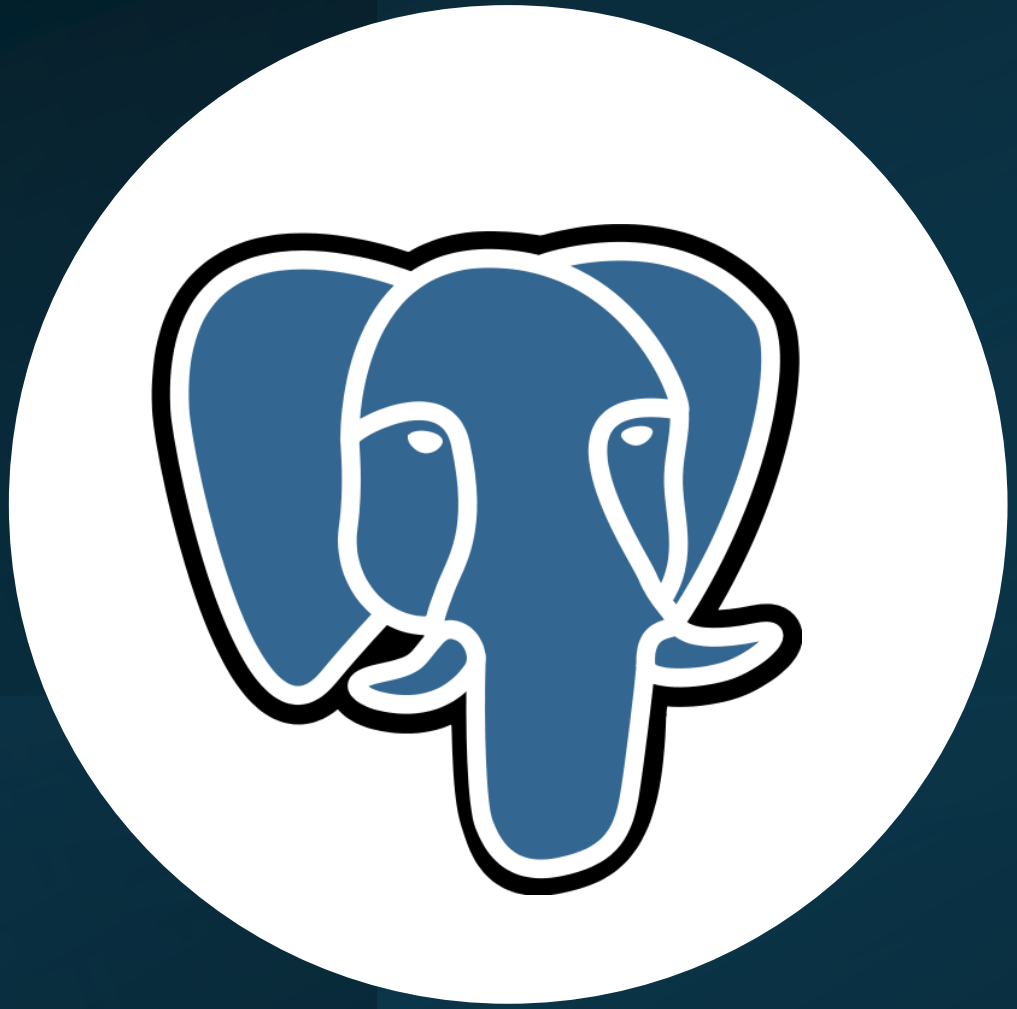
Scaling Logical Replication: Parallel Apply & Centralized Decoding



PGConf.dev Vancouver, Canada

(20 May 2026)

Speakers

```
postgres=# ¥x
Expanded display is on.

postgres=# SELECT * FROM speakers ;
-[ RECORD 1 ]----------------------
name        | Amit Kapila
affiliation | Apple Inc.
title       | Committer
-[ RECORD 2 ]----------------------
name        | Hayato Kuroda
affiliation | Fujitsu Ltd.
title       | Recognized Contributor
```

What We'll Cover

- Why Logical Replication Needs to Scale
- Current bottlenecks
  - Subscriber side
  - Publisher side
- Scaling solutions
  - Parallel apply (ordered commits)
  - Parallel apply (out-of-order commits)
  - Centralized decoding

