

ADAM



BORISS

WHEN ZERO-DOWNTIME SAVES LIVES

The reality of Active-Active Postgres

EDB POSTGRES AI

©EDB 2026 — ALL RIGHTS RESERVED.

THE USE CASE

112 AT CANARY ISLANDS

One emergency line, seven islands.



> Every call to 112 has to reach a dispatcher — police, ambulance, fire — no matter which island it comes from. The system behind that line runs on PostgreSQL.



©EDB 2026 — ALL RIGHTS RESERVED.

THE PROBLEM

WHY STANDARD HA IS NOT A SOLUTION



Classic primary/standby was built for failure recovery, not for zero downtime.

01

FAILOVER TAKES TIME

Promoting a standby is seconds to minutes — an eternity for a call that's ringing right now.

02

ONE WRITER AT A TIME

Only the primary accepts writes. Every other island is read-only until a failover happens.

03

DISTANCE ADDS RISK

Islands are physically apart. A single regional outage can isolate the only writable node.

©EDB 2026 — ALL RIGHTS RESERVED.

THE IDEA

WHY BI-DIRECTIONAL REPLICATION IS THE SOLUTION



Let every island keep its own writable node — and replicate in every direction.

STANDARD HA	ACTIVE-ACTIVE (BDR)
— One primary, others read-only	— Every node accepts writes
— Failover = downtime window	— A node loss is a non-event elsewhere
— Writes tied to one location	— Writes happen where the users are

112 Canary Islands understood this 10 years ago with BDR1

©EDB 2026 — ALL RIGHTS RESERVED.

LET'S TRY IT

A PING-PONG DEMO

Two plain PostgreSQL nodes, bi-directional logical replication, one table.

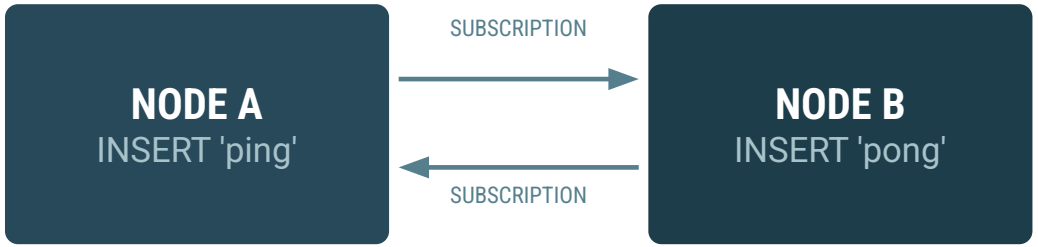


THE SETUP

```
-- on BOTH nodes   (wal_level = logical)
CREATE TABLE pingpong (
  id  int PRIMARY KEY,
  msg text);

CREATE PUBLICATION pp
  FOR TABLE pingpong;

-- on A   (on B: point it at A)
CREATE SUBSCRIPTION from_b
  CONNECTION 'host=node-b dbname=demo'
  PUBLICATION pp
  WITH (origin = none,
        copy_data = false);
```



Each node publishes the table and subscribes to the other. Write on either side; the row shows up on both.

WHAT YOU NEED

PostgreSQL 16 or later, wal_level = logical (on 19, replica also works), and the same table definition on both nodes. No extensions.

TAKEAWAY Six statements, stock PostgreSQL, both nodes writable. Now let's see how long that lasts.

©EDB 2026 — ALL RIGHTS RESERVED.

WHY IT DOESN'T LOOP

ORIGIN = NONE: THE LINE THAT MAKES IT WORK



PostgreSQL 16 lets a subscription ask for local changes only, so nothing replicated is sent back.

ORIGIN = ANY (BEFORE V16)

∞

1. INSERT on A
2. A sends it to B; B applies it
3. B sees a "new" change and sends it back to A
4. A applies it again, and sends it on… forever

ORIGIN = NONE (V16+)

1 hop

1. INSERT on A (no origin: it's local)
2. A sends it to B; B applies it, tagged with a replication origin
3. B's publisher skips it: it isn't local to B
4. Done. One hop, no loop

WHAT'S HAPPENING UNDER THE HOOD

Every change applied by a subscription is stamped with a replication origin. With origin = none, the publisher sends only changes that carry no origin — the ones written locally.

DEMO MOMENT Flip the subscription to origin = any on a table with no primary key, and watch the row count climb.

