

Postgres schema migrations using the expand/contract pattern

PGDay UK, September 9th 2025

Speaker



Andrew Farries
Staff Software Engineer @ Xata
andrew.farries@xata.io

Outline

01 Schema change is hard

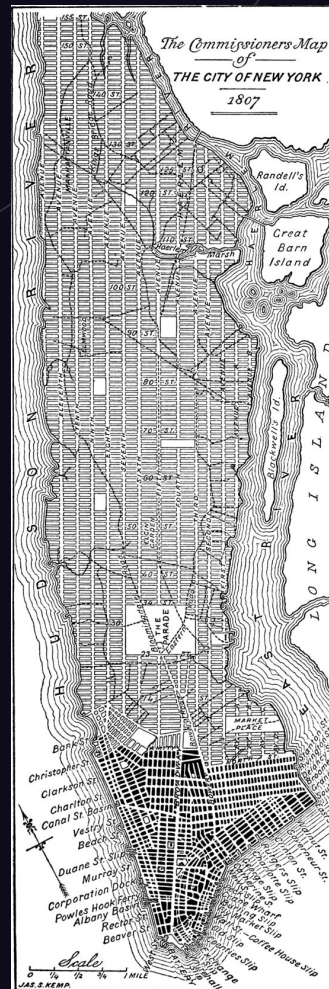
02 Common pitfalls

03 Expand contract migrations

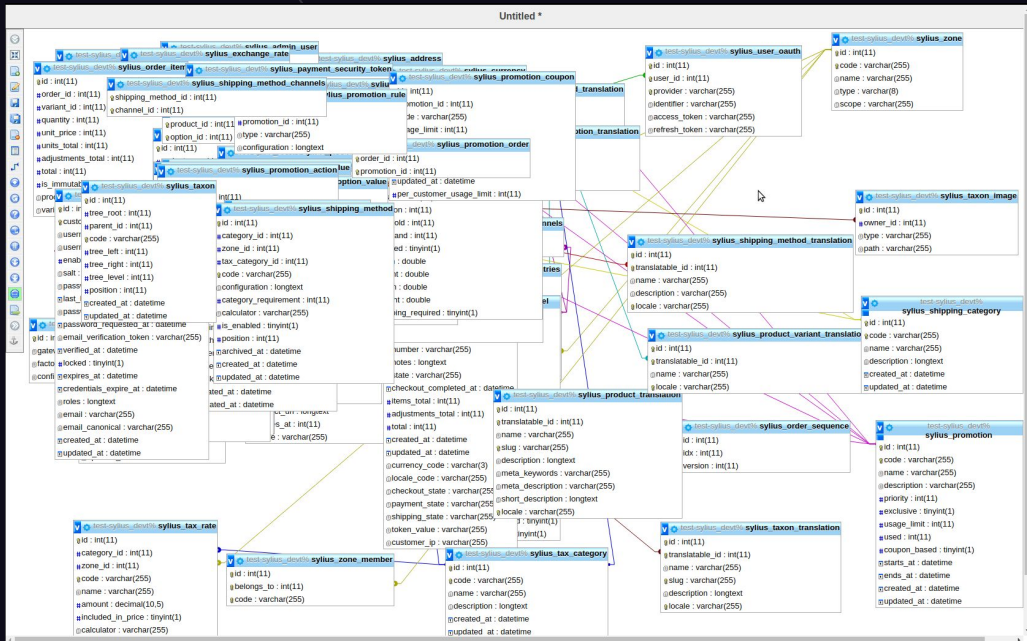
04 Expand/contract with pgroll



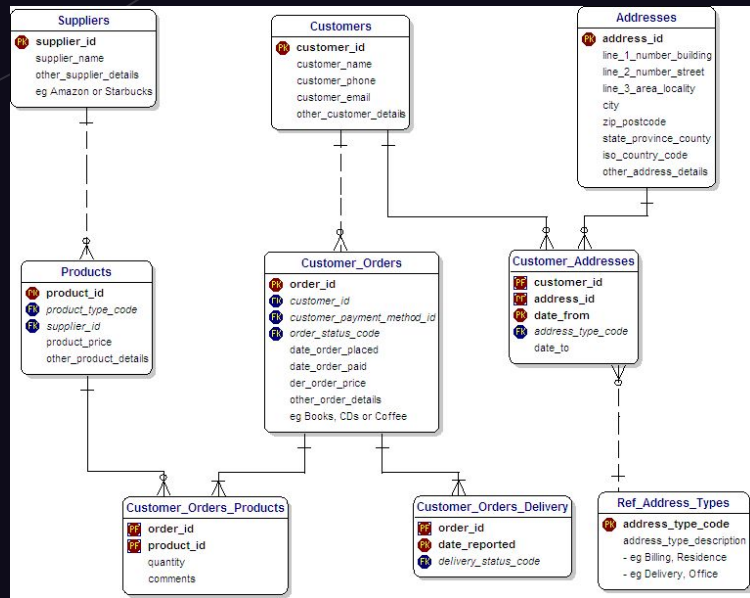
(image from londonist.com)