Why Logical Replication Needs to Scale

Performance comparisons between two replications types

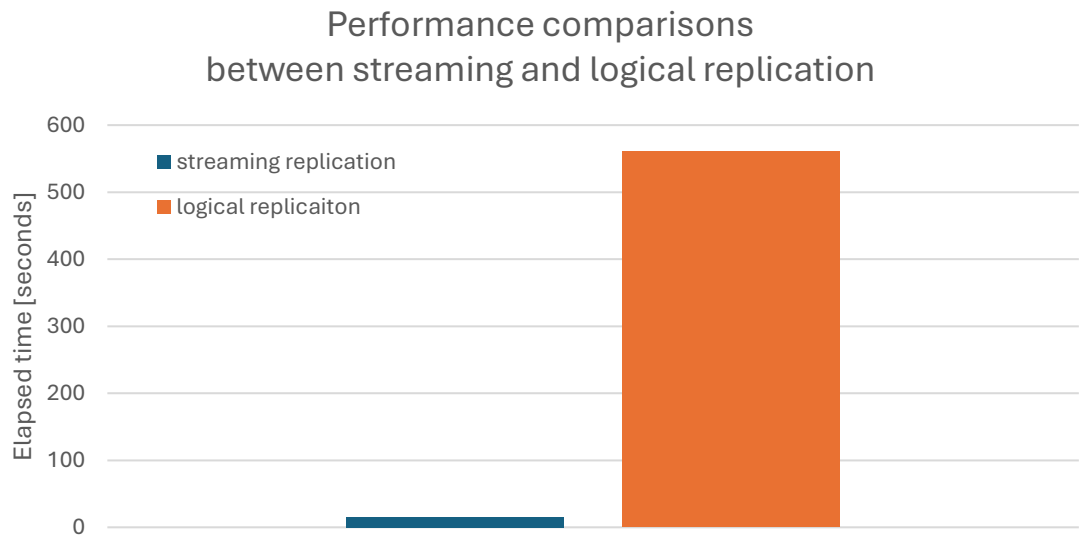
- **Setup**: streaming replication or logical replication system on the same machine with:

```
shared_buffers = 30GB
max_wal_size = 10GB
min_wal_size = 5GB
autovacuum = off
```

- **Workload**: pgbench (tpcb-like) executed on the primary/publisher with:

```
Scale=40
Clients/jobs=30
Duration 5 minutes
```

- **Measurement**: measured till all changes are replicated
- **Result**: Logical replication took about **34 times longer** than streaming replication.



Machine Details

- Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz
- 56 logical cores
- 251 GB RAM

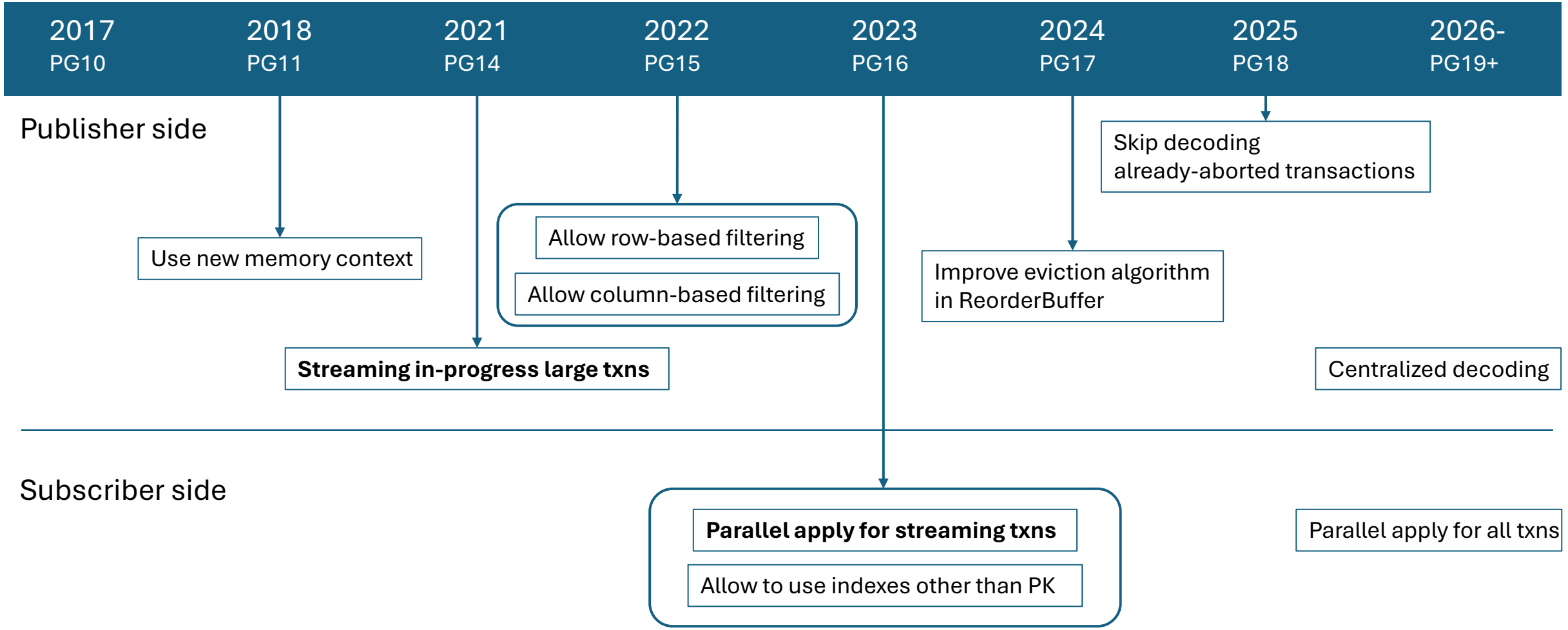
Performance comparisons between two replications types

