A journey into PostgreSQL logical replication

José Neves - December 2023

PostgreSQL Monolithic Database

Track evolved on top of a monolithic transactional database. And we have been facing more and more challenges on the data side of things. Especially when it comes to building reporting features over normalized data structures.

Moving data around

We want to build bigger, faster, more complete reporting features.

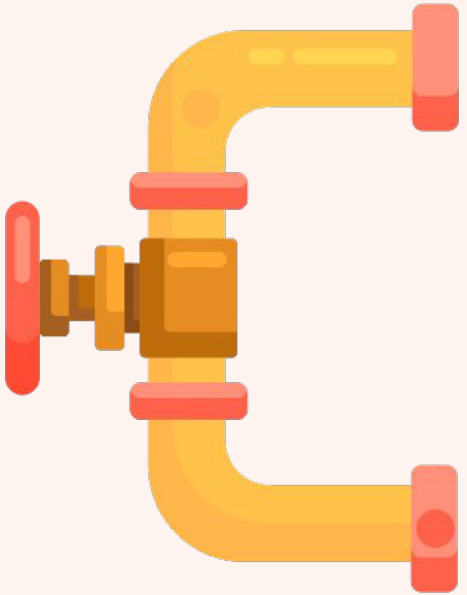
For that, we need to provide fast access to data across different dimensions.

In sum, we needed to modernize our data stack, from ingestion, and warehousing, to finally building the visualization tools that our users love.

CDC pipeline

We needed to capture data changes happening on our transactional databases. Not really a single, or periodic, extraction.

We need a **CDC pipeline**, providing us with a reliable data stream of changes happening on the transactional side of things in close to real-time.



Extract, transform, … serve

Workers would tap into the data change stream, and make sure that the transformations that we chose to do, were reflecting the last changes that users made to their data.

The resulting data, would then be ready to be used and served with as little computing as possible.

Postgres Logical Replication

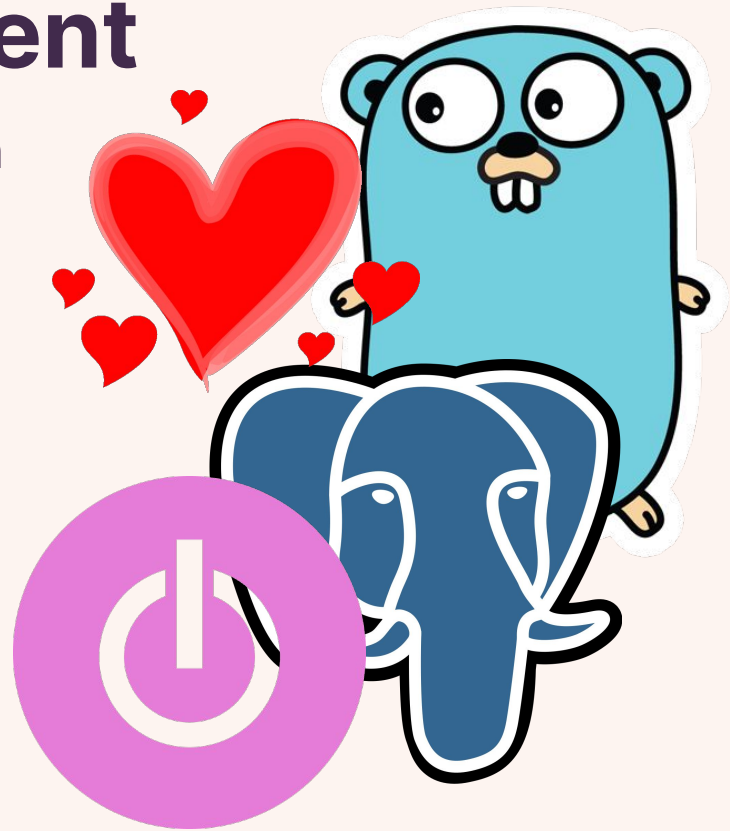
We leveraged PostgreSQL logical replication to stream data to a messaging service. Creating our CDC pipeline.

We then transform that data to create OLAP data structures and keep them up-to-date in close to real-time.

At the other end, we still use Postgres to transform, keep, and serve our report-friendly (OLAP) datasets. However, modularization is in place to allow us to replace any of the technologies used in the different gears of the system.