London



Manhattan



"Database schemas are notoriously volatile, extremely concrete, and highly depended on. This is one reason why the interface between OO applications and databases is so difficult to manage, and why schema updates are generally painful."

Robert C. Martin, Clean Architecture

Common pitfalls

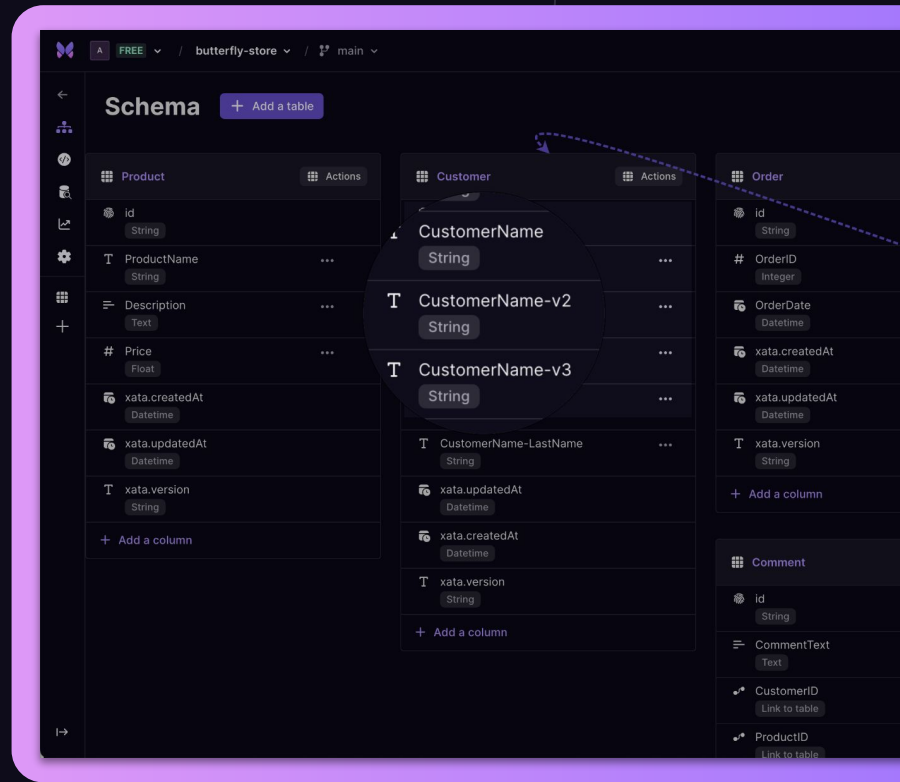
Additive-only changes and schema debt

What is it?

The act of never modifying or removing columns, only adding new ones

Why is this bad?

- Bugs and performance implications
- General confusion
- Long-lived compatibility code



The locking minefield

What is it?

PostgreSQL offers good ways to control locking your database, but you need to know what you're doing

Why is this bad?

- Tables are inaccessible
- Query queuing
- Testing is hard

Non-conflicting lock modes can be held concurrently by many transactions. Notice in the diagram that some lock modes are self-conflicting (for example, an ACCESS EXCLUSIVE lock can be held by multiple transactions).



Table-Level Lock Modes



ACCESS SHARE (AccessShareLock)

Conflicts with the ACCESS EXCLUSIVE lock mode only.

The SELECT command acquires a lock of this mode on referenced tables. In general, any query that only *reads* a table and does not modify it can be held by multiple transactions.



ROW SHARE (RowShareLock)

Conflicts with the EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

The SELECT command acquires a lock of this mode on all tables on which one of the FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, or FOR KEY SHARE locks on any other tables that are referenced without any explicit FOR ... locking option).



