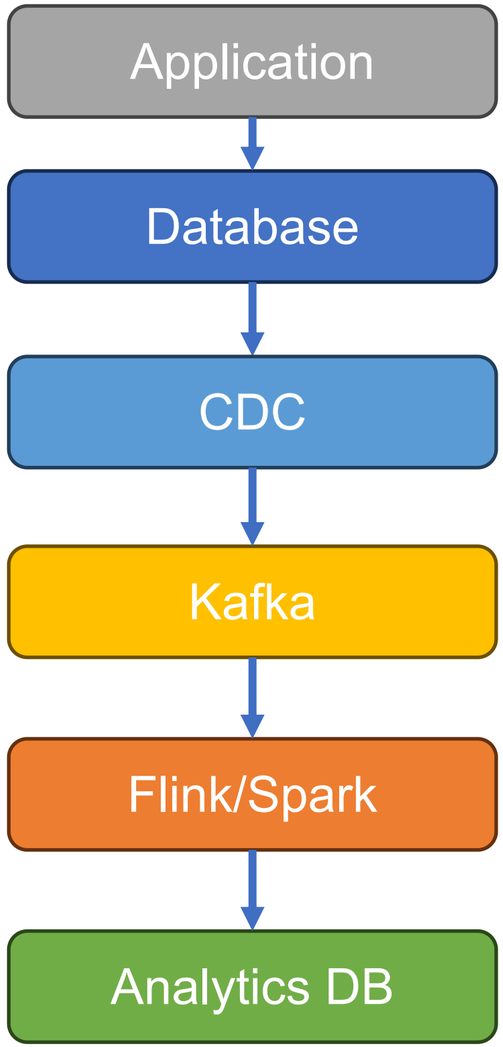
PostgreSQL Streaming Model: From Implicit to Explicit

How Domino Unlocks Continuous Computing

What if PostgreSQL is already a streaming system?

Why streaming pipelines are painful?



Five Systems just to compute a simple metric

Problems with this architecture

MATRIX

System Complexity

High Operational Costs

Data Consistency Challenges

Accumulated Pipeline Latency

We are moving data out of the database,
only to compute something that originated inside it, then store them into database again.

Streaming SQL in PostgreSQL

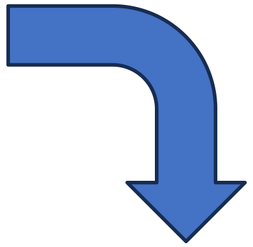
MATRIX

```
CREATE STREAM sales_olap
AS (
    SELECT s.*, r.region, n.nation
    FROM STREAMING sales s
    JOIN dim_region r ON s.region_id = r.id
    JOIN dim_nation n ON s.nation_id = n.id
);
```

- CREATE **STREAM** syntax extension
- FROM **STREAMING** syntax extension
- Vanila SQL for streaming computation

From Query Plan to Delta Plan

```
Insert on sales_olap
  -> Nested Loop
     Join Filter: (s.nation_id = n.id)
     -> Nested Loop
        Join Filter: (s.region_id = r.id)
        -> Seq Scan on sales s
        -> Materialize
           -> Seq Scan on dim_region r
     -> Materialize
        -> Seq Scan on dim_nation n
```



```
Insert on sales_olap
  -> Nested Loop
     Join Filter: (__delta_scan.nation_id = n.id)
     -> Nested Loop
        Join Filter: (__delta_scan.region_id = r.id)
        -> Function Scan on __delta_scan_241444
        -> Materialize
           -> Seq Scan on dim_region r
      -> Materialize
         -> Seq Scan on dim_nation n
```

