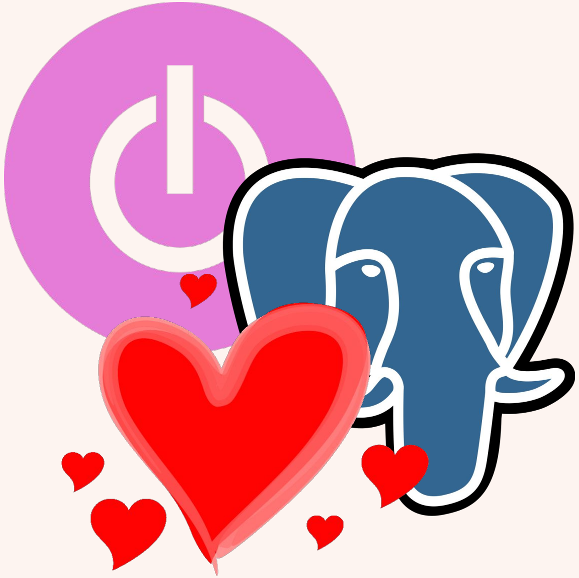
toggl

A journey into PostgreSQL logical replication

The Next Chapter

José Neves - October 2024

The journey: prelude



Toggl Track evolved on top of a monolithic transactional PostgreSQL database.

With an ever-increasing dataset, we started to feel the need to move away from the transactional - normalized - data structure to provide the analytical features that users were asking for.

The journey: the plan

Move it, transform it. Use it.

Leverage logical replication as a means of facilitating close to real-time transformation of our transactional data. Make it the source of a CDC pipeline.

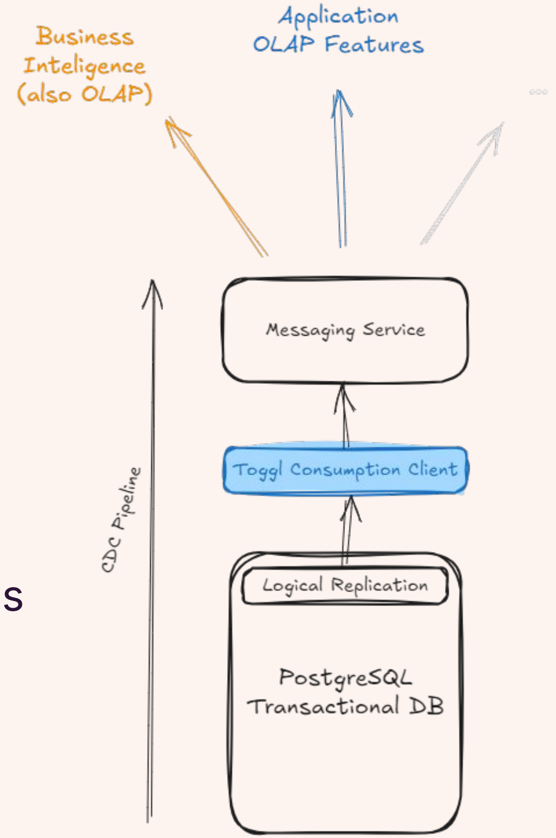
We implemented our own logical replication client in GoLang, using **pglogrepl** *(github.com/jackc/pglogrepl)*.

The journey: the plan

A major product requirement for our analytic features was near **real-time** availability.

Logical replication was our best bet to achieve it. And while at it, we managed to provide the means to “liberalize” access to “data changes”.

Later on used to decrease latency in other processes such as internal BI tools, email notifications, or simply data propagation from sources of truth to other services.



The journey: the bad turn

Caring for data-changing events - only - turned out to be a **bad turn**.

Logical replication streams data that is already committed.

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

The journey: consumption client

While developing our consumption client, our line of thinking was: **no use keeping track of events that don’t change data**.

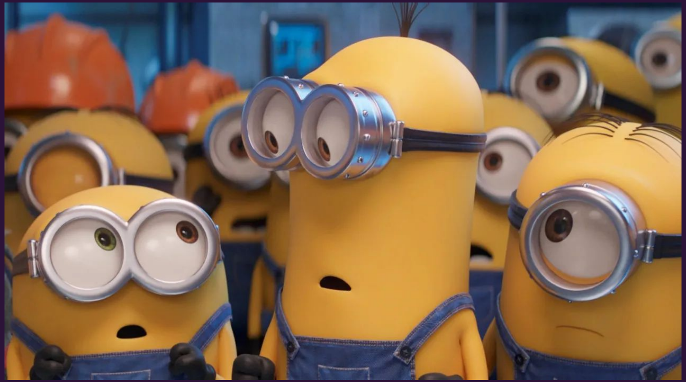
Like transaction begins and commits.

