

# Anatomy of Table-Level Locks in PostgreSQL

Gülçin Yıldırım Jelinek, Staff Engineer @ Xata

Jan 2025, PGDay CERN

## Select \* from me;

---

### Current:

- Staff Engineer at [Xata](#)
- [Postgres Contributor](#)
- Co-founder of [Prague PostgreSQL Meetup](#)
- Co-founder & General Coordinator of [Kadin Yazilimci](#) (Women Devs Turkey)
- Co-founder & Chair of [Diva: Dive into AI Conference](#)
- [Postgres Europe Diversity Committee](#)

### Past:

- Board Member at [Postgres Europe](#)

Select \* from me;



# Agenda

**01** MVCC

**04** Reducing Locking Impact

**02** DDL Locks

**05** pgroll

**03** Postgres Lock Queue

# Locks

- Concurrency primitive
- Ensures conflicting actions don't happen in parallel
- Used everywhere

## Postgres lock avoidance

- Uses MVCC for DML
- Writes make new copy of data
- Reads don't block writes, writes don't block reads
- But even reads still lock objects (tables, types, views)

# MVCC example

Transaction 1 (txid: 100)	Transaction 2 (txid: 101)	Row Versions
BEGIN;		(0,1)   1   Alice   50000   99   null
SELECT ctid, id, name, salary, xmin, xmax FROM employees WHERE id = 1; - Sees: salary = 50000		(0,1)   1   Alice   50000   99   null
	BEGIN;	(0,1)   1   Alice   50000   99   null
	UPDATE employees SET salary = 60000 WHERE id = 1;	(0,1)   1   Alice   50000   99   101 (0,2)   1   Alice   60000   101   null
SELECT ctid, id, name, salary, xmin, xmax FROM employees WHERE id = 1; - Sees: salary = 50000		(0,1)   1   Alice   50000   99   101 (0,2)   1   Alice   60000   101   null
	COMMIT;	(0,1)   1   Alice   50000   99   101 (0,2)   1   Alice   60000   101   null
SELECT ctid, id, name, salary, xmin, xmax FROM employees WHERE id = 1; - Sees: salary = 60000		(0,1)   1   Alice   50000   99   101 (0,2)   1   Alice   60000   101   null
COMMIT;		(0,1)   1   Alice   50000   99   101 (0,2)   1   Alice   60000   101   null

## Why we lock

- Reduces the throughput
- May increase latency - Loss of performance
- Correctness - Different Isolation Levels



# DDL

- Often needs stronger lock modes on objects
- Especially things like ALTER TABLE, VACUUM FULL
- May block other DDL, DML or even SELECTs accessing same object
- Every DDL command (and sometimes sub-command) is different

## Takeaway #1:

**MVCC will protect you from writes blocking reads, but not from object locks taken by DDL.**

Different variants of the same DDL command may need very different lock strength.

# Table-level lock modes

ACCESS SHARE - SELECTs

ROW SHARE - SELECT FOR UPDATE/SHARE

ROW EXCLUSIVE - DML (INSERT/UPDATE/DELETE/MERGE)

SHARE UPDATE EXCLUSIVE - VACUUM, ANALYZE, CREATE INDEX CONCURRENTLY

SHARE - CREATE INDEX

SHARE ROW EXCLUSIVE - CREATE TRIGGER

EXCLUSIVE - REFRESH MATERIALIZED VIEW CONCURRENTLY

ACCESS EXCLUSIVE - DROP TABLE, TRUNCATE, some forms of ALTER TABLE, VACUUM FULL

## Table-level lock modes

- Different modes conflict with different other modes
- ACCESS EXCLUSIVE conflicts with everything, including ACCESS SHARE (SELECT)
- Postgres has many optimizations to take weaker lock modes when it can
- But nothing is perfect, it will still take strong locks on some DDL
- Once a transaction takes a lock, it holds it even when the statement has finished

## Takeaway #2:

**DDL may block writes and/or reads for the whole run time of the transaction.**

Don't mix commands that need strong locks with other commands in the same transaction.

## Lock queue

- When requested lock mode conflicts with already acquired lock mode by different backend, it needs to wait
- By default, it waits forever and can stall everything unless you specify `lock_timeout`
- Waiting locks form a lock queue
- The queue is not visible in `pg_locks`, use `pg_blocking_pids()` to see what other backends blocks a specific backend
- Locks that are ahead in the queue can block locks that are behind them in the queue

## Takeaway #3:

**Use `lock_timeout` to limit how long something waits for lock.**

Using `lock_timeout` for DDL commands is often enough. You must be able to handle failures, for example retry the DDL again.

