PostgreSQL's Distributed Evolution

What changed and what stayed the same in the cloud

Raluca Constantin

Amazon Aurora DSQL Database Engineer

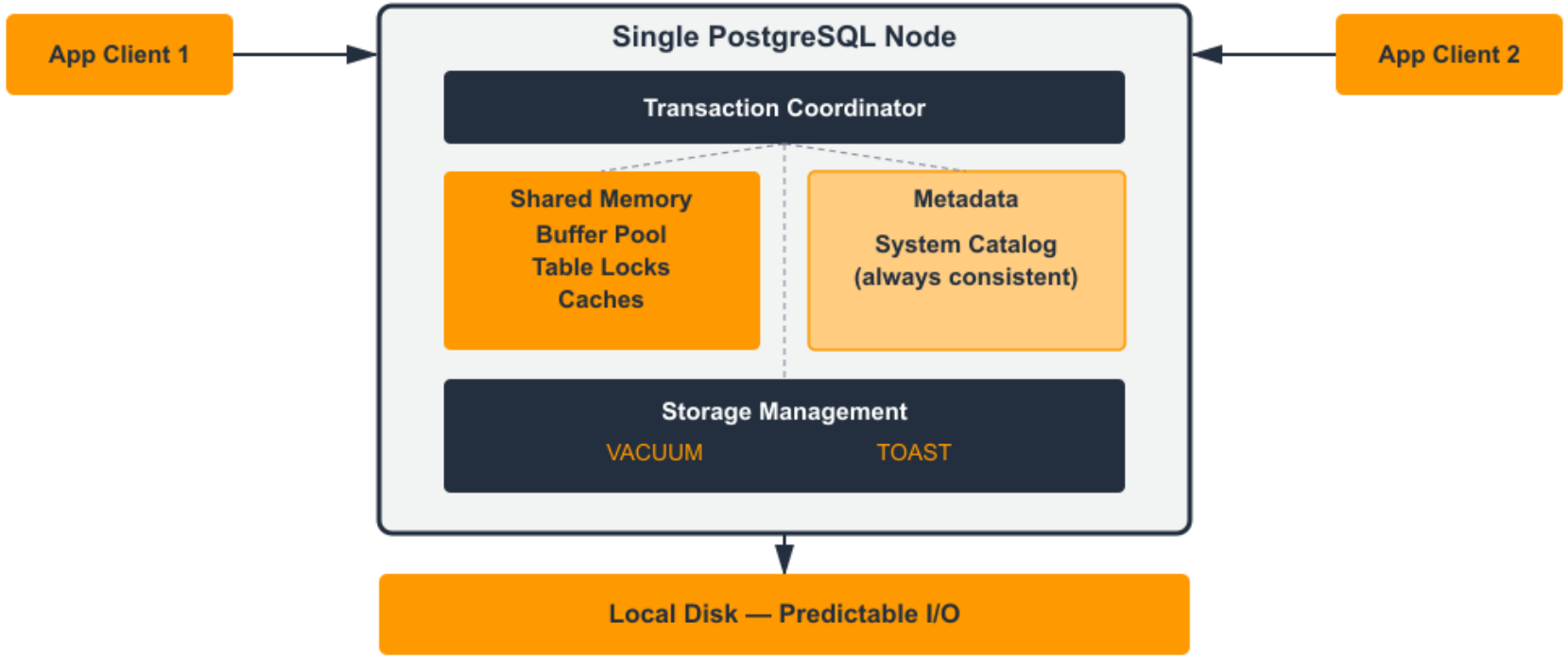
Agenda

- Context
- MVCC / Snapshot visibility
- Transaction coordination
- DDL / Catalog consistency
- Statistics / Query planning
- Multi-Region writes
- Wins, lessons and open questions

Why we built Aurora DSQL

- Scalability
- High availability
- Multi-Region Active-Active
- Serverless elasticity

Single-node design choices that don't extend to distributed



Design choices optimized for a single node — why they don't distribute

- Shared memory — buffers, locks, caches in one address space
- One coordinator — global visibility into all transactions
- Local disk — predictable latency, sequential I/O optimization
- Centralized catalog — metadata lookups are instant
- VACUUM & TOAST — storage management tied to local disk
- MVCC — xmin/xmax visibility via proc-array and snapshots

MVCC / Snapshot Visibility

PG design choice	32-bit XIDs + xmin/xmax + procarray Single address space Multiple isolation levels (typically Read Committed)
DSQL approach	Timestamps (τ_start, τ_commit) via HLC Synchronized clocks Fixed isolation level: Strong Snapshot (~Repeatable read)
Alternatives	Other versions of HLC (Hybrid Logical Clock)