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

Spoiler! It was a crappy idea.

toggl

toggl



Developing our Consumption Client

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

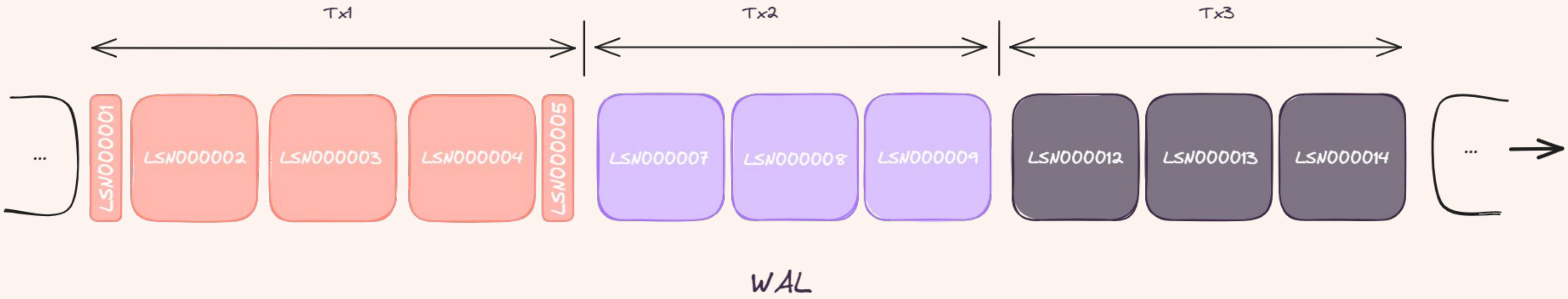
BEGIN 4/98EE65C0	START TRANSACTION;
INSERT 4/98EE65C0	INSERT INTO track (description, duration) VALUES (‘Reading’, 360000);
UPDATE 4/98EE66D8	UPDATE track_total SET duration = duration + 360000;
UPDATE 4/98EE6788	UPDATE user SET entries = entries + 1;
COMMIT 4/98EE6830	COMMIT;
BEGIN 4/98EE6950	START 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	

Without Concurrency

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

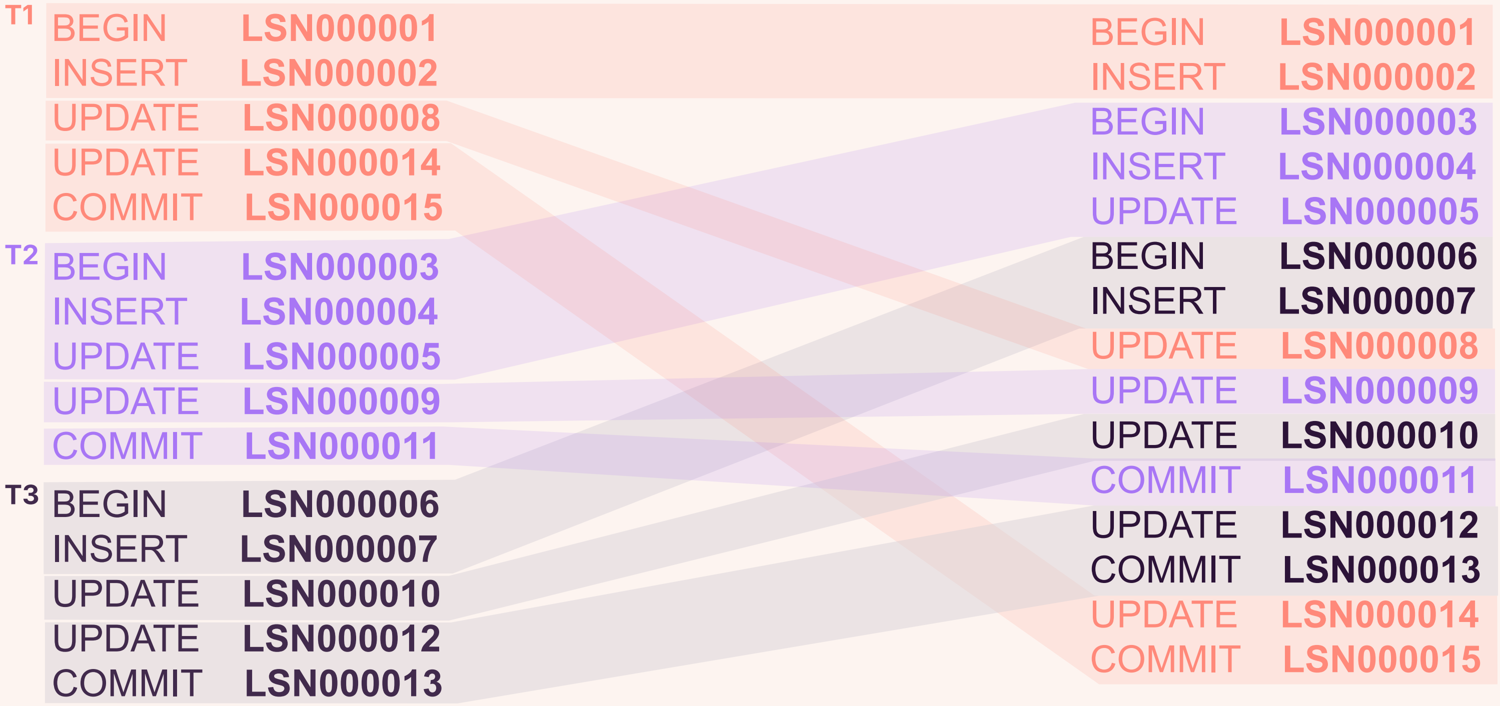
BEGIN	LSN000001
INSERT	LSN000002
UPDATE	LSN000003
UPDATE	LSN000004
COMMIT	LSN000005
BEGIN	LSN000006
INSERT	LSN000007
UPDATE	LSN000008
UPDATE	LSN000009
COMMIT	LSN000010
BEGIN	LSN000011
INSERT	LSN000012
UPDATE	LSN000013
UPDATE	LSN000014
COMMIT	LSN000015