ROW EXCLUSIVE (RowExclusiveLock)

Conflicts with the SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

The commands UPDATE, DELETE, INSERT, and MERGE acquire this lock mode on the target table (in addition to ACCESS SHARE locks on other tables) if they *modify data* in a table.



SHARE UPDATE EXCLUSIVE (ShareUpdateExclusiveLock)

Conflicts with the SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This lock mode is acquired by VACUUM (without FULL), ANALYZE, CREATE INDEX CONCURRENTLY, CREATE STATISTICS, COMMENT ON, REINDEX CONCURRENTLY, and some forms of ALTER TABLE.

Acquired by VACUUM (without FULL), ANALYZE, CREATE INDEX CONCURRENTLY, CREATE STATISTICS, COMMENT ON, REINDEX CONCURRENTLY, and some forms of ALTER TABLE. (for full details see the documentation of these commands).



SHARE (ShareLock)

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This lock mode is acquired by CREATE INDEX (without CONCURRENTLY).

Acquired by CREATE INDEX (without CONCURRENTLY).



SHARE ROW EXCLUSIVE (ShareRowExclusiveLock)

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This lock mode is acquired by CREATE TRIGGER and some forms of ALTER TABLE.

Acquired by CREATE TRIGGER and some forms of ALTER TABLE.

Testing schema migrations

What is it?

Confidence that a migration will succeed in production is hard to obtain with limited data available in lower environments.

Why is this bad?

- Failures only detected in production
- Problems provisioning realistic data-sets
- Lack of confidence in migrations



Rolling back your changes

What is it?

We're all human; mistakes occur. Having to roll back the changes you made in production can be painful

Why is this bad?

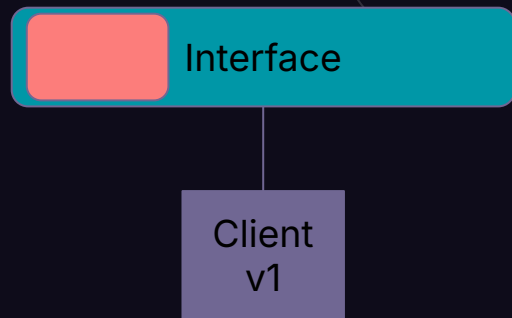
- Unplanned maintenance
- Often untested
- Time consuming