- Logical replication is slower than the streaming replication **by design**

Streaming Replication		Logical Replication
○ Primary simply sends WAL records ○ Replicated messages can be simply applied	Vs.	○ Publisher must **decode WAL records**. This needs to check system catalogs to convert physical->logical information ○ Subscriber must apply replicated messages **using complex logic** – reading system catalogs, checking constraints, search existing tuples etc. ○ All changes must be handled per tuple

- Performance issues are a well-known limitation of logical replication.
- It has improved in each release, but not enough yet

Brief History of Performance Improvements in Logical replication



Existing Known Bottlenecks

Subscriber side

- Bottleneck #1: Transactions are applied in series

Publisher side

- Bottleneck #2: Repeated WAL Decoding on the Primary

Improve Performance on Subscriber

Bottleneck #1: Transactions are applied in series

- A single apply worker receives and applies transactions
  - Exception: streamed transaction could be handled by parallel apply worker

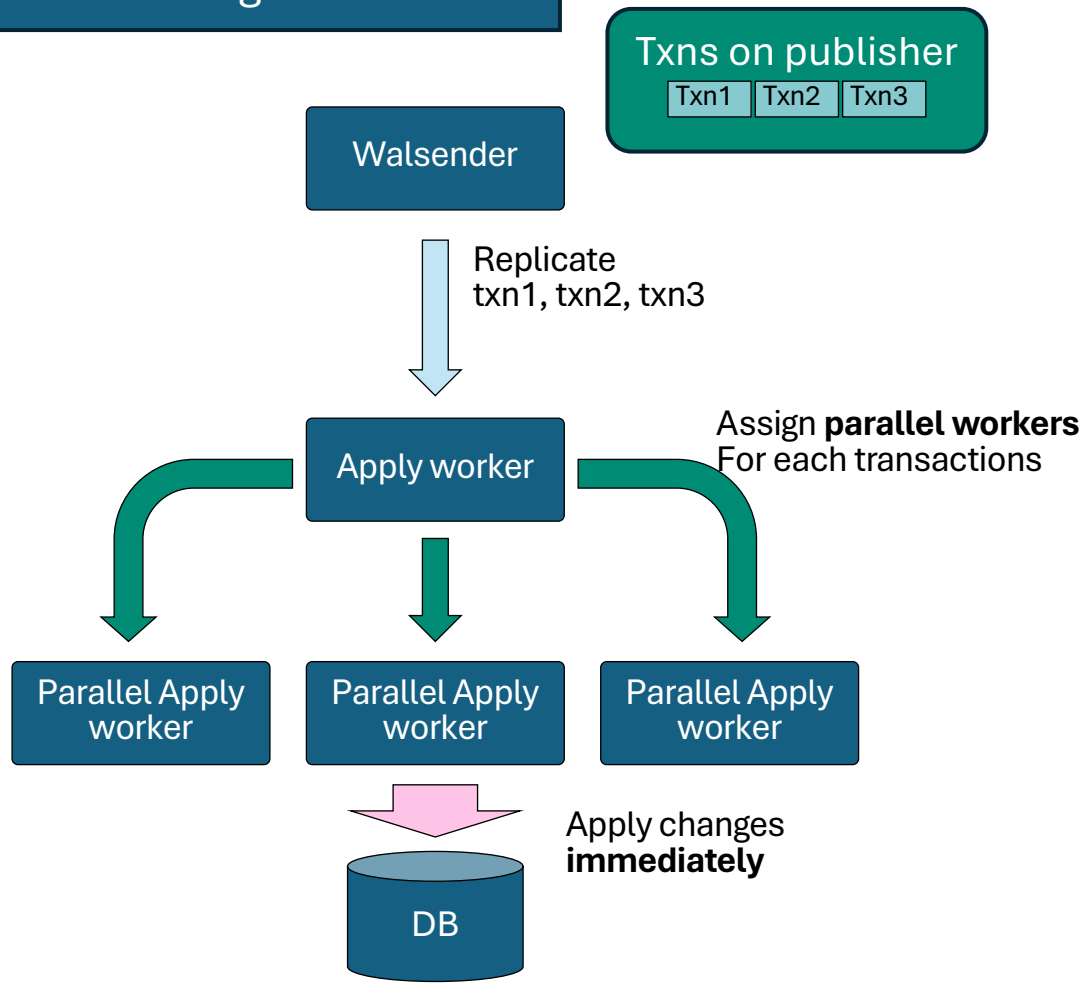
- Ideas for the speed up
  - **Pipeline**: Receive transactions even while applying changes
  - **Extend parallelism**: Use parallel apply workers for all transactions

- Key challenges
  - Dependencies between transactions must be respected.
  - Same result with the non-parallel case must be obtained.

Solution: Parallel Apply

Extends the parallelism to the non-streaming transactions

- A normal apply worker behaves as a leader
  - Communicates with the publisher
  - Assigns txns to parallel apply workers
  - Checks dependencies between txns
- Parallel apply workers applies each transaction independently
  - One apply worker handles one txn
  - Receives transactions from the leader



Why Dependency Tracking Is Critical

T2 must be applied after T1

```
TABLE employees
(id INT PRIMARY KEY, name text);

T1: INSERT INTO employees VALUES (1, 'kuroda');
T2: UPDATE employees SET name = ‘hayato' WHERE id = 1;
```

- We need to detect the dependencies of txns, and serialize these applications
- Checks must be done on the subscriber side, because the subscriber may have additional unique constraints. If rows conflict on a subscriber-only unique key, they must be applied in the same order as non-parallel apply. Otherwise, the result may differ.

