

Postgres schema migrations using the expand/contract pattern

September 30, 2024

Speaker

Xata



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

The plan

01 Common pitfalls

02 Tools and techniques

03 Expand contract migrations

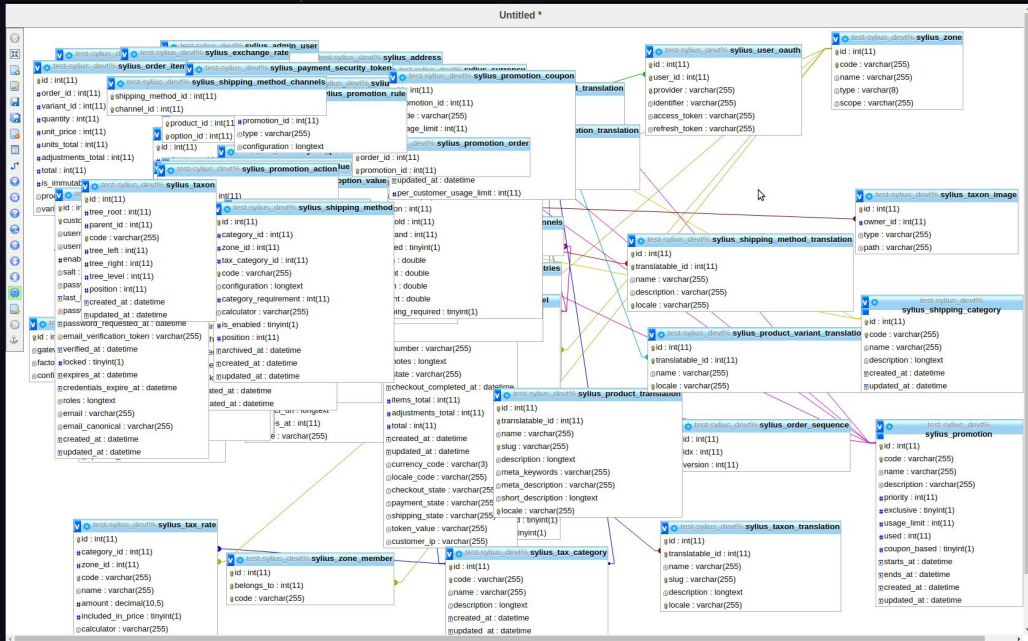
04 Expand/contract with pgroll



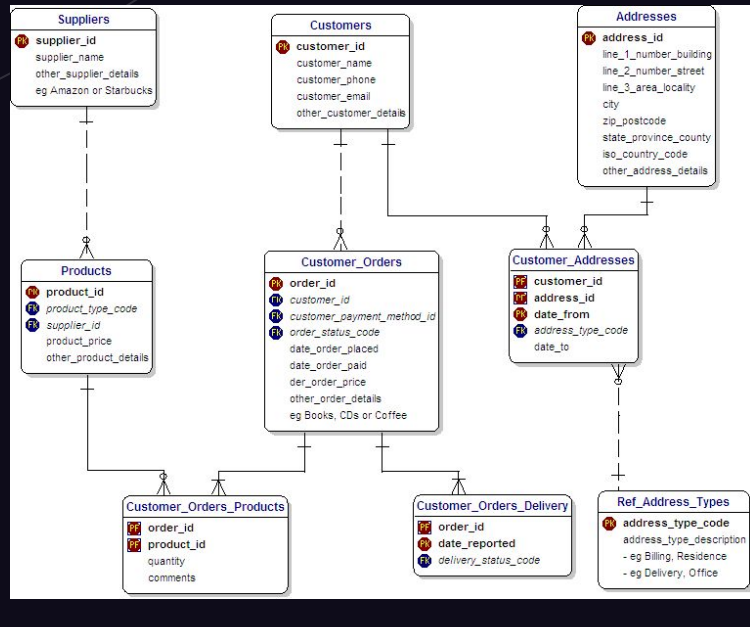
(image from londonist.com)



