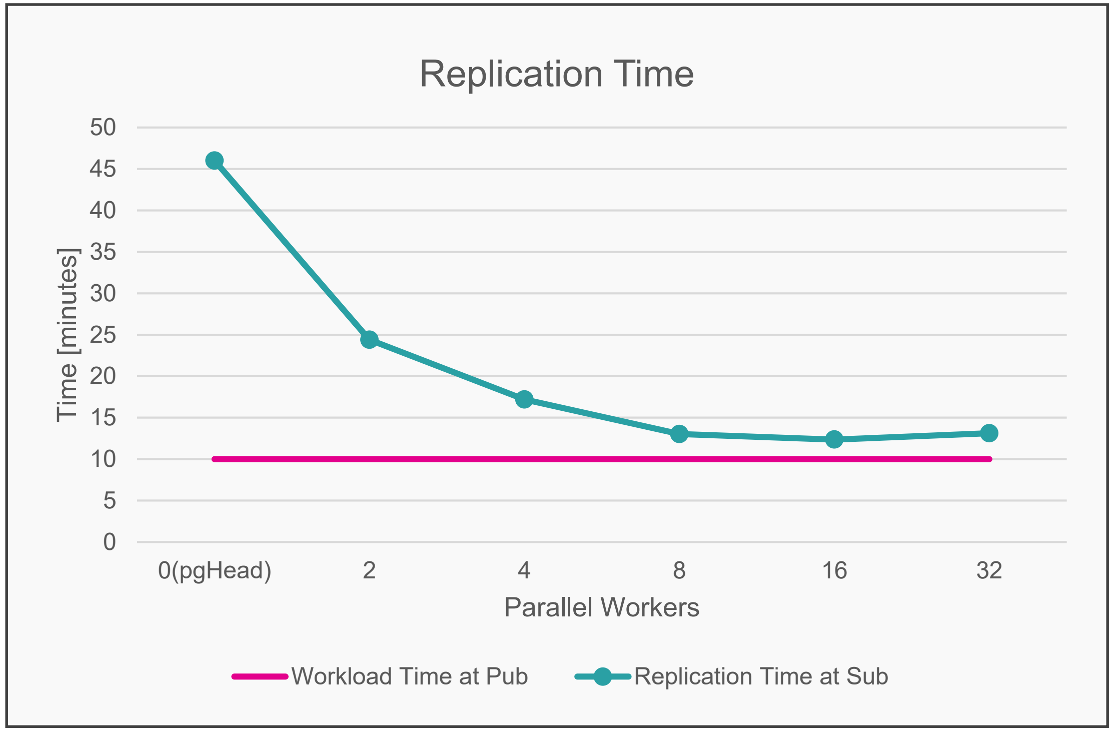
- Replication time improvements measured using pgbench workloads.
- **Setup:** Publisher → Subscriber on the same machine with:

```
shared_buffers = 30GB
max_wal_size = 20GB
min_wal_size = 10GB
autovacuum = off
```

- **Workload:** pgbench (read-write) executed on the publisher with:

```
Scale = 300
Clients = 40
Duration = 10 minutes
```

- **Time measured:** Time taken by the apply worker to replicate all changes to the subscriber.
- **Results:**
  - On **HEAD** (no parallel workers): replication of publisher's workload takes **~45 minutes**.
  - With the patch: replication lag reduces sharply as the number of workers increases.
  - With **16+ workers**, replication completes in only **~12 minutes**.



Test details: Logical replication time improvements

Parallel Apply (all transactions)

