Peaceful Django Migrations

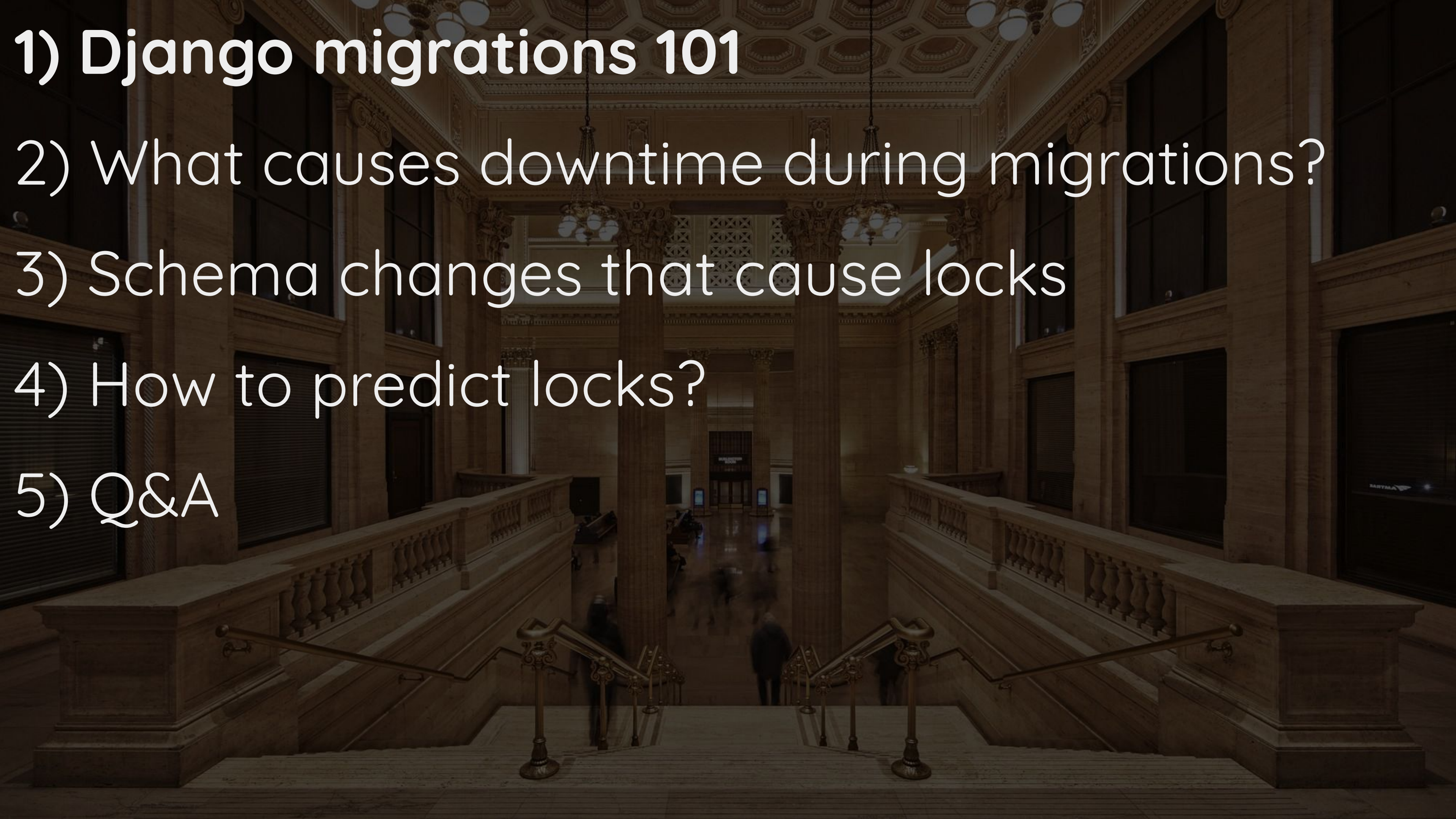
Efe Öge

Hey, I’m Efe Öge

- Pronounced as “Eh-feh”
- Originally from **Türkiye**, based in **The Netherlands**
- **Past:** Founding engineer in early phase startups
- **Now:** Senior engineer @ Sendcloud 📦
- Writes at **efe.me** 🔗
- DjangoNL & Mentor at Django Girls Amsterdam (previously PyCon Turkey, Python Istanbul)
- Django, Python, LLMs, Backend
- Boring Technology Advocate

2024

'00 10 12



1) Django migrations 101

2) What causes downtime during migrations?

3) Schema changes that cause locks

4) How to predict locks?

5) Q&A

Django (Database) Migrations 🕊️

noun. A way of **tracking** and **applying** changes to your **database schema** so it stays in sync with your **models**.

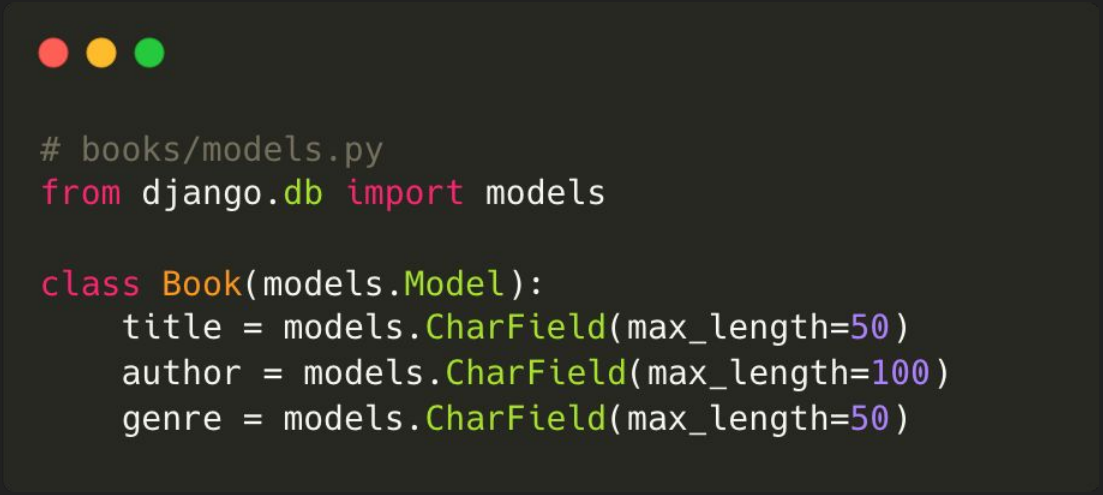
It is not about a migration,

- from **PHP** to **Python (Django)**.
- from **API v2** to **API v3**.
- from **Django 4** to **Django 5**.
- from **FastAPI** to **Django**.
- from **PostgreSQL 16** to **17**.

A library 📖

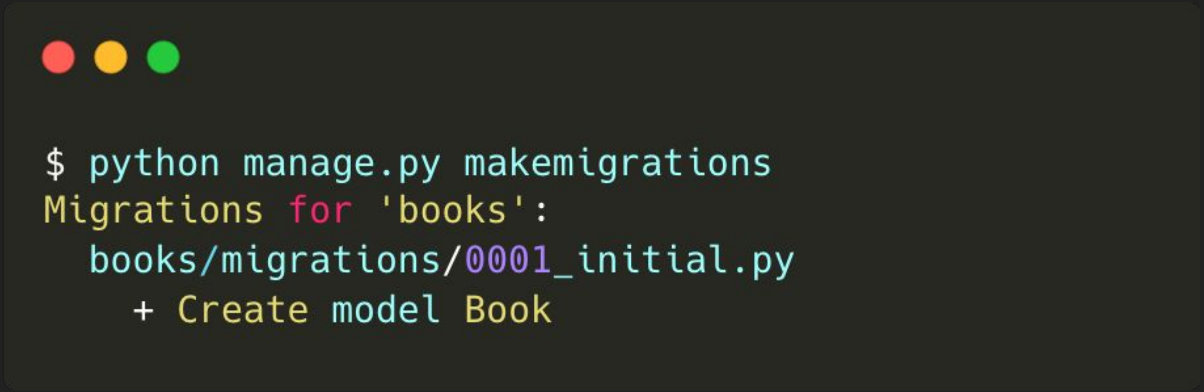
case. help people find, borrow, and return books efficiently while keeping track of the entire collection.

Every book record has **title**, **author**, **genre**.

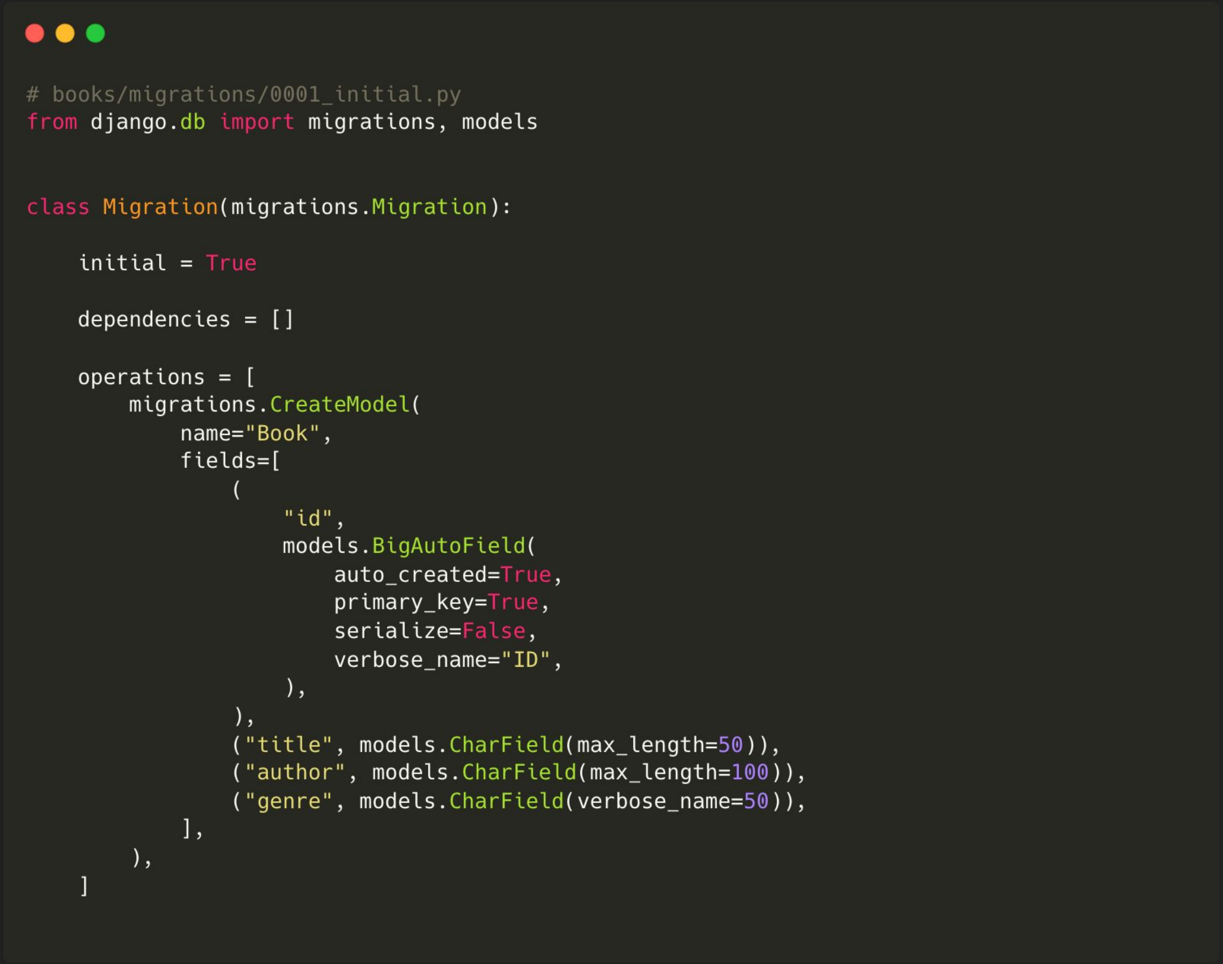


```python
# books/models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=50)
    author = models.CharField(max_length=100)
    genre = models.CharField(max_length=50)
```