London



New York

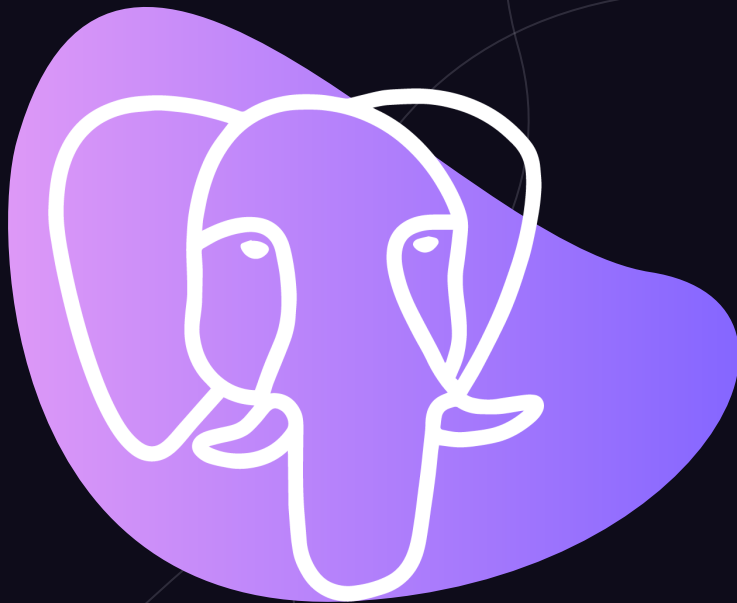


“Database schemas are notoriously volatile, extremely concrete, and highly depended on. This is one reason why the interface between OO applications and databases is so difficult to manage, and why schema updates are generally painful.”

Robert C. Martin, Clean Architecture

Some assumptions we'll be making

- ✓ Postgres as your primary database
- ✓ Database running in production
- ✓ Application code is live
- ✓ Zero downtime is important to you



Common pitfalls

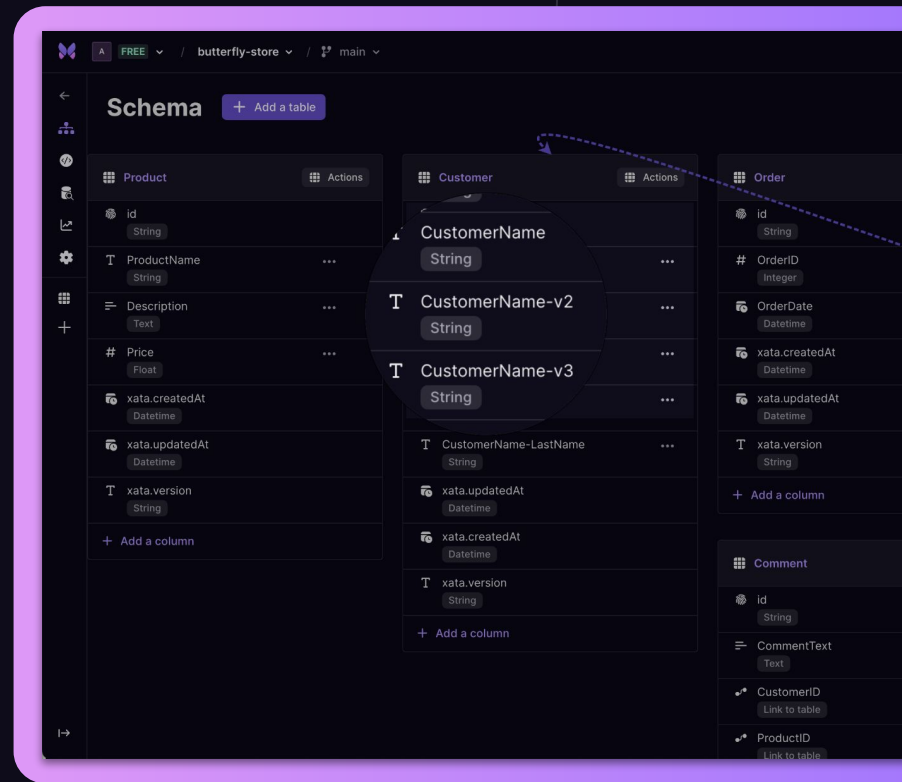
Additive-only changes and schema debt

Wait, what?

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

Why is this bad?

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



The locking minefield

Wait, what?

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 table below that some lock modes are self-conflicting (for example, an ACCESS EXCLUSIVE lock mode 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 data 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.
- SHARE (ShareLock)**
 - Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.
 - 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 self-exclusive so that only one session can hold it at a time.
 - Acquired by CREATE TRIGGER and some forms of ALTER TABLE.

Testing schema migrations

Wait, what?

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

Wait, what?

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
- Likely untested
- Time consuming



Deploying application changes

Wait, what?

