pgroll - Zero-downtime, reversible, schema changes for Postgres

Tudor Golubenco

November, 2024

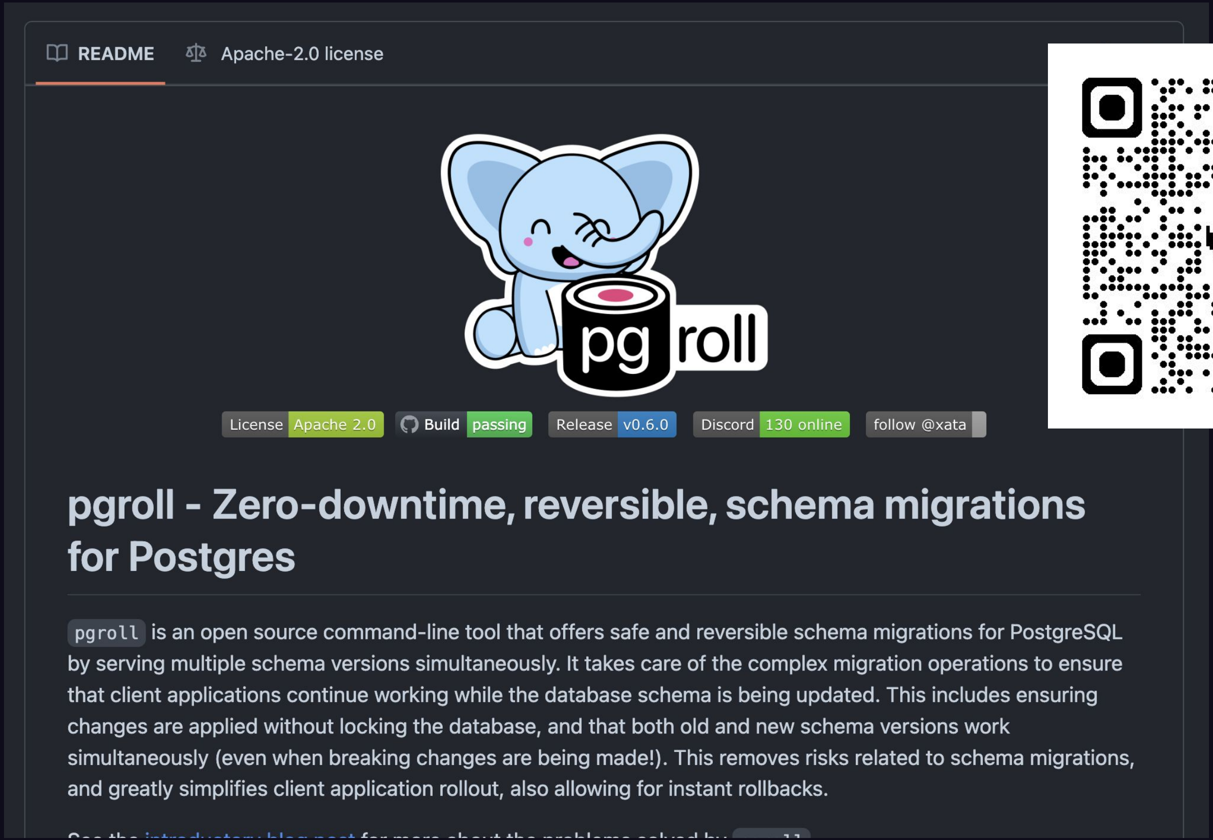
xata

xata.io
Zero-downtime Postgres Platform

Fully managed Postgres service

+ Branching
+ Zero-downtime, reversible, schema changes
+ Developer workflow for safe schema changes
+ Excellent support

https://github.com/xataio/pgroll



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

`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.