Logical Replication Client

Set out to implement a small logical replication wrapper in GoLang, using **pglogrepl** *(github.com/jackc/pglogrepl)*.



Simplistic Implementation

```
rawMsg, _ := w.getReplicationMessage()
msg, _ := rawMsg.(*pgproto3.CopyData)

switch msg.Data[0] {
case pglogrepl.PrimaryKeepaliveMessageByteID:
pkm, _ := pglogrepl.ParsePrimaryKeepaliveMessage(msg.Data[1:])
if pkm.ReplyRequested {
      if err = w.sendStandbyStatus(w.ctx); err != nil {
            return err
      }
}
case pglogrepl.XLogDataByteID:
      return w.processDataMessage(msg.Data[1:])
}
```

```
switch m := logicalMsg.(type) {
      case *pglogrepl.RelationMessage:
            set.Add(*m)
            return nil
      case *pglogrepl.InsertMessage:
            return doSomeInsertStuff()
      case *pglogrepl.UpdateMessage:
            return doSomeUpdateStuff()
      case *pglogrepl.DeleteMessage:
            return doSoneDeleteSuff()
}
```

Data changing events

Logical replication streams data that is already committed.

It won't be rolled back. And **DDL changes will not be there anyway**.

So, we thought: no use keeping track of events that don’t change data. Like transaction begins and commits.

We cared only for: Inserts, updates, deletes, maybe truncates.

Spoiler! It was a crappy idea.

Logical replication slot

The slot is persistent, regardless of an active connection. And will store the consumption status, using two offsets, restart, and flush LSN*.

LSNs are pointers to given locations in the WAL. Logical replication clients must periodically push consumption status to update the slot.

* https://www.postgresql.org/docs/current/datatype-pg-lsn.html

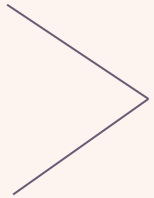
LSN Examples

BEGIN 4/98EE65C0	START TRANSACTION;
INSERT 4/98EE65C0	INSERT INTO track (description, duration) VALUES (‘Reading’, 360000);
UPDATE 4/98EE66D8	UPDATE track_total SET duration = duration + 360000;
UPDATE 4/98EE6788	UPDATE user SET entries = entries + 1;
COMMIT 4/98EE6830	COMMIT;
BEGIN 4/98EE6950	START TRANSACTION;
UPDATE 4/98EE6AD8	UPDATE track SET duration = duration + 360000;
UPDATE 4/98EE6D28	UPDATE track_total SET duration = duration + 360000;
UPDATE 4/98EE6DD8	UPDATE users SET entries = entries + 1;
COMMIT 4/98EE6F30	COMMIT;
BEGIN 4/98EE6F68	
INSERT 4/98EE6F68	INSERT INTO users (email, password) VALUES (‘@.com’, ‘...’);
COMMIT 4/98EE7040	

👀Simplifying notation

Let’s simplify references to LSNs from this point forward, so sequence it’s more perceptible. Dropping the hexadecimal representation.