FKs must be also considered

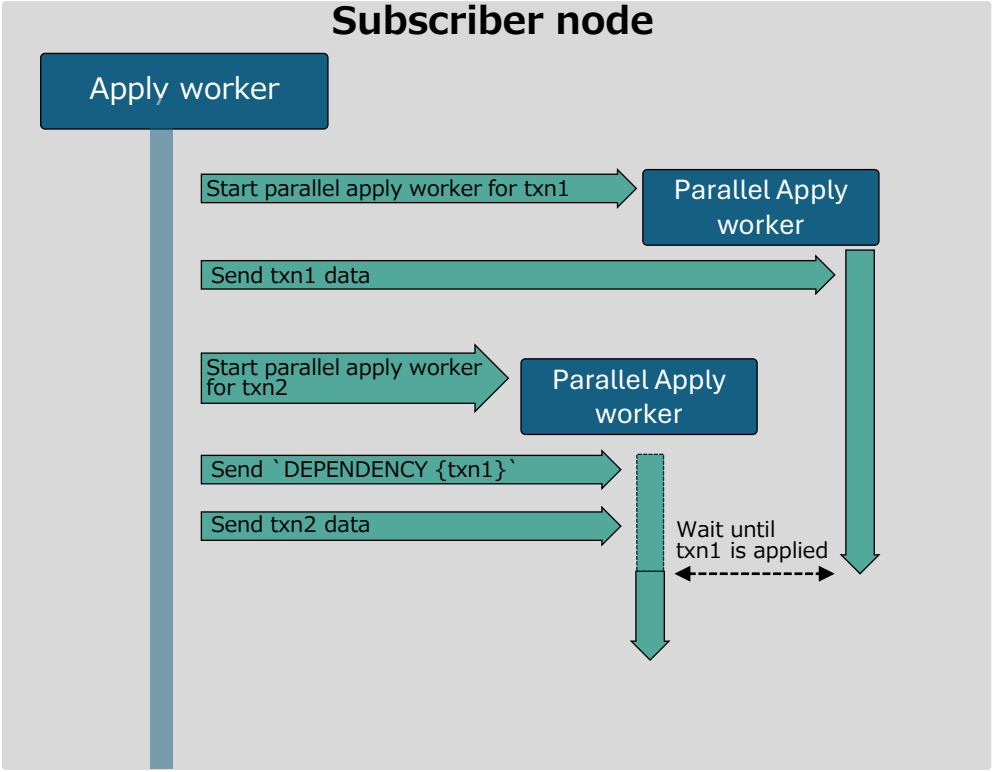
```
TABLE owner(user_id INT PRIMARY KEY);
TABLE car(car_name TEXT, user_id INT REFERENCES owner);

Ta: INSERT INTO owner(1)
Tb: INSERT INTO car('bz', 1)
```

- Here, Tb must be applied after Ta. Otherwise, Tb would fail due to the foreign key violation

- COMMIT ordering of txns can be also preserved for the safety, but can be extended in future versions

Dependency Tracking Mechanism

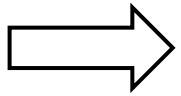
- A leader worker computes dependencies when it shares each changes to each parallel worker
- When the leader detects dependencies, it adds an additional replication message (DEPENDENCY). This message contains a list of remote XIDs on the publisher.
- When parallel apply workers receive DEPENDENCY messages, they wait until all listed transactions are applied by other parallel apply workers.
- A small Key/value store (KVS) is used for the dependency tracking



How KVS Enables Dependency Tracking

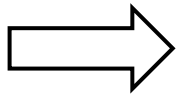
- {relid, hash value} pair is a key, remote XID touching a tuple is a value
  - Hash value is computed from column values in PK/other unique indexes
  - KVS itself is implemented by a simple hash
- When the leader receives changes from the pub, it computes the hash and tries to find an entry on KVS
  - If PK column is modified, both old/new values would be registered
- If nothing found, it just registers the entry and passes to the parallel apply worker
  - Checking is done once per PK or unique key
- If found, it checks the remote XID and add DEPENDENCY with the given XID.
  Then update the XID to the applying transaction
- The KVS is periodically cleaned up. When the leader sends a feedback or the number of entries exceeds a limit

```
T101: INSERT INTO employee VALUES (10, 'kuroda');
T102: INSERT INTO employee VALUES (15, 'amit');
T103: DELETE account WHERE id = 15;
T104: UPDATE account SET balance += 10 WHERE id = 30;
```



Key ({relid, hash})	Value (XID)
{employee, hash(10)}	101
{employee, hash(15)}	102
{account, hash(15)}	103
{account, hash(30)}	104

```
T201: UPDATE employee SET name = ‘hayato’ WHERE id = 10;
```



Key ({relid, hash})	Value (XID)
{employee, hash(10)}	**201**
{employee, hash(15)}	102
{account, hash(15)}	103
{account, hash(30)}	104

`DEPENDENCY {101}` goes to the parallel apply worker

Special Cases and Fallback Behavior

- The leader also applies transactions itself if no parallel apply workers are available
- Preserving order is done by appending a dependency to the last transaction
- `TRUNCATE` statements block all the other changes for the table. Any transactions that do `INSERT`/`UPDATE`/`DELETE` to the table depend on the `TRUNCATE`.

