Living on the Edge

Distributed Postgres Clusters and You

Shaun Thomas
Software Engineer, pgEdge

April 25th, 2025

Who are pgEdge?

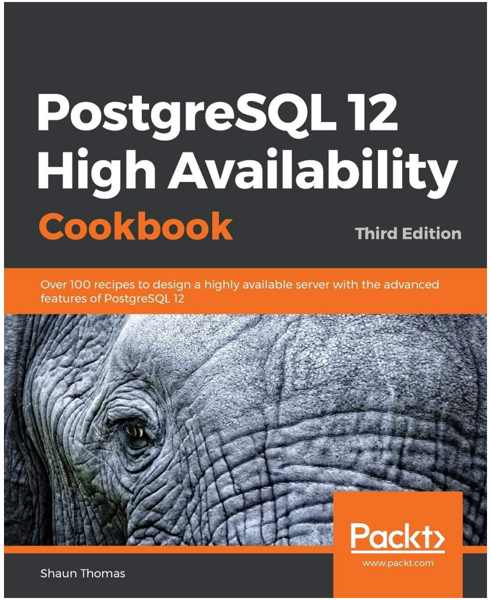
- Distributed Postgres
- Active-Active clusters
- Cloud Services
- Platform Automation
- Ultra High Availability

www.pgedge.com

Who am I?

- Author
- Speaker
- Blogger
- Mentor
- Dev

shaun.thomas@
pgedge.com



Agenda

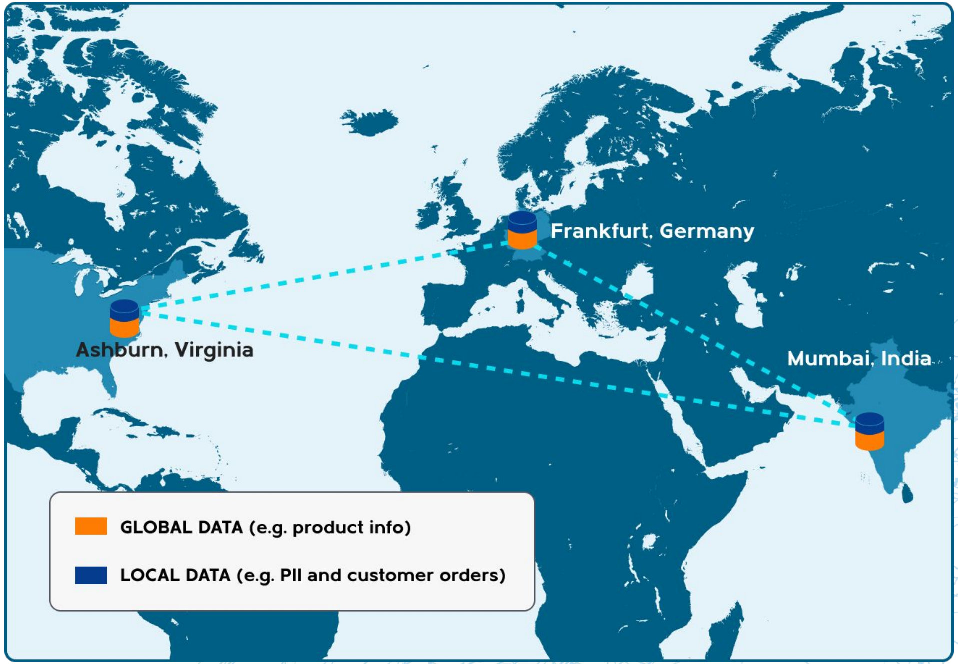
Material We'll Be Covering

1. Why Distributed Postgres?
2. Multi-Master Cluster Theory
3. Conflict Types
4. Conflict Management

Widely Distributed

Comparing Cluster Types

A Distributed Cluster



In Theory

Thinking CAP

CAP is *not* ACID

- **C**onsistency
- **A**vailability
- **P**artition Tolerance

Partitioned (distributed) clusters have either availability or consistency, never both.