5/5

Synchronous replication throughput improvements

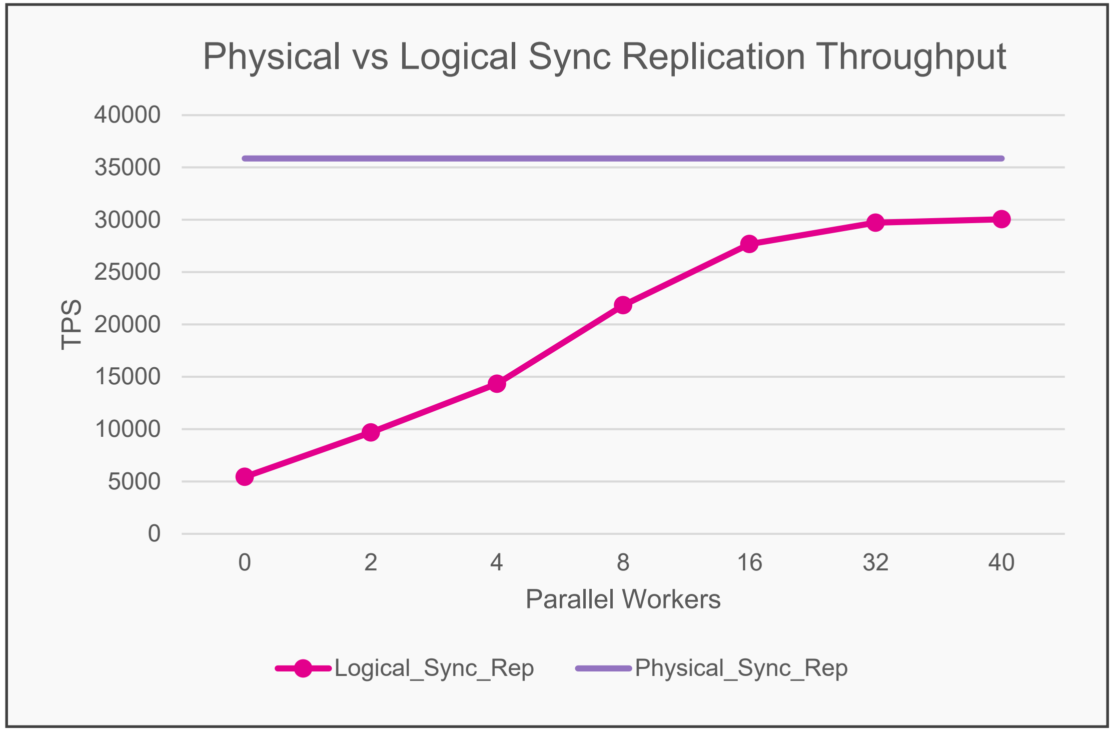
- Compared the throughput of physical synchronous replication vs logical synchronous replication
- **Setup:** Primary → Standby and Publisher → Subscriber on the same machine with:

```
shared_buffers = 30GB
max_wal_size = 20GB
min_wal_size = 10GB
autovacuum = off
```

- **Workload:** pgbench (read-write) executed on the primary/publisher with:

```
Scale = 300
Clients = 40
Duration = 20 minutes
```

- Measured pgbench throughput when *synchronous_commit='remote_apply'* in both physical and logical cases.
- **Results:**
  - On **HEAD** (no parallel workers), logical replication throughput is **5-6× lower** than physical replication.
  - With the patch, increasing the number of parallel workers reduces apply lag and boosts publisher throughput.
  - With **40 workers**, throughput **improved by ~5×**, reaching very close to physical synchronous replication throughput.



Test details: [Physical vs logical synchronous replication]()

Failover Logical Slots

- Synchronize the logical replication slots from Primary to Standby
- Minimize the downtime of logical replica during failover by eliminating the need of re-creating the subscription and re-synching the data
- Configurations to make the logical slots eligible for failover
  - Set while CREATE SUBSCRIPTION or slot creation

```
CREATE SUBSCRIPTION sub ... PUBLICATION pub WITH (failover = true);
```

– or –

```
pg_create_logical_replication_slot('slot','test_decoding', false, true, true);
```

  - Using ALTER SUBSCRIPTION

```
ALTER SUBSCRIPTION sub DISABLE;
ALTER SUBSCRIPTION sub SET (failover=false);
ALTER SUBSCRIPTION sub ENABLE;
```

Sync the Slots to standby

1/2

- Standby server configurations:
  - A physical replication slot (`primary_slot_name`) must be configured
  - `hot_standby_feedback` must be enabled
  - A valid `dbname` must be specified in the `primary_conninfo`

- Primary server configuration
  - `synchronized_standby_slots` must be set to ensure that logical replica waits for the WAL to be first received by the physical standby

Sync the Slots to standby 2/2

- Methods available
  1. Using pg_sync_replication_slots() API (Manual)
     - Connects to primary → fetches eligible slots.
     - Creates/updates synced slots; drops obsolete ones.
     - **Not allowed** if auto-sync is enabled.

  2. Automatic Synchronization
     - Controlled by *sync_replication_slots* GUC
     - Background slotsync Worker: Periodically syncs eligible slots from primary.
     - Adaptive Timing: Sync cycle gap ≥200ms, nap up to 30s based on activity.