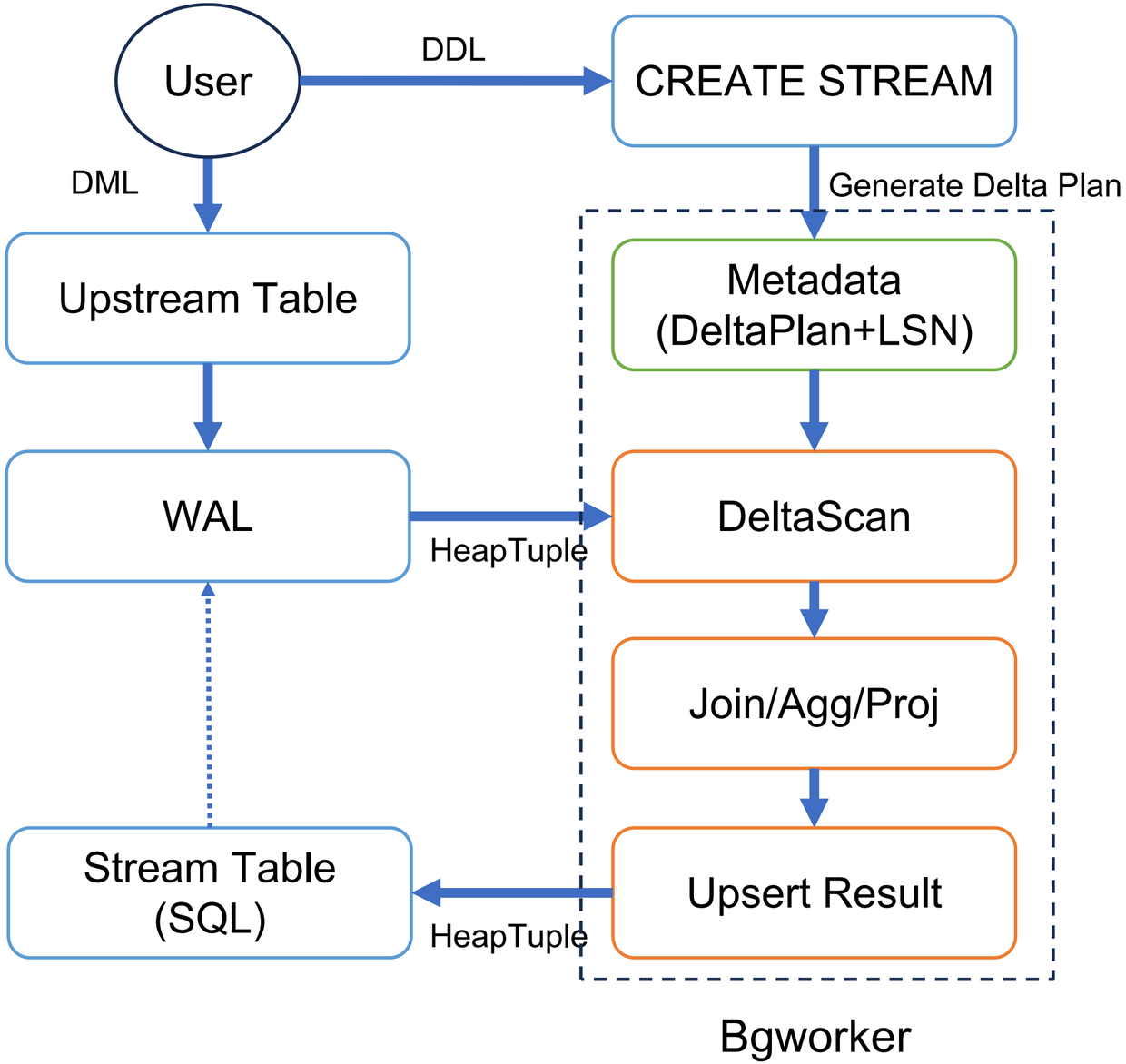
Key Observation: PostgreSQL already has all the pieces for streaming

MATRIX

Why PostgreSQL is Perfect for Streaming?