Benchmark Results

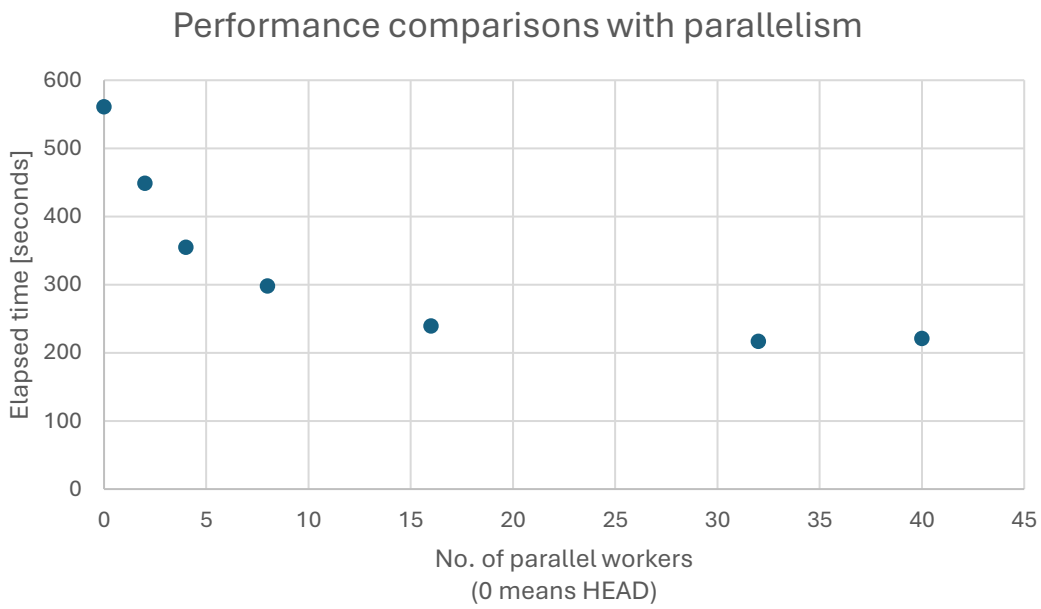
- **Setup**: logical replication system on the same machine with:

```
shared_buffers = 30GB
max_wal_size = 10GB
min_wal_size = 5GB
autovacuum = off
```

- **Workload**: pgbench (tpcb-like) executed on the primary/publisher with:

```
Scale=40
Clients/jobs=30
Duration 5 minutes
```

- **Measurement**: measured till all changes are replicated. Parallelism is varied
- **Result**: The performance would be 2x better than HEAD case.
  - Needs polishing to improve more



Parallel Apply with Out-of-Order Commits

Parallel Apply with Out-of-Order Commits

Ordered Commit Limitation:

- Commit is strictly serialized, even with parallel apply
- Throughput is limited by the slowest transaction
- Workers remain idle → wasted CPU and I/O

Solution: Out-of-Order Commits

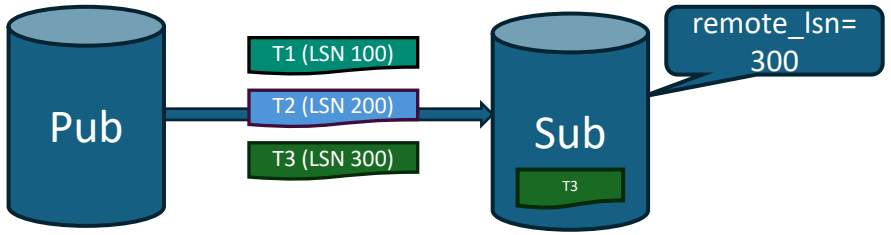
- Allow commits in execution order, not publisher order
- Fast transactions no longer wait behind slow ones
- Parallel Workers stay fully utilized → better resource usage

Dependency tracking is still needed to avoid data inconsistency for interdependent transactions

Challenges with Out-of-Order Commits

Replication Progress Becomes Ambiguous

- PostgreSQL tracks "how far has this subscriber applied?" via a single LSN called *remote_lsn*
- When subscriber restarts, it resume the replication from this *remote_lsn*
- With ordered commits, *remote_lsn* advances monotonically → all prior changes are applied
- Out-of-order commits break this guarantee
- Later transactions may commit while earlier ones are still in-progress
- If *remote_lsn* is advanced with each commit, it may move ahead of unapplied transactions, causing them to be skipped during recovery
- A single LSN is no longer a safe indicator of progress



What Does "Safe Progress" Mean Now?