BEGIN	4/98EE7160
INSERT	4/98EE71C8
UPDATE	4/98EE76D8
UPDATE	4/98EE7788
COMMIT	4/98EE7830

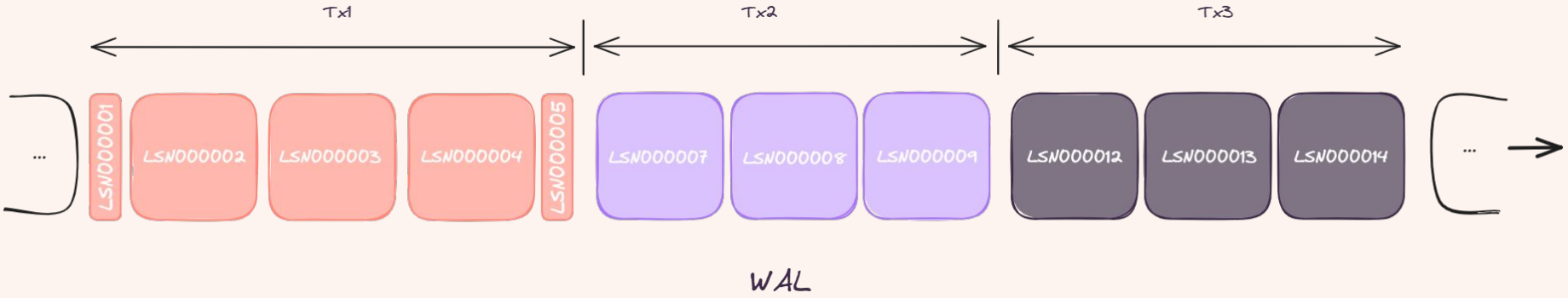


BEGIN	**LSN000001**
INSERT	**LSN000002**
UPDATE	**LSN000003**
UPDATE	**LSN000004**
COMMIT	**LSN000005**

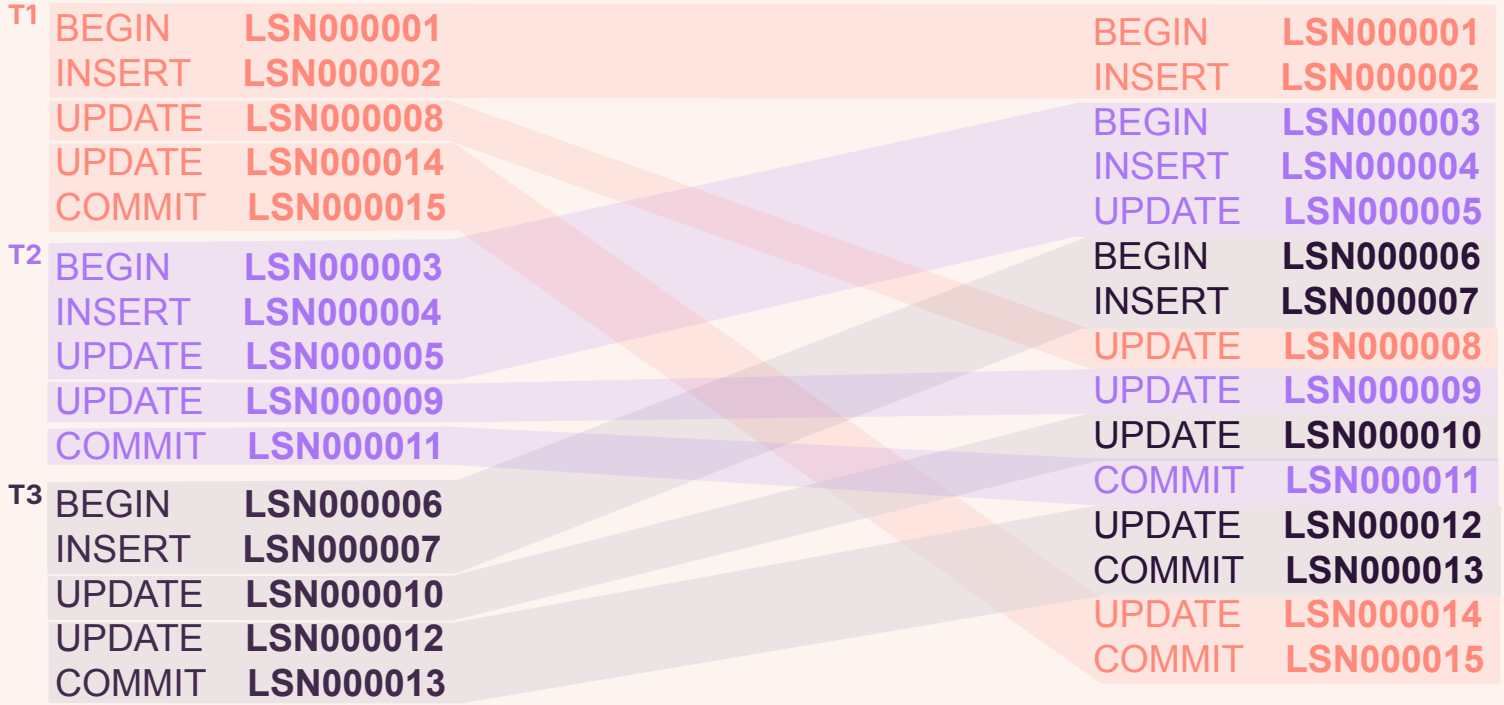
Without Concurrency

T1	BEGIN	LSN000001	BEGIN	LSN000001
	INSERT	LSN000002	INSERT	LSN000002
	UPDATE	LSN000003	UPDATE	LSN000003
	UPDATE	LSN000004	UPDATE	LSN000004
	COMMIT	LSN000005	COMMIT	LSN000005
T2	BEGIN	LSN000006	BEGIN	LSN000006
	INSERT	LSN000007	INSERT	LSN000007
	UPDATE	LSN000008	UPDATE	LSN000008
	UPDATE	LSN000009	UPDATE	LSN000009
	COMMIT	LSN000010	COMMIT	LSN000010
T3	BEGIN	LSN000011	BEGIN	LSN000011
	INSERT	LSN000012	INSERT	LSN000012
	UPDATE	LSN000013	UPDATE	LSN000013
	UPDATE	LSN000014	UPDATE	LSN000014
	COMMIT	LSN000015	COMMIT	LSN000015

Without Concurrency



With Concurrency



With Concurrency

toggl

Concurrency

As we were intentionally disregarding transactions, all we had to work with were data-changing events and their offsets.

Operation Order:

INSERT	LSN000002
UPDATE	LSN000008
UPDATE	LSN000011
INSERT	LSN000004
UPDATE	LSN000005
UPDATE	LSN000009
INSERT	LSN000007
UPDATE	LSN000010
UPDATE	LSN000014

Log:

INSERT	LSN000002
INSERT	LSN000004
UPDATE	LSN000005
INSERT	LSN000007
UPDATE	LSN000008
UPDATE	LSN000009
UPDATE	LSN000010
UPDATE	LSN000011
UPDATE	LSN000014

Replication Stream:

INSERT	LSN000004
UPDATE	LSN000005