Motivation

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)

Adding a column

- Adding a nullable column is safe:

```
ALTER TABLE foo ADD COLUMN col1 text;
```

✅ Minimal locking ✅ No backfill ✅ Single step ✅Backwards compatible

Adding a NOT NULL column

- Adding a not null column:

  `ALTER TABLE foo ADD COLUMN col2 text NOT NULL;   -- won’t work`

  ✅ Minimal locking, ❗ Backfill required, ❗ Multiple steps, ✅Backwards compatible

- Adding a not null column with a default value:

  `ALTER TABLE foo ADD COLUMN col2 text NOT NULL DEFAULT 'hi';`

  ✅ Minimal locking, ✅ No backfill, ✅ Single step, ✅Backwards compatible

- Adding a not null column with a volatile default value:

  `ALTER TABLE foo ADD COLUMN col4 text DEFAULT timeofday();  -- will take long`

  ❗ Locking, ❗ Backfill required, ✅Single step, ✅Backwards compatible

xata

Adding a UNIQUE column

- Adding a nullable + unique column:

  ```
  ALTER TABLE foo ADD COLUMN col4 text UNIQUE;  -- will take long
  ```

  ❗ Locking, ✅ No backfill, ✅ Single step, ✅Backwards compatible
- Adding a not null + unique column:

  ```
  ALTER TABLE foo ADD COLUMN col5 text NOT NULL UNIQUE; -- will error out after a long time
  ```

  ❗ Locking, ✅ No backfill, ❗ Single step, ✅Backwards compatible
- To solve:
  - Add the column without UNIQUE
  - Ensure the data is actually unique
  - Create an unique index with the CONCURRENTLY keyword

Adding a column with a custom constraint

- Adding a column with a custom constraint:

  ALTER TABLE foo ADD COLUMN col6 text CHECK (LENGTH(col6) < 140); -- will take long

  ❗ Locking, ❗ Backfill needed, ✅ Single step, ✅Backwards compatible

- To avoid locking, we need multiple steps:
  - ALTER TABLE foo ADD COLUMN col6 text;
  - ALTER TABLE foo ADD CONSTRAINT foo_col6_length CHECK (LENGTH(col6) < 140) NOT VALID;
  - ALTER TABLE foo VALIDATE CONSTRAINT foo_col6_length;

Deleting a column

- Adding a nullable + unique column:

  `ALTER TABLE foo DROP COLUMN col6;`

  ✅ Minimal locking, ✅ No backfill, ✅ Single step, ❗ Backwards incompatible
- To solve:
  - First remove the usage of the column from the app
  - Then drop the column

Renaming a column

- Renaming a column:

  ALTER TABLE foo RENAME COLUMN col2 TO col6;

  ✅ Minimal locking, ✅ No backfill, ✅ Single step, ❗ Backwards incompatible
- Won’t work with rolling application upgrade
- To solve in a backwards compatible way:
  - Add the new column
  - Backfill the data
  - Have the app write in both at the same time
  - Upgrade the app
  - Drop the old column

Changing the type of a column

- Changing the type of a column:

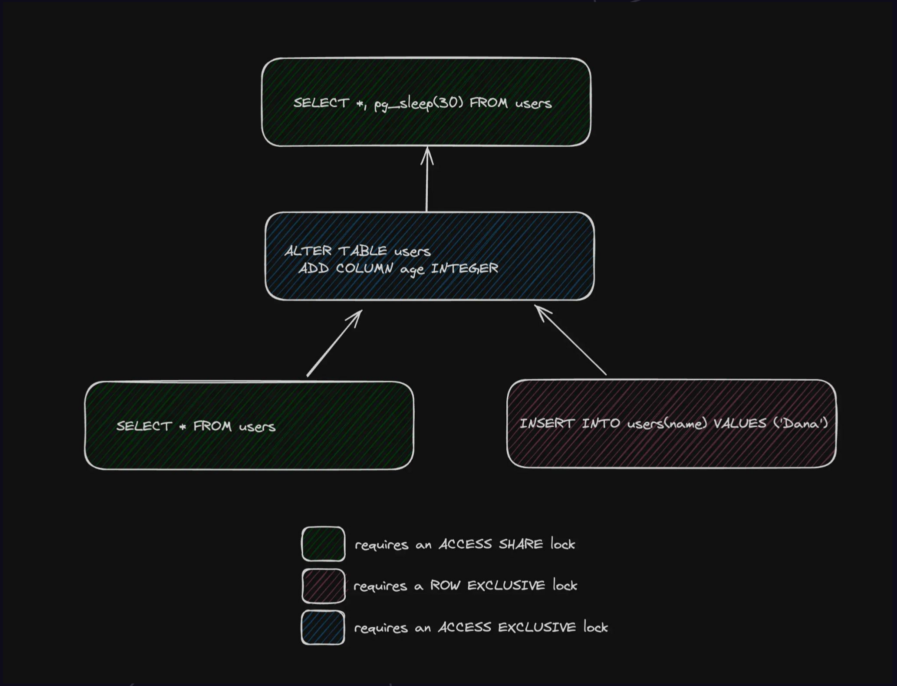
  ```
  ALTER TABLE foo ALTER COLUMN col7 TYPE INTEGER USING col7::integer;
  ```