## Lock queue blocking example

---

Let's reiterate: Locks that are ahead in the queue can block locks that are behind them in the queue

1. Long running SELECT holds ACCESS SHARE LOCK
2. ALTER TABLE DETACH PARTITION needs brief ACCESS EXCLUSIVE LOCK
3. They conflict so ALTER TABLE is put into lock queue
4. Another ~30 backends try to do simple primary key lookup SELECTs
5. They conflict with the ALTER TABLEs lock, so they are put into the lock queue behind it
6. All access to the given table is now queued behind and no processing happens until the long running SELECT and ALTER TABLE both finish



## Takeaway #4:

**Any long-running query can cause blocking during schema changes.**

The cumulative waiting effect can be mitigated by `lock_timeout` (remember takeaway #3).

# Multiple ways to achieve the same result #1

---

Sometimes, there are less impactful ways to do something

- Use CONCURRENTLY commands
  - CREATE INDEX CONCURRENTLY
  - ALTER TABLE DETACH PARTITION CONCURRENTLY
- They use less locking, however
  - They take longer
  - Not transactional (can't be in transaction block, can't be rolled back)
  - Leave half-done work on failure (there are commands to clean up or finish the work)
- As a result: Treat them with care

## Multiple ways to achieve the same result #2

---

Some actions can be split

- ALTER TABLE mytable ADD COLUMN newcol NOT NULL DEFAULT clock\_timestamp()
  - ACCESS EXCLUSIVE lock
  - Blocks everything else
  - Table rewrite - holds lock for a long time if table is big
- Can be done in steps
  - ALTER TABLE mytable ADD COLUMN newcol TYPE timestampz DEFAULT clock\_timestamp()
  - UPDATE TABLE mytable SET newcol = clock\_timestamp() WHERE newcol IS NULL
    - You actually want to do the update in batches, remember any long running query can cause problems
  - ALTER TABLE mytable ALTER COLUMN newcol SET NOT NULL

## Multiple ways to achieve the same result #2 continued

---

Some actions can be split

- ALTER TABLE mytable ALTER COLUMN newcol SET NOT NULL
  - Still takes long time and blocks writes
    - But does not block reads unlike the original command
- Can be further split into
  - ALTER TABLE mytable ADD CONSTRAINT CONSTRAINT mytable\_newcol\_not\_null CHECK (newcol IS NOT NULL) NOT VALID
  - ALTER TABLE mytable VALIDATE CONSTRAINT mytable\_newcol\_not\_null
  - This way the scan during VALIDATE CONSTRAINT does not block writes

## Takeaway #5:

**Try to find an approach that does less locking.**

Postgres manual contains all the CONCURRENTLY commands.

Splitting actions takes expertise and some things are impossible (or very hard) to do without heavy locking from plain SQL.

## Postgres improves over time

---

Let's look at slightly modified example from before

- `ALTER TABLE mytable ADD COLUMN newcol int NOT NULL DEFAULT 1`
- Still takes `ACCESS EXCLUSIVE` lock
  - Blocks everything else
- Does not rewrite table because 1 is constant and can be stored as metadata
  - Locks for very short time
- This used to rewrite in the old versions of Postgres just like the previous example

## Takeaway #6:

**Make sure you are running the newest version of Postgres.**

Improvements in locking and even how long the command takes (and holds the lock) happens in newer versions.

New CONCURRENTLY command variants are added in newer versions.



## Enter pgroll

Zero-downtime, reversible schema changes for Postgres



# Motivation

---

(Some) Postgres schema changes are difficult

- Locking issues (most ALTER statements take the ACCESS EXCLUSIVE lock)
- Data backfill (e.g. add a column with unique constraints)
- Require multiple steps (e.g renaming a column)
- Backwards incompatible with old or new versions of the application (e.g. dropping a column)



How does pgroll work?

## How does pgoroll work?

---

- Higher level operations
- Automatic Expand/Contract pattern
- Multi-version schema views



# Higher level operations

- Instead of ALTER statements, pgoroll uses higher level operations:
  - Add column
  - Rename column
  - Change type of column
  - Add index
  - Add constraint
  - Etc.
- Backfilling of data is represented in the JSON