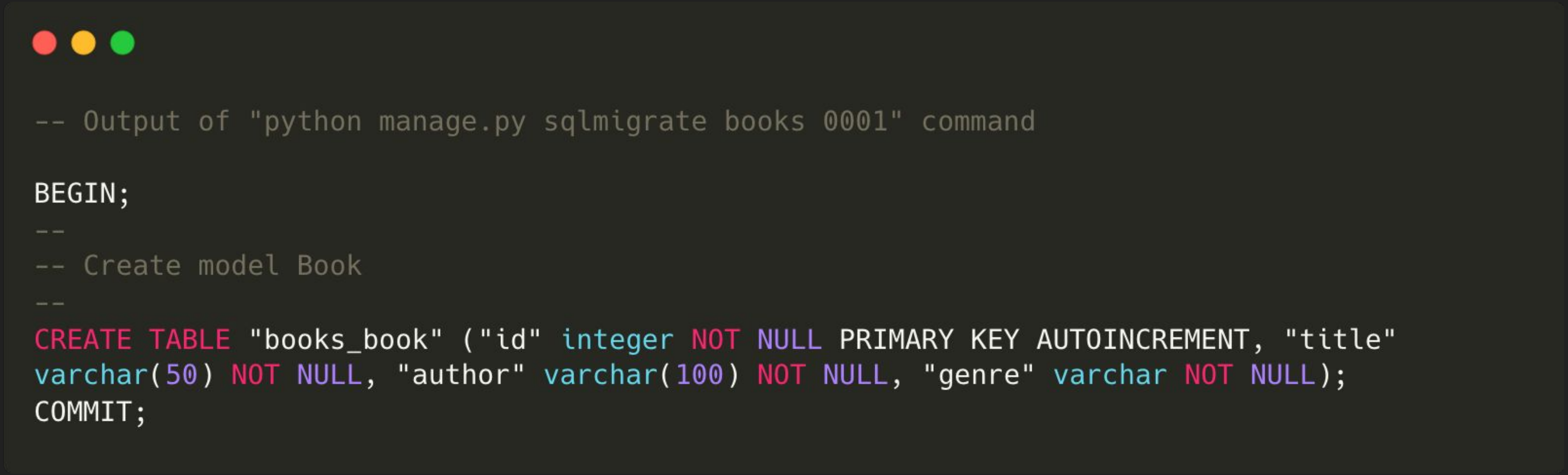
```
$ python
manage.py makemigrations
```



```
$ python manage.py makemigrations
Migrations for 'books':
  books/migrations/0001_initial.py
    + Create model Book
```

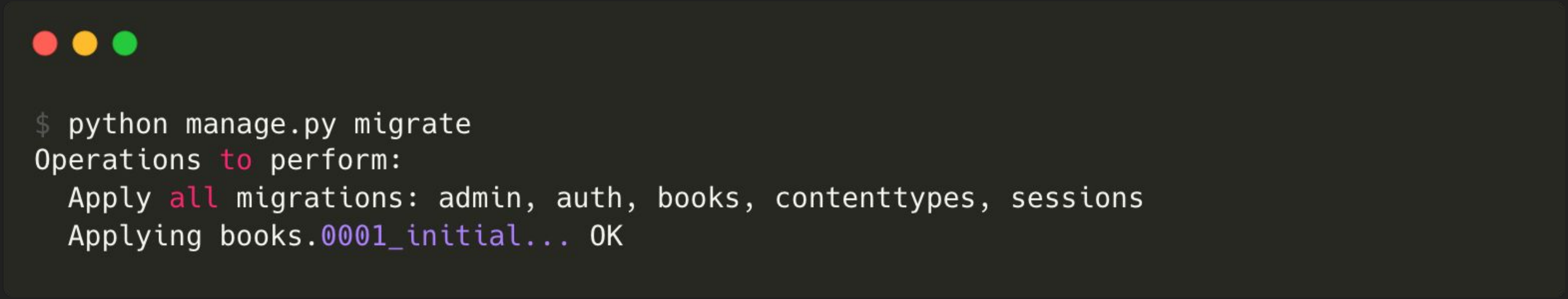


```python
# books/migrations/0001_initial.py
from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = []

    operations = [
        migrations.CreateModel(
            name="Book",
            fields=[
                (
                    "id",
                    models.BigAutoField(
                        auto_created=True,
                        primary_key=True,
                        serialize=False,
                        verbose_name="ID",
                    ),
                ),
                ("title", models.CharField(max_length=50)),
                ("author", models.CharField(max_length=100)),
                ("genre", models.CharField(verbose_name=50)),
            ],
        ),
    ]
```



```
-- Output of "python manage.py sqlmigrate books 0001" command

BEGIN;
--
-- Create model Book
--
CREATE TABLE "books_book" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "title"
varchar(50) NOT NULL, "author" varchar(100) NOT NULL, "genre" varchar NOT NULL);
COMMIT;
```

```
$ python
manage.py migrate
```

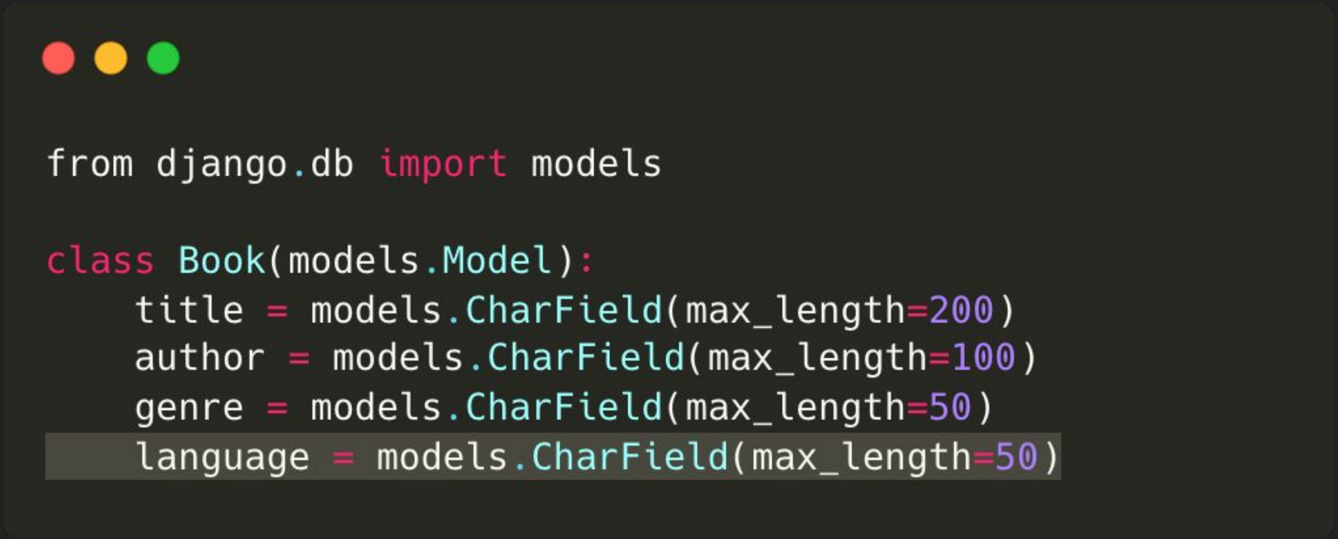


```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, books, contenttypes, sessions
  Applying books.0001_initial... OK
```

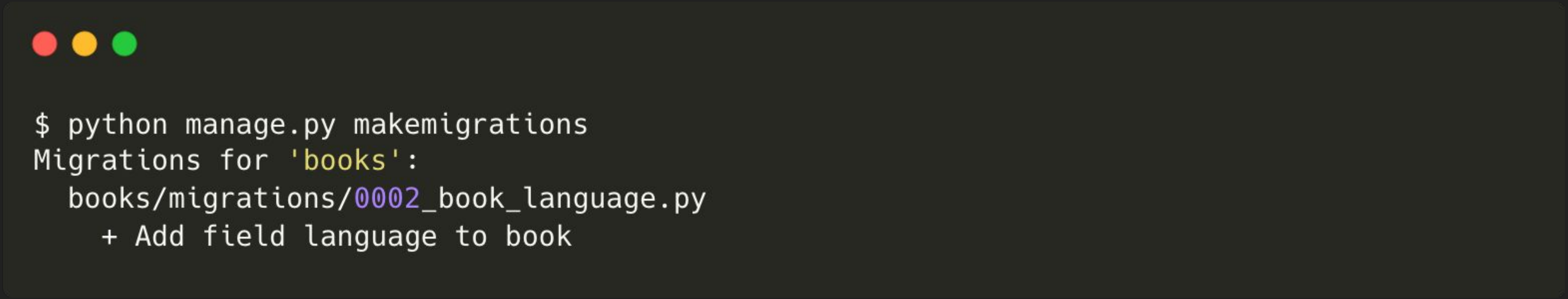
List books in Turkish language 💡

A library 📖

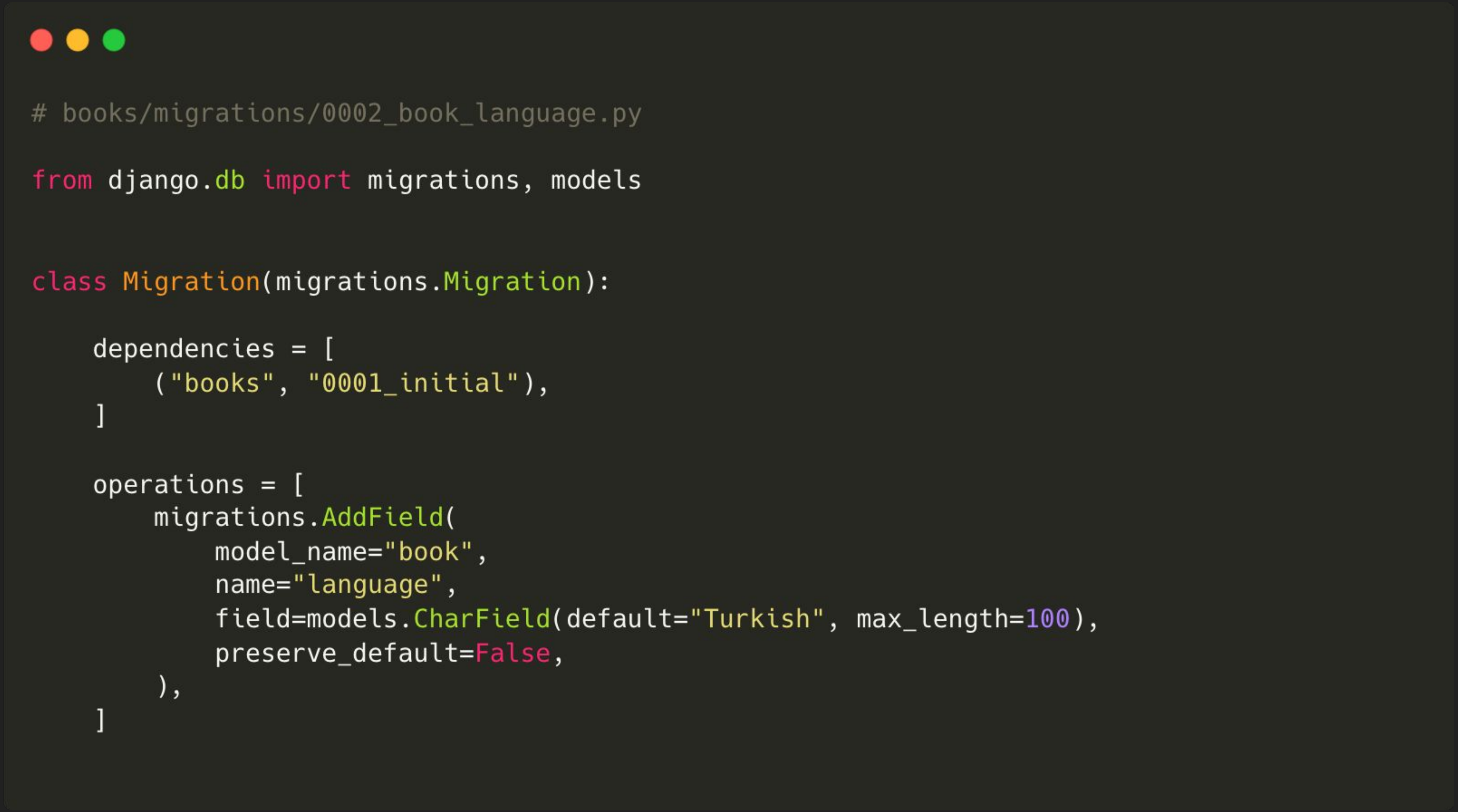
wish. I would like to list books in Turkish language.