- Safe progress = the highest LSN below which all transactions are guaranteed applied
- With out-of-order commits, this lags behind the highest committed LSN
- We need to know:
    a. The lowest committed transaction's remote LSN
    b. All the "holes" (already committed but out-of-order) LSNs above it
- Neither of these is tracked by current single-LSN progress tracking

Solution: Progress Tracking with Boundaries

- Replace single remote_lsn with multi-point progress tracking:

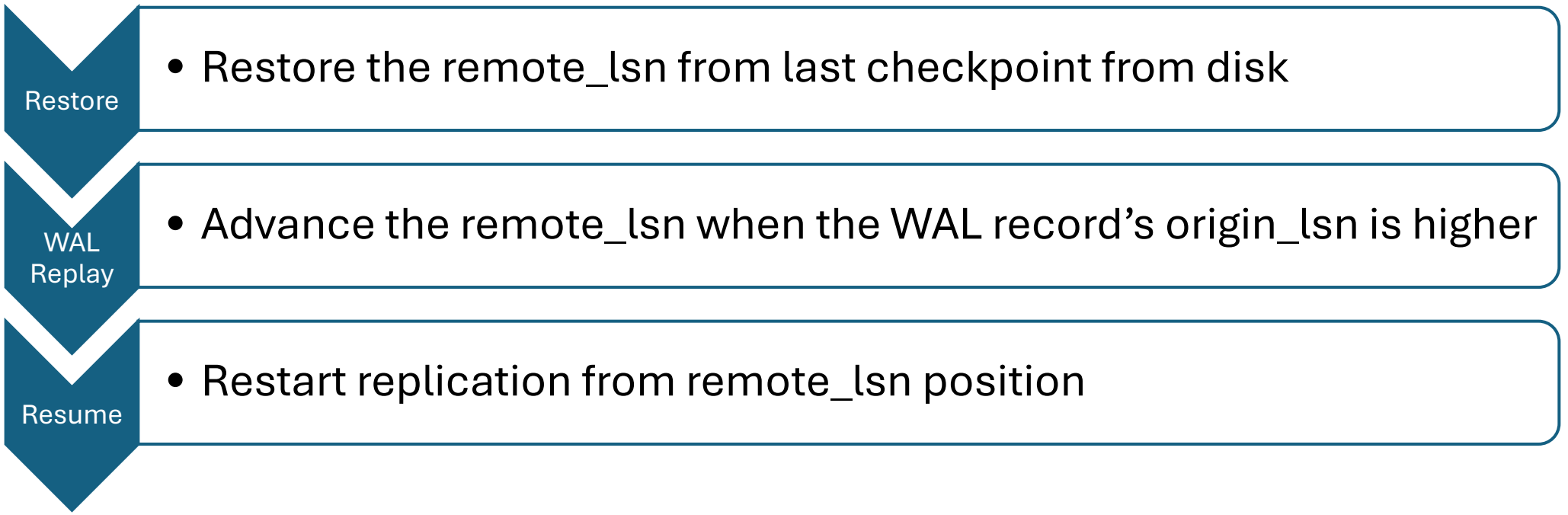
Field	Meaning
lowest_remote_lsn	The safe replication restart point, everything below this is fully applied
highest_remote_lsn	The highest remote LSN that has been committed locally
list_remote_lsn	Sorted list of all committed LSNs between lowest and highest (the "holes" map)

- The gap between lowest and highest represents in-progress or out-of-order commits

Challenge: Crash Recovery of Subscriber

Current Crash Recovery Model

- Subscriber's apply progress (remote_lsn) is periodically persisted via checkpoints
- WAL commit records carry the origin_lsn (transaction's remote LSN)
- On crash recovery:



Crash Recovery Challenges (1/2)

1. **Incomplete Replication State Persistence**
   - Persisting only *remote_lsn* is not sufficient as replication may resume from an incorrect position after recovery
   - **Risk:**
     - Missing the earlier in-progress transactions
   - **Solution: Persist complete replication state**
     - Serialize all the new replication progress state (lowest_remote_lsn, highest_remote_lsn and list_remote_lsn) to disk on checkpoint
     - Replication can start from the safe lowest_remote_lsn
     - Repeated transactions can now be skipped if exists in the list_remote_lsn