  ❗ Locking, ✅ No backfill, ✅ Single step, ❗ Backwards incompatible
- To do it without locking, the Expand and Contract pattern:
  - Add a new column with the new type
  - Copy/cast the data from the old column (backfill)
  - Remove the old column

Even minimal locking can be an issue

- Most ALTER statements take ACCESS EXCLUSIVE lock
- They need to wait for any existing query to finish
- It will queue behind any long query
- Any new reads and writes will be queued after the ALTER
- → **Downtime**

- Solution:
  - **SET lock_timeout TO '1000ms';**


SELECT *, pg_sleep(30) FROM users
ALTER TABLE users ADD COLUMN age INTEGER
SELECT * FROM users
INSERT INTO users(name) VALUES ('Dana')
requires an ACCESS SHARE lock
requires a ROW EXCLUSIVE lock
requires an ACCESS EXCLUSIVE lock

Intermediary conclusions

- Locking behaviour of ALTER can be sometimes surprising
- To avoid locking, you often need multiple steps
- To avoid backwards compatibility issues, you can use expand/contract pattern
- When you do multiple steps, you often need to backfill the data

What if I told you...

There is a way so that all schema changes are:

✅ Lock safe

✅ Single step

✅ Automated Backfill

✅ Backwards compatible

✅ Rollback is easy

How does pgroll work?

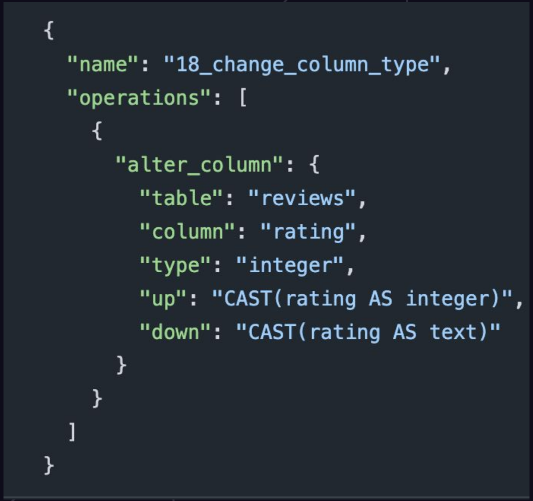
How does pgroll work?

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

Higher level operations

- Instead of ALTER statements, pgroll 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

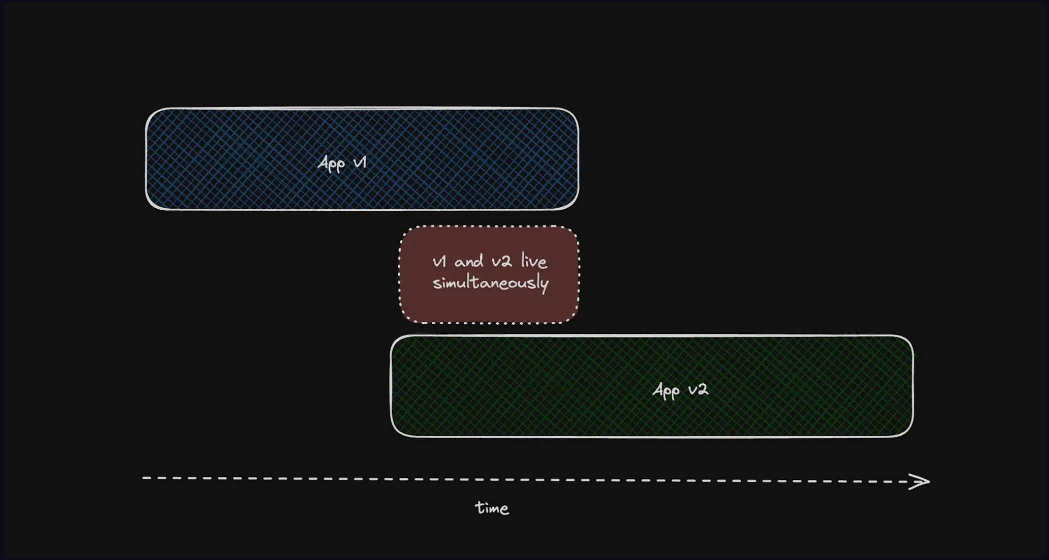


```
{
  "name": "18_change_column_type",
  "operations": [
    {
      "alter_column": {
        "table": "reviews",
        "column": "rating",
        "type": "integer",
        "up": "CAST(rating AS integer)",
        "down": "CAST(rating AS text)"
      }
    }
  ]
}
```

