Zero-Downtime PostgreSQL Migrations

Version PostgreSQL 18 · Alembic 1.20 · verified 2026-11-14

Change a live PostgreSQL schema without an outage — lock_timeout, expand-migrate-contract, batched backfills, NOT VALID constraints and Alembic.

Contents

Mental model 2

Locks and lock_timeout 2

Expand, migrate, contract 3

Adding columns 4

Backfilling in batches 5

NOT NULL and foreign keys 7

Indexes and unique constraints 8

Changes that rewrite or break 9

Alembic and other tools 10

Further reading 12

Mental model

A migration fails in production for one of two reasons. Either it takes a lock that queues every other query behind it, or it changes the schema in a way the application code still running cannot handle. Schema and code never deploy at the same instant, so every migration must work with both the old and the new version of the app, and every statement must either hold its lock briefly or give up quickly.

Locks and lock_timeout

Statement	Lock	Blocks
Most `ALTER TABLE` forms	`ACCESS EXCLUSIVE`	Everything, reads too
`ADD FOREIGN KEY`	`SHARE ROW EXCLUSIVE`	Writes, on both tables
`VALIDATE CONSTRAINT`	`SHARE UPDATE EXCLUSIVE`	Other DDL, not DML
`CREATE INDEX`	`SHARE`	Writes
`CREATE INDEX CONCURRENTLY`	`SHARE UPDATE EXCLUSIVE`	Other DDL, not DML

Even an instant `ALTER TABLE` must first wait for every transaction already touching the table. While it waits, every new query on that table queues behind it — a one-millisecond change behind a five-minute report is a five-minute outage. `lock_timeout` makes the statement abort instead of waiting, so you retry later rather than block traffic.

```
BEGIN;
SET LOCAL lock_timeout = '2s';
SET LOCAL statement_timeout = '30s';
ALTER TABLE orders
  ADD COLUMN region text;
COMMIT;
-- on timeout: "canceling statement due
-- to lock timeout" — retry with backoff
```

> **Gotcha:** Plain `SET` lasts for the session. Through a connection pooler, the timeout stays on that connection and later fails some unrelated application query. Use `SET LOCAL` inside the migration's transaction, or set it per role for the migration user.

Expand, migrate, contract

Release	Schema	Code
1. Expand	Add new column or table	Writes old and new
2. Migrate	Backfill existing rows	Still writes both
3. Switch	Nothing	Reads new, stops writing old
4. Contract	Drop old column	Uses new only

Never rename or change a column in place while code depends on it. Add the new shape beside the old one, move the data, move the code, and remove the old shape in a later release — once no running instance,

background worker or rollback target still reads it. Each step is deployable and reversible on its own.

```
-- Rename email -> contact_email, step 1:
ALTER TABLE customers
  ADD COLUMN contact_email text;
-- step 2: backfill in batches (see below)
-- step 4, releases later:
ALTER TABLE customers DROP COLUMN email;
```

> **Warning:** A rollback of the app redeploys code that expects the old schema. If you contract in the same release as the switch, a rollback hits a dropped column. Keep at least one release between them.

Adding columns

Column added	Cost
Nullable, no default	Catalog only, instant
Non-volatile default, e.g. `'eu'`, `now()`	Catalog only (PG 11+)
Volatile default, e.g. `gen_random_uuid()`	Rewrites table and indexes
Identity or stored generated column	Rewrites table and indexes
Virtual generated column (PG 18)	Catalog only

With a non-volatile default, PostgreSQL evaluates it once, stores the result in the catalog and returns it for existing rows, so no row is

touched. The lock is still `ACCESS EXCLUSIVE`, which is why `lock_timeout` still matters even for an instant change.

```
ALTER TABLE orders
  ADD COLUMN source text NOT NULL
  DEFAULT 'web';           -- instant

ALTER TABLE orders
  ADD COLUMN created_on timestamptz
  DEFAULT now();           -- instant
```

> **Gotcha:** `now()` is not volatile — it returns the transaction's start time — so every existing row gets the **same** timestamp, the moment of the migration. If old rows need real values, backfill them; if each row needs its own value, add the column nullable and backfill.

Backfilling in batches

Approach	Effect
One `UPDATE` of every row	Long row locks, huge WAL, table bloat
Batches of 1k–10k rows, one transaction each	Short locks, vacuum keeps up
Pause between batches	Replicas and autovacuum catch up

An `UPDATE` writes a new version of every row it touches, so updating 50 million rows in one statement holds their row locks until commit, doubles the table on disk, and produces one burst of WAL that replicas must replay. Batches keyed on the primary key, each committed on its own, keep every lock short and let vacuum reclaim space as you go. Make the batch idempotent so a crash means “run it again”.

```
UPDATE customers
SET contact_email = email
WHERE id IN (
  SELECT id FROM customers
  WHERE contact_email IS NULL
    AND email IS NOT NULL
  ORDER BY id
  LIMIT 5000
);
-- repeat until it reports UPDATE 0
```

> **Tip:** Run the loop from a script or job, not from inside one migration transaction — a single transaction around every batch undoes the whole point. Monitor replica lag and stop if it grows.