UPDATE	LSN000009
INSERT	LSN000007
UPDATE	LSN000010
UPDATE	LSN000014
INSERT	LSN000002
UPDATE	LSN000008
UPDATE	LSN000011

Attempting to live with it

Still set on using only data-changing events we attempted to keep track of logical replication offsets, and make sense of the conflicting results, not realizing that such an approach **would always lead to data losses**, happening in the most varied ways.

😱 Bad assumptions

We assumed that LSNs were incremental cross-transactions.

Every logical replication event comes with a LSN offset which corresponds to a location in the WAL, but logging happens concurrently.

We cared only about data events.

But this only works if your client is always up and running, and never runs into trouble.

Commutative Property

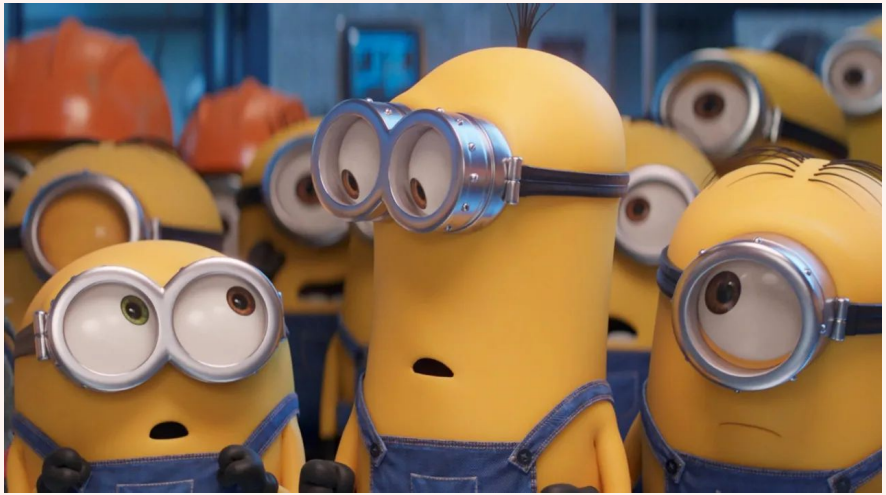
Due to the nature of the transformations that we were implementing, data order didn’t really matter.
Deletes for us were always updates, translated to +0-value.

INSERT	**LSN000002**	**+1**	**+1**
INSERT	**LSN000004**	+3	-3+1
UPDATE	**LSN000005**	-3+1	-1+2
INSERT	**LSN000007**	+2	+3
UPDATE	**LSN000008**	-1+2	-1+4
UPDATE	**LSN000009**	-1+4	-1+3
UPDATE	**LSN000010**	-2+1	-1+2
UPDATE	**LSN000011**	-1+3	-2+1
UPDATE	**LSN000014**	-1+2	+2
		10	**10**

Unraveling inconsistent data mysteries

Working as expected in a controlled environment, but we lacked conditions for testing under concurrency.