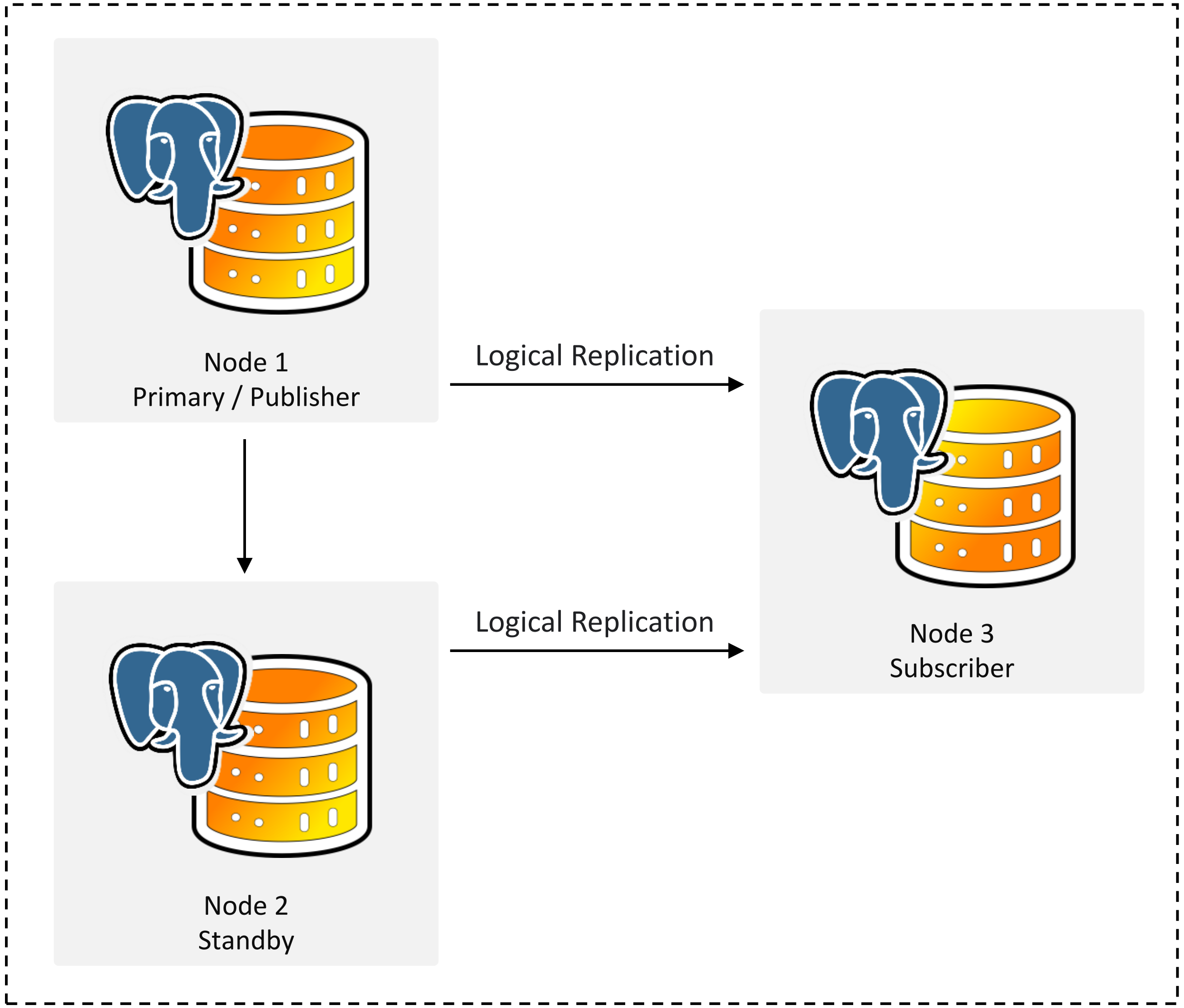
≤PG16: Availability of logical replica on failover