Without Concurrency



toggl

With Concurrency

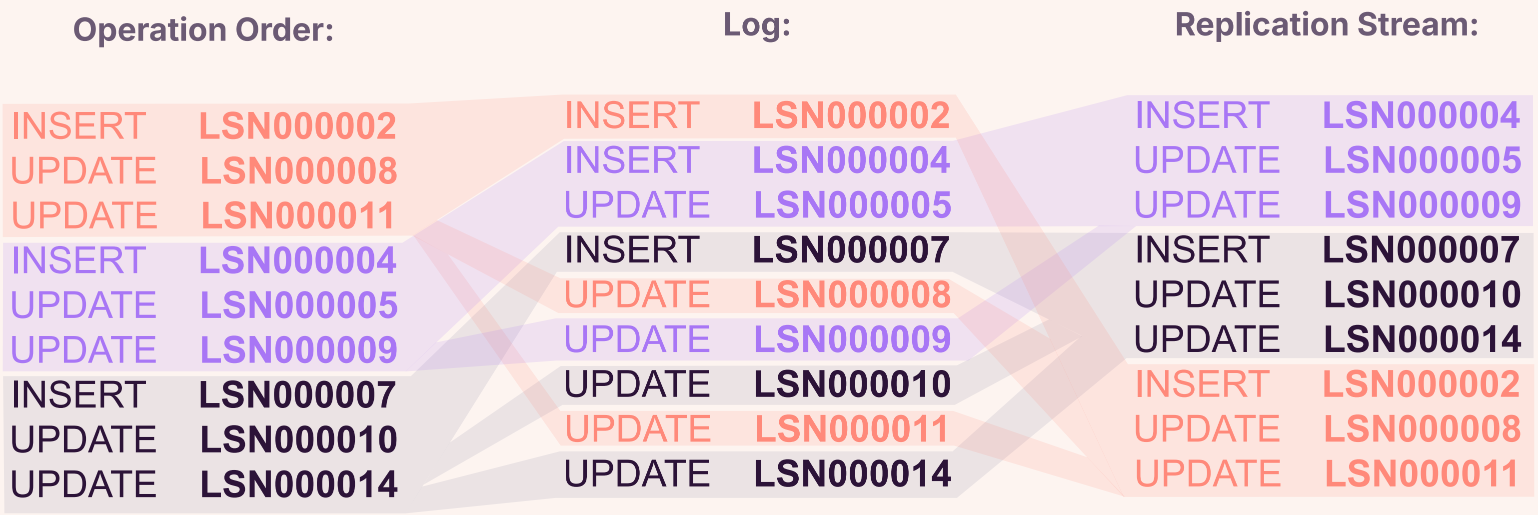


With Concurrency

toggl

Concurrency

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



Fast forwarding...

We attempted to live with the conditions that we found outside dev environments.

A mix of **bad assumptions**, the **ideal conditions for confusion** due to our first use case *(summing up tracked time / commutative property)*, and **lack of knowledge** about logical replication inner works led to dark times trying to figure out why our brand new OLAP data was - sometimes - inconsistent.

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**

Bad: Assuming incremental LSNs

We assumed that LSNs were incremental cross-transactions.

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

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

Bad: Assuming incremental LSNs

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

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

In this example by filtering for LNS > “5” we would discard the first data event from the next transaction.

Bad: Commiting 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 duplicate it on reconnection, by committing operation offsets.

