PostgreSQL Migrations Without Drama

data egret

Your remote PostgreSQL DBA team

Daria Nikolaenko

contact@dataegret.com

SECURING YOUR POSTGRESQL DATABASE AVAILABILITY AND HIGH PERFORMANCE

- **Migration**
- Performance **audit**
- **Cloud Cost** management
- **Backup & restore**
- **Architecture** review
- **DataOps/CDC** projects
- **Consulting** for data science and analytics teams
- **PostgreSQL Courses** for DBA and Developers

Contributing sponsor to PostgreSQL

Open Alliance
For PostgreSQL Education

OSB Open Source Business ALLIANCE
Bundesverband Für digitale Souveränität e.V.
MITGLIED

EXPERTISE

Senior DBA with 10+ years of experience in PostgreSQL administration.

DEVELOPMENT

PostgreSQL Contributors involved in new PostgreSQL feature and extention development.

CUSTOMISATION

Flexible approach and dedicated team focused on success of your project.

COMMUNITY

Recognised as Significant Contributing Sponsor to PostgreSQL.

Daria Nikolaenko

- PostgreSQL DBA **@Data Egret**
  - PostgreSQL consulting, support, and training

What Are DDL/DML Migrations

- **DDL** — changes structure (tables, indexes, constraints, …)
- **DML** — changes data (INSERT, UPDATE, DELETE)

Performance Impact

- **Generally safe operations:**
  - Adding a column with constant `DEFAULT`
  - Creating new indexes *CONCURRENTLY*
- **Risky or unsafe operations:**
  - `ALTER TABLE … TYPE`
  - `CREATE INDEX` (*without* CONCURRENTLY)
  - `UPDATE` millions of rows

What Can Go Wrong?

- **Long locks:** queries blocked
- **Heavy WAL generation:** disk / replication issues / high I/O
- **Large temporary files:** high I/O
- **Long transactions:**
  - VACUUM problems,
  - increased risk of transaction ID exhaustion.

Why Locks Exist

PostgreSQL uses locks to guarantee **ACID** properties (Atomicity, Consistency, Isolation, Durability):

- prevent data corruption,
- ensure consistency between transactions,
- coordinate concurrent changes.

How Blocking Happens

- Every query takes locks
- Some locks conflict
- Conflicts = waiting
- Waiting = blocking

Lock Examples

- **DDL migrations:**
  - `ALTER TABLE`: often ACCESS EXCLUSIVE
  - `CREATE INDEX`: SHARE
  - `CREATE INDEX CONCURRENTLY`: SHARE UPDATE EXCLUSIVE
- **DML migrations:**
  - `INSERT / UPDATE / DELETE / MERGE`: ROW EXCLUSIVE
  - modified rows are locked individually

Conflicting Locks

Req\Exist	ACC SH.	ROW SH.	ROW EXCL.	SH.UPD.EXCL.	SH.	SH.ROW.EXCL.	EXCL.	ACC EXCL.
ACC SH.								X
ROW SH.							X	X
ROW EXCL.					X	X	X	X
SH.UPD.EX.				X	X	X	X	X
SH.			X	X		X	X	X
SH.ROW.EX.			X	X	X	X	X	X
EXCL.		X	X	X	X	X	X	X
ACC EXCL.	X	X	X	X	X	X	X	X

Identify Locks Your Query Takes

- Check PostgreSQL documentation - Explicit Locking
- Run the statement on a test system and observe locks in `pg_locks`

Identify Locks Your Query Takes. Example

```
SELECT pg_backend_pid();

BEGIN;

ALTER TABLE table1
ALTER COLUMN value TYPE varchar(200);

-- other session
SELECT
    l.locktype,
    c.relname,
    l.mode,
    l.granted
FROM pg_locks l
LEFT JOIN pg_class c ON c.oid = l.relation
WHERE l.pid = your_pid;

   locktype    | relation  |        mode          | granted
---------------+-----------+----------------------+---------
 relation      | table1    | AccessExclusiveLock  | t
 ...
```

What AccessExclusiveLock Conflicts With

Req\Exist	ACC SH.	ROW SH.	ROW EXCL.	SH.UPD.EXCL.	SH.	SH.ROW.EXCL.	EXCL.	ACC EXCL.
ACC EXCL.	X	X	X	X	X	X	X	X

Who Is Blocking Whom?

```
ALTER TABLE table1 ALTER COLUMN value TYPE varchar(200); -- pid 1145445
SELECT * FROM table1; -- pid 1145588

SELECT
    waiting.pid AS blocked,
    waiting.usename AS b_user,
    blocker.pid AS blocker,
    blocker.usename AS bl_user,
    waiting.query AS blocked_q,
    blocker.query AS blocker_q
FROM pg_stat_activity AS waiting
CROSS JOIN LATERAL unnest(pg_blocking_pids(waiting.pid)) AS blocking_pid(pid)
JOIN pg_stat_activity AS blocker
    ON blocker.pid = blocking_pid.pid
WHERE waiting.wait_event_type = 'Lock';
```

```
 blocked | b_user   | blocker | bl_user  |        blocked_q          |        blocker_q
---------+----------+---------+----------+---------------------------+-------------------------------
1145588  | postgres |1145445  | postgres | SELECT * FROM table1;     | ALTER TABLE table1          +
         |          |         |          |                           | ALTER COLUMN value TYPE ...;
```

Before Migration

Before running DDL/DML changes in production, always double-check.