©EDB 2026 — ALL RIGHTS RESERVED.

FAILURE #1: WRITE CONFLICTS



Update the same row on both nodes at the same moment.

QUIET KIND

t=0 A: UPDATE call SET holder = 'A' B: UPDATE call SET holder = 'B'

t=2 each node applies the other's change

t=3 A holds 'B'. B holds 'A'. No error, nothing in the log.

LOUD KIND

Duplicate keys stop the apply worker until someone acts. Fix the local row and it retries on its own, or run ALTER SUBSCRIPTION … SKIP (lsn = …) with the LSN from the log. SKIP drops the whole remote transaction, not just the row.

WHAT YOU COULD DO

Give every row a home node and route writes by key or region.

Write reconciliation jobs, or model the data as append-only.

Turn on track_commit_timestamp. PostgreSQL 18 logs the conflict, so you can at least see it.

STILL MISSING A rule that picks the same winner on every node, and routing you don't have to build yourself.

©EDB 2026 — ALL RIGHTS RESERVED.

CONFLICT DETECTION: POSTGRES NOW SEES THE COLLISION



PostgreSQL 18 detects and logs apply conflicts. PostgreSQL 19 adds the one it couldn't see before.

DETECT AND LOG (v18)

7 types

1. Stops apply (ERROR): insert_exists, update_exists, multiple_unique_conflicts
2. Applied, remote wins (LOG): update_origin_differs, delete_origin_differs
3. Skipped (LOG): update_missing, delete_missing
4. Each one counted in pg_stat_subscription_stats (confl_*)

+ UPDATE_DELETED (v19)

+1

1. A deletes a row while B updates it
2. By default, even on 19: reported as update_missing
3. WITH (retain_dead_tuples = true): reported as update_deleted
4. Cost: a slot holds VACUUM back; max_retention_duration caps it

WHAT IT STILL DOESN'T DO

Nothing picks a winner. An incoming update is applied as it arrives, so the last change to arrive wins, and two nodes that crossed still end up holding different values. (origin_differs and update_deleted are only detected with track_commit_timestamp = on.)

DEMO MOMENT Rerun the call swap with track_commit_timestamp = on. Both logs report update_origin_differs. The values still swap.

©EDB 2026 — ALL RIGHTS RESERVED.

FAILURE #2: IDENTIFIERS



Both nodes use a plain local sequence for the primary key.

WHAT BREAKS

```
-- Node A                -- Node B
INSERT ... id = 5        INSERT ... id = 5
ERROR: conflict detected on relation
  "public.t": conflict=insert_exists  (both sides)
```

Both nodes call nextval() and get 5. Both insert id = 5. Each replicates to the other and hits an insert_exists conflict. Both apply workers stop, and replication is down in both directions.

WHAT YOU COULD DO

Odd / even: A counts 1, 3, 5 and B counts 2, 4, 6 (INCREMENT 2, different start).

Add a third node and all three move to INCREMENT 3, restarted above max(id): A 7, 10, 13 · B 8, 11, 14 · C 9, 12, 15.

Or give each node a range to hand out, or use UUIDs (PostgreSQL 18 has uuidv7()).

STILL MISSING Ids that don't depend on how many nodes exist, with nobody refilling ranges or changing key types.

©EDB 2026 — ALL RIGHTS RESERVED.

SEQUENCE SYNC: CATCH THE SUBSCRIBER UP



PostgreSQL 19 can copy sequence values to a subscriber. One way, on demand, built for cutover.

BEFORE V19

id = 1

1. Publisher has handed out ids 1 to 1000; rows replicate
2. Sequences don't, so the subscriber's is still at its start
3. Cut writes over to the subscriber
4. nextval() returns 1: duplicate key on the first insert

FOR ALL SEQUENCES (V19)

id = 1001

1. CREATE PUBLICATION p FOR ALL SEQUENCES
2. CREATE SUBSCRIPTION copies each sequence's value once
3. Before cutover: ALTER SUBSCRIPTION s REFRESH SEQUENCES
4. Refresh is async: wait for 1000, then nextval() returns 1001

WHAT'S HAPPENING UNDER THE HOOD

Sequence changes aren't logically decoded. A sequencesync worker reads pg_get_sequence_data() on the publisher and sets the value locally, the same as setval(). It's one way, all sequences or none, nothing moves between refreshes, and a refresh overwrites the subscriber's value even if it had gone higher.