2 Stop the primary node

```
$ pg_ctl -D node1 stop
```

3 Promote the standby

```
$ pg_ctl -D node2 promote
```



1 Disable all subscriptions

```
ALTER SUBSCRIPTION sub_xx DISABLE
```

4 Truncate all tables

```
TRUNCATE XXX…
```

5 Re-create subscriptions

```
DROP SUBSCRIPTION sub_xx
CREATE SUBSCRIPTION sub_xx
```

⌛ It takes time to recopy TB/PB again

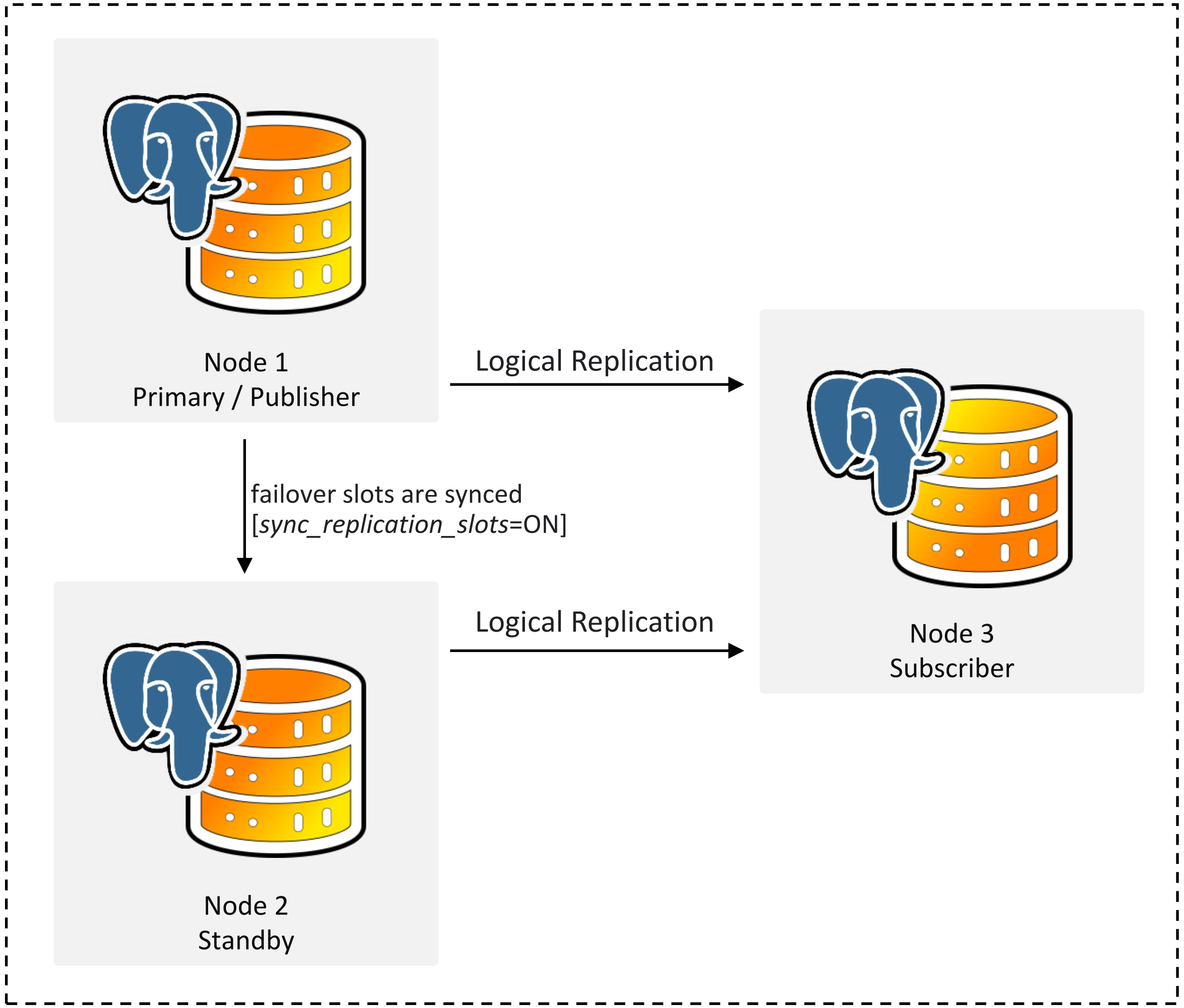
PG17: HA of logical replica during failover

1 Stop the primary node

```
$ pg_ctl -D node1 stop
```

2 Promote the standby

```
$ pg_ctl -D node2 promote
```



3 Change connection info for subscriptions to the new primary

```
ALTER SUBSCRIPTION sub_xxx
connection 'node2…'…
```

Online Upgrade of Replica

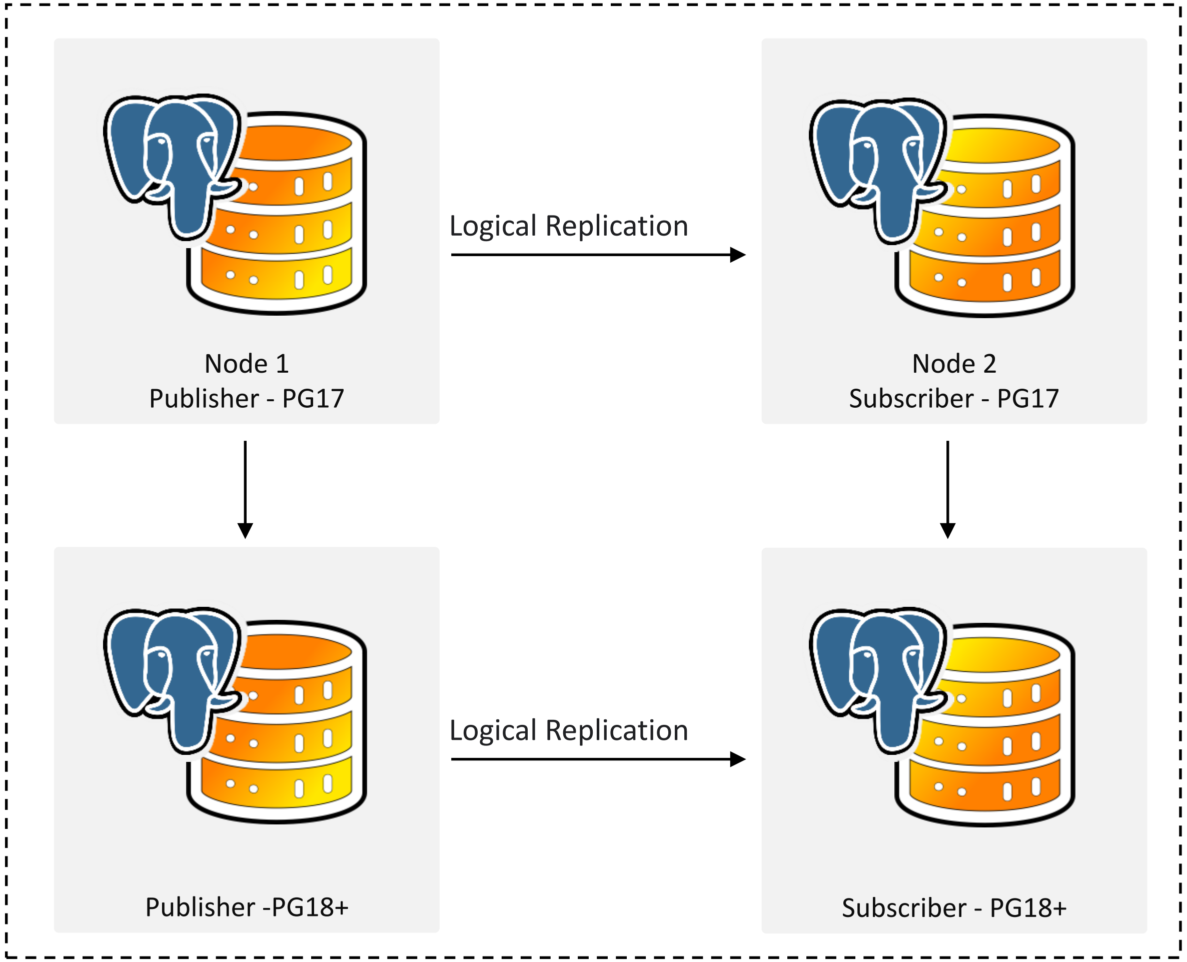
- **Before PG17:**
  - Major upgrades interrupted replication.
  - Slots were lost and writes had to be paused until setups were manually reconstructed.
- PG17 or later supports upgrading logical replication setups *(publisher/subscriber)* **without blocking writes**, without needing to recreate replication slots or reattach subscriptions.
- **Key Features:**
  - **Logical slot migration:** slots can be migrated to the upgraded publisher
  - **Preserved subscription state:** subscriber retains subscription metadata, allowing re-enable after upgrade without data loss.
- Use case:
  - **Case 1:** Online upgrade of a logical replication cluster.
  - **Case 2:** Upgrade of a streaming replica by **temporarily converting it to a logical replica**, enabling seamless upgrade.