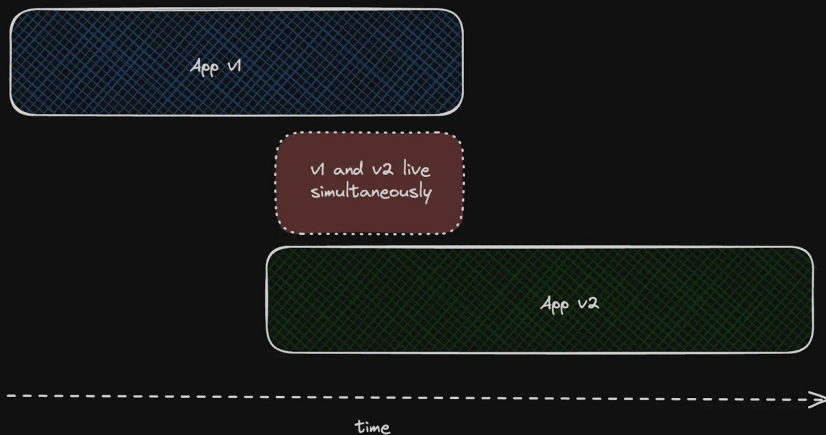
```
{
  "name": "18_change_column_type",
  "operations": [
    {
      "alter_column": {
        "table": "reviews",
        "column": "rating",
        "type": "integer",
        "up": "CAST(rating AS integer)",
        "down": "CAST(rating AS text)"
      }
    }
  ]
}
```

# Automated Expand and Contract pattern

---

- Temporary columns are added to the physical table
- Data is backfilled and transformed in background
- Views hide or show the different columns
- Temporary columns are deleted when no longer needed

# Multiple-schema versions via views

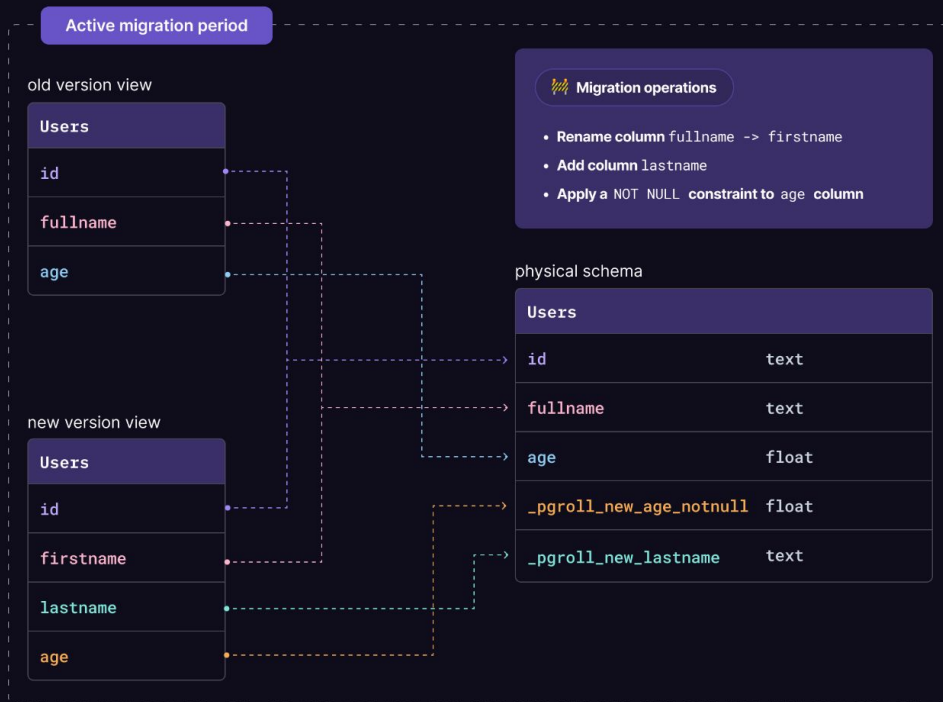


Workflow is always the same:

- Start migration
- Do a (rolling) upgrade of your application
- Finalize the migration

<https://xata.io/blog/multi-version-schema-migrations>

# Different version of the schema are exposed via views



- Temporary columns are added to the physical table
- Data is backfilled and transformed in background
- Views hide or show the different columns

## How - Application selects its version by setting the `search\_path`

```
-- Switch back the new schema, which disallows `NULL`s in the `name` field
SET search_path TO mig_cq778qtl0oe0bpredl0;

-- Attempt to insert a `NULL` value in the name field
INSERT INTO users(name) VALUES (NULL)
-- ERROR null value in column "name" of relation "users" violates not-null constraint

-- Switch back the old schema, which allows `NULL`s in the `name` field
SET search_path TO mig_cq778jdl0oe0bpredk0;

-- Attempt to insert a `NULL` value in the name field
INSERT INTO users(name) VALUES (NULL)

-- Retrieve the data from the `users` table
SELECT * FROM users ORDER BY name DESC;
```



## How - automatic backfilling

- The “up” SQL expression is used to convert or generate the required data
- You can control the batch size and rate

```
{
  "name": "18_change_column_type",
  "operations": [
    {
      "alter_column": {
        "table": "reviews",
        "column": "rating",
        "type": "integer",
        "up": "CAST(rating AS integer)",
        "down": "CAST(rating AS text)"
      }
    }
  ]
}
```

## How - triggers update and downgrade data in both directions

What about new writes to the table?