Mitigation measures were actually so effective that were masking the issue.



The wrongs: Incremental LSNs

We first attempted to solve the issue by tracking the current LSN and discarding data with offsets smaller than the current position. We were under the wrong impression that we would have incremental LSN offsets.

T1	BEGIN	LSN000001
	INSERT	LSN000002
	UPDATE	LSN000005
	COMMIT	LSN000006
T2	BEGIN	LSN000003
	UPDATE	LSN000004
	UPDATE	LSN000007
	COMMIT	LSN000008

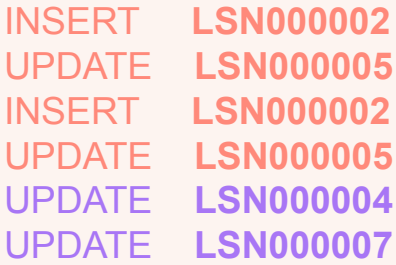
INSERT	LSN000002
UPDATE	LSN000005
~~UPDATE~~	~~LSN000004~~
UPDATE	LSN000007

In this example by using “5” offset, we would discard the first data event of the next transaction.

The wrongs: commit ops offsets

After figuring out that our data events presented “out-of-order” offsets. We fell into another nuance when we stopped discarding data based on expecting an incremental offset:
We would duplicated it on reconnection, by committing operation offsets.

If our logical replication client exited at “5”, we would commit LSN “5” to pg.



But on reconnection we would receive the last transaction all over, duplicating data.

The realization

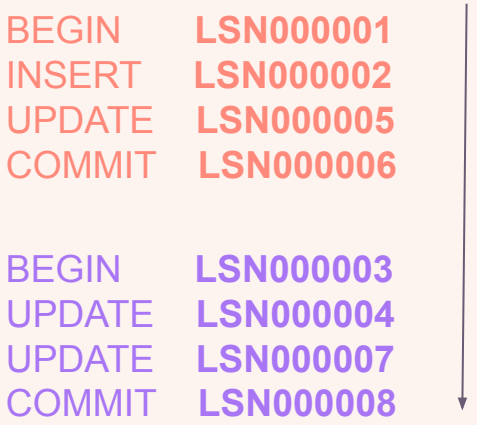
A few key points became clear laying the path to a proper implementation:

- ✓ **Logical replication works over TCP**
- ✓ **COMMIT LSN offsets are ensured to be incrementally-sequential**. And only committing that offset to the replication slot will mark the transaction as consumed.
- ✓ Replication transaction stream is sorted by transaction end offsets

The realization

- ✓ When we commit LSN offsets that pertain to mid-transaction events, pg **will resent the whole transaction again** upon reconnection.

- ✓ Events for a given transaction are always streamed together, regardless of log positioning.



Incremental COMMIT LSNs

The only LSN offsets warrantied to be “incrementally-sequential” between transactions are COMMIT offsets. These mark the end of a transaction.

Incremental COMMIT LSNs

Committing offset “6” doesn’t prevent all the data in the next transaction from being sent.

All transaction events are resent if the offset that we committed to the replication slot is not a transaction end - or bigger.

Transactional integrity

We always commit to the replication slot the transaction end LSN.

And for that, we make sure that we process all data-changing events respecting the transactional integrity, in order to report progress abiding by Postgres rules.

Message sizes

Messaging services will have limited message sizes, but a single update can change millions of rows, generating transactions with millions events.

Possibility of data duplication on chunk split, either on plug-off events, or general erroring.

Data duplication possibilities

The proper use of LSN event offsets fully prevents data loss, it doesn’t prevent data duplication.

In plug-off events, messages that were already delivered to our event messaging service may not end up committed to PostgreSQL replication slot.

Embracing data duplication

We embrace the possibility of data duplication, by keeping track of processed LSNs in our transformation process. While the proper usage of LSN commits to the replication slot ensures that we will not lose any data.

Similar to the process that PostgreSQL itself does on server-to-server logical replication.

Thank you.

toggl

José Neves
@rafalneves

PostgreSQL DBA for Toggl Track