Case1: Upgrade logical replication cluster (PG17)

2 Stop and upgrade

```
$ pg_ctl -D node1 stop
$ pg_upgrade -d node1 …
```

Here we want to upgrade logical replication clusters from PG17 to PG18+…



1 Disable all subscriptions

```
ALTER SUBSCRIPTION
sub1_node1_node2 DISABLE
```

3 Stop and upgrade

```
$ pg_ctl -D node2 stop
$ pg_upgrade -d node2 …
```

4 Define tables that were created in node1 during upgrade

```
CREATE TABLE …
```

(if needed)

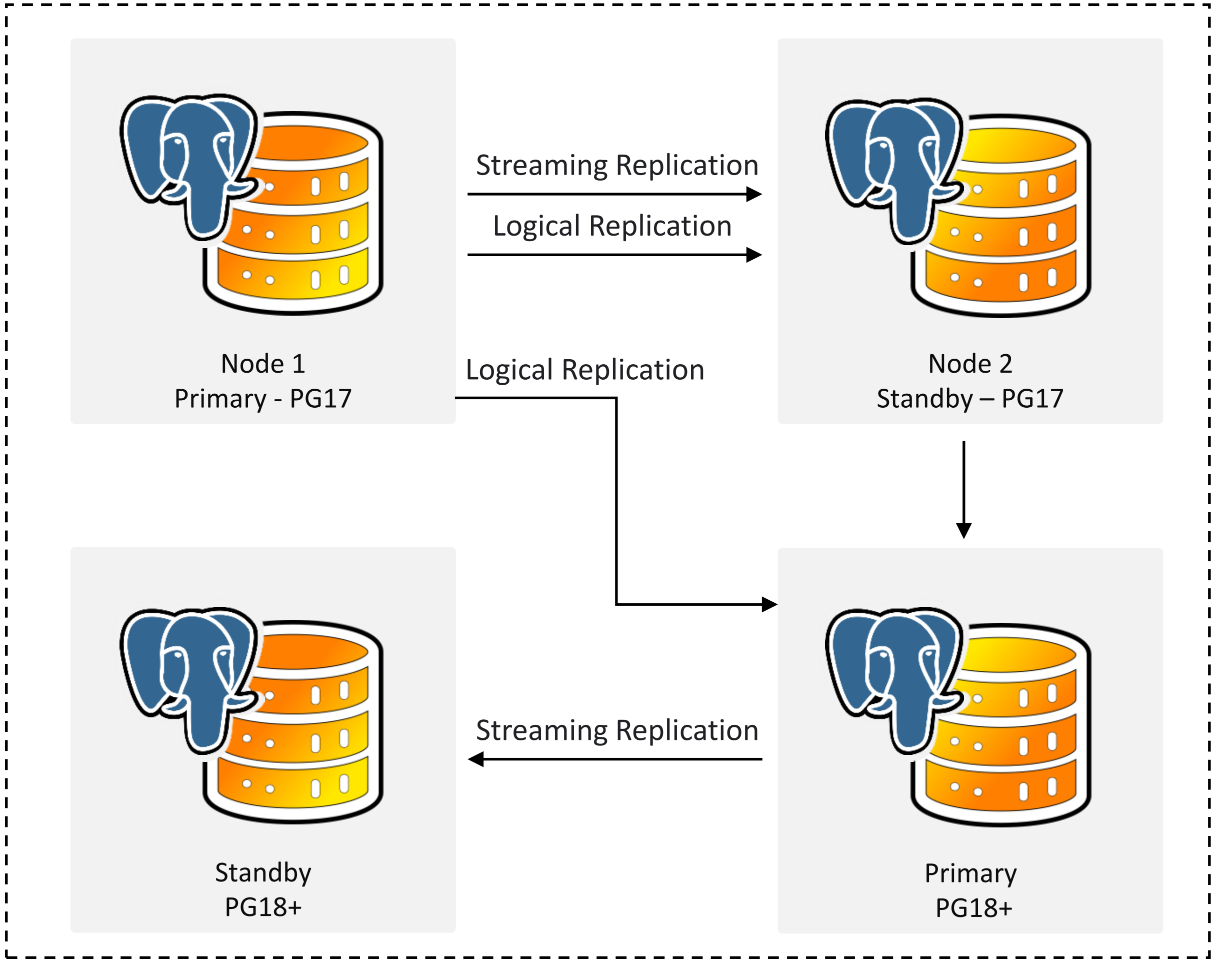
5 Enable and refresh subscriptions for all tables

```
ALTER SUBSCRIPTIONS sub_node1_node2
 ENABLE
ALTER SUBSCRIPTIONS sub_node1_node2
 REFRESH PUBLICATION
```