INSERT	**LSN000002**
UPDATE	**LSN000005** ←
UPDATE	**LSN000004**
UPDATE	**LSN000007**

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

INSERT	**LSN000002**
UPDATE	**LSN000005** ←
INSERT	**LSN000002**
UPDATE	**LSN000005**
UPDATE	**LSN000004**
UPDATE	**LSN000007**

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

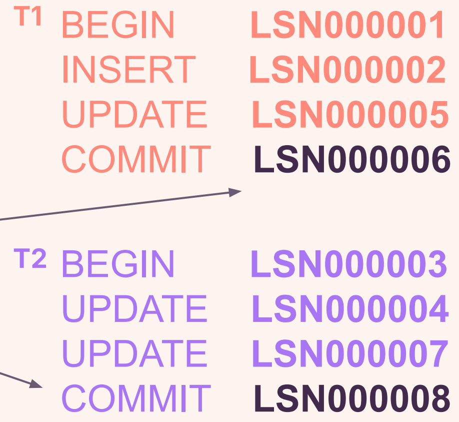
toggl

Key points

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

- ✓ **Logical replication works over TCP**

- ✓ **Only COMMIT LSN offsets are incrementally-sequential**. And **only committing that offset** to the replication slot will mark the transaction as consumed.



Key points

- The replication stream data is sorted data by transaction end offsets

- 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.

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

Key points

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

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

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

Conclusion

By always committing the transaction end LSN we make sure that we process all data-changing events respecting their transactional integrity.

Only **committing the appropriated offset will allow us to correctly manage the slot state**. *That is, without additional stateful logic on the client side.*

The Next Chapter

Our vision

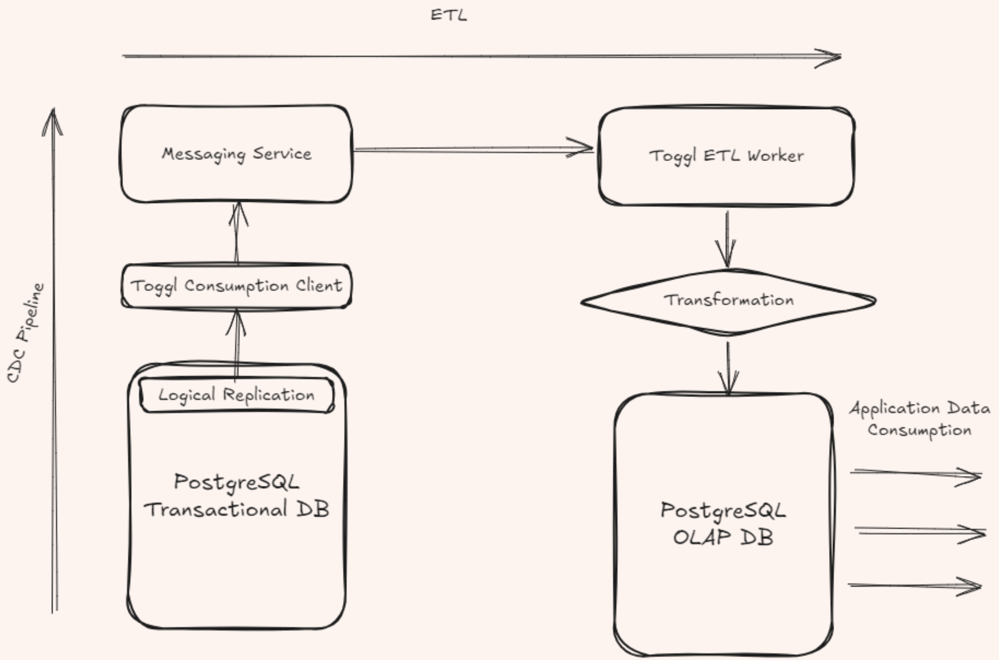
The CDC pipeline created by our logical replication client became the source of data for many of our endeavors:

- Generalist **data aggregations**.
- User-generated reporting.
- Real-time business intelligence reporting.
- Our **authorization system**.

Our vision

On top of our CDC pipeline, we can then achieve whatever transformations are needed.

Scale horizontally and, if we choose to, segregate by feature or general domain.