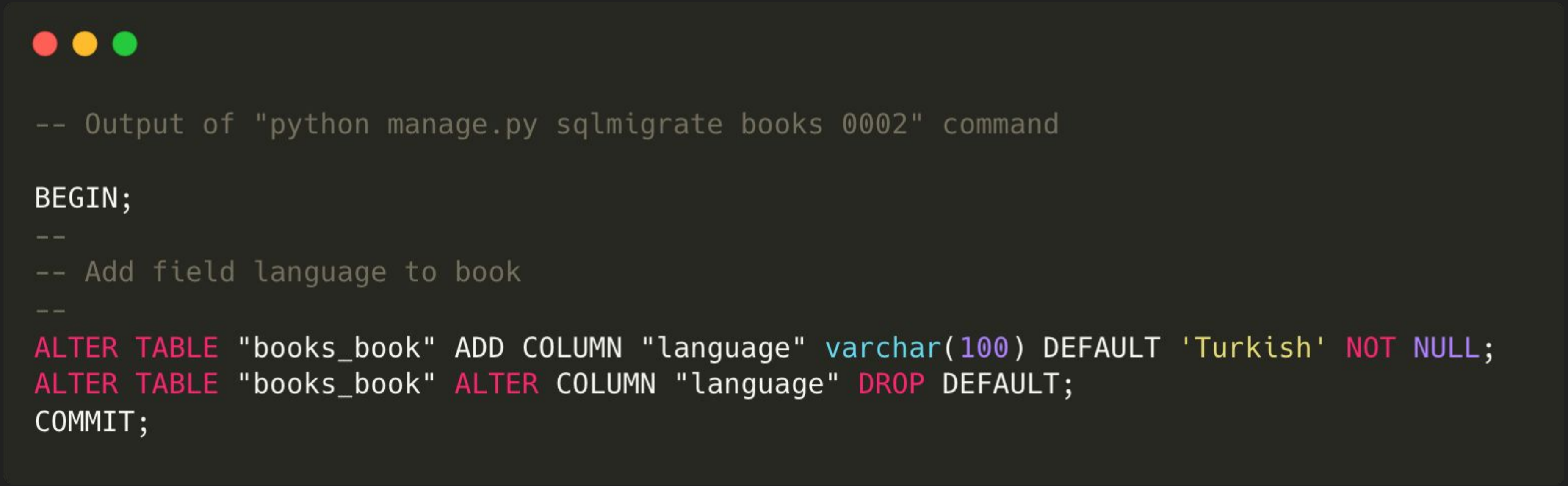
```python
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    genre = models.CharField(max_length=50)
    language = models.CharField(max_length=50)
```



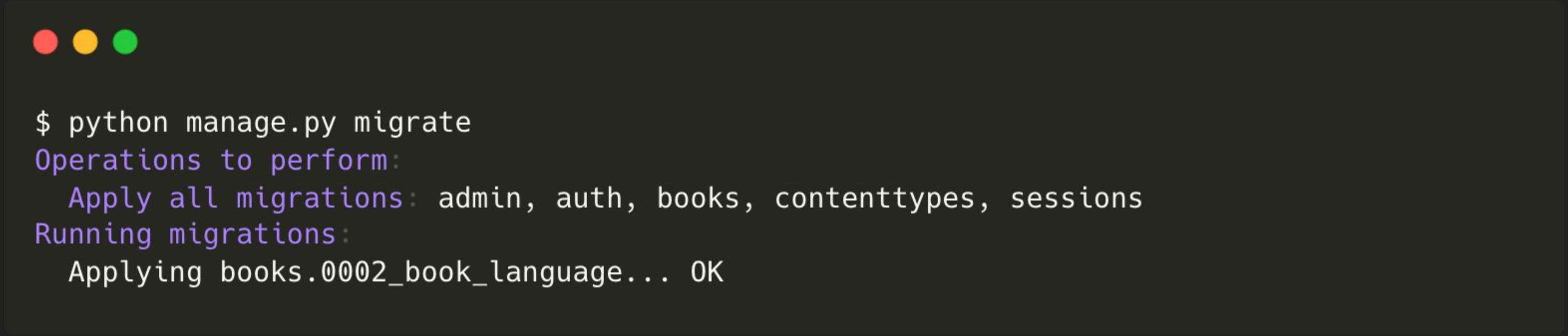
```
$ python manage.py makemigrations
Migrations for 'books':
  books/migrations/0002_book_language.py
    + Add field language to book
```



```python
# books/migrations/0002_book_language.py

from django.db import migrations, models


class Migration(migrations.Migration):

    dependencies = [
        ("books", "0001_initial"),
    ]

    operations = [
        migrations.AddField(
            model_name="book",
            name="language",
            field=models.CharField(default="Turkish", max_length=100),
            preserve_default=False,
        ),
    ]
```

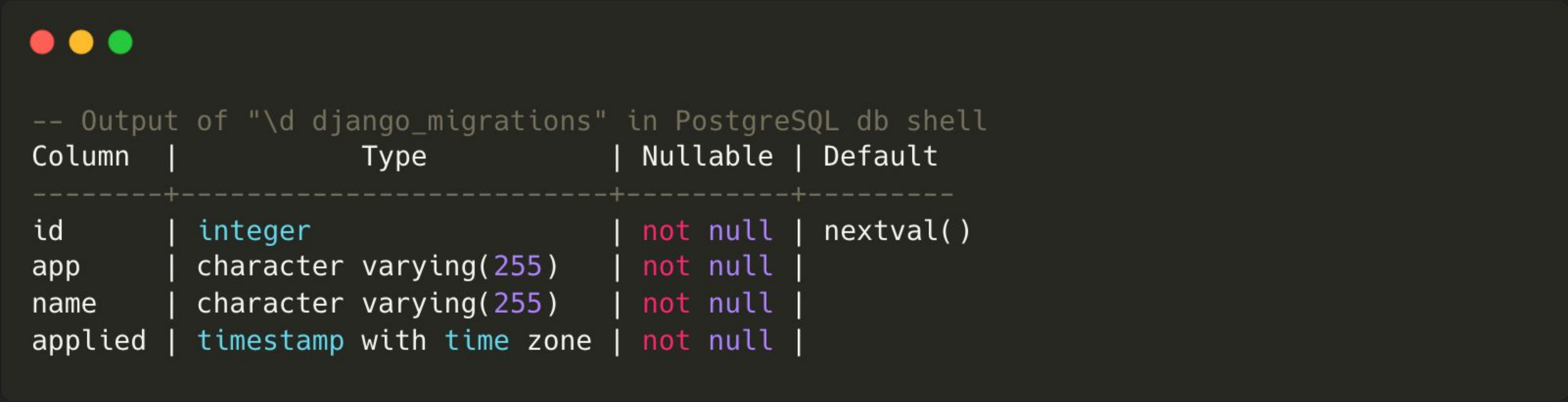


```sql
-- Output of "python manage.py sqlmigrate books 0002" command

BEGIN;
--
-- Add field language to book
--
ALTER TABLE "books_book" ADD COLUMN "language" varchar(100) DEFAULT 'Turkish' NOT NULL;
ALTER TABLE "books_book" ALTER COLUMN "language" DROP DEFAULT;
COMMIT;
```

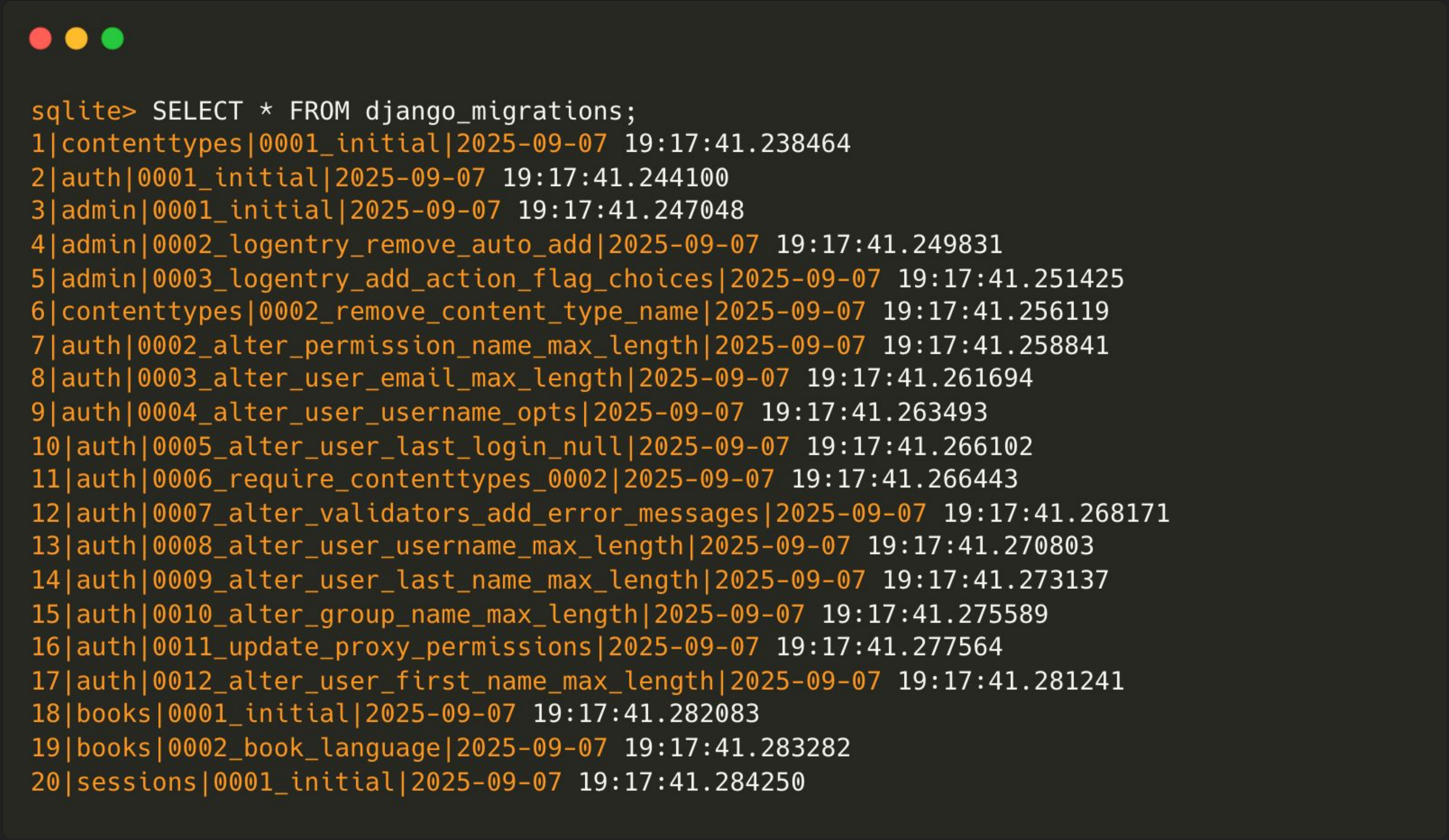


```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, books, contenttypes, sessions
Running migrations:
  Applying books.0002_book_language... OK
```

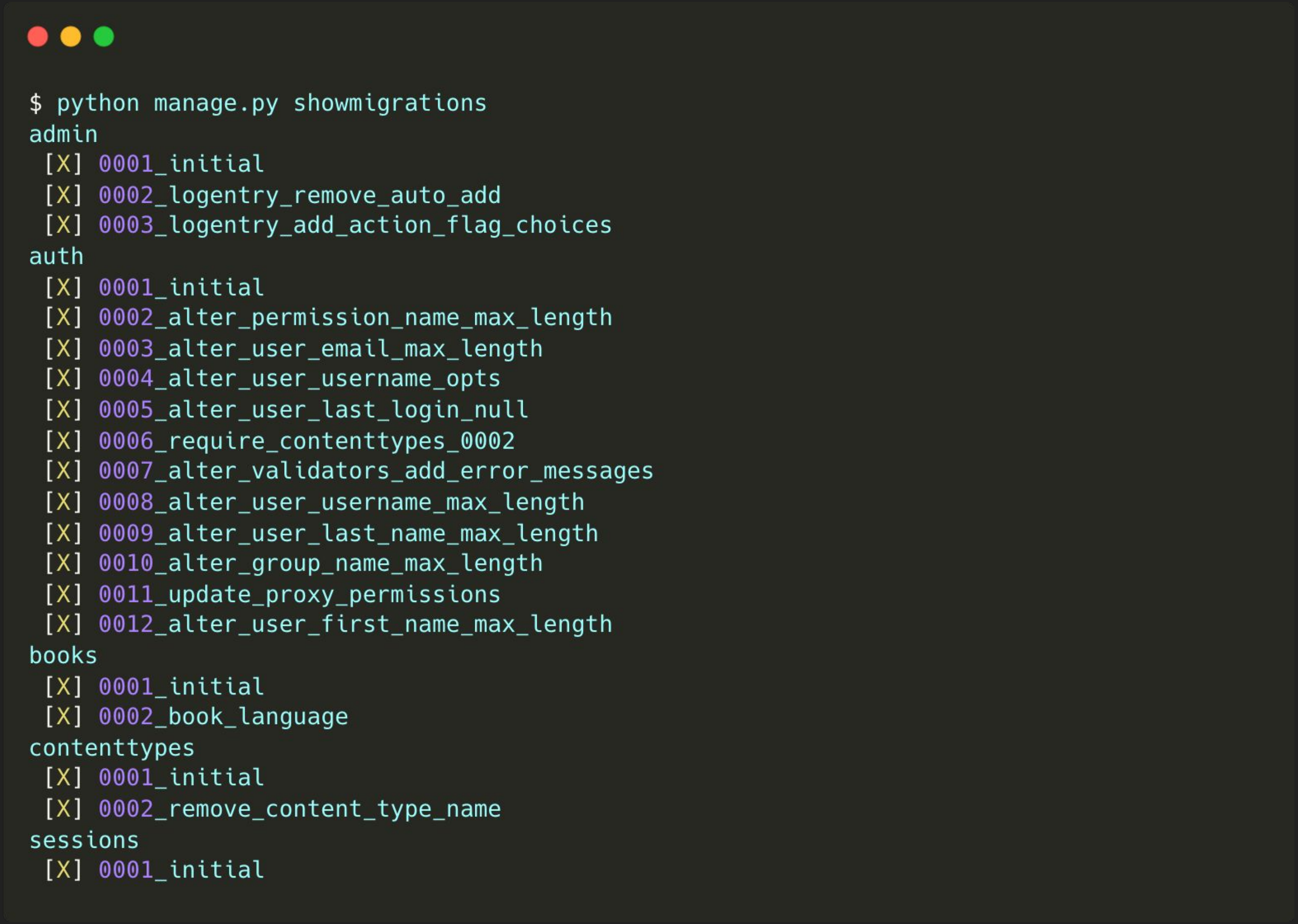
How does Django know to apply only the 0002 migration? 🤔



```
-- Output of "\d django_migrations" in PostgreSQL db shell
Column  |           Type           | Nullable | Default
--------+--------------------------+----------+---------
id      | integer                  | not null | nextval()
app     | character varying(255)   | not null |
name    | character varying(255)   | not null |
applied | timestamp with time zone | not null |
```