- One node’s upgrade won’t block writes on other
- Subscriber can resume replication after upgrade without any data loss

Case2: Upgrade streaming replication cluster (PG17)

Here we want to upgrade streaming replication clusters from PG17 to PG18+…



1 Run pg_createsubscriber

```
$ pg_createsubscriber -D node2 …
```

2 Stop and upgrade

```
$ pg_ctl -D node2 stop
$ pg_upgrade -d node2 …
```

3 Stop Node1

```
$ pg_ctl -D node1 stop
```

4 Run pg_basebackup

```
$ pg_basebackup -D node2_upgraded -R…
```

5 Drop subscriptions

```
DROP SUBSCRIPTION subXX…
```

Replication of Sequences

1/2

- Currently logical replication is used for upgrade of physical replication cluster, but one of the problem is that the table data will be replicated but the sequences are still at the initial values, requiring some custom solution that moves the sequences forward enough to prevent duplicities after upgrade.
- Sequences can now be synced from publisher to subscriber.
- New command to support:
  - `CREATE PUBLICATION for ALL SEQUENCES;`
    - A subscription created for publication for “ALL SEQUENCES” allow subscriber to fetch and sync the sequences from publisher
  - `ALTER SUBSCRIPTION <sub> REFRESH REPLICATION SEQUENCES;`
    - REFRESH re-synchronize all the existing sequence value with the updated sequence value from the publisher

Replication of Sequences (2/2)

- The apply worker spawns a **sequence sync worker** to fetch & copy sequences from publisher.
- The page LSN of the sequence from the publisher is also captured and stored in pg_subscription_rel.
- This LSN will reflect the state of the sequence at the time of synchronization.
- By comparing the current LSN of the sequence on the publisher (via pg_sequence_state()) with the stored LSN on the subscriber, users can detect if the sequence has advanced and is now out-of-sync.
- This comparison will help determine whether resynchronization is needed for a given sequence.

Use case: Primarily designed to simplify **upgrades**.

Online wal_level change

- Motivation
  - Users often run DB instance with *wal_level = replica* for lower WAL volume and better performance.
  - Enabling logical decoding later (for migration, analytics, CDC) currently requires a server restart, causing downtime.
- Design Highlights
  - Enable logical decoding **dynamically** in replica mode whenever a logical slot is created, eliminating the need for a server restart when switching to logical replication.
  - New GUC *effective_wal_level* to monitor the actual WAL level in effect.
  - Ensure all running sessions start writing WAL at the logical level before decoding is allowed.
  - Keep *wal_level = logical* for backward compatibility.

Use case: Run in replica mode for minimal overhead, then enable logical decoding on demand without restart, for high availability.

Conflict Detection and Logging

- Introduced new types of conflicts detection

INSERT_EXISTS **UPDATE_EXISTS**	Inserting/Updating a row that violates a NOT DEFERRABLE unique constraint
UPDATE_ORIGIN_DIFFERS **DELETE_ORIGIN_DIFFERS**	Updating/deleting a row that was previously modified by another origin. Note that this conflict can only be detected when *track_commit_timestamp* is enabled on the subscriber
UPDATE_MISSING **DELETE_MISSING**	The tuple to be updated/deleted was not found
UPDATE_DELETED	The tuple to be updated was deleted by another origin
MULTIPLE_UNIQUE_CONFLICTS	Inserting or updating a row violates multiple NOT DEFERRABLE unique constraints

UPDATE_DELETED

Required to ensure eventual consistency in a bidirectional setup.