How it's going: domain segregation

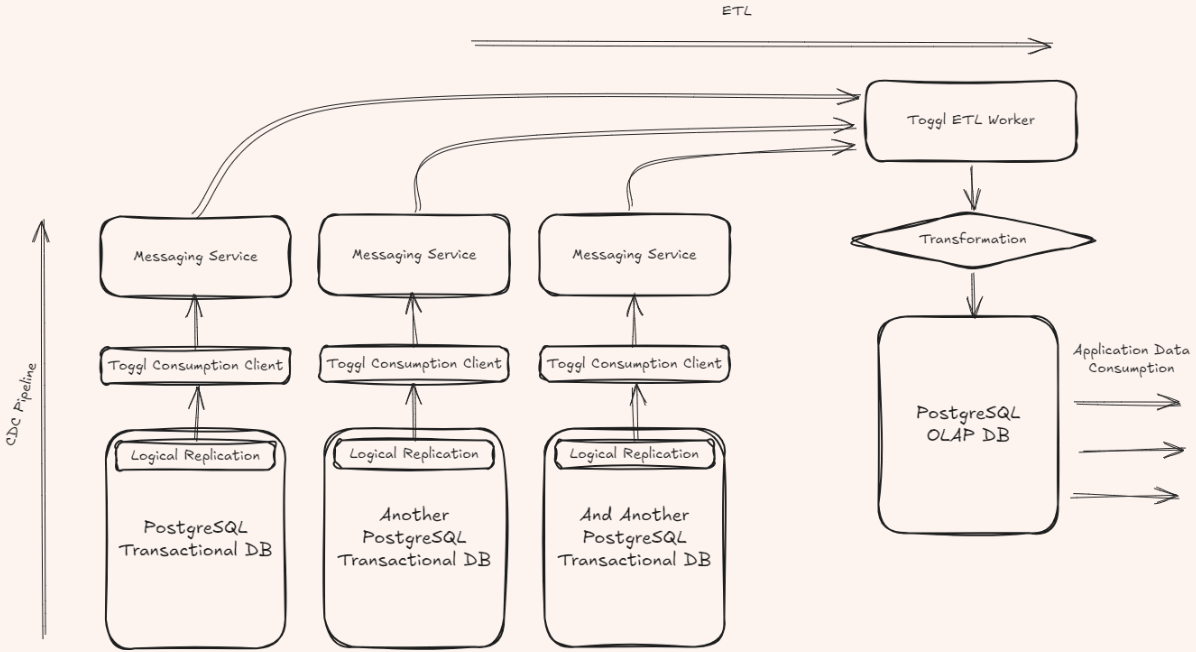
To better scale, we also started **decoupling domains from our main service**.

As a result, our **authorization system** needed to **source data from different DBs**:

- Authorization *(roles, permissions, and their relations to users)*.
- Subscriptions *(plans, features, status)*.
- Remaining transactional data *(subject to business logic)*.

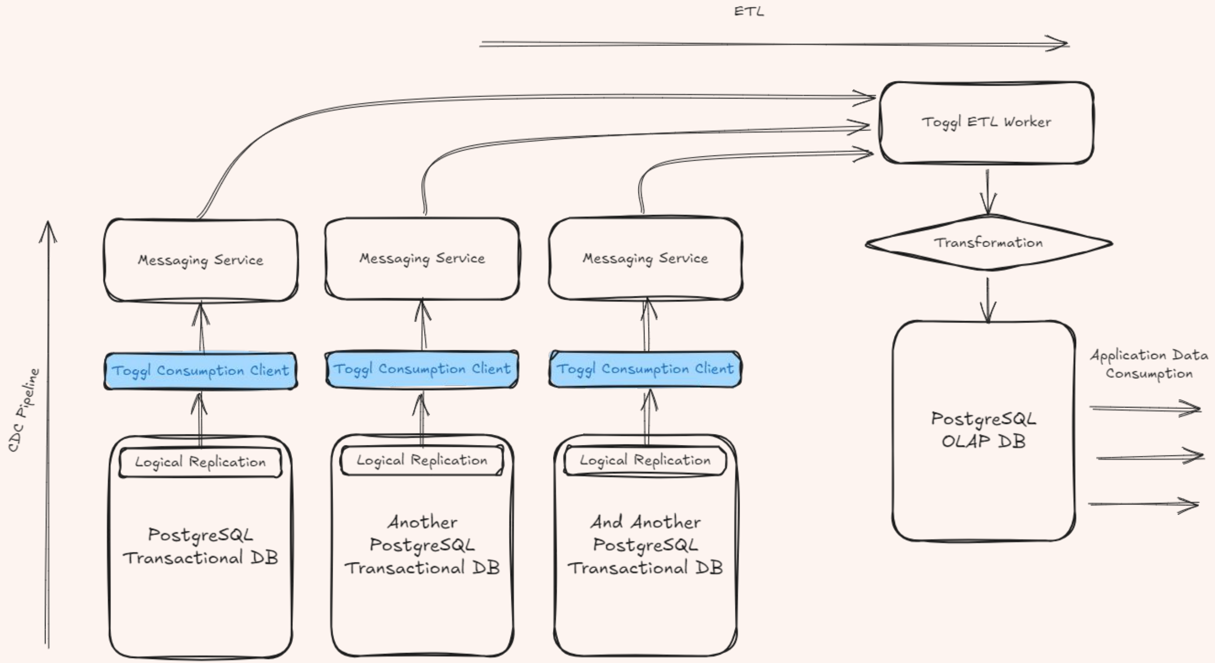
How it's going

As we split domain responsibilities into different services, more sources of data were added.