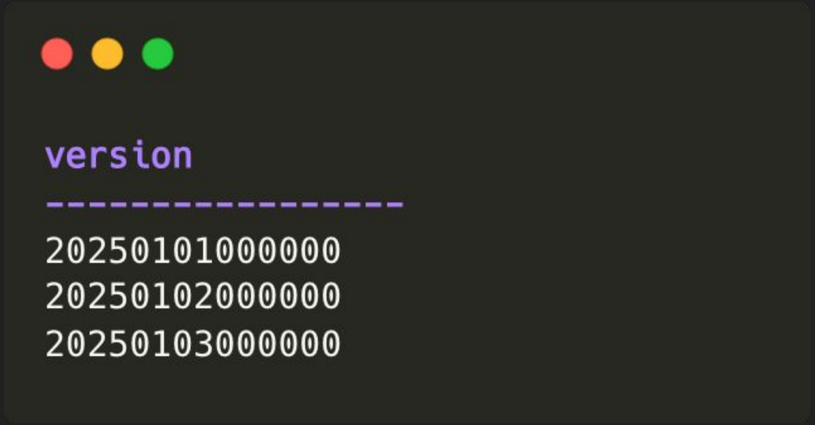


```
sqlite> SELECT * FROM django_migrations;
1|contenttypes|0001_initial|2025-09-07 19:17:41.238464
2|auth|0001_initial|2025-09-07 19:17:41.244100
3|admin|0001_initial|2025-09-07 19:17:41.247048
4|admin|0002_logentry_remove_auto_add|2025-09-07 19:17:41.249831
5|admin|0003_logentry_add_action_flag_choices|2025-09-07 19:17:41.251425
6|contenttypes|0002_remove_content_type_name|2025-09-07 19:17:41.256119
7|auth|0002_alter_permission_name_max_length|2025-09-07 19:17:41.258841
8|auth|0003_alter_user_email_max_length|2025-09-07 19:17:41.261694
9|auth|0004_alter_user_username_opts|2025-09-07 19:17:41.263493
10|auth|0005_alter_user_last_login_null|2025-09-07 19:17:41.266102
11|auth|0006_require_contenttypes_0002|2025-09-07 19:17:41.266443
12|auth|0007_alter_validators_add_error_messages|2025-09-07 19:17:41.268171
13|auth|0008_alter_user_username_max_length|2025-09-07 19:17:41.270803
14|auth|0009_alter_user_last_name_max_length|2025-09-07 19:17:41.273137
15|auth|0010_alter_group_name_max_length|2025-09-07 19:17:41.275589
16|auth|0011_update_proxy_permissions|2025-09-07 19:17:41.277564
17|auth|0012_alter_user_first_name_max_length|2025-09-07 19:17:41.281241
18|books|0001_initial|2025-09-07 19:17:41.282083
19|books|0002_book_language|2025-09-07 19:17:41.283282
20|sessions|0001_initial|2025-09-07 19:17:41.284250
```



```
$ python manage.py showmigrations
admin
 [X] 0001_initial
 [X] 0002_logentry_remove_auto_add
 [X] 0003_logentry_add_action_flag_choices
auth
 [X] 0001_initial
 [X] 0002_alter_permission_name_max_length
 [X] 0003_alter_user_email_max_length
 [X] 0004_alter_user_username_opts
 [X] 0005_alter_user_last_login_null
 [X] 0006_require_contenttypes_0002
 [X] 0007_alter_validators_add_error_messages
 [X] 0008_alter_user_username_max_length
 [X] 0009_alter_user_last_name_max_length
 [X] 0010_alter_group_name_max_length
 [X] 0011_update_proxy_permissions
 [X] 0012_alter_user_first_name_max_length
books
 [X] 0001_initial
 [X] 0002_book_language
contenttypes
 [X] 0001_initial
 [X] 0002_remove_content_type_name
sessions
 [X] 0001_initial
```

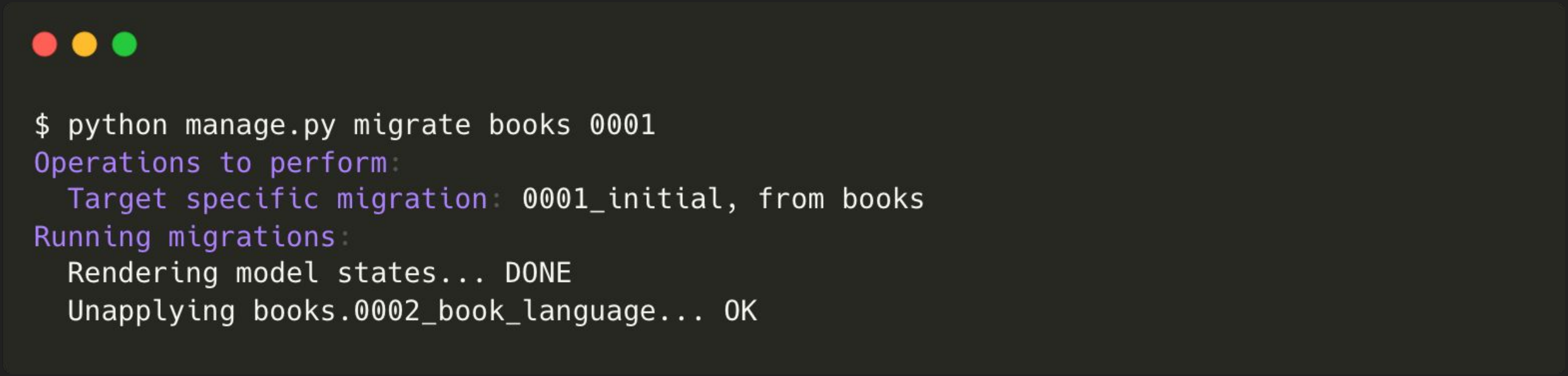
How does rails work?

- Migrations have timestamp as prefix.
    - 20250101000000_create_users.rb
- Keeps in “schema_migrations” table.
- 

```
version
-----------------
20250101000000
20250102000000
20250103000000
```

```
$ python manage.py
migrate books 0001
```

(to revert a migration)



```
$ python manage.py migrate books 0001
Operations to perform:
  Target specific migration: 0001_initial, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0002_book_language... OK
```

Django Migrations Terminology

Django models

describe the structure of the data we keep (like title, author for books).

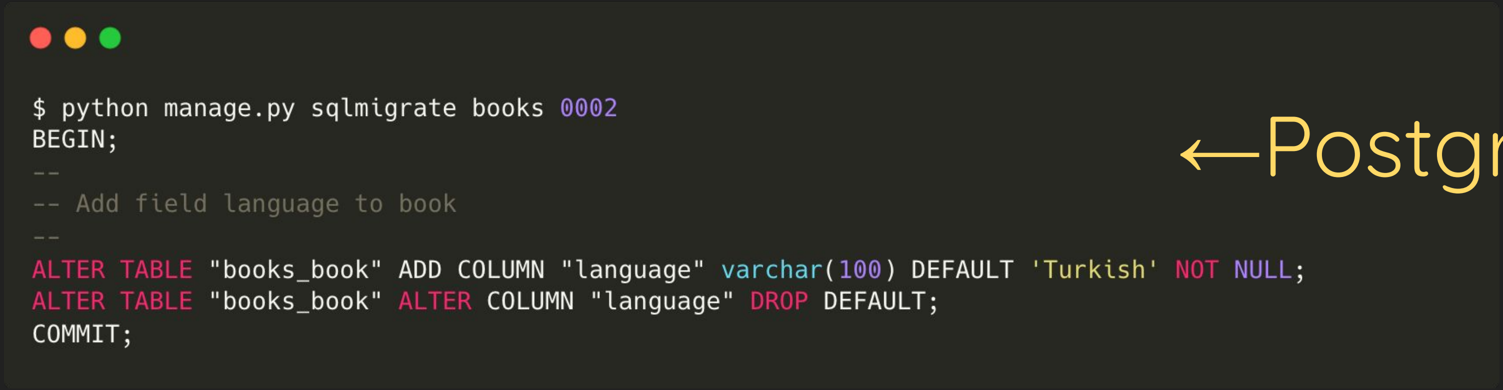
Django migration files

act as a “to-do list” for the database: what needs to be created, updated, or removed. (like add language to books)

Django Migrations Command Cheat Sheet

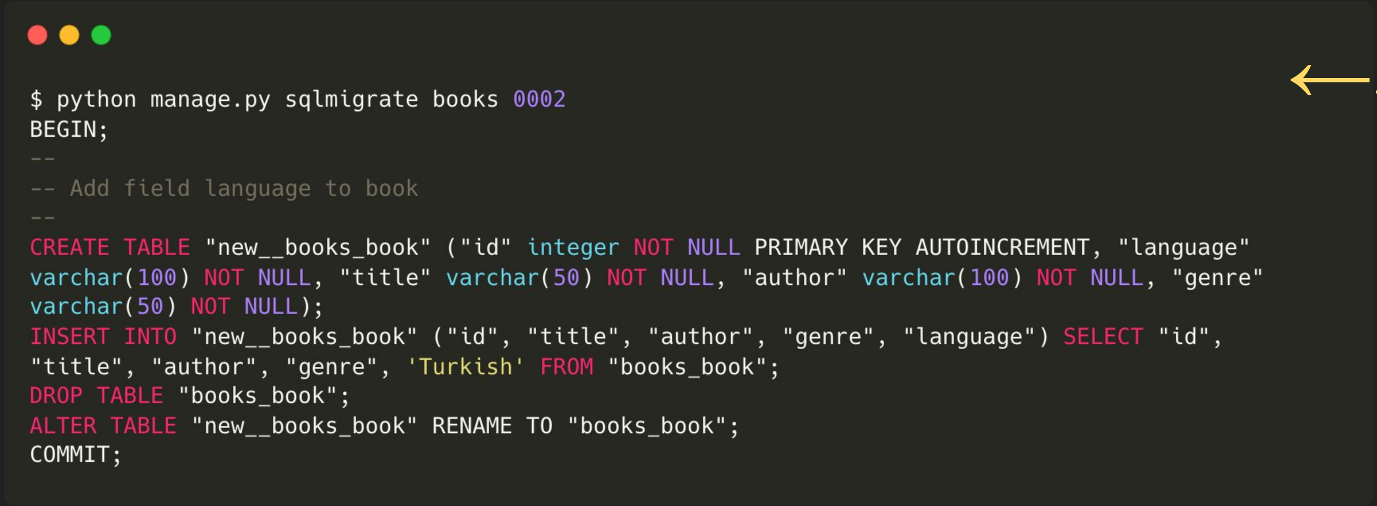
- **manage.py makemigrations** (after updating models.py)
- **manage.py showmigrations** (to see which migrations are applied in the database)
- **manage.py migrate** (to apply migration files in the database)
- **manage.py sqlmigrate** (to see SQL equivalence of migration file)

Quiz:
Why are they different?



```
$ python manage.py sqlmigrate books 0002
BEGIN;
--
-- Add field language to book
--
ALTER TABLE "books_book" ADD COLUMN "language" varchar(100) DEFAULT 'Turkish' NOT NULL;
ALTER TABLE "books_book" ALTER COLUMN "language" DROP DEFAULT;
COMMIT;
```