STILL NOT MULTI-WRITER Sync it in both directions and a refresh can rewind a node. It copies one value, one way. It doesn't hand out unique ids.

©EDB 2026 — ALL RIGHTS RESERVED.

DDL

FAILURE #3: SCHEMA CHANGES



Add a column on Node A, then write to it, and the stream to Node B stops.

WHAT BREAKS

```
-- Node A only
ALTER TABLE pingpong ADD COLUMN note text;
INSERT INTO pingpong VALUES (3,'ping','hello');
-- Node B log:
ERROR: target relation "public.pingpong"
  is missing replicated column: "note"
```

Row 3 is written on Node A after the ALTER; row 4 is written on Node B. B's apply worker fails on row 3, even though it doesn't use the new column, and keeps retrying, so A → B is stuck while B → A keeps flowing. Node A ends with 1, 2, 3, 4 and Node B with 1, 2, 4, and only B's log says why.

WHAT YOU COULD DO

Run every migration on every node, in an agreed order, through one tool.

Publish an explicit column list: a new column on Node A stays local until you add it on Node B, then to the list.

STILL MISSING A global order, a lock, a drain of in-flight writes, and a way to notice a node has drifted.

©EDB 2026 — ALL RIGHTS RESERVED.

WHY "JUST RUN IT EVERYWHERE" ISN'T ENOUGH



Even the careful DIY answer runs into four problems.

01

ORDER & CONCURRENCY

Once one node has the new column, its next write to that table breaks every node that doesn’t. Two admins running conflicting DDL on two nodes can stop replication outright.

02

THE STATEMENT, NOT THE RESULT

Replaying the statement means it must behave identically on every node: same encoding, same settings, same extension versions. Anything non-deterministic diverges.

03

DDL VS. ROWS IN FLIGHT

Add a NOT NULL, unique or CHECK constraint while rows are queued and those rows can fail to apply. Something has to drain the writes first.

04

NOT EVERYTHING REPLICATES

VACUUM, ANALYZE, LOCK, temp tables and ALTER SYSTEM are per node by nature. Long-running DDL and mixed major versions (rolling upgrades) need special care.

A production answer needs one global ordering, a lock, and a queue drain. That is a distributed system, and a runbook doesn't get you there.

©EDB 2026 — ALL RIGHTS RESERVED.



WHY IS PGD RELEVANT

©EDB 2026 — ALL RIGHTS RESERVED.

THE PATTERN

EVERY WORKAROUND IS A CLUSTER-MANAGEMENT TASK



Look at what you were doing by hand.

WHAT YOU DID BY HAND	THE CAPABILITY IT STANDS FOR	WHERE PGD PUTS IT
Decided which node writes which rows	**Ownership and conflict policy**	Conflict resolution, groups, write leaders
Split the ids between the nodes	**Allocation**	Global sequences: galloc, SnowflakeId
Ran every migration on every node, in order	**Agreement and ordering**	Raft-ordered global DDL lock
Reconfigured it all when a node joined	**Membership and routing**	Node groups, Connection Manager

THE POINT PostgreSQL gives you logical replication. Active-active needs a cluster manager on top of it.

©EDB 2026 — ALL RIGHTS RESERVED.

THE BIGGER PICTURE

LOGICAL REPLICATION GETS BETTER BUT DOESN'T MANAGE THE CLUSTER



Ten releases of steady progress. It's still a transport, the same way streaming replication is.

v16
origin = none, parallel apply, decode on standby

v17
Failover slots, pg_createsubscriber, slots survive pg_upgrade

v18
Conflict detection and stats, generated columns

v19
Sequence sync, EXCEPT, update_deleted, logical without a restart

WHAT POSTGRES SHIPS	WHAT YOU STILL NEED AROUND IT	WHO DOES IT
Physical Streaming replication	Leader election, fencing, automatic failover, a stable endpoint	**Patroni (with etcd, HAProxy)** **EFM** **CloudNativePG (Kubernetes)**
Logical Streaming Replication	Consensus, DDL ordering, unique ids, conflict resolution, write routing, durability rules	**PGD**

THE POINT Streaming replication didn't make Patroni unnecessary. Better logical replication doesn't make a cluster manager unnecessary.

©EDB 2026 — ALL RIGHTS RESERVED.

THE CONTROL PLANE

WHAT A CLUSTER MANAGER ADDS



Consensus agrees. Groups organise. Connection Manager routes. Commit scopes decide durability.