One does not simply deploy database changes to production, there's a natural order of things

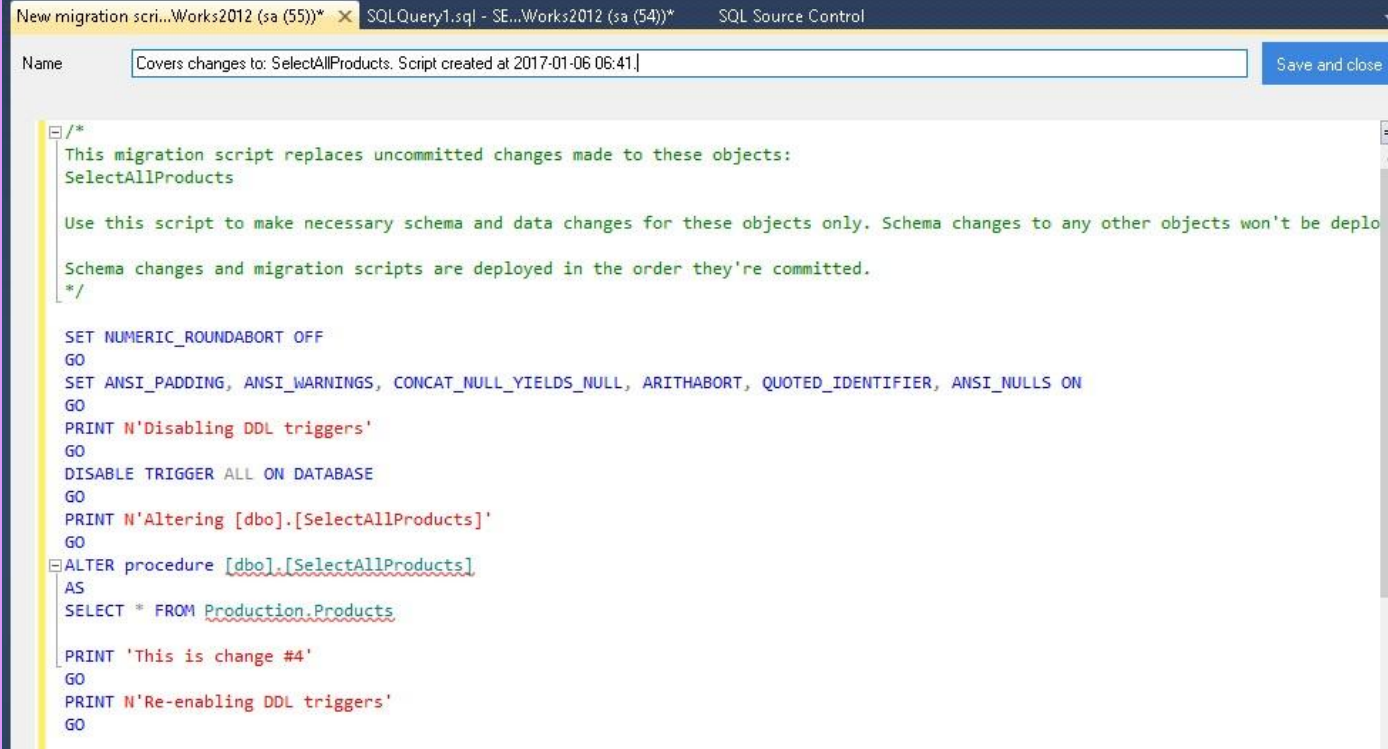
Why is this bad?

- Data consistency
- Mismatched schemas and application code
- Angry users 😡



Production rollout strategies

Version controlled SQL scripts



The screenshot shows a window titled 'New migration scri...Works2012 (sa (55))* x SQLQuery1.sql - SE...Works2012 (sa (54))* SQL Source Control'. The 'Name' field contains the text 'Covers changes to: SelectAllProducts. Script created at 2017-01-06 06:41'. A 'Save and close' button is visible in the top right. The main area contains a SQL script with a multi-line comment and several SQL commands.

```
/*
This migration script replaces uncommitted changes made to these objects:
SelectAllProducts

Use this script to make necessary schema and data changes for these objects only. Schema changes to any other objects won't be deplo

Schema changes and migration scripts are deployed in the order they're committed.
*/

SET NUMERIC_ROUNDABORT OFF
GO
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT, QUOTED_IDENTIFIER, ANSI_NULLS ON
GO
PRINT N'Disabling DDL triggers'
GO
DISABLE TRIGGER ALL ON DATABASE
GO
PRINT N'Altering [dbo].[SelectAllProducts]'
GO
ALTER procedure [dbo].[SelectAllProducts]
AS
SELECT * FROM Production.Products

PRINT 'This is change #4'
GO
PRINT N'Re-enabling DDL triggers'
GO
```

Frameworks & ORMs

The Django logo, featuring the word "django" in a white, lowercase, sans-serif font on a dark green rounded rectangular background.

```
python manage.py makemigrations
```

The Rails logo, consisting of a red stylized sun or gear icon above the word "RAILS" in a bold, red, uppercase, sans-serif font.

```
rails generate migration AddPartNumberToProduct
```

The Prisma logo, featuring a white stylized triangle icon to the left of the word "Prisma" in a white, sans-serif font.

```
prisma migrate dev --name init
```