A sturdy PACELC

PACELC is CAP with Latency

- Standard CAP applies
- **E**lse even when working normally
- Choose between **L**atency or
- Loss of **C**onsistency
- Most clusters choose the latter

Remember your Pack-Elk

What is Single-Master?

Consider playing chess. The game state can only have one true value, mediated by the board itself.

That's a standard Postgres cluster

What is Multi-Master?

A swarm of starlings can operate independently, and the group continues to thrive despite the absence of any single bird.

That's a distributed Postgres cluster

Disagreeable Outcomes

How Conflicts Fit In

- Multi-Master = asynchronous operation
- Asynchronous operation = simultaneous changes
- Simultaneous changes = conflicts
- Conflicts = 😭

Four Categories of Conflict

1. Naturally convergent conflicts
2. Resolvable conflicts
3. Divergent conflicts
4. (Bonus) Phantom conflicts

Convergent Conflicts

It doesn't matter how you got here

We're just glad you made it

Update - Delete

- Update happened first? It gets deleted
- Delete happened first? Nothing to update
- End state: the row is deleted

Delete - Delete

- End state: the row is deleted

Update - Truncate

- Update happened first? The table gets truncated
- Truncate happened first? Nothing to update
- End state: the table is truncated

Delete - Truncate

- Delete happened first? The table gets truncated
- Truncate happened first? Nothing to delete
- End state: the table is truncated

Truncate - Truncate

- End state: the table is truncated

Resolvable Conflicts

We got to the cabin,

but what happened to Jim?

Insert - Insert

- Caused by this sequence:
  - Node A: `INSERT ... (id, col1) VALUES (2, 10)`
  - Node B: `INSERT ... (id, col1) VALUES (2, 100)`
- Last "update" wins (default resolution method)
- Result: one `INSERT` is discarded / overwritten

Insert - Insert (Multiple Unique Keys)

- Caused by this sequence:
  - Node A: `INSERT ... (id, email) VALUES (1, 'bob@smith.com')`
  - Node B: `INSERT ... (id, email) VALUES (2, 'bob@smith.com')`
- Last "update" wins (default resolution method)
- Results:
  - One `INSERT` is discarded / overwritten
  - Lose a primary key
  - Potentially orphan foreign keys

Why Insert - Insert Conflicts Matter

- Losing one INSERT is technically data loss
- What did that INSERT contain?
- All nodes have the same data
- But what happened to Jim?

Update - Update - Type 1

- Node A: `UPDATE t SET col1=100 WHERE id=4`
- Node B: `UPDATE t SET col1=500 WHERE id=4`
- Last "update" wins (default resolution method)
- Result: contents of `col1` are discarded / overwritten

Update - Update - Type 2

- Node A: `UPDATE t SET col1=100 WHERE id=4`
- Node B: `UPDATE t SET col2='stuff' WHERE id=4`
- Last "update" wins (default resolution method)
- Result: contents of `col1` *or* `col2` are discarded / overwritten

How Update-Update Conflicts Work

- Postgres logical replication copies the entire result tuple!
- Old tuple: `{id: 4, col1: 50, col2: 'wow'}`
- Node A: `UPDATE t SET col1=100 WHERE id=4`
- Logical replication sees: `{id: 4, col1: 100, col2: 'wow'}`
- Node B: `UPDATE t SET col2='stuff' WHERE id=4`
- Logical replication sees: `{id: 4, col1: 50, col2: 'stuff'}`
- **Only one full tuple can be applied**
- What happened to Jim?

Divergent Conflicts

Who are you

and how did you get in here?