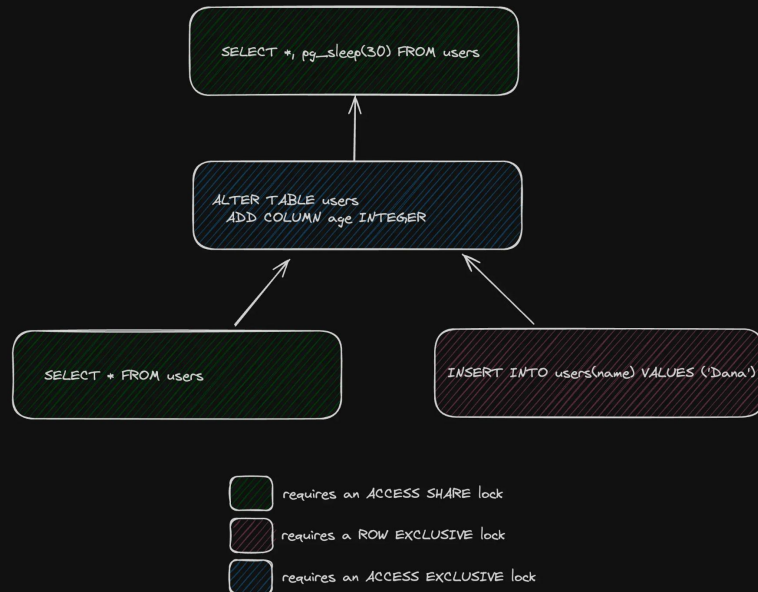
- Triggers are installed to convert the data "up" and "down"

Note: dual write at the column level is necessary here, but you'd have to do it anyway.

```
{
  "name": "18_change_column_type",
  "operations": [
    {
      "alter_column": {
        "table": "reviews",
        "column": "rating",
        "type": "integer",
        "up": "CAST(rating AS integer)",
        "down": "CAST(rating AS text)"
      }
    }
  ]
}
```

# The "trylock" trick is built-in

- Generated ALTER is prefixed with a SET lock\_timeout command
- Avoids issues with the lock queue



# Benefits

---

- Rollback is easy - just drop the views and intermediary columns.
- The tool takes care of locking issues and common issues.
- The merging workflow is always the same:
  - Start the pgroll migration
  - Roll-out the application upgrade (can be blue-green)
  - Complete the pgroll migration



## Final Takeaway:

**Smart tools like pgroll can help you avoid many common pitfalls.**

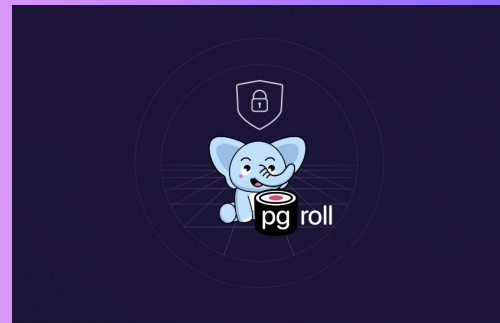
Zero-downtime, reversible schema changes are possible.

# For more

---



<https://github.com/xataio/pgroll>

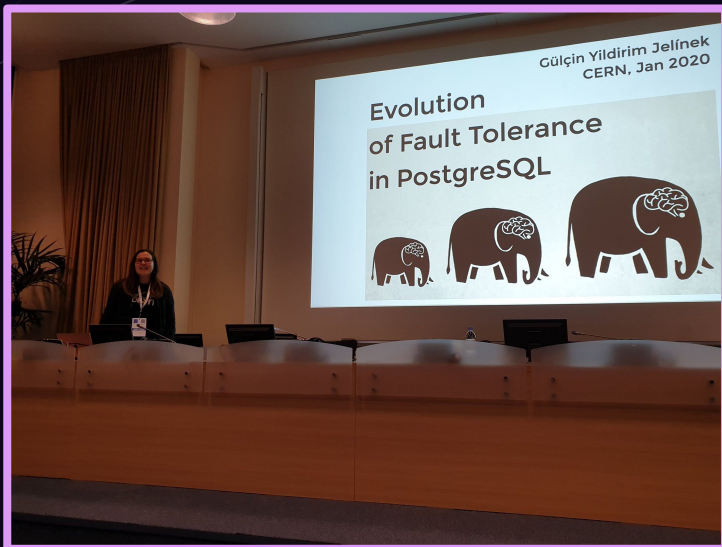


## Anatomy of Table-Level Locks in PostgreSQL

By Gulcin Yildirim Jelinek | Jan 13, 2025

This blog explains locking mechanisms in PostgreSQL, focusing on table-level locks that are required by Data Definition Language (DDL) operations.

<https://pgroll.com/blog>



# Thank you! Until next time!