APPLICATIONS connect to the local endpoint and read/write locally

CONNECTION MANAGER routing · pooling · read/write and read-only ports, inside Postgres on every node

TOP GROUP cluster-wide Raft consensus · DDL ordering · global sequences

BIDIRECTIONAL LOGICAL REPLICATION BETWEEN GROUPS · ONE CLUSTER

©EDB 2026 — ALL RIGHTS RESERVED.

THE CONTROL PLANE

WHAT A CLUSTER MANAGER ADDS

Consensus agrees. Groups organise. Connection Manager routes. Commit scopes decide durability.



APPLICATIONS connect to the local endpoint and read/write locally

CONNECTION MANAGER routing · pooling · read/write and read-only ports, inside Postgres on every node

TOP GROUP cluster-wide Raft consensus · DDL ordering · global sequences

DATA GROUP · LAS PALMAS

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · TENERIFE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · LANZAROTE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

BIDIRECTIONAL LOGICAL REPLICATION BETWEEN GROUPS · ONE CLUSTER

EDB POSTGRES AI

©EDB 2026 — ALL RIGHTS RESERVED.

THE CONTROL PLANE

WHAT A CLUSTER MANAGER ADDS



Consensus agrees. Groups organise. Connection Manager routes. Commit scopes decide durability.

APPLICATIONS connect to the local endpoint and read/write locally

CONNECTION MANAGER routing · pooling · read/write and read-only ports, inside Postgres on every node

TOP GROUP cluster-wide Raft consensus · DDL ordering · global sequences

DATA GROUP · LAS PALMAS

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · TENERIFE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · LANZAROTE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

BIDIRECTIONAL LOGICAL REPLICATION BETWEEN GROUPS · ONE CLUSTER

CONSENSUS (RAFT)

Orders DDL, hands out sequence ranges, elects leaders. A majority decides, and it stays out of your write path.

GROUPS & SUBGROUPS

A cluster is a group of groups. HA, routing and policy attach to the group, not the host.

CONNECTION MANAGER

Sends writes to the current write leader and reads to the rest. Replaces the proxy and the pooler.

COMMIT SCOPES

Durability as a rule on groups, e.g. a MAJORITY of the origin group plus ANY 1 of another.

EDB POSTGRES AI

©EDB 2026 — ALL RIGHTS RESERVED.

THE CONTROL PLANE

WHAT A CLUSTER MANAGER ADDS



Consensus agrees. Groups organise. Connection Manager routes. Commit scopes decide durability.

APPLICATIONS connect to the local endpoint and read/write locally

CONNECTION MANAGER routing · pooling · read/write and read-only ports, inside Postgres on every node

TOP GROUP cluster-wide Raft consensus · DDL ordering · global sequences

DATA GROUP · LAS PALMAS

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · TENERIFE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · LANZAROTE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

BIDIRECTIONAL LOGICAL REPLICATION BETWEEN GROUPS · ONE CLUSTER

CONSENSUS (RAFT)

Orders DDL, hands out sequence ranges, elects leaders. A majority decides, and it stays out of your write path.

GROUPS & SUBGROUPS

A cluster is a group of groups. HA, routing and policy attach to the group, not the host.

CONNECTION MANAGER

Sends writes to the current write leader and reads to the rest. Replaces the proxy and the pooler.

COMMIT SCOPES

Durability as a rule on groups, e.g. a MAJORITY of the origin group plus ANY 1 of another.

©EDB 2026 — ALL RIGHTS RESERVED.

THE CONTROL PLANE

WHAT A CLUSTER MANAGER ADDS



Consensus agrees. Groups organise. Connection Manager routes. Commit scopes decide durability.

APPLICATIONS connect to the local endpoint and read/write locally

CONNECTION MANAGER routing · pooling · read/write and read-only ports, inside Postgres on every node

TOP GROUP cluster-wide Raft consensus · DDL ordering · global sequences

DATA GROUP · LAS PALMAS

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · TENERIFE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · LANZAROTE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

BIDIRECTIONAL LOGICAL REPLICATION BETWEEN GROUPS · ONE CLUSTER

CONSENSUS (RAFT)

Orders DDL, hands out sequence ranges, elects leaders. A majority decides, and it stays out of your write path.

GROUPS & SUBGROUPS

A cluster is a group of groups. HA, routing and policy attach to the group, not the host.