Insert - Update

- Involve 3+ nodes
- Caused by this sequence:
  - Node A: INSERT -> Node B
  - Node B: UPDATE -> Node C
  - Node C: ignores UPDATE
  - Node A: Insert -> Node C
- Result: Node C has **diverged**

Insert - Delete

- Involve 3+ nodes
- Caused by this sequence:
  - Node A: INSERT -> Node B
  - Node B: DELETE -> Node C
  - Node C: ignores DELETE
  - Node A: Insert -> Node C
- Result: Node C has **diverged**

Insert - Truncate

- Involve 3+ nodes
- Caused by this sequence:
  - Node A: INSERT -> Node B
  - Node B: TRUNCATE -> Node C
  - Node C: ignores TRUNCATE
  - Node A: Insert -> Node C
- Result: Node C has **diverged**

Phantom Conflicts

What is the sound

of one hand clapping?

What is a “Phantom” Conflict?

- The "not a conflict" conflict
- No divergence
- No action collisions
- No latency problems
- No logs
- What is it, then?

The Conflict that Wasn’t

1. App: `INSERT; COMMIT` -> Node A
2. Node A: `COMMIT` -> WAL -> Node B
3. Node A: Confirm !-> App
4. Node A: Crash
5. Node B: Becomes new write target
6. App: Connection aborted? Retry insert!
7. App: `INSERT` -> Node B
8. Duplicate record now exists

No More Ghosts

- Don't auto-increment surrogate (sequence) keys
  - Use natural keys
  - Use application-generated IDs
  - Fetch sequence value *before* INSERT
- Confirm COMMITS actually failed
  - SELECT after re-establishing connection
  - Trust, but verify
- This is also necessary in non-distributed clusters

Preventing the Inevitable

Don’t Do That

The only winning move
is not to play

“Don’t do That” ?!

- “An ounce of prevention is worth a pound of cure”
- What is the primary cause of conflicts?
- Concurrent operation on the same keys
- Consider possible ways that may happen

Avoiding "Doing That"

- Use "sticky" sessions
- Assign app servers to specific (regional) nodes
- Interact with specific (regional) data
- Avoid unnecessary cross-node activity

Prefer Ledgers to Cumulative Totals

Rhymes with Credit

An elegant datatype

for a more civilized age

What is a CRDT?

Designed specifically for distribution

- **C**onflict-free
- **R**eplicating
- **D**ata
- **T**ype

I call them Conflict Resistant Data Types

How do CRDTs Work?

Two basic approaches:

1. Apply a diff between the incoming and existing values
2. Use a custom data type with per-node “hidden” fields

This only works for numerical columns!