How it's going: domain segregation

Dealing with logical replication inner works was just the beginning of the journey.



toggl

Authorization use-case

Our freshly developed CDC couldn’t look better to achieve this goal.

We figured that we could simply ingest data from all three sources and pre-compute user session data.

We don’t want users to have to wait very long for their 50k project list to load.

toggl

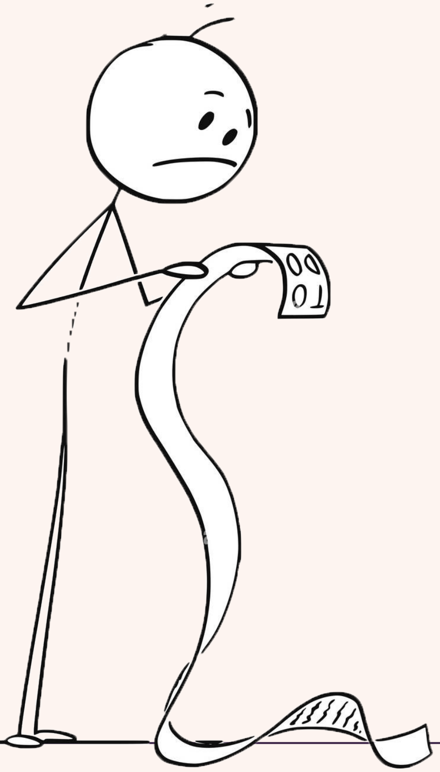
Authorization use-case

Pre-computed Sessions

Will tell us what workspaces a user has access to, with which permissions, and the projects that he can see in them.

Behind the scenes, our **OLAP transformation is pre-generating user sessions** by applying subscription, authorization, and business logic to data changes.

Authorization use-case



Sessions are expected to be **updated fast**. Users won't want to wait very long for their newly created project to be available for use.

That's **challenging**.

Especially if you have users with 100k project lists.

Tips & Tricks

Cross our implementations, while striving for near “real-time” data transformations, we collected a few do’s and don’t do’s.

Tips & Tricks: Embrace duplication

While the proper use of LSN event offsets **fully prevents data loss, it doesn’t prevent data duplication**. On plug-off events, it will happen.

Embrace that possibility. Take advantage of event atomicity on the OLAP transformation and make sure that the operations are idempotent.

Tip?

Use of an **updated** timestamp in your transactional tables. Make sure that every change correctly updates it *(ie.: through triggers)*.
And make use of it in your transformations to dedup.

Tips & Tricks: Bulk updates

Processing data changes individually is challenging.

Prepare your pipeline to deal with events in bulk.

Tip?

Add a small delay in your data collection and bulk upload changes.
Ingest and transform data in bulk.
Use upsert to keep it a single operation.

Tips & Tricks: Bulk updates

```
INSERT INTO transformation (
  user_id,
  total_time,
  updated_at
)
SELECT
  (d->>'user_id')::int  AS user_id,
  (d->>'total_time')::int  AS total_time,
  (d->>'updated_at')::timestamp without time zone  AS updated_at
FROM JSON_ARRAY_ELEMENTS($1::jsonb) d
ON CONFLICT ON CONSTRAINT transformation_pkey
DO UPDATE SET
  total_time = excluded.total_time,
  updated_at = excluded.updated_at
WHERE
  transformation.updated_at <
    excluded.updated_at;
```

toggl

Tips & Tricks: Statement triggers

If we update multiple rows, the transformation will run multiple times with **row-based triggers**. Make sure that logic uses statement changes instead.

Tip?

Use **FOR EACH STATEMENT** triggers to the detriment of **FOR EACH ROW** triggers. If you must use row triggers, use conditioning to make sure that the transformation is only executed if the change is relevant to it.

Tips & Tricks: Statement triggers

```
CREATE TRIGGER after_update AFTER UPDATE ON transformation
  REFERENCING NEW TABLE AS new_table OLD TABLE AS old_table
  FOR EACH STATEMENT EXECUTE FUNCTION olap.update_team_goals();

CREATE OR REPLACE FUNCTION olap.update_team_goals() RETURNS trigger LANGUAGE plpgsql AS $$
  BEGIN
    WITH changes AS (
      SELECT t.team_id, SUM(new_table.total_time - COALESCE(old_table.total_time, 0)) AS total_time
      FROM relations.user_teams t
      JOIN new_table ON t.user_id = new_table.user_id
      LEFT JOIN old_table ON new_table.user_id = old_table.user_id
      GROUP BY t.team_id
    )
    UPDATE olap.team_goals tg SET total_time = total_time + changes.total_time
    FROM changes WHERE tg.team_id = changes.team_id;

    RETURN NULL;
  END;
$$;
```

toggl

Tips & Tricks: Inclusive Indexes

When creating OLAP structures from normalized data changes, you will find yourself looking into relationship tables to find parent ids, *for instance*.