Check Workload

- Are there concurrent transactions that may conflict?
  - Look for **long-running transactions** on the same table *(on replica too)*
- Is any **maintenance running**? *(e.g., VACUUM, REINDEX)*
  - Some operations may conflict with certain DDL
  - Others may mainly add resource pressure
- Do you have enough **free disk space, I/O capacity, and CPU headroom**?
- What is the **size** of the affected tables?

Replication Awareness

- Is the table part of:
  - logical replication / CDC
  - physical streaming replication
- DDL operations may **break** logical replication *(e.g., dropping or renaming columns)*
- Heavy DML generates WAL: can cause **replication lag**

Additional Checklist

- Are there scheduled **backups** or other jobs that might overlap?
- Do you have a **rollback plan**?
- Did you **test this migration** on realistic staging data?

Timeout as a Precaution

- `lock_timeout` — limits lock waiting
- `statement_timeout` — limits total execution time

Timeouts to Prevent Lock Queues

```
-- session 1
SELECT * FROM table1; -- runs for 5 min

-- session 2 (migration)
ALTER TABLE table1 ADD COLUMN col1 int; -- waits for lock from session 1

-- session 3, 4, 5, ..
SELECT * FROM table1; -- waits for lock to be acquired and then released by session 1
```

Setting Timeouts

```
BEGIN;

SET LOCAL lock_timeout = '3s';
SET LOCAL statement_timeout = '60s';
...
COMMIT;

-- or

SET statement_timeout = '60s';
...
RESET statement_timeout;
```

Safe Migration Patterns

- DDL
- DML

Adding Constraints

```
-- ACCESS EXCLUSIVE lock
ALTER TABLE subscriptions
ADD CONSTRAINT subscriptions_valid_period
CHECK (ends_at > starts_at);
```

- Validation runs inside the same command
- Blocks everything

Adding CHECK constraint safely

```
-- brief ACCESS EXCLUSIVE lock
ALTER TABLE subscriptions
ADD CONSTRAINT subscriptions_valid_period
CHECK (ends_at > starts_at) NOT VALID;

-- SHARE UPDATE EXCLUSIVE lock
ALTER TABLE subscriptions
VALIDATE CONSTRAINT subscriptions_valid_period;
```

- Validation is separated from the DDL
- Avoids long `ACCESS EXCLUSIVE`

Foreign keys: same pattern

```
-- basic approach: SHARE ROW EXCLUSIVE lock
ALTER TABLE orders
    ADD CONSTRAINT orders_user_id_fk
    FOREIGN KEY (user_id) REFERENCES users(id);

-- safer approach: ROW SHARE, ACCESS SHARE, SHARE UPDATE EXCLUSIVE locks
-- new writes are still checked immediately
ALTER TABLE orders
    ADD CONSTRAINT orders_user_id_fk
    FOREIGN KEY (user_id) REFERENCES users(id) NOT VALID;

ALTER TABLE orders
    VALIDATE CONSTRAINT orders_user_id_fk;
```

NOT NULL before PostgreSQL 18

```
ALTER TABLE users
    ADD CONSTRAINT users_email_not_null_check
    CHECK (email IS NOT NULL) NOT VALID;

ALTER TABLE users
    VALIDATE CONSTRAINT users_email_not_null_check;

ALTER TABLE users
    ALTER COLUMN email SET NOT NULL;

ALTER TABLE users
    DROP CONSTRAINT users_email_not_null_check;
```

- No direct `NOT VALID` support for `NOT NULL`
- Use CHECK as a workaround

NOT NULL in PostgreSQL 18

```
ALTER TABLE users
    ADD CONSTRAINT users_email_not_null
    NOT NULL email NOT VALID;

ALTER TABLE users
    VALIDATE CONSTRAINT users_email_not_null;
```

- Native support for `NOT VALID`
- No workaround needed
- Fewer steps, fewer mistakes

ADD UNIQUE

```
-- basic approach
ALTER TABLE users
    ADD CONSTRAINT users_email_key UNIQUE (email);

-- safe approach
CREATE UNIQUE INDEX CONCURRENTLY users_email_idx
    ON users (email);
ALTER TABLE users
    ADD CONSTRAINT users_email_key
    UNIQUE USING INDEX users_email_idx;
```

- Index build can be tuned (`maintenance_work_mem` , parallel workers)
- Same approach for primary keys