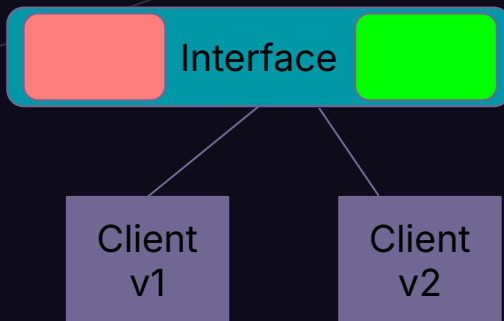
The expand / contract pattern

Expand & contract

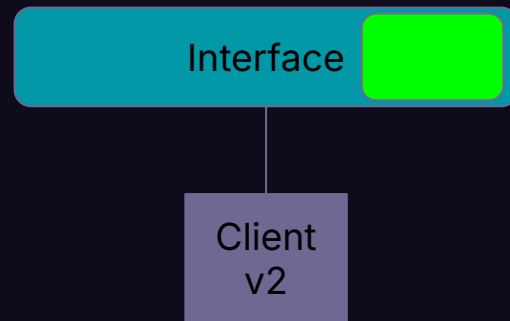
Start



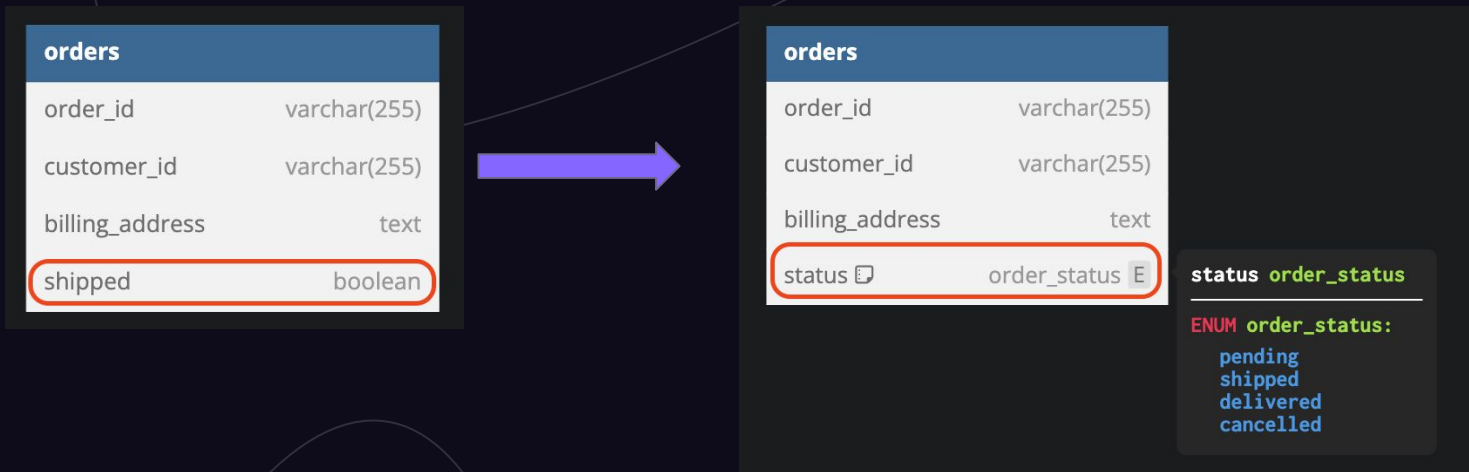
Expand



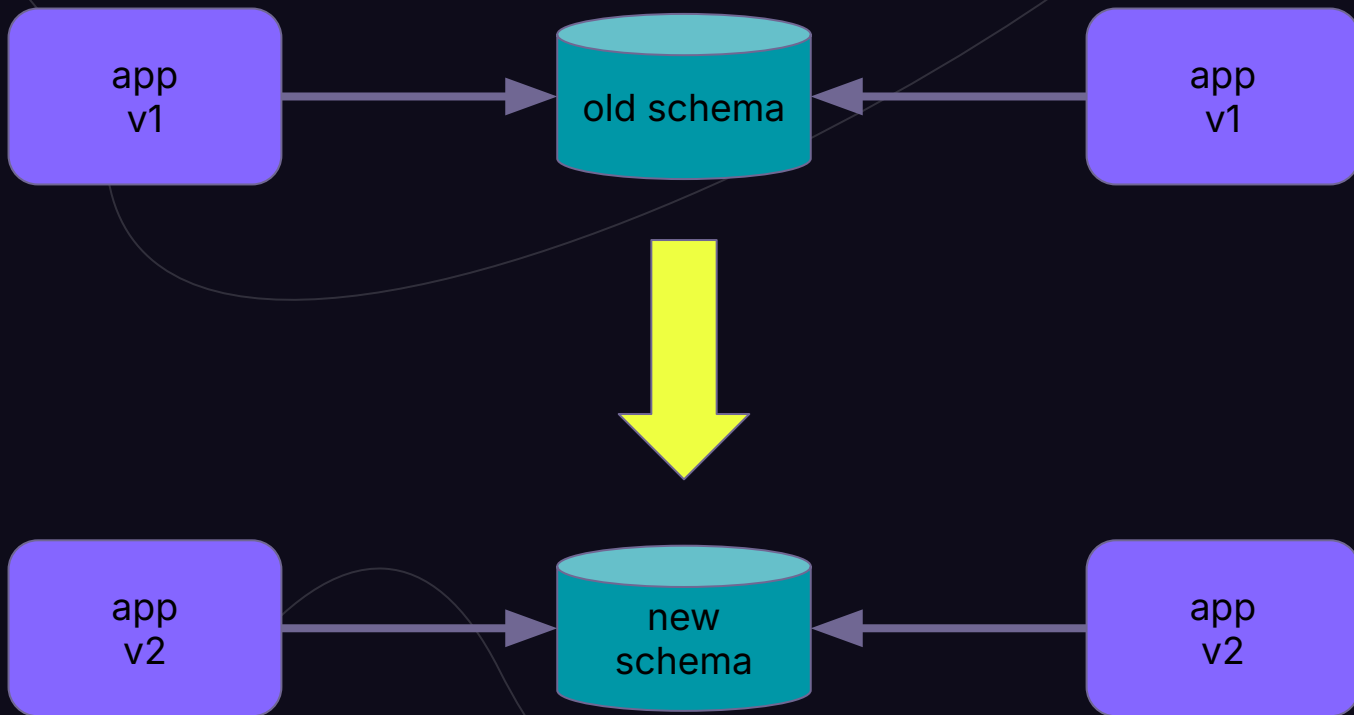
Contract



Expand & contract

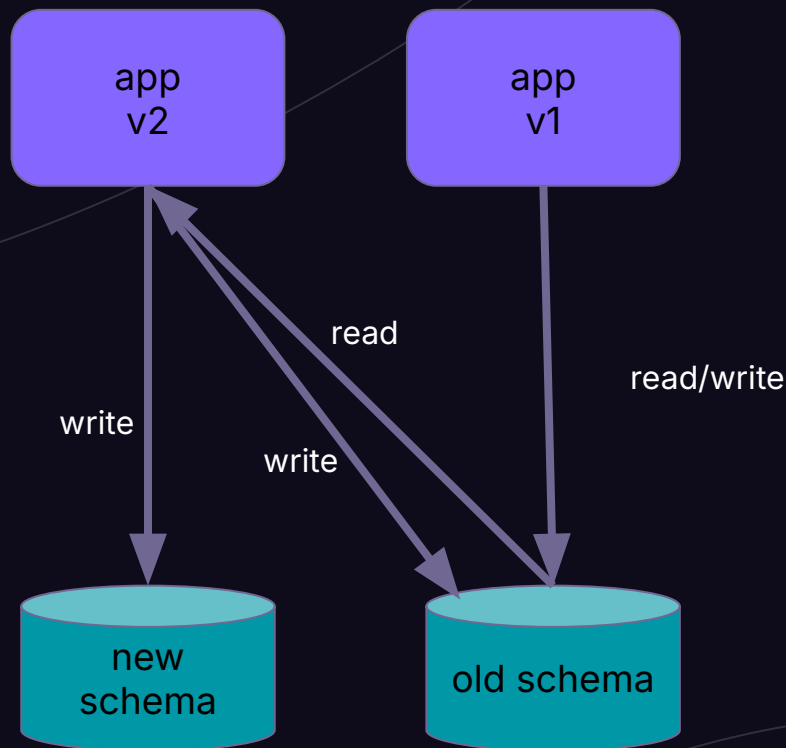


Big bang migration



expand/contract - dual write

id	customer_id	billing_address	shipped	status
3	5678	456 Somewhere Lane	False	<null>
4	1234	123 Somewhere Street	True	<null>



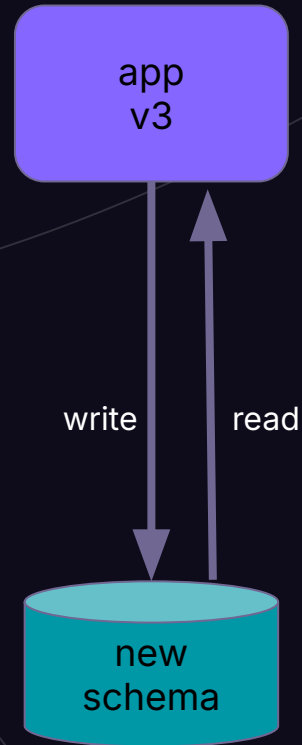
expand/contract - migrate

- Wait for the rollout of v2 to complete
- Run a data migration to backfill the `status` field

id	customer_id	billing_address	shipped	status
1	1234	123 Somewhere Street	True	shipped
2	5678	456 Somewhere Lane	False	pending



expand/contract - read new



expand/contract - contract

- Once the rollout of v3 is complete, drop the `shipped` field
- The migration is complete



id	customer_id	billing_address	shipped	status
1	1234	123 Somewhere Street	True	shipped
2	5678	456 Somewhere Lane	False	pending

Expand / contract - complete



3 migrations required:

- Add the new field
- Backfill the new field
- Remove the old field

2 new application versions:

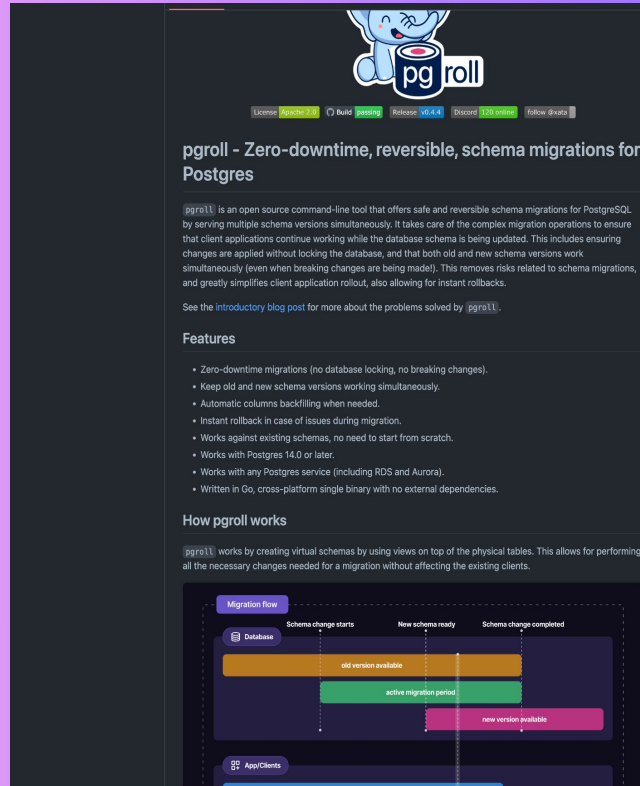
- Dual write
- Final version



**Zero-downtime, reversible, schema
migrations for Postgres**

pgroll - design goals

- Build around the expand/contract pattern
- Keep migration logic out of the application layer
- Easy rollbacks
- No nasty surprises around locking behaviour
- Postgres only
- Open source



The screenshot shows the pgroll website. At the top is the pgroll logo, which features a blue cartoon character holding a paint can with 'pg roll' written on it. Below the logo are links for 'License: Apache 2.0', 'Build: passing', 'Release: v0.4.4', 'Discord: 128 online', and 'Follow @xata'. The main heading is 'pgroll - Zero-downtime, reversible, schema migrations for Postgres'. Below this is a paragraph describing pgroll as an open source command-line tool that offers safe and reversible schema migrations for PostgreSQL by serving multiple schema versions simultaneously. It mentions that it takes care of complex migration operations to ensure client applications continue working while the database schema is being updated, including ensuring changes are applied without locking the database and that both old and new schema versions work simultaneously. A link to an introductory blog post is provided. The 'Features' section lists several bullet points: zero-downtime migrations (no database locking, no breaking changes), keeping old and new schema versions working simultaneously, automatic columns backfilling when needed, instant rollback in case of issues during migration, working against existing schemas (no need to start from scratch), working with Postgres 14.0 or later, working with any Postgres service (including RDS and Aurora), and being written in Go as a cross-platform single binary with no external dependencies. The 'How pgroll works' section explains that pgroll works by creating virtual schemas by using views on top of the physical tables, allowing for all necessary changes needed for a migration without affecting existing clients. Below this is a 'Migration flow' diagram showing a timeline for a database migration. The timeline has four stages: 'Schema change starts', 'New schema ready', and 'Schema change completed'. The 'old version available' bar spans from the start to the 'New schema ready' stage. The 'active migration period' bar spans from the start to the 'Schema change completed' stage. The 'new version available' bar starts at the 'New schema ready' stage and continues until the end of the timeline. The 'App/Clients' bar at the bottom shows a period of downtime during the migration.

pgroll is an open source command-line tool that offers safe and reversible schema migrations for PostgreSQL by serving multiple schema versions simultaneously. It takes care of the complex migration operations to ensure that client applications continue working while the database schema is being updated. This includes ensuring changes are applied without locking the database, and that both old and new schema versions work simultaneously (even when breaking changes are being made). This removes risks related to schema migrations, and greatly simplifies client application rollout, also allowing for instant rollbacks.

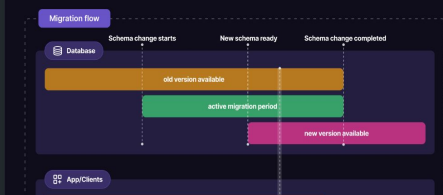
See the [introductory blog post](#) for more about the problems solved by `pgroll`.

Features

- Zero-downtime migrations (no database locking, no breaking changes).
- Keep old and new schema versions working simultaneously.
- Automatic columns backfilling when needed.
- Instant rollback in case of issues during migration.
- Works against existing schemas, no need to start from scratch.
- Works with Postgres 14.0 or later.
- Works with any Postgres service (including RDS and Aurora).
- Written in Go, cross-platform single binary with no external dependencies.