The Drizzle logo, featuring three white slanted lines to the left of the word "Drizzle" in a white, sans-serif font.

```
pnpm drizzle-kit generate:mysql
```

Application specific process

GitLab Docs

What's new? v16.9 Get free trial

- Database load balancing
- Database migration pipeline
- Database review guidelines
- Database check-migrations job
- Delete existing migrations
- Enums
- Foreign keys and associations
- Introducing a new database migration version
- Layout and access patterns
- Maintenance operations
- Migrations style guide
- Multiple databases >
- Ordering table columns
- Pagination guidelines >
- Post-deployment migrations
- Query comments with Marginalia
- Query Recorder
- Single Table Inheritance
- SQL guidelines
- Strings and the Text data type

◦ Creating a new table, example: `create_table`.

◦ Adding a new column to an existing table, example: `add_column`.

3. Batched background migrations. These aren't regular Rails migrations, but application code that is executed via Sidekiq jobs, although a post-deployment migration is used to schedule them. Use them only for data migrations that exceed the timing guidelines for post-deploy migrations. Batched background migrations should *not* change the schema.

Use the following diagram to guide your decision, but keep in mind that it is just a tool, and the final outcome will always be dependent on the specific changes being made:

```

graph TD
    A{Schema changed?} -- Yes --> B{Critical to speed or behavior?}
    A -- No --> C{Is it fast?}
    B -- Yes --> D{Is it fast?}
    B -- No --> E[Post-deploy migration]
    D -- Yes --> F[Regular migration]
    D -- No --> G[Post-deploy migration + feature flag]
    C -- Yes --> H[Post-deploy migration]
    C -- No --> I[Background migration]
    
```

How long a migration should take

In general, all migrations for a single deploy shouldn't take longer than 1 hour for GitLab.com. The following guidelines are not hard rules, they were estimated to keep migration duration to a minimum.

ⓘ Keep in mind that all durations should be measured against GitLab.com.

Migration Type	Recommended Duration	Notes

On this page

- Choose an appropriate migration type
- How long a migration should take
- Decide which database to target
- Create a regular schema migration
 - Regular schema migrations to add new models
- Schema Changes
- Avoiding downtime
- Reversibility
- Atomicity and transaction
 - Heavy operations in a single transaction
 - Temporarily turn off the statement timeout limit
 - Disable transaction-wrapped migration
- Naming conventions
 - Truncate long index names
 - Migration timestamp age
 - Best practice
- Migration helpers and versioning
 - Retry mechanism when acquiring database locks
 - Usage with transactional migrations
 - Removing a column
 - Multiple changes on the same table
 - Removing a foreign key
 - Changing default value for a column
 - Creating a new table with a foreign key
 - Creating a new table when we have two foreign keys
 - Usage with non-transactional migrations (`disable_ddl_transaction!`)
 - When to use the helper method
 - How the helper method works
 - Lock-retry methodology at the SQL level

⏪ Collapse sidebar

Planned downtime



The banner features the Etsy logo with 'beta' in a smaller font on an orange background. To the right, the text 'Technical difficulties' is overlaid on a photograph of server racks.

Etsy beta Technical difficulties

Don't worry, Haim's working on it!