CONNECTION MANAGER

Sends writes to the current write leader and reads to the rest. Replaces the proxy and the pooler.

COMMIT SCOPES

Durability as a rule on groups, e.g. a MAJORITY of the origin group plus ANY 1 of another.

©EDB 2026 — ALL RIGHTS RESERVED.

WHAT A CLUSTER MANAGER ADDS



Consensus agrees. Groups organise. Connection Manager routes. Commit scopes decide durability.

APPLICATIONS connect to the local endpoint and read/write locally

CONNECTION MANAGER routing · pooling · read/write and read-only ports, inside Postgres on every node

TOP GROUP cluster-wide Raft consensus · DDL ordering · global sequences

DATA GROUP · LAS PALMAS

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · TENERIFE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

DATA GROUP · LANZAROTE

WRITE LEADER | DATA | DATA

own Raft subgroup · own write leader

```
commit scope: MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

BIDIRECTIONAL LOGICAL REPLICATION BETWEEN GROUPS · ONE CLUSTER

CONSENSUS (RAFT)

Orders DDL, hands out sequence ranges, elects leaders. A majority decides, and it stays out of your write path.

GROUPS & SUBGROUPS

A cluster is a group of groups. HA, routing and policy attach to the group, not the host.

CONNECTION MANAGER

Sends writes to the current write leader and reads to the rest. Replaces the proxy and the pooler.

COMMIT SCOPES

Durability as a rule on groups, e.g. a MAJORITY of the origin group plus ANY 1 of another.

©EDB 2026 — ALL RIGHTS RESERVED.

FIX #1

AUTOMATIC CONFLICT RESOLUTION



A rule decides the winner — every time, on every node, the same way.

DEFAULT: LAST-UPDATE-WINS

Each row change carries a commit timestamp. When two updates collide, the later timestamp wins on every node — deterministically, without a coordinator needing to get involved.

WHEN YOU NEED MORE

Custom conflict triggers can apply business logic (merge, sum, reject) instead of picking a single winner — and every conflict is logged, so nothing resolves silently and invisibly.

WHY THEY AGREE Core Postgres applies whichever change arrives last, so crossed updates swap. PGD keeps the one that committed last, on every node.

RESULT The call swap can't happen again: every node keeps the same winner.

©EDB 2026 — ALL RIGHTS RESERVED.

FIX #2

GALLOC & SNOWFLAKEID



Two ways to make sure no two nodes ever hand out the same id.

GALLOC SEQUENCES

Each node is handed a private range of values to hand out from a shared pool, and requests a new chunk automatically before it runs out.

Ordered within each node, gap-tolerant, no coordination per insert.

```
      2
      3
2000002
2000003
      4
```

SNOWFLAKEID

Every id is built from a timestamp plus the node's own identifier, encoded into one 64-bit value.

Globally unique with zero coordination at all, and roughly sortable by time of creation.