CRDTs in the Spock Extension

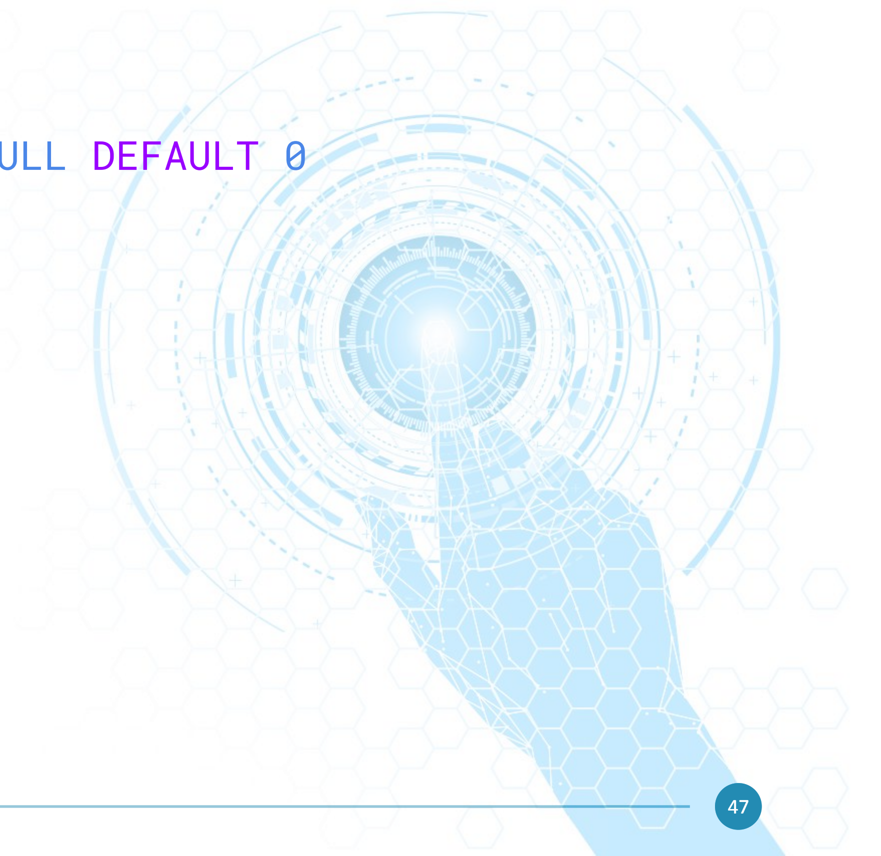
```
CREATE TABLE account (
  id     BIGINT PRIMARY KEY,
  total  BIGINT NOT NULL DEFAULT 0
);

ALTER TABLE account
ALTER COLUMN total
  SET (LOG_OLD_VALUE=true,
       DELTA_APPLY_FUNCTION=spock.delta_apply);
```

CRDTs in the BDR Extension

```sql
CREATE TABLE account (
  id     BIGINT PRIMARY KEY,
  total  bdr.crdt_delta_counter NOT NULL DEFAULT 0
);
```

- The BDR extension uses custom data types
- Several options to choose from



How Do CRDT Deltas Work?

- Old tuple: `{id: 5, total: 100}`
- `UPDATE account SET total = total + 100 WHERE id = 5`
- New tuple: `{id: 5, total: 200}`
- Old *and* new values sent to remote node
- The delta function or type calculates:
  - total = local.total + remote.new.total - remote.old.total
  - total = 100 + 200 - 100 = 200

How Deltas Avoid Conflicts

- Old tuple: `{id: 5, total: 100}`
- Node A adds 100, sends: `{id: 5, total: 200}`
- Node B adds 400, sends: `{id: 5, total: 500}`
- Node A: total = 200 + (500 - 100) = 600
- Node B: total = 500 + (200 - 100) = 600

How CRDT Aggregates Work

1. Two node cluster
2. Initial tuple: `{id: 5, total: (0, 0)}`
3. Node A adds 100
4. Node A tuple: `{id: 5, total: (100, 0)}`
5. Node B tuple: `{id: 5, total: (100, 0)}`
6. Each node only interacts with its own “column”

How CRDT Aggregates Avoid Conflicts

1. Starting tuple: `{id: 5, total: (100, 0)}`
2. Total displayed as 100
3. Node A adds 100
4. Node B subtracts 50
5. New tuple: `{id: 5, total: (200, -50)}`
6. Total displayed as 150

Common Aggregate CRDT Caveat

CRDT aggregate need a "reset" function:

1. Starting tuple: `{id: 5, total: (200, -50)}`
2. Total displayed as 150
3. Node A sets total to 0
4. Node B does *nothing*
5. New tuple: `{id: 5, total: (0, -50)}`
6. Total displayed as -50

Key Management

Like two ships

passing in the night

Preventing Insert - Insert Conflicts

Four methods for avoiding key collisions:

1. Sequence offsets
2. Globally unique keys
3. External key generator
4. Global allocations

What are Sequence Offsets?