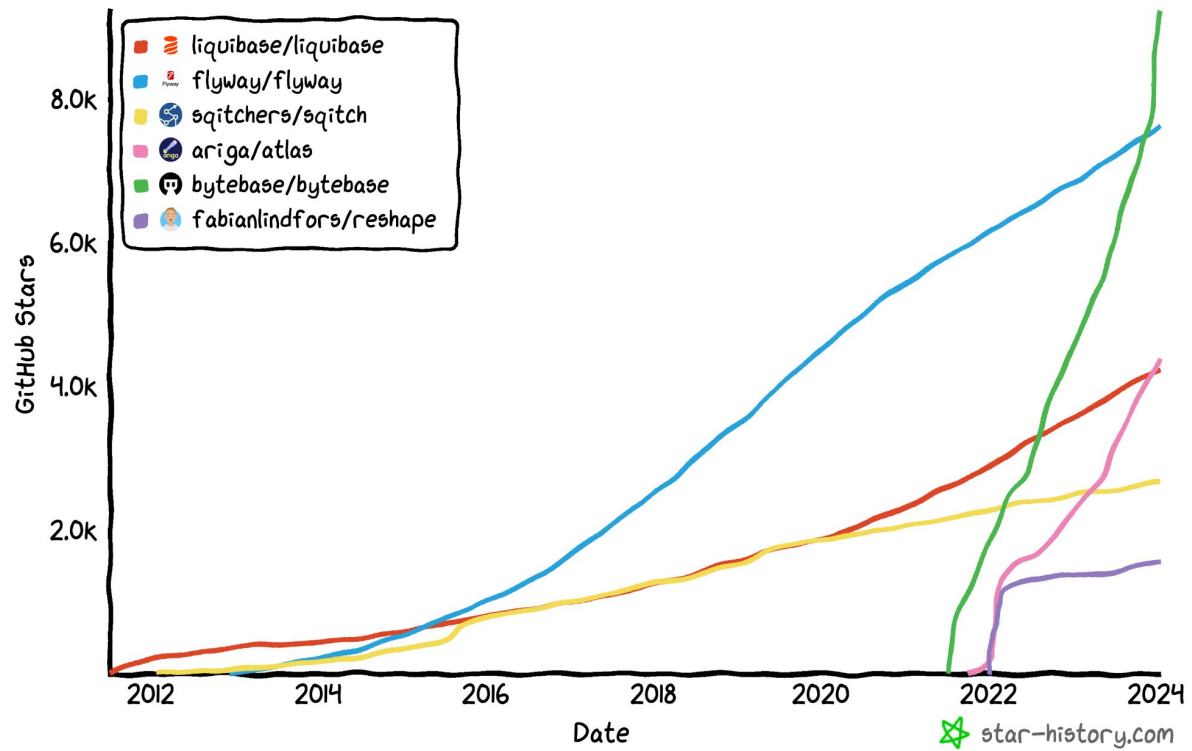


A pixelated cartoon character with glasses and a beard is shown from the chest up, wearing a red t-shirt. He is surrounded by bright red and orange flames, suggesting he is on fire.

What's happening: Fixing some ugly emergency server issues. We'll be back asap.
Estimated downtime: 12:00AM - 4:00AM EDT

Tools available

Star History



The expand / contract pattern

Expand & contract

In vivo: information ecosystem:

Database types

Comparing relational and document databases

Relational Databases

What is an ORM?

Comparing SQL, query builders, and ORMs

What are database migrations?

Strategies for deploying database migrations

Using the expand and contract pattern for schema changes

Comparing common database infrastructure patterns

Document Databases

PostgreSQL

The benefits of PostgreSQL

Getting to know PostgreSQL

5 ways to host PostgreSQL databases

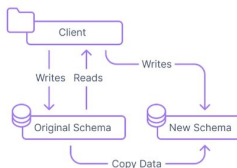
Setting up a local PostgreSQL database

How to configure a PostgreSQL database on RDS

Connecting to PostgreSQL databases

Authentication and authorization

Step 3: Migrate existing data to the new schema



The new data schema is in place within the database, but it does not yet have any of the actual data that the original schema holds. To prepare the new schema for actual use, you need to migrate the data from the existing columns or tables to the new ones.

In some cases, this step may require just copying the existing values to the new structure, but often, you may need to modify the data in some way. This might involve modifying data types.

It is essential to think carefully about how the new schema is semantically equivalent to their original `color` column stored values to colors stored in another table, then migrate some records.

Other times, however, it might be necessary to split a field into `first_name` and `last_name`, for example. For example, while you could write a script that may not do the right thing with new fields relate to the original field, you might need to migrate some records.

How to rename a column on PlanetScale

1. Create a new column with the new name.
2. Update the application to write to both columns with new data.
3. Backfill all the data in the new column for rows that are still missing that information.
4. Optionally, add constraints like **NOT NULL** to the new column once all the data is backfilled.
5. Update the application to only use the new column, and remove any references to the old column name.
6. Drop the old column.

This means at least two deploy requests are needed (potentially more if you want to enforce **NOT NULL** without a **DEFAULT**), where you first add the newly named column and then drop the old one.

Expand & contract

orders	
order_id	varchar(255)
customer_id	varchar(255)
billing_address	text
shipped	boolean