- Example: If a remote update arrives after a local delete, the system must detect the delete and compare commit times to decide the action.
- With only update_missing, users can't distinguish between these two cases:
  - When the INSERT (convert update->insert) has to be applied.
  - When the UPDATE has to be skipped.

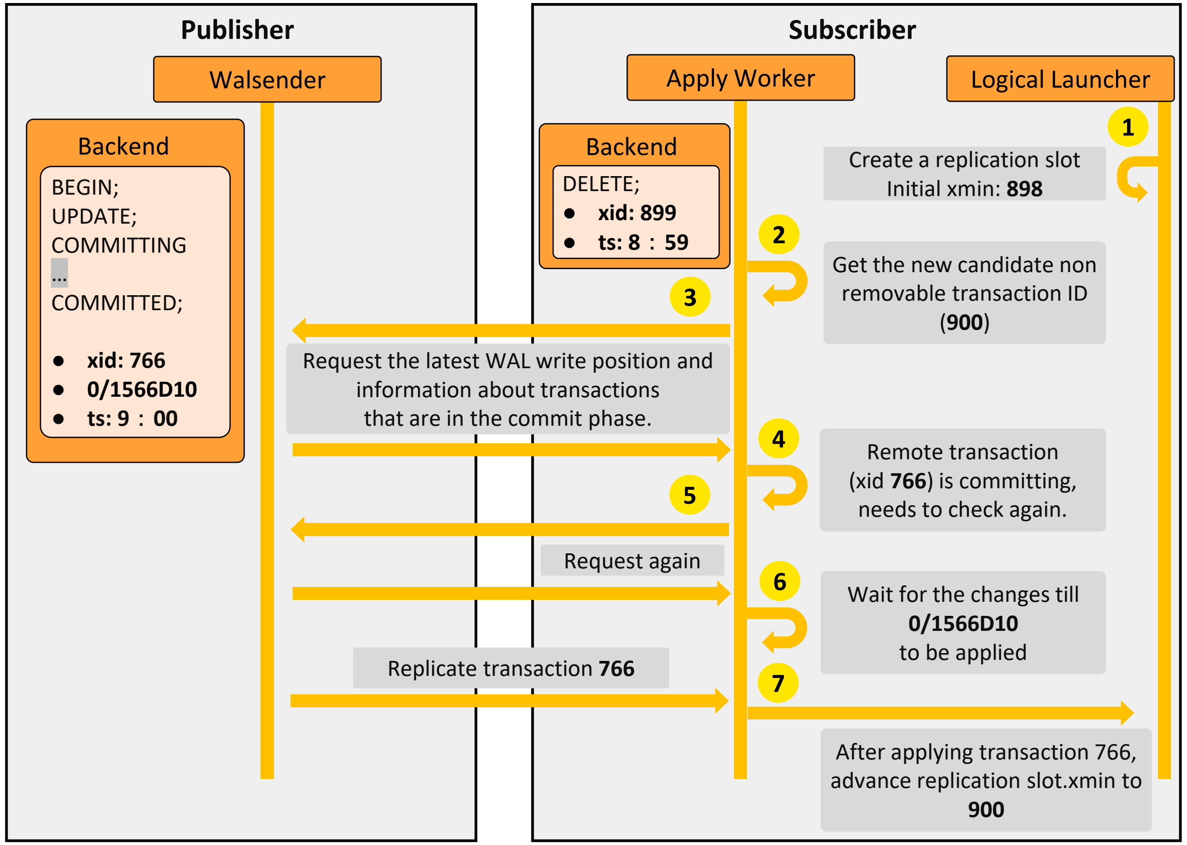
How?

- If remote update < local delete → skip update.
- If remote update > local delete → re-insert row (last-update-wins).
- To support this, deleted tuples must be retained until all concurrent remote txns are flushed locally.

Configuration (as subscription options):

- *retain_dead_tuples* → to enable detection.
- *max_retention_duration* → limit how long dead tuples are kept if apply worker lags; 0 (default) = no limit.

Retention Duration


Publisher
Walsender
Backend
BEGIN;
UPDATE;
COMMITTING
...
COMMITTED;
● xid: 766
● 0/1566D10
● ts: 9 : 00
Subscriber
Apply Worker
Logical Launcher
Backend
DELETE;
● xid: 899
● ts: 8 : 59
1
Create a replication slot
Initial xmin: 898
2
Get the new candidate non removable transaction ID (900)
3
Request the latest WAL write position and information about transactions that are in the commit phase.
4
Remote transaction (xid 766) is committing, needs to check again.
5
Request again
6
Wait for the changes till 0/1566D10 to be applied
Replicate transaction 766
7
After applying transaction 766, advance replication slot.xmin to 900

- A replication slot is created to retain dead tuples.
- Dead tuples are retained until all remote transactions that occurred concurrently with the tuple DELETE are applied and flushed locally.
- The apply worker requests the walsender to get the latest WAL write position and information about transactions that are committing.
- Do not proceed if there are concurrent remote transactions that are committing, because these transactions might have been assigned an earlier commit timestamp.