Node 1:

```
ALTER SEQUENCE foo_id_seq
      RESTART WITH 1001
      INCREMENT BY 10;
```

Node 2:

```
ALTER SEQUENCE foo_id_seq
      RESTART WITH 1002
      INCREMENT BY 10;
```

Sequence Offset Results

- Node 1: 1001, 1011, 1021, 1031 …
- Node 2: 1002, 1012, 1022, 1032 …
- Must be done for every sequence
- More difficult to add new nodes
- Increment size determines maximum node count
- Backward compatible with existing clusters

What are Globally Unique Keys?

Anything guaranteed to be unique across the cluster

- UUID: `CREATE EXTENSION uuid-ossp`
- Snowflake IDs
  - https://github.com/pgEdge/snowflake
  - Provide replacement for `nextval(), currval()`
  - Arbitrarily large 64-bit (bit-packed) values
  - May not be compatible with some front-end frameworks

What are External Key Generators

- Some external service that generates keys
- May add latency to each insert
- Could be a single point of failure

What are Global Allocations?

- Nodes assign chunks of values by consensus
  - Node A: 1 - 2,000,000
  - Node B: 2,000,001 - 4,000,000
  - Node C: 4,000,001 - 6,000,000
- Nodes always keep a current and future chunk
- Chunk size based on column type (INT, BIGINT)
- BDR calls this sequence type `galloc`

Sequence Safety

Can Sequences

Ever Be Safe?

Best Practices: Don’t use SERIAL?

We’ve all been told not to do this:

```
CREATE TABLE serial_example (
  id BIGSERIAL PRIMARY KEY
);
```

This is How Postgres Does SERIAL

```
CREATE TABLE serial_example (
  id BIGINT PRIMARY KEY
);

CREATE SEQUENCE serial_example_id_seq;

ALTER SEQUENCE serial_example_id_seq
      OWNED BY serial_example.id;

ALTER TABLE ONLY public.serial_example
ALTER COLUMN id
  SET DEFAULT nextval('public.serial_example_id_seq');
```

Problems with Serial

- Lots of under-the-hood magic
- It’s just a column default
- Values can be overridden easily by users
- Not standard SQL
- May forget to transfer sequence during migrations
- What are we told to use instead?

Best Practices: Identities?

```sql
CREATE TABLE ident_example (
  id BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY
);
```

These are better, right?

Serial Killer

- No more magical `nextval` shenanigans
- The word `ALWAYS` means what it says
- Standard SQL
- Extra syntax to set starting point, increment size, and others
- It’s great, right?

Protected Identity

```
INSERT INTO ident_example (id) VALUES (2);

ERROR:  cannot insert a non-DEFAULT value into column "id"
DETAIL:  Column "id" is an identity column defined as GENERATED
ALWAYS.
```

Distributed Identities

- What happens in Active-Active clusters?
- We can no longer substitute our own `nextval` function
- Stuck with monotonically advancing sequential values
- Can't use snowflake, timeshard, or other numerical replacements
- Ironically *too* inflexible

Unexpected Sequence Conclusion

- Keep using SERIAL and BIGSERIAL
- Or just use DEFAULT directly
- May need to *reverse*-migrate tables using IDENTITY

Shameless Self Promotion

You don't need him,

I'm the upgraded model

Handling Phantom Updates

UPDATEs can lead to divergent conflicts

1. UPDATE arrives after initial INSERT
2. UPDATE ignored

How to fix?

1. Convert UPDATE to INSERT
2. Ignore “old” INSERT when it arrives

UPDATE Conversion Safety

This is safe, provided:

1. Timestamps are tightly synchronized
2. Retain transaction IDs in converted statement

Historical Relevance

What do you want

on your tombstone?

Handling Data Deletion

DELETEs or can lead to divergent conflicts

1. Data gets deleted
2. Old INSERT statements may then succeed

How to fix?

1. Retain the old row as a “tombstone” (Soft delete)
2. Don’t allow inserts on tombstone keys

Tombstone Persistence

How long should tombstones last?

- At least as long as potential node latency
- Could work similarly to `hot_standby_feedback`
- Manage cleanup through node consensus

Questions?