Using **inclusive indexes saves an extra trip to the table** to retrieve the information itself.

Tips & Tricks: Inclusive Indexes

```
CREATE SCHEMA relations;
CREATE TABLE relations.user_teams (
      user_id integer NOT NULL,
      team_id integer NOT NULL,
      CONSTRAINT user_teams_pkey PRIMARY KEY (user_id, team_id)
);

INSERT INTO relations.user_teams (user_id, team_id)
SELECT (random() * 10000)::int AS user_id, (random() * 1000)::int AS team_id
FROM generate_series(1, 1000000)
GROUP BY 1, 2;

SELECT user_id, COUNT(1) FROM relations.user_teams GROUP BY user_id ORDER BY 2 DESC LIMIT 1;

CREATE INDEX simple_user_index ON relations.user_teams (user_id);
EXPLAIN (ANALYSE, VERBOSE) SELECT team_id FROM relations.user_teams WHERE user_id = 1560;

CREATE INDEX inclusive_user_index ON relations.user_teams (user_id) INCLUDE (team_id);
EXPLAIN (ANALYSE, VERBOSE) SELECT team_id FROM relations.user_teams WHERE user_id = 1560;
```

Tips & Tricks: Inclusive Indexes

```
EXPLAIN (ANALYSE, VERBOSE) SELECT team_id FROM relations.user_teams  WHERE user_id = 1560
-----------------------------------------------------------------------------------------
Index Scan using simple_user_index on relations.user_teams  (cost=0.42..4.29 rows=95 width=4)
(actual time=12.193..12.220 rows=128 loops=1)
  Output: team_id
  Index Cond: (user_teams.user_id = 1560)
Planning Time:  0.116 ms
Execution Time:  12.253 ms

vs

Index Only Scan using inclusive_user_index on relations.user_teams  (cost=0.42..3.19 rows=95
width=4) (actual time=0.077..0.089 rows=128 loops=1)
  Output: team_id
  Index Cond: (user_teams.user_id = 1560)
  Heap Fetches: 0
Planning Time:  0.129 ms
Execution Time:  0.112 ms
```

toggl

Tips & Tricks: Generated Columns

In a situation where **sourced data is unstructured**, we may want to expose some properties, so they are directly indexable, for instance.

Make use of generated columns to avoid scanning through unstructured data at run time.

Tips & Tricks: Generated Columns

```
CREATE SCHEMA olap;
CREATE TABLE olap.unstructured (
     dump jsonb NOT NULL
);

INSERT INTO olap.unstructured (dump)
SELECT JSONB_BUILD_OBJECT('user_id', (random() * 10000)::int, 'total_time', (random() * 1000)::int,
'goal', (random() * 1000)::int + 1 ) FROM generate_series(1, 100000000);

CREATE INDEX unstructured_gin_index ON olap.unstructured USING GIN (dump);

EXPLAIN (ANALYSE, VERBOSE) SELECT COUNT(1) FROM olap.unstructured
WHERE (dump->>'total_time')::int >= (dump->>'goal')::int;

ALTER TABLE olap.unstructured ADD COLUMN goal_completed boolean
GENERATED ALWAYS AS ((dump->>'total_time')::int >= (dump->>'goal')::int ) STORED;
CREATE INDEX goal_completed_index ON olap.unstructured (goal_completed);

EXPLAIN (ANALYSE, VERBOSE) SELECT COUNT(1) FROM olap.unstructured WHERE goal_completed;
```

Tips & Tricks: Generated Columns

```
EXPLAIN (ANALYSE, VERBOSE) SELECT COUNT(1) FROM olap.unstructured
WHERE (dump->>'total_time')::int >= (dump->>'goal')::int;
-----------------------------------------------------------------------------------------------
-
Finalize Aggregate  (cost=2292731.42..2292731.43 rows=1 width=8) (actual time=74036.280..74047.599
rows=1 loops=1)
  Output: count(1)
  ->  Gather  (cost=2292731.00..2292731.41 rows=4 width=8) (actual time=74036.072..74047.584
rows=5 loops=1)
        Output: (PARTIAL count(1))...
        ->  Partial Aggregate  (cost=2291731.00..2291731.01 rows=1 width=8) (actual
time=74029.976..74029.977 rows=1 loops=5)
              Output: PARTIAL count(1)...
              ->  Parallel Seq Scan on olap.unstructured  (cost=0.00..2270481.00 rows=8500000
width=0) (actual time=0.477..73068.822 rows=10190650 loops=5)
                    Filter: (((unstructured.dump ->> 'total_time'::text))::integer >=
((unstructured.dump ->> 'goal'::text))::integer)
                    Rows Removed by Filter: 10209350...
Planning Time: 10.810 ms
Execution Time: 74048.038 ms
```