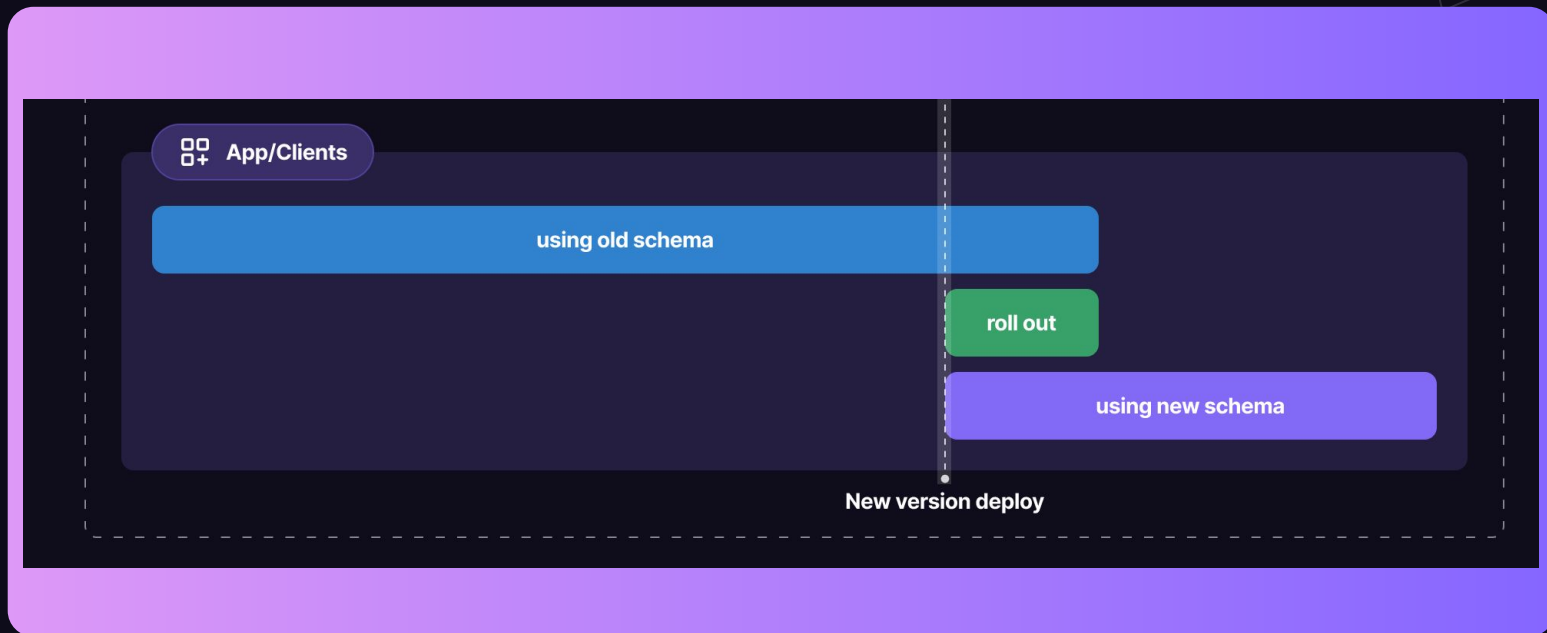
How pgroll works

`pgroll` works by creating virtual schemas by using views on top of the physical tables. This allows for performing all the necessary changes needed for a migration without affecting the existing clients.



The diagram illustrates the migration flow. It shows a timeline with three main stages: 'Schema change starts', 'New schema ready', and 'Schema change completed'. The 'old version available' bar spans from the start to the 'New schema ready' stage. The 'active migration period' bar spans from the start to the 'Schema change completed' stage. The 'new version available' bar starts at the 'New schema ready' stage and continues until the end of the timeline. The 'App/Clients' bar at the bottom shows a period of downtime during the migration.

Application rollouts



Demo

Lesson learned

- Expand contract is a powerful technique for schema change
- Migration tools should operate at a higher level than raw SQL
- Migrations are long-lived processes and migration tools should manage them end to end
- Data migrations should be handled by migration tools, not at the application level



Thank you!

 @xata

 xataio/pgroll

 @xata.io

 xata.io/discord