NOT NULL and foreign keys

Step	Lock held	Scans table
`ADD CONSTRAINT ... CHECK (...) NOT VALID`	Brief `ACCESS EXCLUSIVE`	No
`VALIDATE CONSTRAINT`	`SHARE UPDATE EXCLUSIVE`	Yes, writes continue
`ALTER COLUMN ... SET NOT NULL`	Brief `ACCESS EXCLUSIVE`	No, uses the valid check
`DROP CONSTRAINT` the check	Brief `ACCESS EXCLUSIVE`	No

`SET NOT NULL` on its own scans the whole table under an exclusive lock. A `NOT VALID` constraint is enforced for new writes immediately but skips checking existing rows; `VALIDATE` checks them later under a lock that lets reads and writes continue. `SET NOT NULL` then skips its scan because a valid `CHECK` already proves no nulls exist. Foreign keys use the same two steps.

```
ALTER TABLE customers
  ADD CONSTRAINT contact_email_nn
  CHECK (contact_email IS NOT NULL)
  NOT VALID;
ALTER TABLE customers
  VALIDATE CONSTRAINT contact_email_nn;
```

```
ALTER TABLE customers
  ALTER COLUMN contact_email SET NOT NULL;
ALTER TABLE customers
  DROP CONSTRAINT contact_email_nn;
```

> **Note:** PostgreSQL 18 can add the not-null constraint itself as `NOT VALID` — `ADD CONSTRAINT c NOT NULL col NOT VALID`, then `VALIDATE CONSTRAINT c` — which removes the `CHECK` detour. On 12 to 17, use the four steps above.

Indexes and unique constraints

Goal	Safe path
New index	`CREATE INDEX CONCURRENTLY`
New unique constraint	Unique index concurrently, then `ADD CONSTRAINT ... USING INDEX`
Failed concurrent build	`DROP INDEX CONCURRENTLY`, then retry

`CREATE INDEX CONCURRENTLY` cannot run inside a transaction block, and most migration tools wrap each migration in one, so it needs its own migration with the transaction turned off. Keep it to one index per migration: if the build fails it leaves an `INVALID` index behind, and a migration that half-succeeded is easier to clean up when it did only one thing.

```
CREATE UNIQUE INDEX CONCURRENTLY
  customers_contact_email_key
  ON customers (contact_email);

ALTER TABLE customers
  ADD CONSTRAINT customers_contact_email_key
  UNIQUE USING INDEX
    customers_contact_email_key;
```

> **Gotcha:** `IF NOT EXISTS` on a retried concurrent build matches the invalid index by name and does nothing, so the migration reports success with an unusable index. Check `pg_index.indisvalid` after every concurrent build.

Changes that rewrite or break

Change	Risk	Safer path
`ALTER COLUMN ... TYPE`	Rewrite under exclusive lock	New column, backfill, switch
`RENAME COLUMN` / table	Instant, breaks running code	Expand and contract
`DROP COLUMN`	Instant, breaks code still reading it	Stop reading first

Change	Risk	Safer path
`ADD PRIMARY KEY`	Builds index under lock	Unique index concurrently, then `USING INDEX`
`VACUUM FULL`, `CLUSTER`	Rewrite, blocks reads	`pg_repack`, or accept bloat

A type change rewrites the table unless the new type is binary-compatible — `varchar(50)` to `varchar(100)` or to `text` is free, `int` to `bigint` is a full rewrite. `DROP COLUMN` itself is fast because it only hides the column; the space comes back when rows are rewritten.

```
-- Free: binary-compatible widening
ALTER TABLE customers
  ALTER COLUMN name TYPE text;
```

> **Gotcha:** ORMs that cache a table's column list — or `SELECT *` mapped onto a fixed struct — break when a column disappears, even if no code mentions it. Remove the column from the model and deploy before the migration that drops it.

Alembic and other tools

Need	Alembic
One transaction per migration	`env.py` flag — see the Warning
Step outside the transaction	`autocommit_block()`

Need	Alembic
Concurrent index	`postgresql_concurrently`
Partial or covering index	`postgresql_where`, `postgresql_include`

In a Python codebase, Alembic with SQLAlchemy is the usual choice; plain `.sql` files applied in order and tracked in version control work too. Whatever the tool, review the SQL it emits — autogenerate does not know which operations lock — and use the patterns above.

```
from alembic import op


def upgrade():
    ctx = op.get_context()
    with ctx.autocommit_block():
        op.create_index(
            "orders_region_idx",
            "orders",
            ["region"],
            postgresql_concurrently=True,
        )
```

> **Warning:** `autocommit_block()` first commits the transaction that was open, including earlier migrations in the same run. Combine it with `transaction_per_migration=True` so that commit boundary matches a migration boundary.

Further reading

- PostgreSQL — ALTER TABLE, notes
- PostgreSQL — Table-level lock modes
- PostgreSQL — lock_timeout
- PostgreSQL — Building indexes concurrently
- Alembic — Operation reference
- Alembic — Runtime objects