- WAL
- Logical Decoding
- Planner Executor
- MVCC
- bgworker

Domino: The PostgreSQL Streaming Computation Engine



- WAL produces stream
- DeltaPlan turns SQL into incre-computation
- Stream tables are continuously updated views
- Background worker controls the exec flow, reads LSN, execute incremental plan, update LSN, and commit results.

Domino: Plugin Framework

MATRIX

```
CREATE STREAM s1
AS (
    SELECT s.*
          ,r.region
          ,n.nation
    FROM STREAMING sales s
    JOIN dim_region r
      ON s.region_id = r.id
    JOIN dim_nation n
      ON s.nation_id = n.id
);
```

```
CREATE STREAM s2
AS (
  SELECT name
        ,count(1)
  FROM STREAMING upstream
  GROUP BY name;
);
```

```
CREATE STREAM s3
AS (
    SELECT *
    FROM STREAMING s1
    JOIN STREAMING s2
    ON s1.id1 = s2.id2
);
```

domino_one

Stream ⋈ Dimension Join

domino_agg

Incremental Aggregation

domino_join

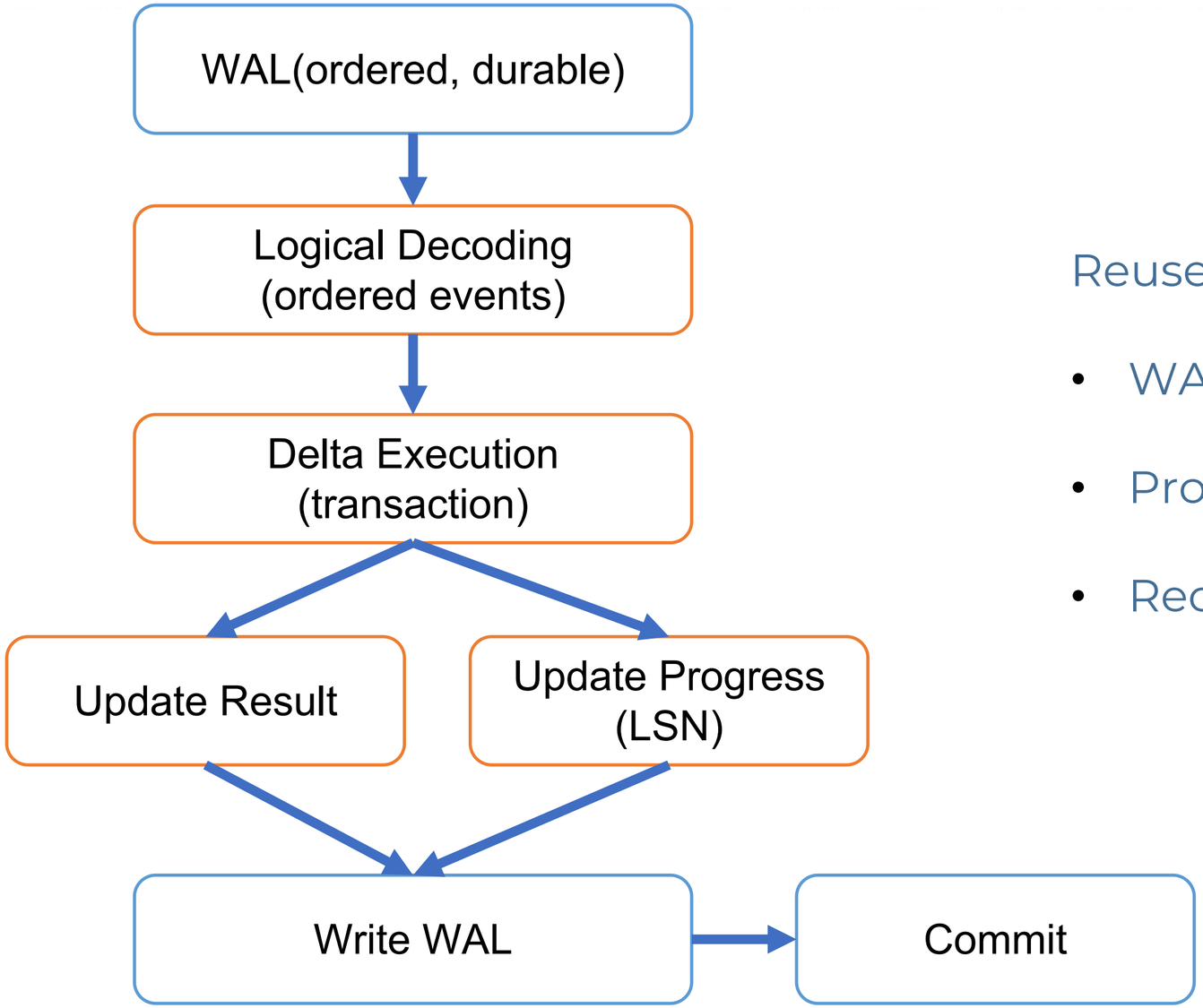
Dual-Stream Join

Hard Problems in In-Database Streaming

MATRIX

- Exactly-Once
- Stream Agg
- Stream Join
- Update Delete

Exactly Once Design



Reuse WAL to guarantee exactly-once:

- WAL provides total order
- Progress is transactional
- Recovery is WAL-driven

Exactly Once on Failure Recovery

MATRIX

Crash Recovery

- **Load Checkpoint**

  Restore last LSN into shared memory

- **Replay WAL**

  Rebuild latest in-memory state.