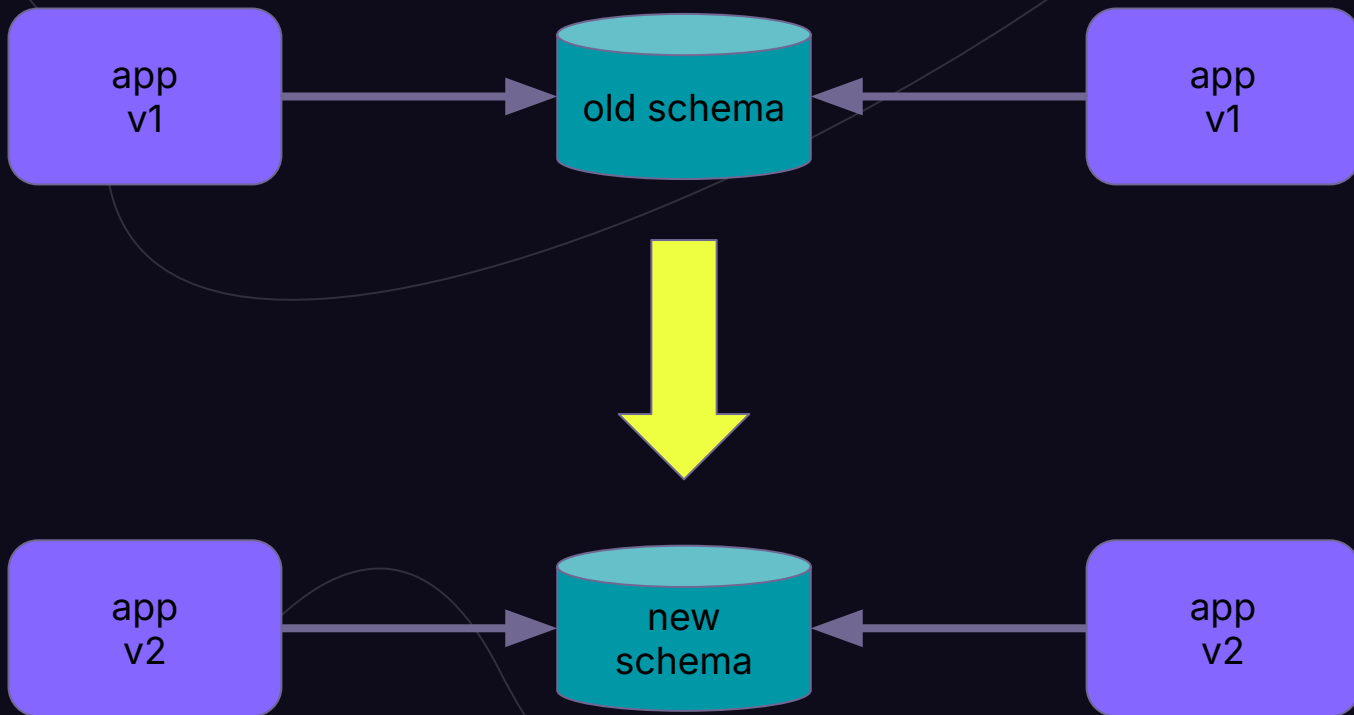
orders	
order_id	varchar(255)
customer_id	varchar(255)
billing_address	text
status	order_status

status order_status

ENUM order_status:

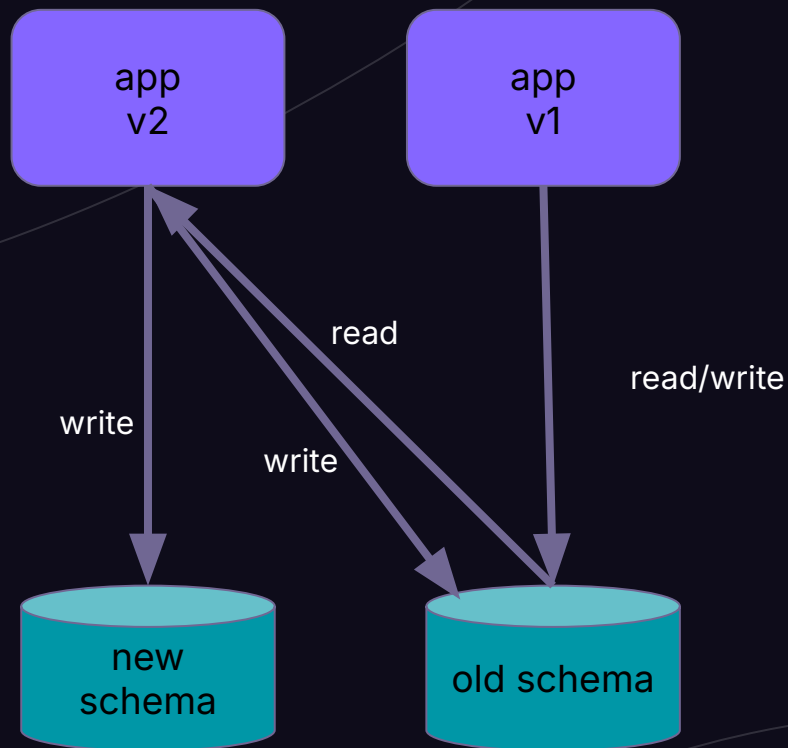
- pending
- shipped
- delivered
- cancelled