Transaction coordination

PG design choice	Single-process transaction manager Shared memory, lock tables, single WAL Pessimistic concurrency control
DSQL approach	Coordination at commit time Isolated, horizontally-scaled Query Processors Optimistic Concurrency Control
Alternatives	PG-native 2PC Raft with write intents or Paxos with locks during execution

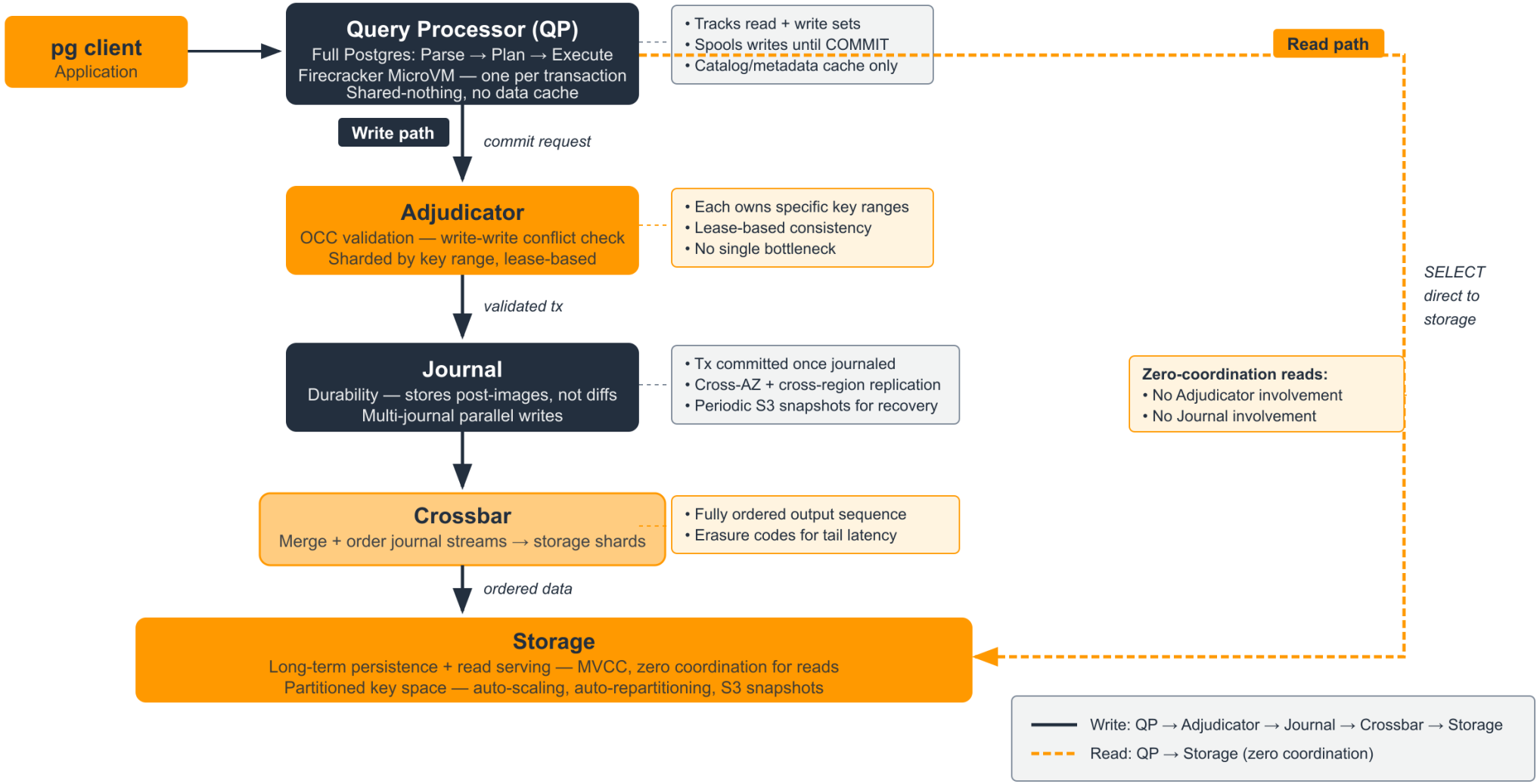
DDL / Catalog consistency

PG design choice	System catalog = regular heap tables with MVCC DDL = DML on catalog tables AccessExclusiveLock for most DDL sinval to invalidate cached copies on modifications
DSQL approach	DDL = distributed transactions updating a catalog version Asynchronous non-blocking DDL
Alternatives	Two-layer catalog – split brain possible F1 schema change protocol Distributed DDL via PG's own extension hooks