toggl

Tips & Tricks: Generated Columns

```
EXPLAIN (ANALYSE, VERBOSE) SELECT COUNT(1) FROM olap.unstructured WHERE goal_completed;
----------------------------------------------------------------------------------------
-
Finalize Aggregate  (cost=692971.65..692971.66 rows=1 width=8) (actual time=3173.567..3195.634
rows=1 loops=1)
  Output: count(1)
  ->  Gather  (cost=692971.23..692971.64 rows=4 width=8) (actual time=3173.488..3195.625 rows=5
loops=1)
        Output: (PARTIAL count(1))...
        ->  Partial Aggregate  (cost=691971.23..691971.24 rows=1 width=8) (actual
time=3165.396..3165.397 rows=1 loops=5)
              Output: PARTIAL count(1)...
              ->  Parallel  Index Only Scan using goal_completed_index on olap.unstructured
(cost=0.57..660276.85 rows=12677752 width=0) (actual time=0.161..2484.704 rows=10190650 loops=5)
                    Output: goal_completed
                    Index Cond: (unstructured.goal_completed = true)
                    Heap Fetches: 0...
Planning Time:  0.234 ms
Execution Time:  3195.690 ms
```

toggl

Tips & Tricks: JSONB

When handling or storing JSON, the binary format can offer advantages, but using **JSONB functions** for smaller - looping - sub-operations **degrades performance** due to the parsing overhead.

Execute operations in JSON, and cast the result to JSONB instead.

Tips & Tricks: JSONB

```
EXPLAIN (ANALYSE, VERBOSE) SELECT JSONB_BUILD_OBJECT('obj', obj) FROM (
      SELECT JSONB_OBJECT_AGG(user_id, tracked_obj) AS obj FROM (
            SELECT user_id, JSONB_OBJECT_AGG(team_id,  JSONB_BUILD_OBJECT('total_time', (random() * 
1000)::int)) AS tracked_obj  FROM (
                  SELECT (random() * 10000)::int  AS user_id, (random() * 10000)::int  AS team_id
                  FROM generate_series(1, 1000000)
            ) gen
            GROUP BY user_id
      ) agg
) foo;
-----------------------------------------------------------------------------------------------
-
Subquery Scan on foo  (cost=50005.01..50005.03 rows=1 width=32) (actual time=6325.727..6368.489 
rows=1 loops=1)
  Output: jsonb_build_object('obj', foo.obj)
  ->  Aggregate  (cost=50005.01..50005.02 rows=1 width=32) (actual time=5503.496..5503.499 rows=1 
loops=1)
        Output: jsonb_object_agg((..., (jsonb_object_agg(..., jsonb_build_object('total_time',...)
        ->  HashAggregate  (... (actual time=3535.802..4119.971 rows=10001 loops=1)...
Planning Time: 0.174 ms
Execution Time:  6412.477 ms
```

toggl

Tips & Tricks: JSONB

```
EXPLAIN (ANALYSE, VERBOSE) SELECT JSONB_BUILD_OBJECT('obj', obj) FROM (
     SELECT JSON_OBJECT_AGG(user_id, tracked_obj) AS obj FROM (
          SELECT user_id, JSON_OBJECT_AGG(team_id, JSON_BUILD_OBJECT('total_time', (random() *
1000)::int)) AS tracked_obj FROM (
               SELECT (random() * 10000)::int AS user_id, (random() * 10000)::int AS team_id
               FROM generate_series(1, 1000000)
          ) gen
          GROUP BY user_id
     ) agg
) foo;
-------------------------------------------------------------------------------------------------
-
Subquery Scan on foo  (cost=50005.01..50005.03 rows=1 width=32) (actual time=3295.757..3345.740
rows=1 loops=1)
  Output: jsonb_build_object('obj', foo.obj)
  ->  Aggregate  (cost=50005.01..50005.02 rows=1 width=32) (actual time=1846.187..1846.190 rows=1
loops=1)
        Output: json_object_agg((..., (json_object_agg(..., json_build_object('total_time',...)
        ->  HashAggregate  (... (actual time=1765.143..1776.300 rows=10001 loops=1)...
Planning Time: 0.104 ms
Execution Time:  3355.383 ms
```

toggl

Thank you.

toggl

José Neves

@rafalneves

DM Tech Lead at Toggl Track