←PostgreSQL

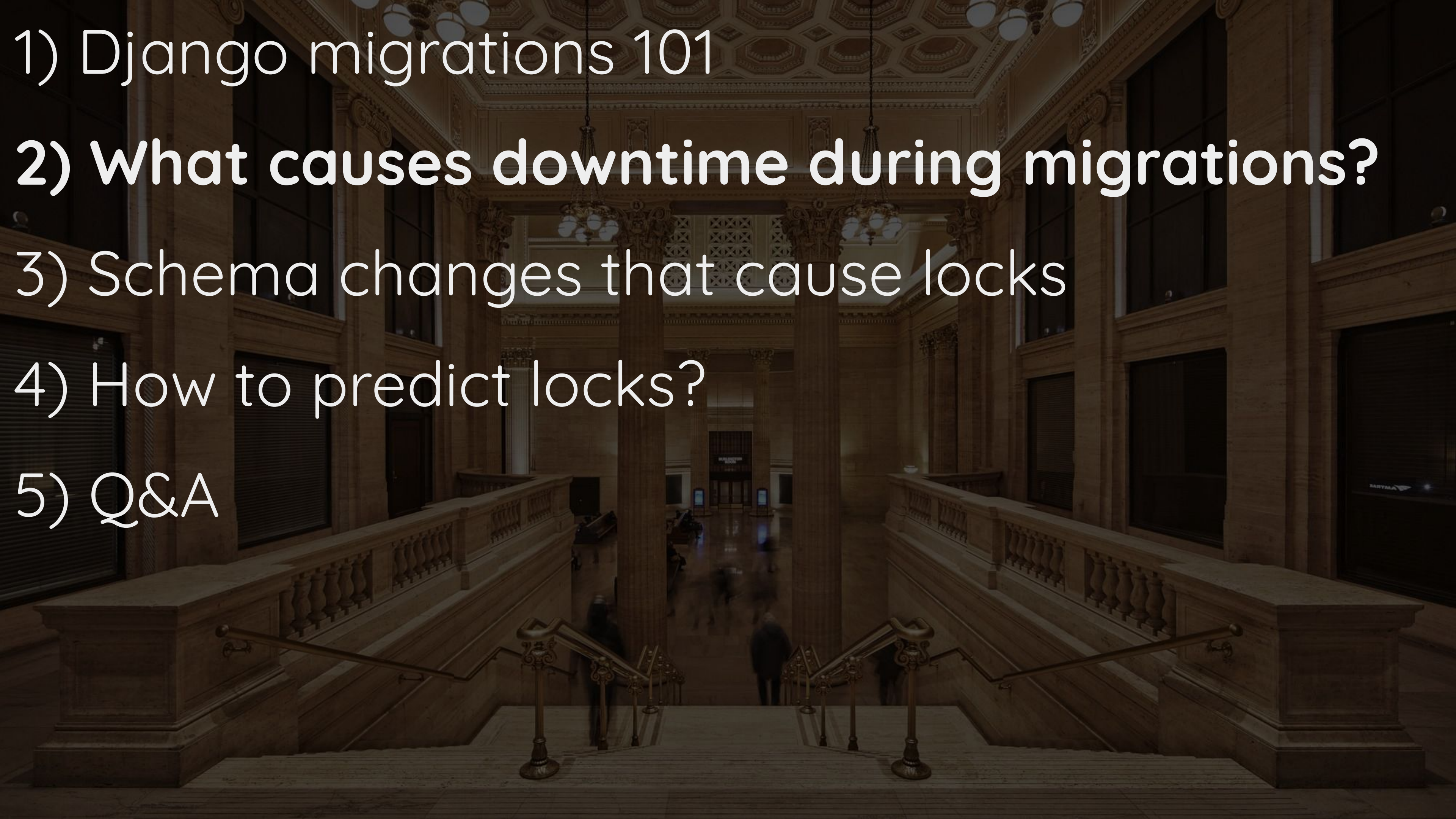


```
$ python manage.py sqlmigrate books 0002
BEGIN;
--
-- Add field language to book
--
CREATE TABLE "new__books_book" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "language"
varchar(100) NOT NULL, "title" varchar(50) NOT NULL, "author" varchar(100) NOT NULL, "genre"
varchar(50) NOT NULL);
INSERT INTO "new__books_book" ("id", "title", "author", "genre", "language") SELECT "id",
"title", "author", "genre", 'Turkish' FROM "books_book";
DROP TABLE "books_book";
ALTER TABLE "new__books_book" RENAME TO "books_book";
COMMIT;
```

←SQLite

SQL of the same migration may vary by database.

(PostgreSQL, SQLite etc.)



1) Django migrations 101

2) What causes downtime during migrations?

3) Schema changes that cause locks

4) How to predict locks?

5) Q&A

😀 😂 🥰 🙂 😐 🙁 😢 😡 😱 😴

- Downtime
- Incident
- Uptime
- Outage
- Availability
- 500
- …

Downtime in Deployment 📖

noun. any period where end users cannot successfully complete their intended actions in the application.

We are interested in the **downtimes during the deployment**.

Reasons of downtime in deployment

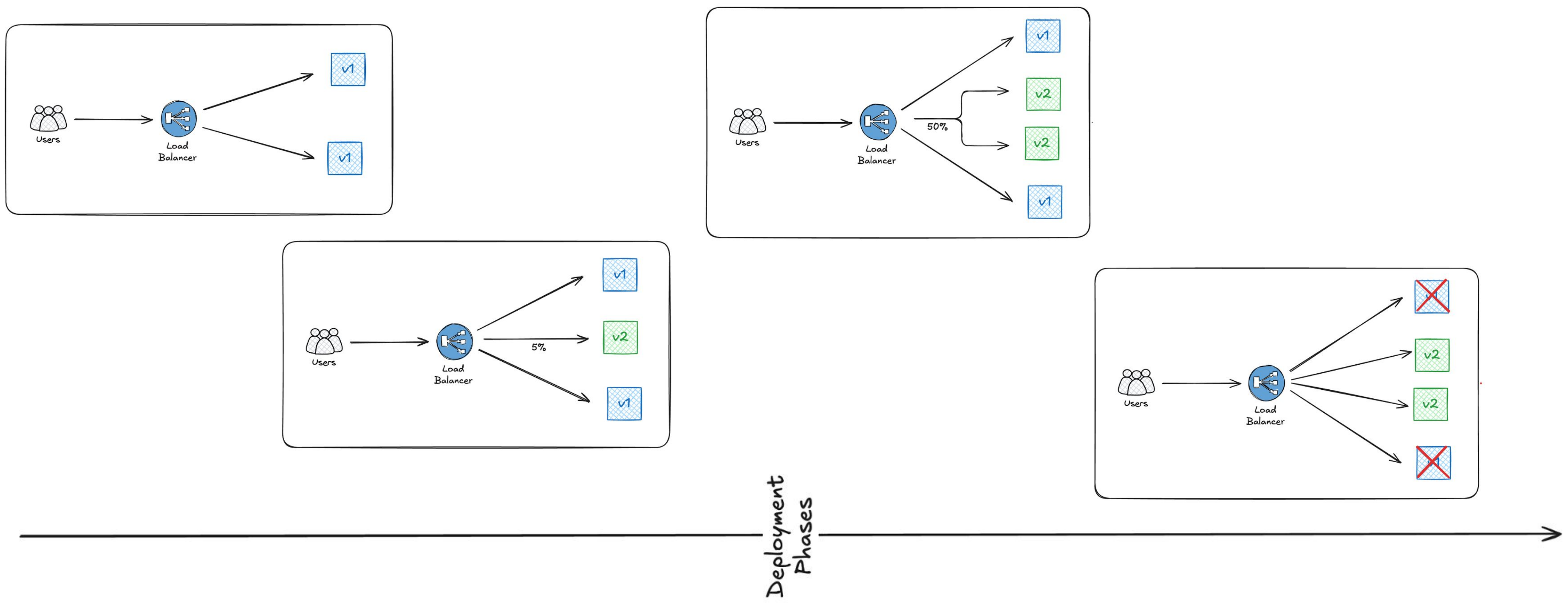
Reason 1:

Schema version mismatch in rolling deployment

Reason 2:

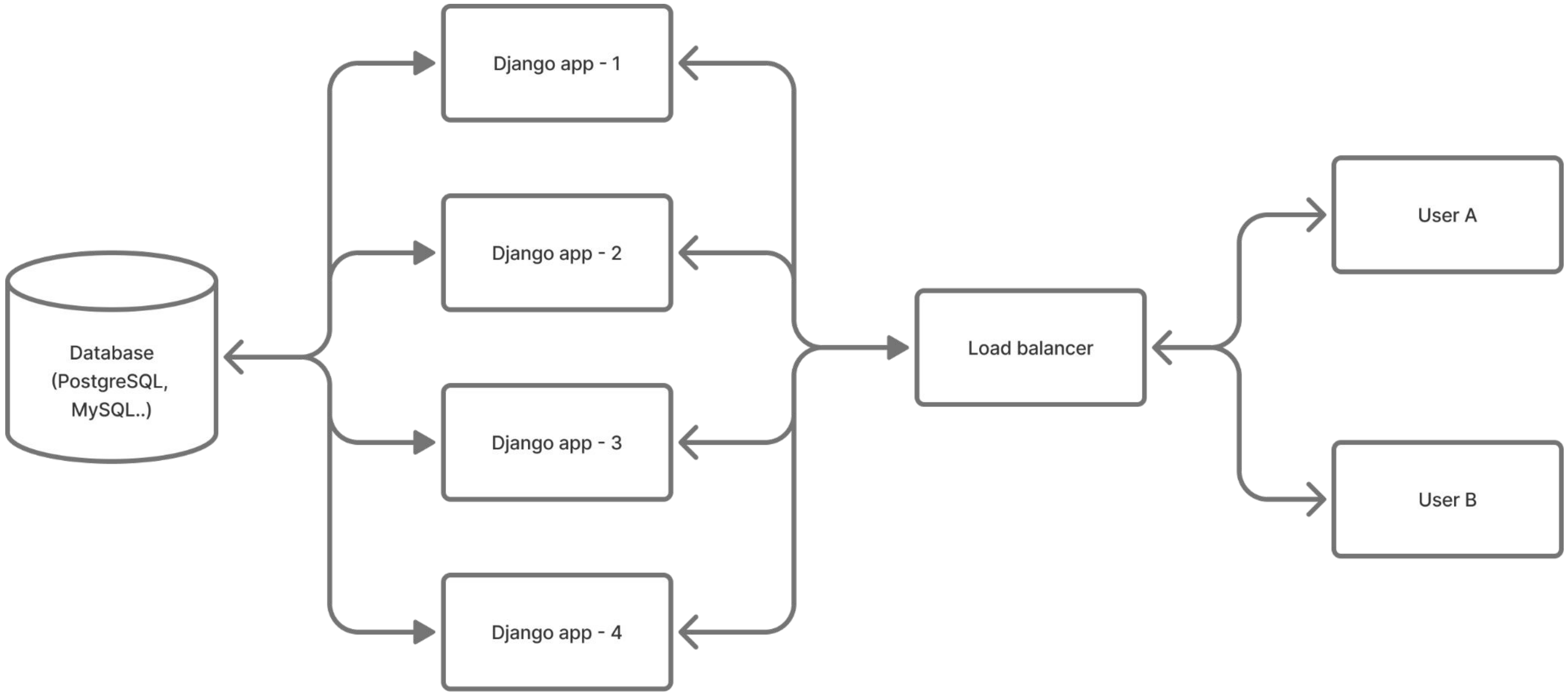
A “locking” migration that takes a long time

An example rolling deployment



*https://www.ykameshrao.com/post/software-deployment-strategies-part-2-stateless-services

One database and **multiple app servers**



*my poor diagram

1: manage.py migrate

- The migration adds the "language" column to Book.

- But while the deploy is in progress, not all app instances are updated yet.

2: Old app instances with old schema

- They may still create new Book objects without setting language.

- If the column is already NOT NULL, those inserts will fail. (book.language must be provided.)

- If it’s nullable, they may insert rows with NULL values that the new app doesn’t expect.

3: New app instances

- They expect every Book to have a language (e.g. "Turkish").

- But they may read rows created by old instances that have NULL, leading to errors or unexpected behavior.

4: Mismatch window

- During the rollout, there’s a period where some app instances speak the “old schema contract” and others speak the “new contract”.

- This creates the temporary incompatibility between application and database.

i) Schema version mismatch in rolling deployment

Schema Change vs. Old App Instances

The new migration adds a **language** column to **Book**. Old app instances, still running during deployment, **may insert new rows without setting language**. If the column is NOT NULL, those inserts will fail; if it's nullable, they may insert NULL values that the updated code does not expect.