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>



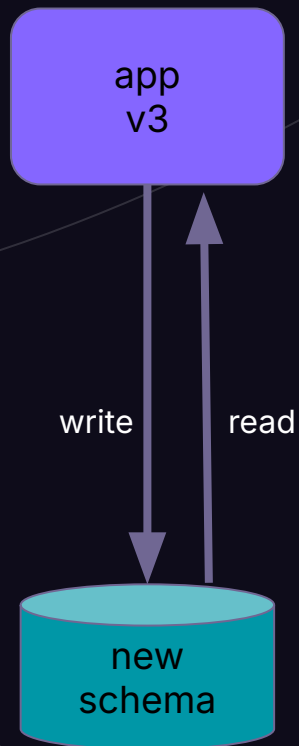
expand/contract - migrate

- 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



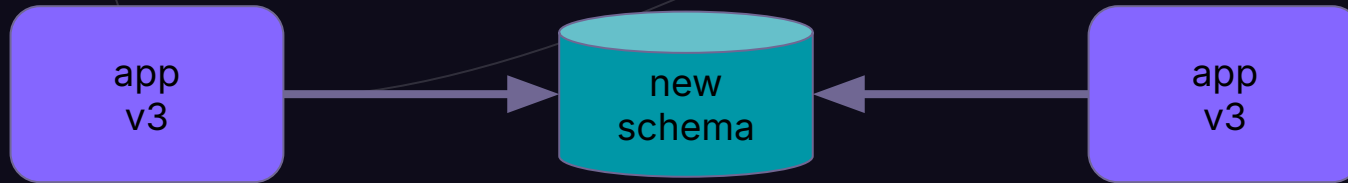
expand/contract - contract

- 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

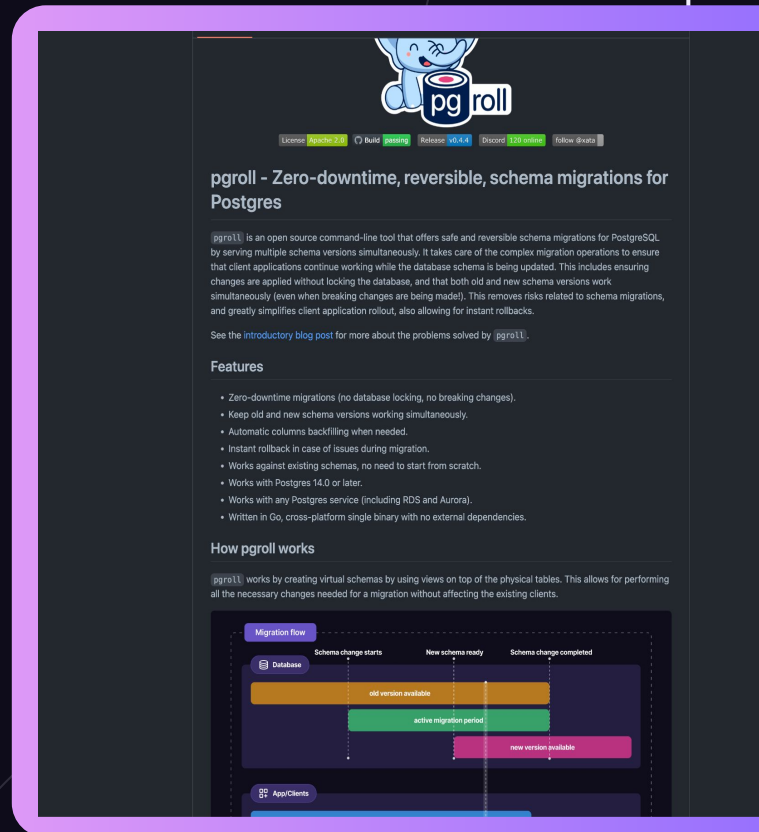




**Zero-downtime, reversible, schema
migrations for Postgres**

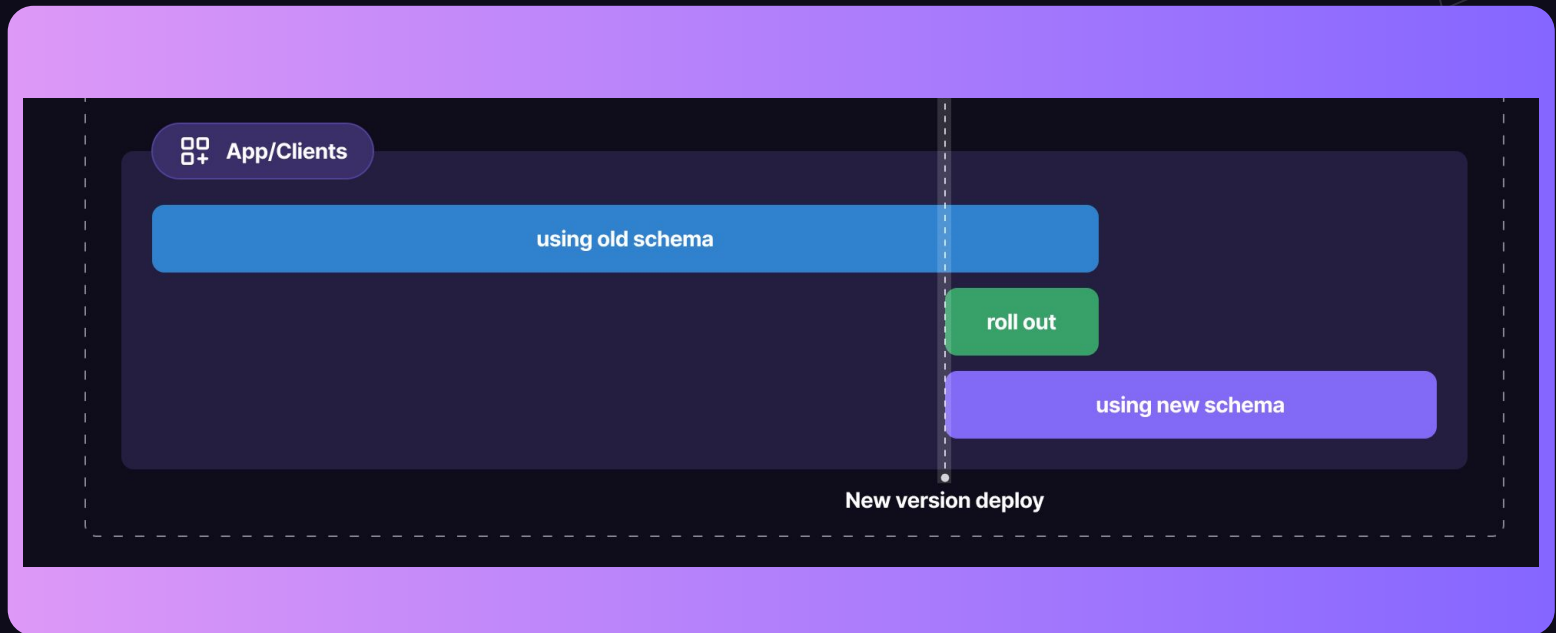
pgroll - design goals

- 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 right is the pgroll logo, which features a blue cartoon character holding a blue mug with the text 'pg roll' on it. Below the logo are navigation links: 'License: Apache 2.0', 'Build: testing', '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 for safe and reversible schema migrations. A 'Features' section lists several capabilities, including zero-downtime migrations, simultaneous old and new schema versions, automatic backfilling, instant rollbacks, and compatibility with Postgres 14.0 and later. A 'How pgroll works' section explains that it uses virtual schemas created with views. At the bottom, a 'Migration flow' diagram illustrates the process: 'Schema change starts' (old version available), 'New schema ready' (active migration period), and 'Schema change completed' (new version available).

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



What's next?

- Higher level migrations
- Multiple in-progress migrations at once
- Dry run data migration