  pg_controldata has the last checkpoint’s LSN

- **Resume workers**

  Continue from recovered LSN

Progress Persistence Model

- **During Execution**

  Progress (WAL LSN) stored in shared_mem

- **On Commit**

  Update progress in shared_mem & Emit progress WAL

- **On Checkpoint**

  Persist progress to disk

- **On Recovery**

  Load progress from disk, Replay WAL to restore state

domino-join: Streaming Join = 3 Incremental Joins

ΔT1 JOIN pre-T2

U

pre-T1 JOIN ΔT2

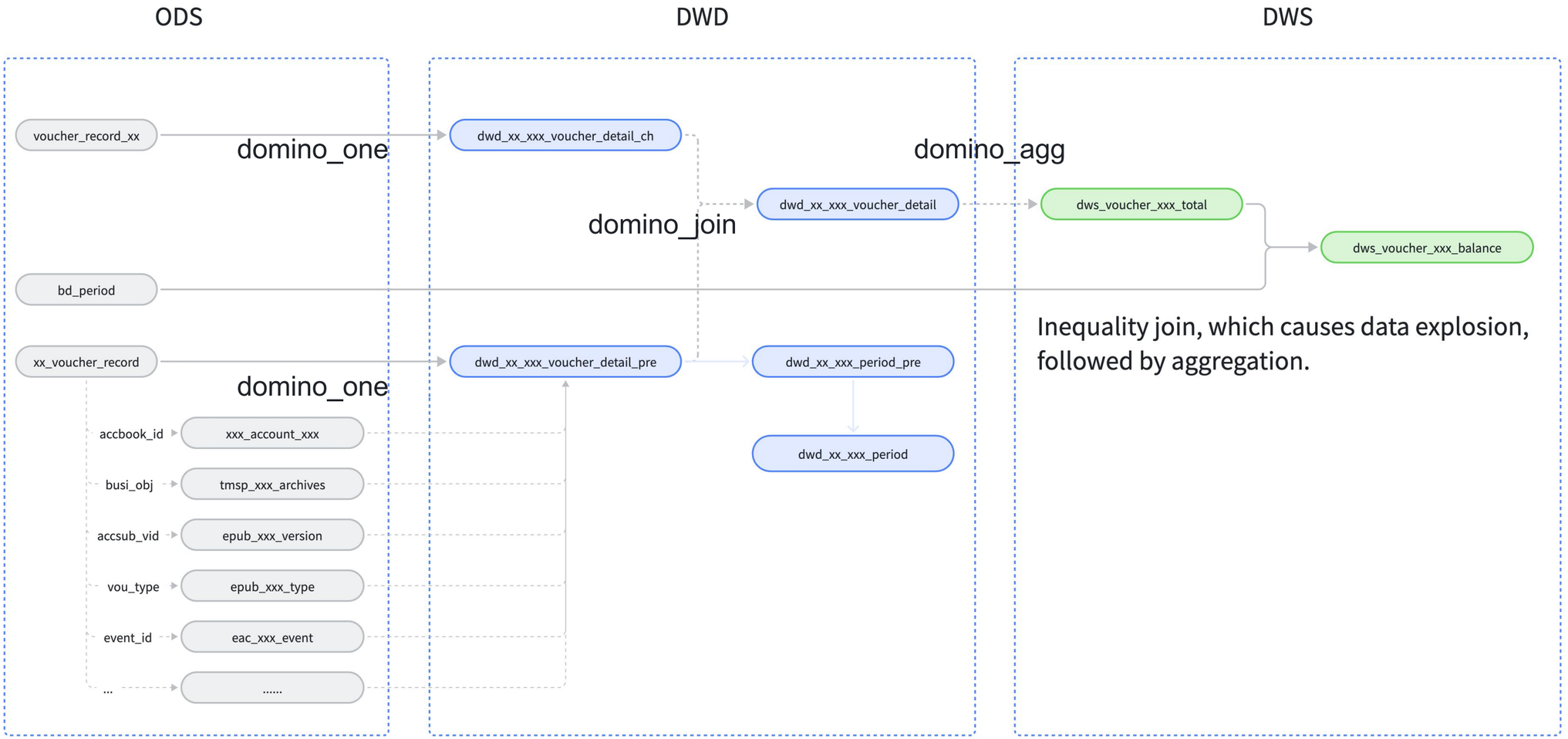
U

ΔT1 JOIN ΔT2

JOIN is hard because both sides are moving

```
Insert on s1
  -> Append
     -> Nested Loop
        -> Shared Scan (share slice:id 0:0)
           -> Function Scan on __delta_scan_241944
        -> Index Scan using t2_id on t2
              Index Cond: (id2 = share0_ref1.id1)
     -> Nested Loop
        -> Shared Scan (share slice:id 0:1)
           -> Function Scan on __delta_scan_241944
        ->  Index Scan using t1_id on t1
              Index Cond: (id1 = share1_ref1.id2)
     ->  Nested Loop
         Join Filter: (share0_ref2.id1 = share1_ref2.id2)
         ->  Shared Scan (share slice:id 0:0)
         ->  Materialize
             ->  Shared Scan (share slice:id 0:1)
```

Customer Use Case: From Lambda to Domino



< 20s
End-to-end
full link

350M rows
270 columns
3000-12000 rows/s

330M rows
240 columns
3000-12000 rows/s

15M rows
170 columns
10K-60K rows/s

Why NOT Just Use Materialized Views?

MATRIX

Materialized views are snapshots, Streams are living data.

Batch recomputes, Streaming reacts:

Feature	MV	Domino
Update Model	Pull	Push
Computation	Full refresh	Incremental
Latency	Minutes/hours	seconds
Trigging source	manual	WAL

Limitations

MATRIX

- MIN/MAX is not supported yet if upstream table has update/delete.
- Window functions are not supported yet.
- CTE/Having/GroupingSet etc are not supported yet.

Summary

MATRIX

PostgreSQL Streaming Computation Model:

- PostgreSQL already has all the pieces needed for streaming.
- Domino turns PostgreSQL into a streaming database.
- SQL becomes a streaming language.

Thank you!

https://ymatrix.ai

WE ARE HIRING Globally!