New App Instances vs. Existing Data

New app instances assume **every Book has a valid language**. However, during the deployment mismatch window, they may encounter NULL values from old app writes, causing errors or unexpected behavior.

ii) A migration that takes a long time

Brief steps of deployment:

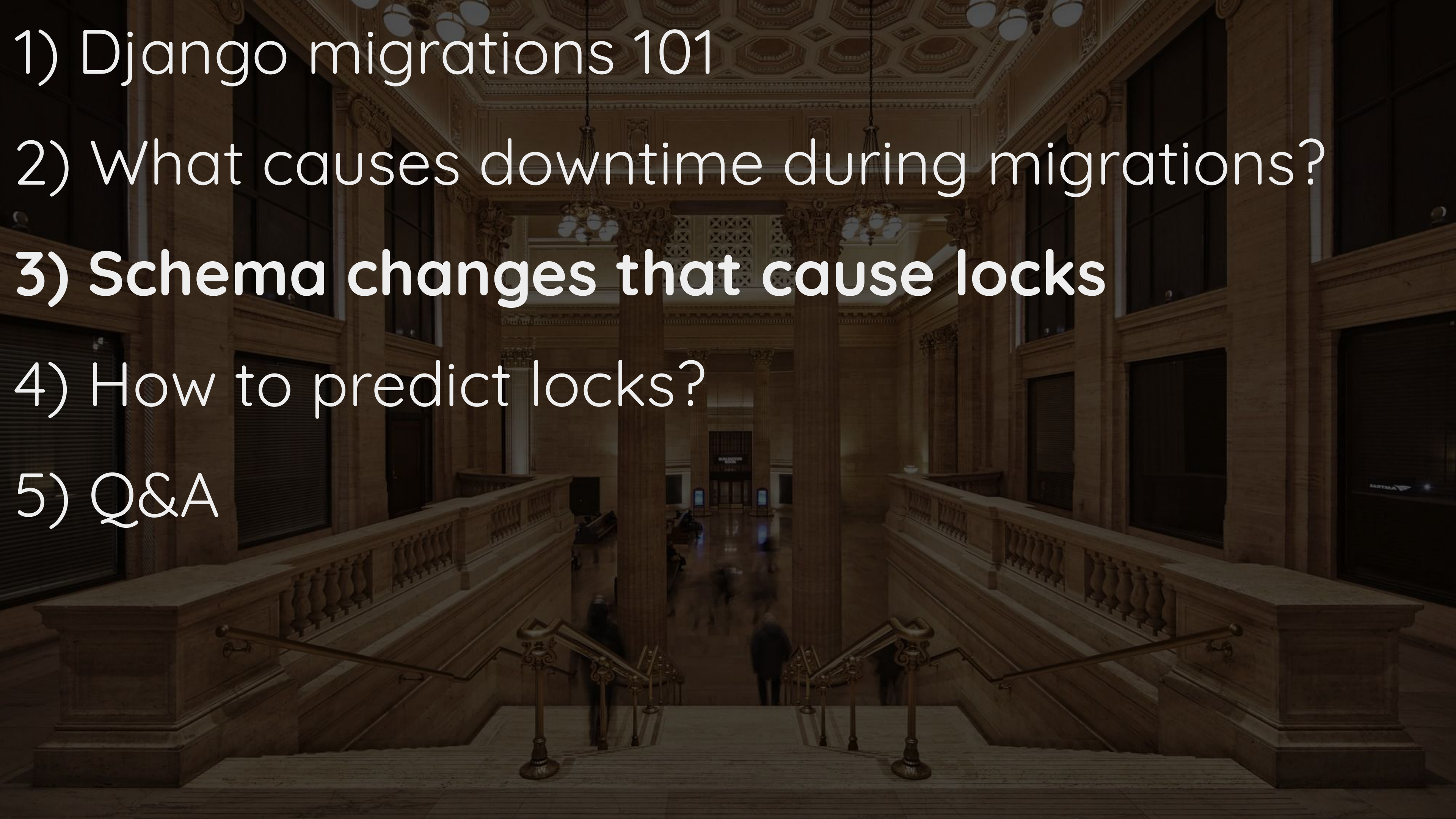
1) Detach first app from load balancer.
2) **manage.py migrate (312 seconds..)**
3) Deploying first app
4) Attach first app to load balancer.
5) Detach nth app from load balancer.
6) Deploying nth app
7) Attach nth app to load balancer.
8) Deployment is completed.

Apps cannot read or write to the database table.

Reasons of downtime in deployment

i) Schema version mismatch in rolling deployment

ii) A “locking” migration that takes a long time



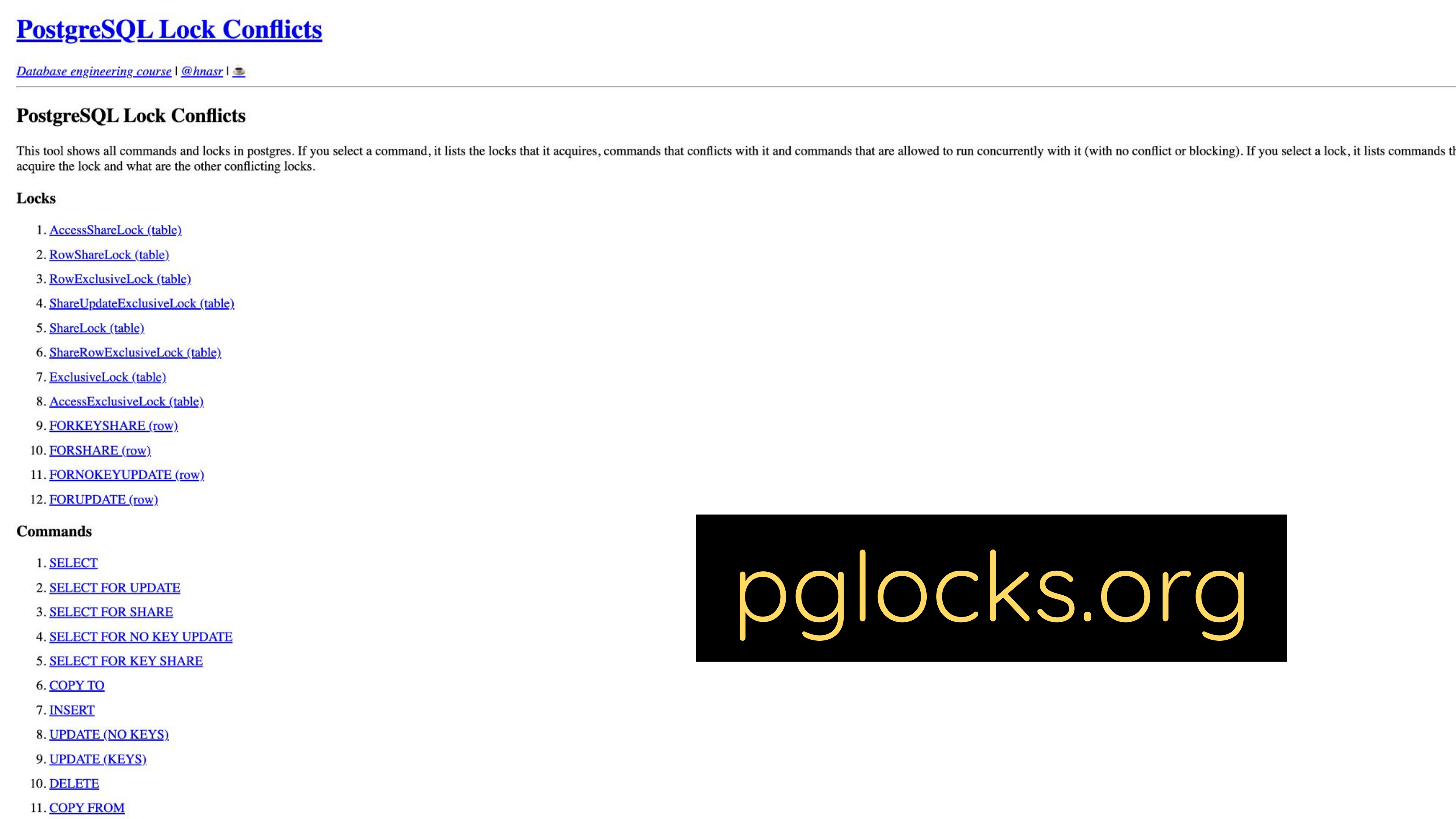
1) Django migrations 101

2) What causes downtime during migrations?

3) Schema changes that cause locks

4) How to predict locks?

5) Q&A



PostgreSQL Lock Conflicts

Database engineering course | *@hnasr* | ☕

PostgreSQL Lock Conflicts

This tool shows all commands and locks in postgres. If you select a command, it lists the locks that it acquires, commands that conflicts with it and commands that are allowed to run concurrently with it (with no conflict or blocking). If you select a lock, it lists commands th
acquire the lock and what are the other conflicting locks.

Locks

1. AccessShareLock (table)
2. RowShareLock (table)
3. RowExclusiveLock (table)
4. ShareUpdateExclusiveLock (table)
5. ShareLock (table)
6. ShareRowExclusiveLock (table)
7. ExclusiveLock (table)
8. AccessExclusiveLock (table)
9. FORKEYSHARE (row)
10. FORSHARE (row)
11. FORNOKEYUPDATE (row)
12. FORUPDATE (row)

Commands

1. SELECT
2. SELECT FOR UPDATE
3. SELECT FOR SHARE
4. SELECT FOR NO KEY UPDATE
5. SELECT FOR KEY SHARE
6. COPY TO
7. INSERT
8. UPDATE (NO KEYS)
9. UPDATE (KEYS)
10. DELETE
11. COPY FROM

pglocks.org

PostgreSQL Lock Conflicts

Database engineering course | *@hnasr* | ☕

SELECT

The SELECT command acquires AccessShareLock table lock . Following are the locks SELECT acquires, the commands that are allowed to run concurrently with its lock AccessShareLock and the commands that conflict with it. The list also includes the conflicting row lo
applicable

Locks acquired by SELECT

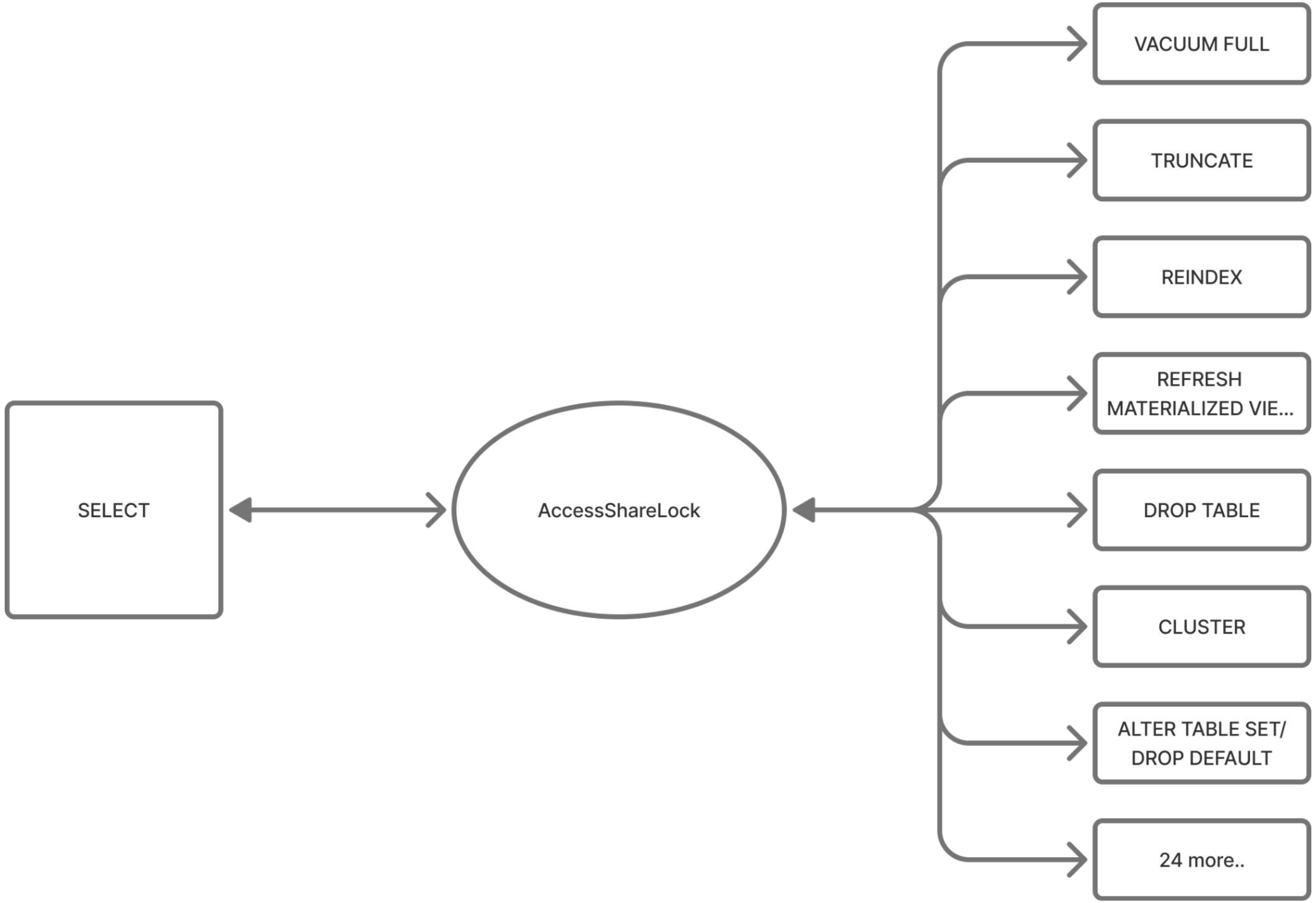
1. AccessShareLock (table)