```
1328604344131522560 | 2026-10-21 05:21:39.550 | 1 | 0
1328604344135716864 | 2026-10-21 05:21:39.551 | 1 | 0
1328604390914793472 | 2026-10-21 05:21:50.704 | 2 | 0
1328604390914793473 | 2026-10-21 05:21:50.704 | 2 | 1
1328604809778958336 | 2026-10-21 05:23:30.569 | 1 | 0
```

RESULT id = 5 on Node A and id = 5 on Node B simply can't happen anymore — by construction.

©EDB 2026 — ALL RIGHTS RESERVED.

FIX #3

DDL REPLICATED, SAFELY



PGD treats a schema change as a cluster-wide operation, not a local one.

CAPTURE

Every DDL statement is intercepted at the node where you run it and queued into the replication stream. No scripts to run on the other nodes.

LOCK & ORDER

A majority of nodes grants a global DDL lock, and Raft fixes the order. Two admins can't run conflicting DDL at once, and every node sees changes in the same sequence.

APPLY

Constraint changes also take a relation-level DML lock and wait for in-flight rows to drain. Then the statement runs on every node, in order.

YOU CHOOSE THE TRADE-OFF: BDR.DDL_LOCKING

`auto (default)`

Leader lock if possible, otherwise all.

`all`

Strictest: global lock from a majority.

`dml`

Safe only if DDL runs from a single node.

`off`

Empty schemas and bulk loads only.

RESULT The ALTER TABLE from the demo just works. Run it on any node, and every node gets it, in order.

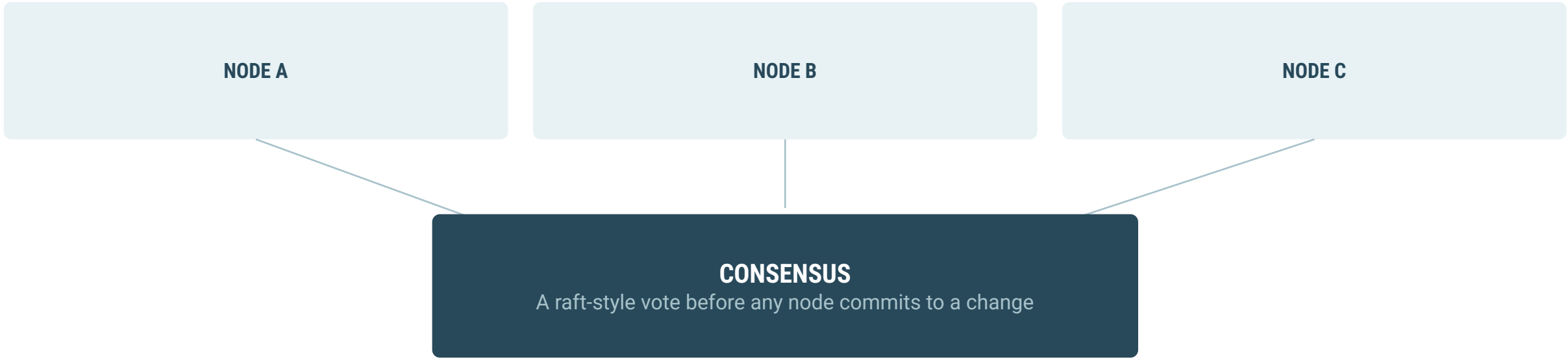
©EDB 2026 — ALL RIGHTS RESERVED.

UNDER THE HOOD

CONSENSUS AT THE CORE OF IT

DDL ordering, leader election and global sequences all lean on one agreement layer.





A majority of nodes has to agree before a DDL change, a sequence chunk, or a leader handover is considered final — that's what keeps every node's view consistent.

©EDB 2026 — ALL RIGHTS RESERVED.

UNDER THE HOOD

WRITE LEADER — GLOBAL AND LOCAL



Some operations still benefit from one node briefly taking the lead.

GLOBAL WRITE LEADER

Elected across the whole cluster for tasks that need a single point of truth for a moment — such as handing out the next galloc chunk — then released the instant the job is done.

LOCAL WRITE LEADER

Elected inside a smaller sub-group — e.g. a pair of nodes running CAMO — so that group can make a strict commit-once guarantee without waiting on the rest of the cluster.

TAKEAWAY Leadership is a role for one task at a time — never a bottleneck that decides who is “the” primary.

©EDB 2026 — ALL RIGHTS RESERVED.

THE PAYOFF

MULTIPLE WRITERS



Every node above is a full read/write member of the cluster — all the time.

LOCAL LATENCY

Each island writes to the node next door instead of crossing the ocean for every request.

NO IDLE HARDWARE

There's no standby quietly waiting for a failure — every node is doing real work today.

GRACEFUL DEGRADATION

Losing a node shrinks capacity; it doesn't stop writes anywhere else in the cluster.

TAKEAWAY This is the property the demo was missing from the start — and the property 112 actually needs.

©EDB 2026 — ALL RIGHTS RESERVED.

WHAT ABOUT CONSISTENCY?



Durability: does the commit survive losing a node? Consistency: do all the nodes agree on what committed?

WHAT POSTGRESQL OFFERS	WHAT IT GIVES YOU	WHAT IT DOESN'T
synchronous_standby_names	A subscription can be a synchronous standby (FIRST n or ANY n). Commit waits until it has flushed the change.	Confirms a peer has the change. Doesn't make two writers agree before they commit.
synchronous_commit on the subscription	Apply-side commit durability. Off is safe: the publisher resends.	Only matters when a publisher is waiting on that subscriber.
synchronized_standby_slots (v17)	Logical subscribers never get ahead of a physical standby, so a failover doesn't strand them.	Protects one primary's failover. Says nothing about several writers.
Conflict detection (v18)	Tells you, after both commits, that two nodes disagreed.	Doesn't prevent it and doesn't settle it.

STILL MISSING A rule for how many nodes, and which ones, must confirm before a commit counts. And a way to agree before anyone commits.

©EDB 2026 — ALL RIGHTS RESERVED.

WITH PGD

COMMIT SCOPES: PICK THE GUARANTEE PER TRANSACTION



Durability and consistency as a rule on groups. Most apps never leave the left half; national infrastructure lives on the right.

MOST APPLICATIONS

LOCAL PROTECT