Statistics / Query Planning

PG design choice	pg_statistic (ANALYZE/autovacuum) Cost model assumes uniform I/O costs seq scan vs index scan calibrated for local disk
DSQL approach	Plug in at the AM layer and decide not to integrate with the buffer pool Probabilistic, stateless Auto Analyze Differences in the EXPLAIN PLAN
Alternatives	New optimizer GUC to expose per-shard plans

Aurora DSQL — Disaggregated transaction flow



Storage Model & EXPLAIN Plans

SELECT amount, status FROM orders WHERE customer_id = '4b18a761-...' — 1M rows, idx ON (customer_id) INCLUDE (amount, status)

PostgreSQL (heap + index)

Case 1: All columns in index → Index Only Scan

```
Index Only Scan using idx_orders_customer on orders
(cost=0.42..4.44 rows=1 width=15)
Index Cond: (customer_id = '4b18a761-...'::uuid)
Heap Fetches: 0
```

One plan node. If visibility map says all-visible, heap is skipped entirely.

Case 2: Column NOT in index → Index Scan (heap fetch implicit)

```
Index Scan using idx_orders_customer on orders
(cost=0.42..8.44 rows=1 width=23)
Index Cond: (customer_id = '4b18a761-...'::uuid)
```

Still one node. The heap fetch is invisible — you can't see its cost separately.

PG hides the heap interaction:

- Index traversal + heap fetch bundled into one plan node
- Cost of heap I/O not visible separately in EXPLAIN
- You can't tell how much latency is index vs. heap

Aurora DSQL (PK = table, no heap)

Case 1: All columns in index → one storage round-trip

```
Index Only Scan using idx_orders_customer on orders
(cost=100.16..104.17 rows=1 width=15)
Index Cond: (customer_id = '4b18a761-...'::uuid)
→ Storage Scan on idx_orders_customer (rows=1)
Projections: amount, status
```

One storage round-trip. Projections show exactly which columns cross the network.

Case 2: Column NOT in index → explicit second round-trip

```
Index Scan using idx_orders_customer on orders
(cost=100.28..208.29 rows=1 width=23)
→ Storage Scan on idx_orders_customer (rows=1)
→ Storage Lookup on orders (rows=1) ← 2nd round-trip
Projections: amount, status, created_at
→ B-Tree Lookup on orders (rows=1)
```

Two storage operations, each with its own cost — overhead is visible and measurable.

DSQL makes every storage interaction explicit:

- Each network round-trip is a separate plan node with its own cost
- Projections show exactly which columns are transferred

Why the plans look different — architectural comparison

Aspect	PostgreSQL	Aurora DSQL
Table storage	Heap (unordered 8KB pages)	B-tree organized by primary key
Primary key	An index that points to heap (TID)	IS the table — all columns inline
Heap fetch visibility	Implicit — hidden inside plan node	Explicit — Storage Lookup is a separate node
Filter cost	Post-fetch filter is cheap (local memory)	Filter location matters (before vs. after network)
Full table scan	Seq Scan — reads heap pages sequentially	Full Scan — traverses PK B-tree (can push filters)

Multi-region writes

PG design choice	A single node is both reader and writer Replicas are read-only Failover is manual or semi-automated No built-in multi-region consistency
DSQL Approach	Active-active across regions Synchronous replication 0 RPO
Alternatives	Specialized hardware Flexible MR model; latency trade-offs

What the design made possible

1. Multi-Region Active-Active
2. Query isolation — no noisy neighbors
3. PostgreSQL's SQL layer separates cleanly
4. Read-only transactions with minimal cost
5. Probabilistic Auto Analyze
6. Asynchronous Non-blocking DDL

Future development

- Foreign keys
- Triggers
- PL/pgSQL, stored procedures
- SAVEPOINT
- … Many more …

Open questions

Transaction lifecycle hooks	Pluggable commit protocols
AM for the catalog tables	Separate catalog storage from semantics
AM for the cost model	Allow Planner to account for network topology
WAL extensibility	Support topology beyond leader-follower
Heap-agnostic AM API	No TID and HOT assumptions

Thank you!

linkedin.com/in/constantinraluca/