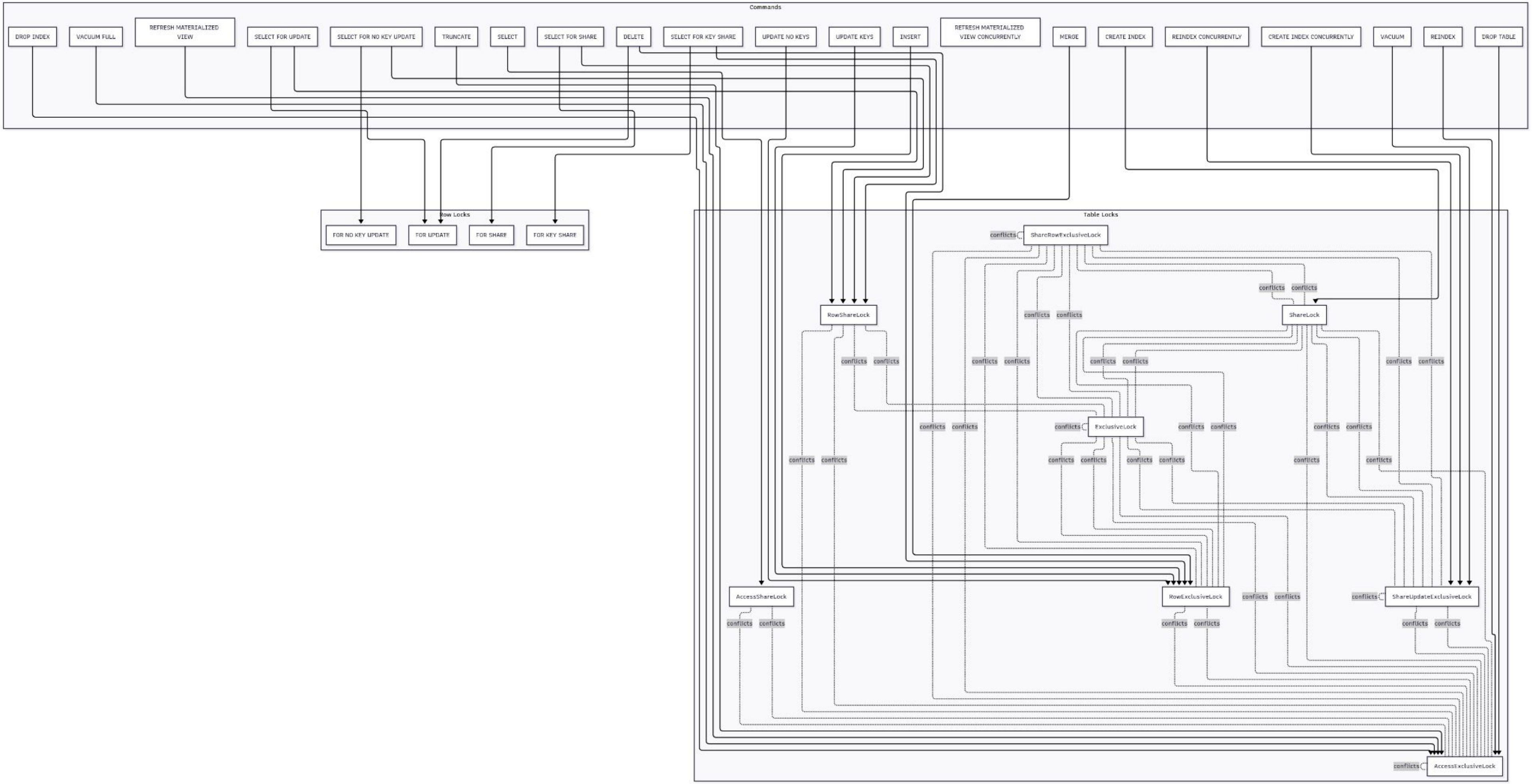
Commands concurrently allowed on the table with SELECT

e.g. If tx1 does a SELECT on the table then tx2 is allowed to do any of the following commands concurrently on the same table without being blocked. Some DMLs executed on the same rows may block, read more below.

1. SELECT
2. SELECT FOR UPDATE
3. SELECT FOR SHARE
4. SELECT FOR NO KEY UPDATE
5. SELECT FOR KEY SHARE
6. COPY TO
7. INSERT
8. UPDATE (NO KEYS)
9. UPDATE (KEYS)
10. DELETE
11. COPY FROM
12. MERGE
13. VACUUM
14. REINDEX CONCURRENTLY
15. REFRESH MATERIALIZED VIEW CONCURRENTLY
16. DROP INDEX CONCURRENTLY
17. CREATE TRIGGER
18. CREATE STATISTICS
19. CREATE INDEX
20. CREATE INDEX CONCURRENTLY
21. COMMENT ON

pglocks.org


SELECT
AccessShareLock
VACUUM FULL
TRUNCATE
REINDEX
REFRESH MATERIALIZED VIE...
DROP TABLE
CLUSTER
ALTER TABLE SET/ DROP DEFAULT
24 more..


Commands
DROP INDEX
VACUUM FULL
REFRESH MATERIALIZED VIEW
SELECT FOR UPDATE
SELECT FOR NO KEY UPDATE
TRUNCATE
SELECT
SELECT FOR SHARE
DELETE
SELECT FOR KEY SHARE
UPDATE NO KEYS
UPDATE KEYS
INSERT
REFRESH MATERIALIZED VIEW CONCURRENTLY
MERGE
CREATE INDEX
REINDEX CONCURRENTLY
CREATE INDEX CONCURRENTLY
VACUUM
REINDEX
DROP TABLE
Row Locks
FOR NO KEY UPDATE
FOR UPDATE
FOR SHARE
FOR KEY SHARE
Table Locks
ShareRowExclusiveLock
RowShareLock
ShareLock
ExclusiveLock
AccessShareLock
RowExclusiveLock
ShareUpdateExclusiveLock
AccessExclusiveLock
conflicts

Anatomy of Table-Level Locks in PostgreSQL

This blog explains locking mechanisms in PostgreSQL, focusing on table-level locks that are required by Data Definition Language (DDL) operations.

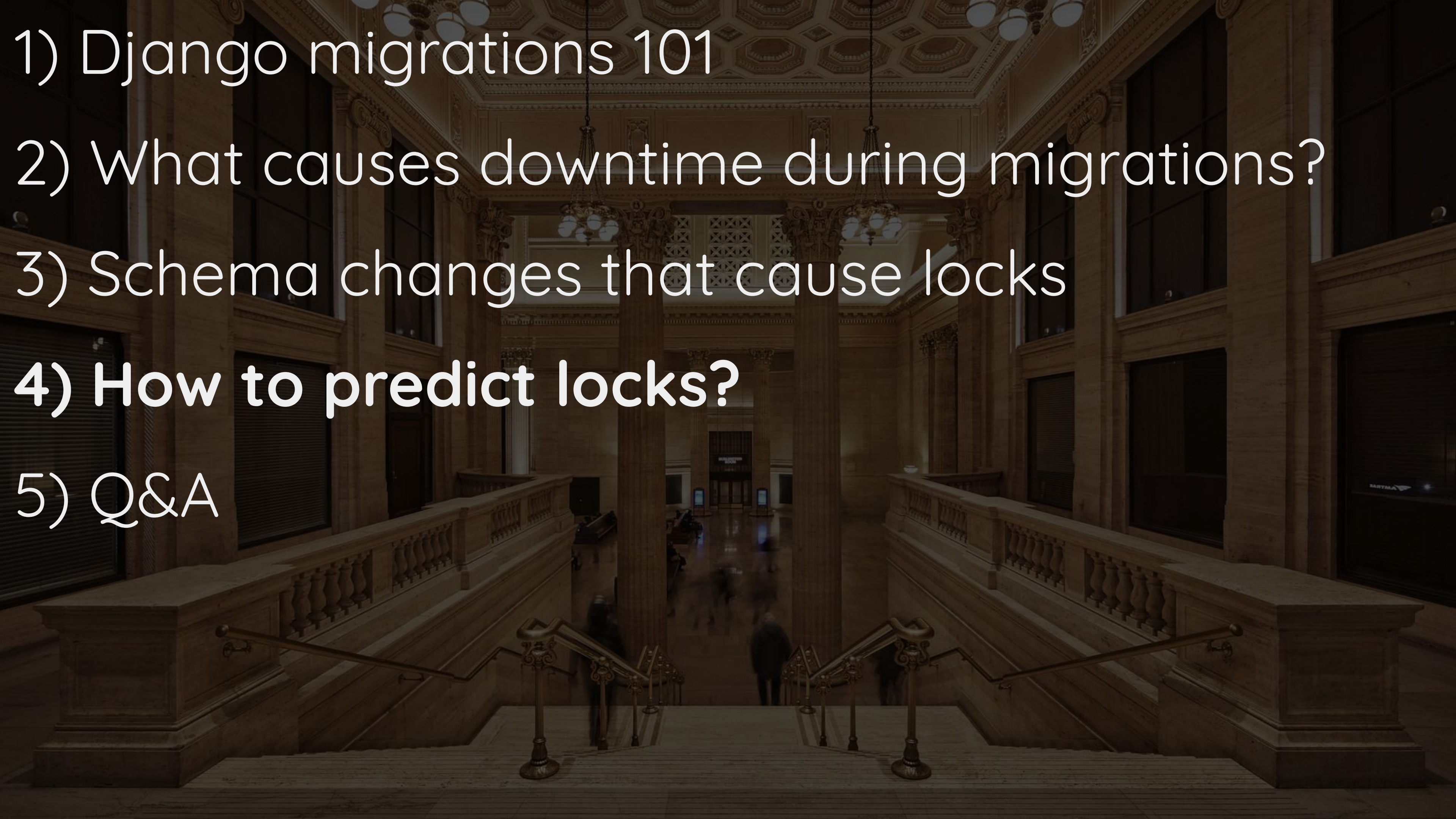
Author
Gulcin Yildirim Jelinek

Date published
Jan 13, 2025

Art of locking or unlocking?

It is common to think about database locks by drawing analogies to physical locks, which might even lead you to order books on the history of locks, Persian locks, and lock-picking techniques. Probably most of us learn by going deeper into the term "locking" to understand a concept in PostgreSQL that doesn't have much to do with physical locks at all, which are primarily about **security**. Postgres locks though are all about **concurrency** and controlling which transaction can hold a lock while another transaction can do its thing, ideally without ever blocking each other. But as we know, no world is ideal, whether it's a door lock or an `AccessShareLock`.

However, now that I've bought those said books about locks, I think I should be allowed to draw some parallels between, let's say, the art of lock-picking and database locking mechanisms. There is one thing I will start with: to be able to pick a lock, any type of lock, you need to have a deep understanding of its inner workings; how the pins, tumbler, and mechanisms interact. By manipulating them, you can find the correct position to unlock the door or safe without a key! In the same way, to be able to manage database locks, you need to understand the internal workings of a database, and mainly how concurrency works in Postgres.



1) Django migrations 101

2) What causes downtime during migrations?

3) Schema changes that cause locks

4) How to predict locks?

5) Q&A

1. Adding a nullable field creates “long” lock?

(True/False)

1. Adding a nullable field creates “long” lock.

False.