```
ASYNCHRONOUS COMMIT
```

Default. Committed once it's in the local WAL. Peers catch up.

LAG PROTECT

```
MAJORITY ORIGIN GROUP
LAG CONTROL
(max_lag_time = 30s)
```

Still async, but writers slow down before a peer falls 30s behind.

ADAPTIVE PROTECT

```
MAJORITY ORIGIN GROUP
SYNCHRONOUS COMMIT
DEGRADE ON (timeout =
10s)
```

Majority confirms each commit. After 10s it drops to async instead of stalling.

EMERGENCY SERVICES, PAYMENTS, NATIONAL INFRASTRUCTURE

MAJORITY PROTECT

```
MAJORITY ORIGIN GROUP
SYNCHRONOUS COMMIT
```

No ack to the app until a majority of the group has the commit.

QUORUM COMMIT

```
MAJORITY ORIGIN GROUP
QUORUM COMMIT
```

A majority agrees before anyone commits. No agreement, rolled back everywhere.

CAMO

```
ALL (pair) CAMO
```

If the connection drops during COMMIT, the app asks the partner whether it landed. No double retries.

```
-- only the dispatch path pays for consensus
BEGIN;
SET LOCAL bdr.commit_scope = 'quorum';
UPDATE incident SET unit = 'AMB-07' WHERE id = 4711;
COMMIT;
```

A scope attaches to a group, a session or a single transaction, so reporting and logging stay async on the same cluster.

Each rule also says when a peer counts: ON RECEIVED, REPLICATED, DURABLE or VISIBLE.

THE POINT Most apps never need the right half. For an emergency line it's the requirement, and it's one SET LOCAL away.

EDB POSTGRES AI

©EDB 2026 — ALL RIGHTS RESERVED.

BACK TO THE USE CASE

112 UNDER A STORM



A real storm season, a real island offline — and no one outside the room noticed.

> The two main islands Tenerife and Las Palmas got disconnected from each other.
>
> In PGD, it's one writer among several — calls routed to that island simply land on its nearest healthy neighbour instead.



EDB POSTGRES AI

©EDB 2026 — ALL RIGHTS RESERVED.

BACK TO THE USE CASE

112 UNDER A STORM

A real storm season, a real island offline — and no one outside the room noticed.



The two main islands Tenerife and Las Palmas got disconnected from each other.

In PGD, it's one writer among several — calls routed to that island simply land on its nearest healthy neighbour instead.



OUTCOME

No failover.
No cutover script.
No dropped call.

EDB POSTGRES AI

©EDB 2026 — ALL RIGHTS RESERVED.

BACK TO THE USE CASE

112 UNDER A STORM

A real storm season, a real island offline — and no one outside the room noticed.



The two main islands Tenerife and Las Palmas got disconnected from each other.

In PGD, it's one writer among several — calls routed to that island simply land on its nearest healthy neighbour instead.



OUTCOME

No failover.
No cutover script.
No dropped call.

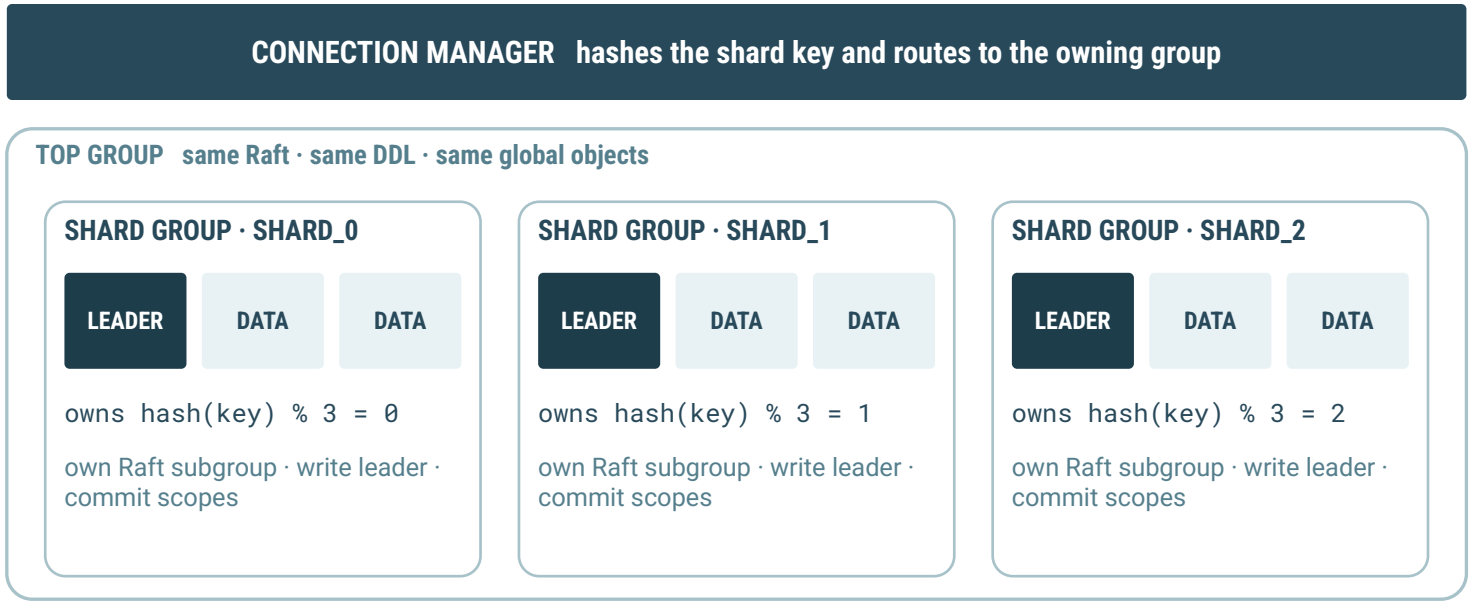
Data gets synchronized

©EDB 2026 — ALL RIGHTS RESERVED.

SHARDING: ONE MORE KIND OF GROUP



Active-active solves availability. Sharding uses the same machinery to solve write scale.



A SHARDED TABLE IS STANDARD DDL