Coming soon to Xata

The screenshot displays the Xata Schema History interface. The top navigation bar shows the project name 'san-diego-get-it-done' and the branch 'main'. The left sidebar contains navigation options: 'Back to workspace', 'Schema', 'Playground', 'Search engine', 'Chat with your data', 'Usage & limits', and 'Settings'. The main content area is titled 'Schema' and includes a '+ Add a table' button. Below this, there are three tabs: 'Schema View', 'Schema History' (which is active), and 'Migration Editor'. The 'Schema History' tab displays a table of migration events:

Date	Summary
Jan 23 02:29:32.09 PM	Renamed table get_it_done_requests_open_datasd → get_it_done_requests_open_data
Jan 23 02:15:27.94 PM	Added column get_it_done_requests_open_datasd.attachments
Jan 23 02:13:25.43 PM	Dropped column get_it_done_requests_open_datasd.iamfloc
Jan 23 02:13:16.47 PM	Dropped column get_it_done_requests_open_datasd.date_closed
Jan 23 02:01:33.65 PM	Added table get_it_done_requests_open_datasd

On the right side, a detailed view of a migration is shown for 'mig_cmo176v4noji046ao52g'. It includes the following metadata:

- Parent: mig_cmo10jv4noji046ao4tg
- Status: true
- Started: Jan 23 02:29:32.09 PM
- Type: pgsroll

The migration details are accompanied by a JSON snippet:

```
{
  "rename_table": {
    "to": "get_it_done_requests_open_data",
    "from": "get_it_done_requests_open_datasd"
  }
}
```

Thank you!

 @xata

 xataio/pgroll

 @xata.io

 xata.io/discord

Thank you