Automated Expand and Contract pattern

- Hidden 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

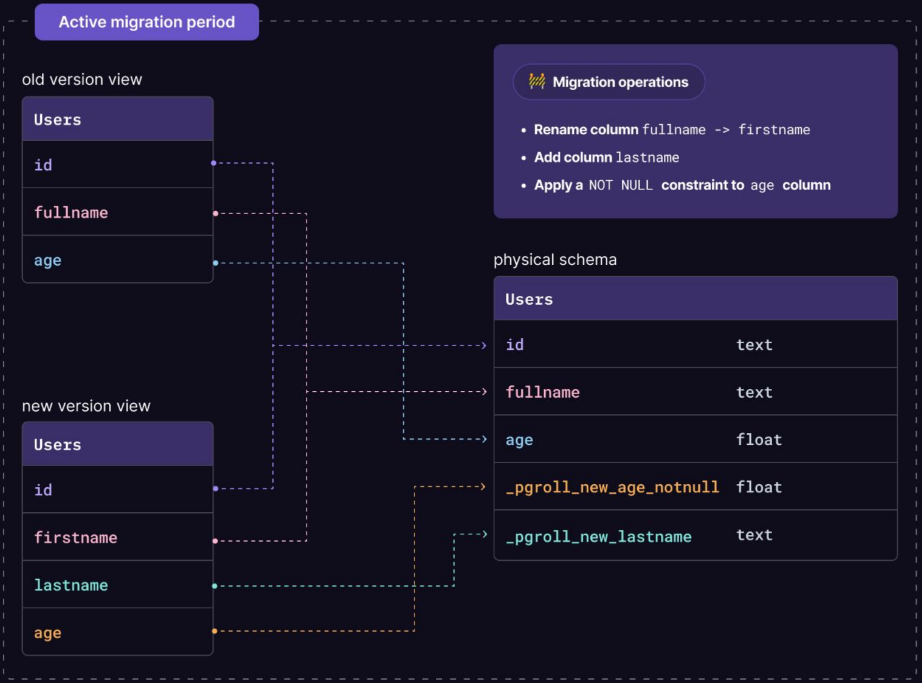


Workflow is always the same:

- 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



Migration operations

- **Rename column** `fullname -> firstname`
- **Add column** `lastname`
- **Apply a** `NOT NULL` **constraint to** `age` **column**