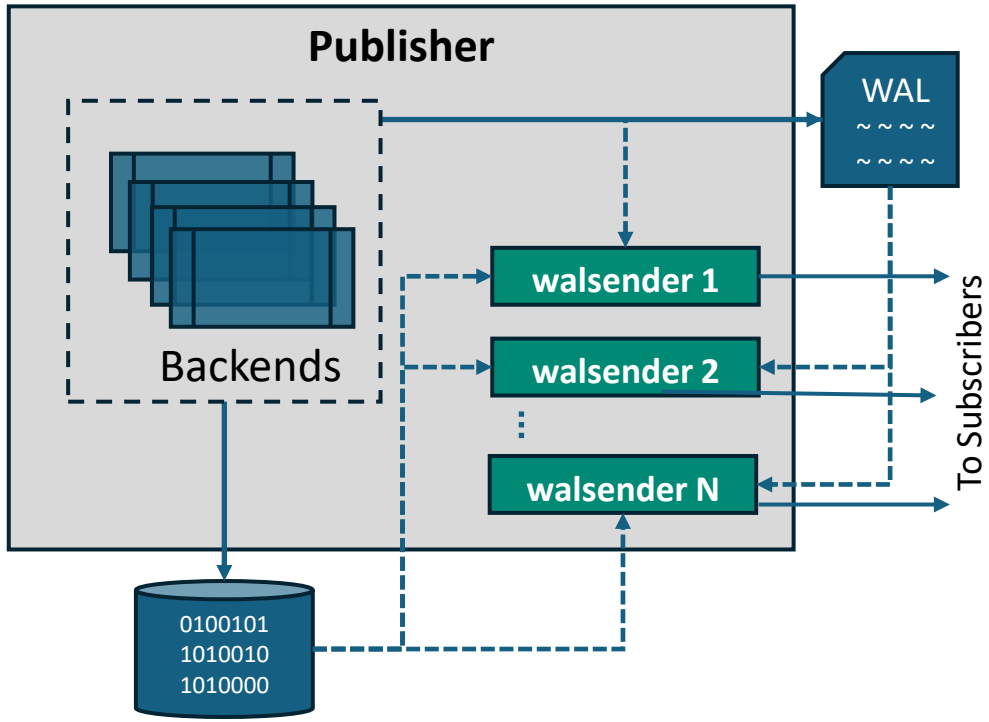
Crash Recovery Challenges (2/2)

2. **Stale Checkpoint State**
   - Checkpoints can be far apart, making stored progress stale, and making the restored lowest_remote_lsn much older than necessary
   - **Risk:**
     - Unnecessary reprocessing and slower recovery
   - **Solution:**
     - Record safe lowest_remote_lsn (when updated) in WAL along with the origin_lsn
     - Restore base state from disk and rebuild incrementally during WAL replay
       - Advance lowest_remote_lsn if WAL has a newer lowest LSN value
       - Advance highest_remote_lsn if WAL has newer origin_lsn for the commit
       - Add origin_lsn to the committed list (if not already present)

Improve Performance On Publisher

Bottleneck #2: Repeated WAL Decoding on the Primary

- Each walsender independently decodes the same WAL
- **CPU overhead on Publisher**
  - Decoding process is expensive and is repeated by all walsenders
- **Scaling issue**
  - Overhead increases linearly with number of subscriptions
- **Increased latency**
  - Excess decoding work can delay change delivery
- **Poor efficiency for high write workloads**
  - Heavy WAL generation amplifies duplication cost.



Centralized Decoding: Design Overview

Core Idea

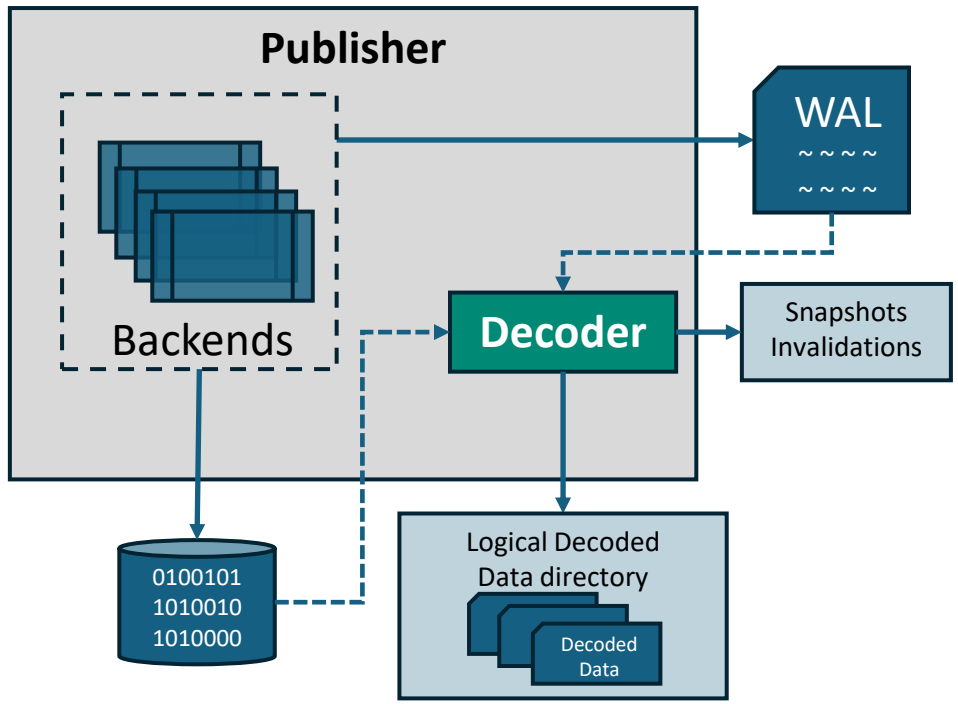
- Decode WAL once in a dedicated process and reuse across all walsenders

Key Design Considerations

- **Memory & Storage**
  - Efficiently share decoded data without exhausting memory or causing excessive disk spill
- **Subscriber Diversity**
  - Support different restart LSN positions and filtering needs across subscribers
- **Synchronization**
  - Enable concurrent access with minimal contention and high scalability
- **Snapshot Consistency**
  - Ensure correct snapshot handoff between decoder and walsenders during initial sync and replication.