ADD COLUMN with DEFAULT

```
\timing

ALTER TABLE table1
    ADD COLUMN created_at timestamptz DEFAULT now();
    ALTER TABLE
Time: 17.554 ms

ALTER TABLE test_big
    ADD COLUMN processed_at timestamptz DEFAULT clock_timestamp();
    ALTER TABLE
Time: 125392.365 ms (02:05.392)
```

- Non-volatile default: metadata only
- Volatile default: full table rewrite
- Use `SET client_min_messages = debug1;` to verify behavior

ADD COLUMN with volatile DEFAULT

```
ALTER TABLE table1
    ADD COLUMN processed_at timestamptz;

UPDATE table1
    SET processed_at = clock_timestamp()
    WHERE processed_at IS NULL;

ALTER TABLE table1
    ALTER COLUMN processed_at SET DEFAULT clock_timestamp();
```

ALTER COLUMN TYPE

```
ALTER TABLE events
ALTER COLUMN title TYPE varchar(255);
```

- **No rewrite**: binary-compatible types (e.g. text ↔ varchar)
- **Rewrite required**: representation changes (e.g. integer → bigint)
- Indexes may still be rebuilt
- Query plans may change, especially with casts

ALTER COLUMN TYPE: safer approach

```
ALTER TABLE orders ADD COLUMN order_id_new bigint;

-- keep new writes in sync (trigger or dual write)
...

-- backfill old rows in batches
UPDATE orders
SET order_id_new = order_id
WHERE order_id_new IS NULL;

-- create indexes and constraints on the new column
...
-- run ANALYZE
...

-- switch
BEGIN;
ALTER TABLE orders RENAME COLUMN order_id TO order_id_old;
ALTER TABLE orders RENAME COLUMN order_id_new TO order_id;
COMMIT;
```

Bulk data changes: indexes and constraints

- Indexes are maintained for each row
  - (`INSERT`: *46s* vs *3m* with one extra index)
- Foreign keys are checked via triggers
  - (`INSERT`: *50s* without FK vs *4ms* with FK)
- `UPDATE` cost depends on index involvement
  - (HOT *3m 50s* vs non-HOT *12m*)

Bulk DELETE

- Best case if partitioned: `DETACH CONCURRENTLY`, `DROP PARTITION`
- If not possible: use batched `DELETE`
  - separate transactions,
  - `pg_repack` after
- Foreign keys need indexes on referencing columns
- Index usefulness depends on selectivity

Batched DELETE

- logical batching (id / created_at): slowest
- ctid batching: faster
  - avoids an additional lookup step during `DELETE`
- ctid range: fastest
  - means working with ranges of table pages
  - for append-only tables

Batched DELETE with ctid

```
WITH delete_batch AS (
  SELECT e.ctid
  FROM events e
  WHERE e.created_at < '2026-02-28 00:00:00+00'
  ORDER BY e.created_at
  FOR UPDATE
  LIMIT 10000
)
DELETE FROM events d
USING delete_batch b
WHERE d.ctid = b.ctid;
```

Batched UPDATE with ctid

```
WITH update_batch AS (
  SELECT w.ctid
  FROM work_item w
  WHERE w.status = 'active'
    AND w.num_retries > 10
  ORDER BY w.retry_timestamp
  FOR UPDATE
  LIMIT 5000
)
UPDATE work_item
SET status = 'failed'
FROM update_batch b
WHERE work_item.ctid = b.ctid;
```

Wrong batching

```
DO $$
DECLARE
  rows_updated integer;
BEGIN
  LOOP
    WITH update_batch AS (
      SELECT ctid
      FROM big_table
      WHERE created_at < DATE '2023-01-01'
      LIMIT 10000
    )
    UPDATE big_table b
    SET status = 'archived'
    FROM update_batch ub
    WHERE b.ctid = ub.ctid;

    GET DIAGNOSTICS rows_updated = ROW_COUNT;
    EXIT WHEN rows_updated = 0;

    PERFORM pg_sleep(0.1);
  END LOOP;
END $$;
```

How to batch correctly

- Each batch must commit separately
- Use a stable way to select rows:
  - `ORDER BY id LIMIT` …
  - or ctid-based selection
- Avoid `OFFSET`-based batching
- Run batches from outside PostgreSQL *(e.g. script or backend job)*
  - track progress
  - add pauses
- Or use a PostgreSQL procedure

During migration

- Monitor live:
  - blocking / waiting queries
  - replication lag
  - system load
- Be ready to stop or adjust
- Don’t leave it unattended

After migration

- Update statistics if needed (`ANALYZE`)
- Check for bloat after large changes and cleanup (`VACUUM` , `pg_repack`)
- Check replication lag and CDC
- Check disk space
- Review logs for errors and warnings

Wrap-up

- Be aware of WAL volume, locks, and data size
- Prefer safer alternatives to blocking DDL
- Test and plan before running
- Control execution:
  - use timeouts
  - batch large changes
- Watch during execution and verify after

Danke!