Conflict Resolution Framework

- Automatic handling of conflicts based on configuration & type.
- Built-in resolvers reduce user effort
- Can be used to achieve eventual consistency in bidirectional replication.
- Flexible control: users specify resolvers per conflict type in subscription DDL.

```
postgres=# CREATE SUBSCRIPTION <subname> CONNECTION <conninfo> PUBLICATION <pubname> CONFLICT RESOLVER
           (conflict_type1 = resolver1, ...);
CREATE SUBSCRIPTION

postgres=# ALTER SUBSCRIPTION <subname> CONFLICT RESOLVER
           (conflict_type1 = resolver1, ...);
ALTER SUBSCRIPTION

postgres=# ALTER SUBSCRIPTION <subname> RESET CONFLICT RESOLVER ALL
ALTER SUBSCRIPTION
```

- The default resolvers for all conflict types are aligned with existing PostgreSQL behavior for consistency.

Built-in Conflict Resolvers

Resolver	Description	Applicable for conflicts
apply_remote	The remote change is applied	● INSERT_EXISTS ● UPDATE_EXISTS ● MULTIPLE_UNIQUE_CONFLICTS ● UPDATE_ORIGIN_DIFFERS ● DELETE_ORIGIN_DIFFERS
apply_or_skip	Remote change is converted to INSERT and is applied. If the complete row cannot be constructed from the info provided by the publisher, then the change is skipped	● UPDATE_MISSING ● UPDATE_DELETED
apply_or_error	Similar to above but raise an ERROR if cannot convert the remote change to INSERT	● UPDATE_MISSING ● UPDATE_DELETED
keep_local	The local version of row is retained	● INSERT_EXISTS ● UPDATE_EXISTS ● MULTIPLE_UNIQUE_CONFLICTS ● UPDATE_ORIGIN_DIFFERS ● DELETE_ORIGIN_DIFFERS
last_update_wins	The change with later commit timestamp wins.	● INSERT_EXISTS ● UPDATE_EXISTS ● UPDATE_ORIGIN_DIFFERS ● DELETE_ORIGIN_DIFFERS ● UPDATE_DELETED
error	Replication is stopped; manual action is needed.	Can be used for any conflict type.

Viewing Conflict History 1/2

FUJITSU

- Currently conflict only visible in logs; hard to analyze
- With this feature a new user specified table will be created to store all the details of a conflict:
  - table and the tuple details
  - conflict type
  - remote XID, LSN and commit_time_stamp.
  - local XID, LSN and commit_time_stamp
  - origin info
- A new subscription option is added to create the user specified table as –

```
postgres=# CREATE SUBSCRIPTION sub CONNECTION 'dbname=postgres port=5432' PUBLICATION
pub WITH(conflict_log_table=myschema.my_conflict_table);
```

Viewing Conflict History 2/2

- Conflict reported in LOG

```
LOG:  conflict detected on relation "public.test": conflict=update_origin_differs

DETAIL:  Updating the row that was modified locally in transaction 776 at 2025-09-22 17:14:48.106542+05:30.
        Existing local row (1, 10); remote row (1, 20); replica identity (a)=(1).

CONTEXT:  processing remote data for replication origin "pg_16391" during message type "UPDATE" for replication target relation
"public.test" in transaction 770, finished at 0/01766E48
```

- Conflict details stored in the table:

```
postgres-# select * from myschema.my_conflict_table;
-[ RECORD 1 ]-----+---------------------------------
relid             | 16385
local_xid         | 776
remote_xid        | 770
local_lsn         | 0/00000000
remote_commit_lsn | 0/01766E48
local_commit_ts   | 2025-09-22 17:14:48.106542+05:30
remote_commit_ts  | 2025-09-22 17:14:53.090079+05:30
table_schema      | public
table_name        | test
conflict_type     | update_origin_differs
local_origin      |
remote_origin     | pg_16391
key_tuple         | {"a":1,"b":20}
local_tuple       | {"a":1,"b":10}
remote_tuple      | {"a":1,"b":20}
```

Features TBD

- Logical replication of DDL commands
- Logical replication of LOB
- Logical replication statistics
- Node initialization, synchronization, resynchronization, pause / resume
- Performance
  - Decoding
  - Lag catch up

Summary

Replication performance	High Availability of Logical Replica	Distributed writes
Parallel Apply: streaming txns	Failover logical slots	Conflict detection & logging
Parallel Apply: all txns	Upgrade of logical replication nodes	Built-in conflict resolvers
	Synchronization of sequences	Conflict storage
	Online wal_level change	

Thank You!