Centralized Decoding: Decoder (1/2)

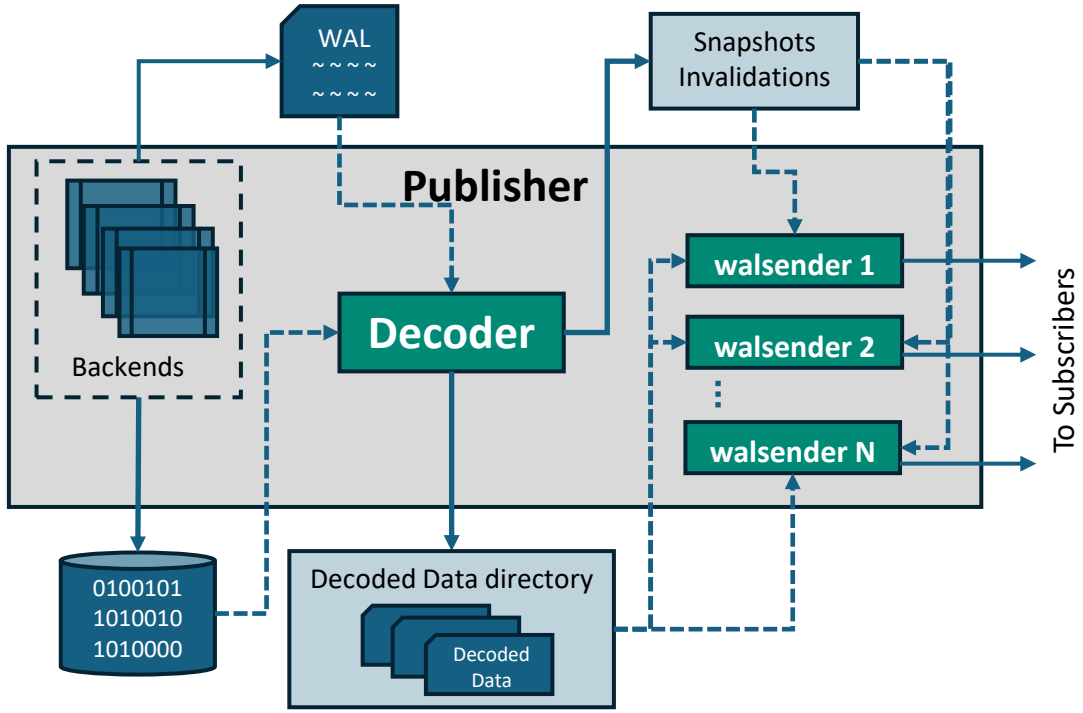
- Builds full snapshot and decodes WAL
- Writes fully decoded transactions (at commit) to shared storage
- Persists:
  - Snapshots
  - Invalidations
- Maintains:
  - Slot to prevent required WAL removal
  - LSN → file offset map to enable walsenders to fetch corresponding decoded tuples



Centralized Decoding: Walsenders (2/2)

Walsender:

- Reads decoded transactions using:
  - start_decoding_lsn / restart_lsn
  - LSN → offset mapping
- Applies publication-based filtering using:
  - Snapshots
  - Invalidations
- Streams only relevant changes to subscriber



Centralized Decoding: Other Challenges

- **Retention of decoded data**
  - Keep decoded files until all subscribers consume them
  - Support multiple files (variable or fixed size)
  - Cleanup based on min required LSN (no partial file cleanup)
- **Progress Tracking & Recovery**
  - Persistently track decoder progress using the last written commit LSN
  - Ensure fast decoder recovery, as it becomes a single point of failure and all walsenders stall until it catches up
- **Usage**
  - Primarily useful for Multi-Consumer Single-Database pattern
  - Pipeline mode can also help in specific workloads where network speed is slow

Centralized Decoding: Further Improvements

- **Parallel Decoding**
  - Share burden of single decoder when WAL generation is high
  - Parallelly decoding of multiple transactions
- **Logical Backup**
  - The decoded_tuple files can be archived and used as logical backup
  - Benefits include:
    - Cross version restore
    - Selective table restore
    - Smaller storage size etc.

Summary

- Logical replication significantly lags physical replication in throughput
- Subscriber-Side Bottlenecks
  - Transactions applied serially limit throughput
  - Solution: Parallel Apply
    - Preserve commit order → baseline improvement
    - Enable out-of-order commits → unlock full parallelism
- Publisher-Side Bottleneck
  - Each walsender independently decodes WAL → redundant work
  - Solution: Centralized Decoding
    - Decode WAL once in a dedicated process
    - Share decoded changes across all walsenders → better CPU efficiency & scalability

Thank You!

Q & A