```
CREATE TABLE orders (
  order_id bigint, customer_id bigint,
  PRIMARY KEY (order_id, customer_id))
  PARTITION BY HASH (customer_id)
  WITH (pgd.table_distribution
        = 'distributed');
```

ROUTE DIRECT BY KEY

```
SET bdr.target_shard = 'bigint:42';
```

INHERITED, UNCHANGED

Raft subgroup, write leader, Connection Manager routing, commit scopes, rolling upgrades, online DDL.

WHAT'S NEW

A flag marking the group as a shard owner, a hash-to-group map, and cross-shard access through a foreign-data wrapper.

WHY IT MATTERS

No coordinator tier in the write path, and the runbooks you built for a data group apply to a shard group.

TAKEAWAY A shard group is a PGD group that owns a slice of the rows. Write scale, without giving the HA story back.

©EDB 2026 — ALL RIGHTS RESERVED.





CLOSING WORDS

Zero downtime isn't a feature request — for some systems, it's the whole point.

EDB POSTGRES AI

©EDB 2026 — ALL RIGHTS RESERVED.

THE PAYOFF

ACTIVE-ACTIVE WHEN IS NEEDED





It’s not just about adding more 9’s

MULTIPLE WRITABLE NODES

Not for performance / No Scale-out
Not for adding 9’s
But because the system needs autonomous writable nodes

THE DATABASE CLUSTER MUST SUPPORT

DDL Replication
Synchronisation of sequences
or Distributed sequences
Conflict Resolution
Replication synchronisation capabilities

EXTRA BENEFITS

It’s a cluster management system
Rolling Major Upgrades
Encryption key rotation

TAKEAWAY PGD solves specific use cases and it has been working in Mission Critical Projects for several years.

©EDB 2026 — ALL RIGHTS RESERVED.

Thank you

PGD https://www.enterprisedb.com/docs/pgd/latest/



https://www.linkedin.com/in/theadamwright/
https://github.com/theadamwright/theadamwright



https://www.linkedin.com/in/boriss-mejías-4637401
https://github.com/bmejias

©EDB 2026 — ALL RIGHTS RESERVED.