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

xata

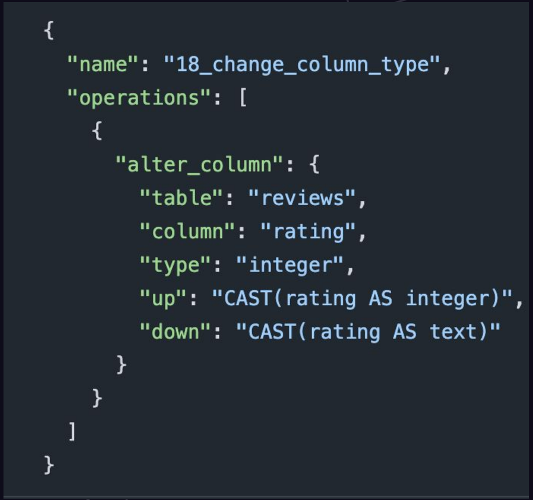
How - Application selects its version by setting the `search_path`

```sql
-- Switch back the new schema, which disallows `NULL`s in the `name` field
SET search_path TO mig_cq778qtlu0oe0bpredl0;

-- 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_cq778jdlu0oe0bpredk0;

-- 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



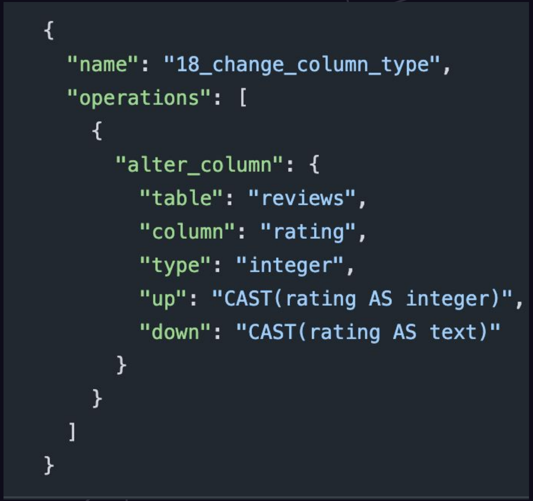
```
{
  "name": "18_change_column_type",
  "operations": [
    {
      "alter_column": {
        "table": "reviews",
        "column": "rating",
        "type": "integer",
        "up": "CAST(rating AS integer)",
        "down": "CAST(rating AS text)"
      }
    }
  ]
}
```

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"

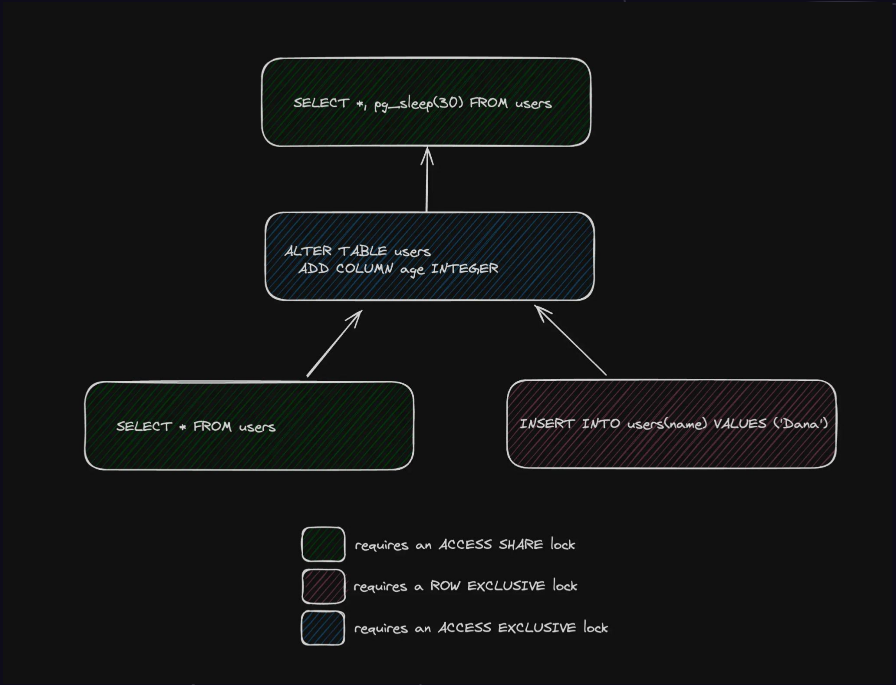
Note: dual write at the column level is necessary here, but you'd have to do it anyway.



```
{
  "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


SELECT *, pg_sleep(30) FROM users
ALTER TABLE users
ADD COLUMN age INTEGER
SELECT * FROM users
INSERT INTO users(name) VALUES ('Dana')
requires an ACCESS SHARE lock
requires a ROW EXCLUSIVE lock
requires an ACCESS EXCLUSIVE lock

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

Demo

Roadmap

- SQL to pgroll json convertor
  - Important for working with existing migration tools
  - E.g. work together with Prisma ORM
- More migration types supported
- Better UX overall
- Public benchmarking at scale

Thanks!