2. Adding an index creates “long” lock?

(True/False)

2. Adding an index creates “long” lock?

- True, CreateIndex
- False, CreateIndexConcurrently

3. Adding a non-nullable field with default value creates “long” lock?

(True/False)

E.23.3.2. Base Backup And Streaming Replication

- Replicate `TRUNCATE` activity when using logical replication (Simon Riggs, Marco Nenciarini, Peter Eisentraut)
- Pass prepared transaction information to logical replication subscribers (Nikhil Sontakke, Stas Kelvich)
- Exclude unlogged tables, temporary tables, and `pg_internal.init` files from streaming base backups (David Steele)

  There is no need to copy such files.
- Allow checksums of heap pages to be verified during streaming base backup (Michael Banck)
- Allow replication slots to be advanced programmatically, rather than be consumed by subscribers (Petr Jelinek)

  This allows efficient advancement of replication slots when the contents do not need to be consumed. This is performed by `pg_replication_slot_advance()`.
- Add timeline information to the `backup_label` file (Michael Paquier)

  Also add a check that the WAL timeline matches the `backup_label` file's timeline.
- Add host and port connection information to the `pg_stat_wal_receiver` system view (Haribabu Kommi)

E.23.3.3. Utility Commands

- Allow `ALTER TABLE` to add a column with a non-null default without doing a table rewrite (Andrew Dunstan, Serge Rielau)

  This is enabled when the default value is a constant.
- Allow views to be locked by locking the underlying tables (Yugo Nagata)
- Allow `ALTER INDEX` to set statistics-gathering targets for expression indexes (Alexander Korotkov, Adrien Nayrat)

  In psql, `\d+` now shows the statistics target for indexes.
- Allow multiple tables to be specified in one `VACUUM` or `ANALYZE` command (Nathan Bossart)

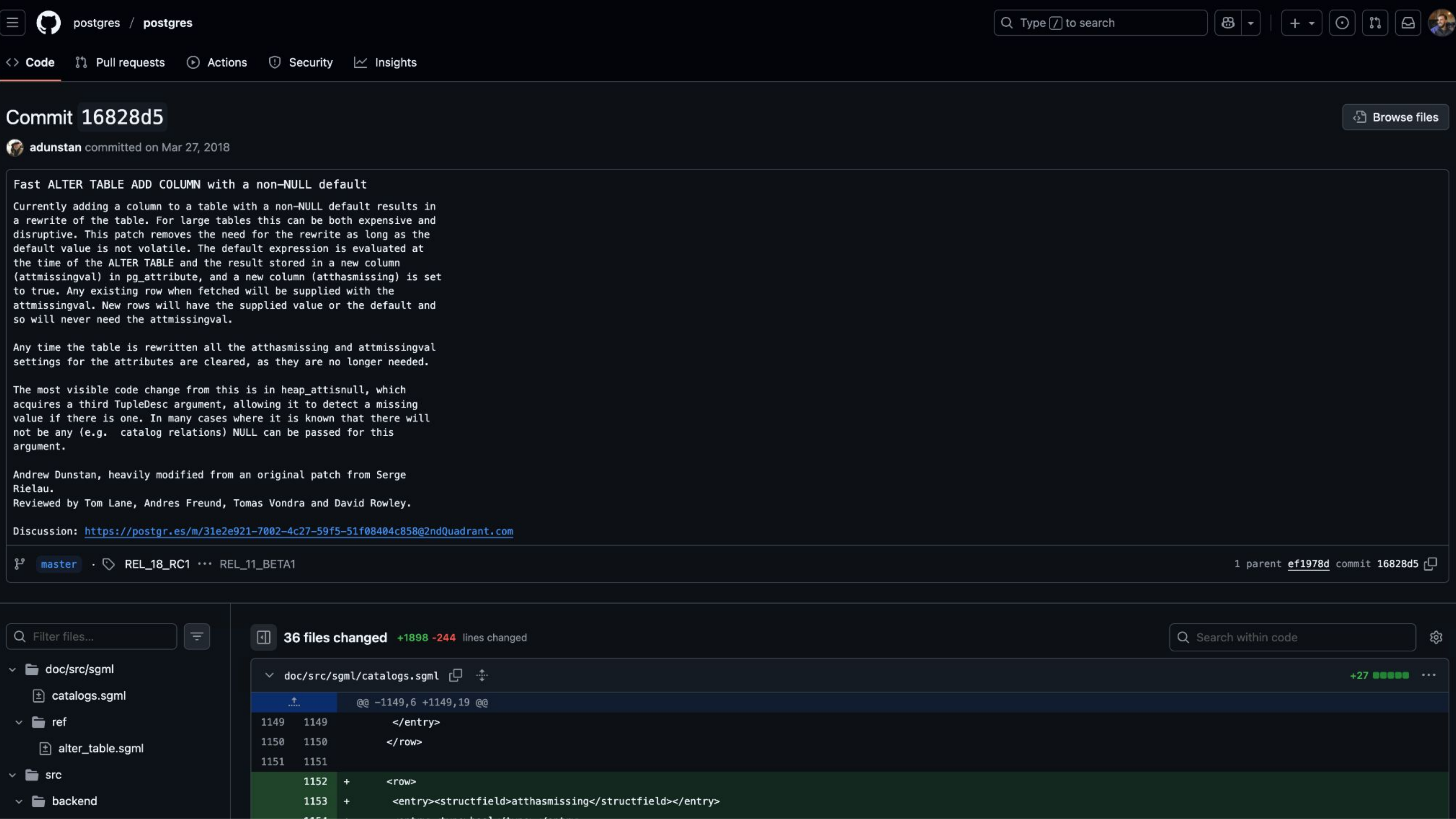
  Also, if any table mentioned in `VACUUM` uses a column list, then the `ANALYZE` keyword must be supplied; previously, `ANALYZE` was implied in such cases.
- Add parenthesized options syntax to `ANALYZE` (Nathan Bossart)

  This is similar to the syntax supported by `VACUUM`.
- Add `CREATE AGGREGATE` option to specify the behavior of the aggregate's finalization function (Tom Lane)

  This is helpful for allowing user-defined aggregate functions to be optimized and to work as window functions.

E.23.3.4. Data Types

- Allow the creation of arrays of domains (Tom Lane)



Commit 16828d5

adunstan committed on Mar 27, 2018

```
Fast ALTER TABLE ADD COLUMN with a non-NULL default

Currently adding a column to a table with a non-NULL default results in
a rewrite of the table. For large tables this can be both expensive and
disruptive. This patch removes the need for the rewrite as long as the
default value is not volatile. The default expression is evaluated at
the time of the ALTER TABLE and the result stored in a new column
(attmissingval) in pg_attribute, and a new column (atthasmissing) is set
to true. Any existing row when fetched will be supplied with the
attmissingval. New rows will have the supplied value or the default and
so will never need the attmissingval.

Any time the table is rewritten all the atthasmissing and attmissingval
settings for the attributes are cleared, as they are no longer needed.

The most visible code change from this is in heap_attisnull, which
acquires a third TupleDesc argument, allowing it to detect a missing
value if there is one. In many cases where it is known that there will
not be any (e.g.  catalog relations) NULL can be passed for this
argument.

Andrew Dunstan, heavily modified from an original patch from Serge
Rielau.
Reviewed by Tom Lane, Andres Freund, Tomas Vondra and David Rowley.

Discussion: https://postgr.es/m/31e2e921-7002-4c27-59f5-51f08404c858@2ndQuadrant.com
```

master · REL_18_RC1 ··· REL_11_BETA1   1 parent ef1978d commit 16828d5

36 files changed +1898 -244 lines changed

doc/src/sgml/catalogs.sgml +27

```
@@ -1149,6 +1149,19 @@
1149 1149       </entry>
1150 1150      </row>
1151 1151
     1152 +    <row>
     1153 +     <entry><structfield>atthasmissing</structfield></entry>
```

3. Adding a non-nullable field with default creates “long” lock?

- True < PostgreSQL 11
- False >= PostgreSQL 11

*if the default value
is constant

(not non-volatile such as
now, gen_random_uuid)

MENTAL
LOAD

“sleeping in the storm”

Lessons from “Aircraft Investigation”

Prevents crashes before they happen

Airplane warning systems have cut crashes by 85% since the 1970s by alerting pilots early.

Saves huge amounts of money

Fixing an airplane problem on the ground costs thousands; fixing it after a crash costs billions.

Lets thousands work together smoothly

Modern aviation safely manages 100,000+ flights daily because every pilot follows the same automated checklists.

NORTHWEST
N309RC

Northwest Airlines Flight 255

Checklist skipped

Pilots forgot to set the flaps and slats—vital for takeoff—because they missed the “taxi” checklist.

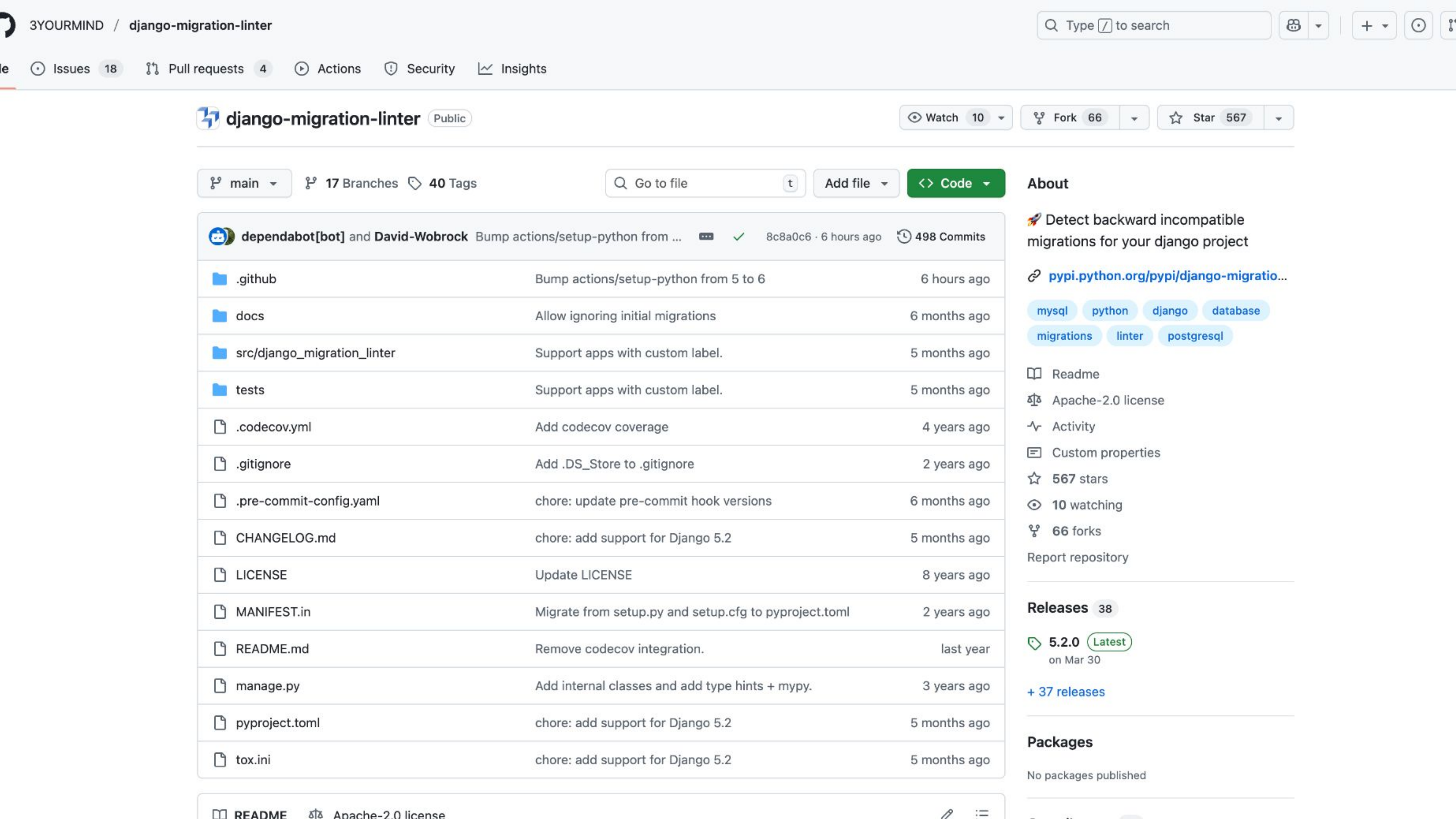
Warning system silent

Normally, an alarm would warn them, but the system had no power and never went off.

Why no power?

Investigators found a key circuit breaker (P-40) was off. It may have been silenced on purpose during taxi to stop a nuisance alarm, but wasn’t reset. It’s unclear if this was deliberate, a malfunction, or an oversight.

sleep in the storm,
no mental load


3YOURMIND / django-migration-linter

Issues 18 · Pull requests 4 · Actions · Security · Insights

django-migration-linter Public

Watch 10 · Fork 66 · Star 567

main · 17 Branches · 40 Tags · Go to file · Add file · Code

dependabot[bot] and David-Wobrock Bump actions/setup-python from ... 8c8a0c6 · 6 hours ago · 498 Commits

.github	Bump actions/setup-python from 5 to 6	6 hours ago
docs	Allow ignoring initial migrations	6 months ago
src/django_migration_linter	Support apps with custom label.	5 months ago
tests	Support apps with custom label.	5 months ago
.codecov.yml	Add codecov coverage	4 years ago
.gitignore	Add .DS_Store to .gitignore	2 years ago
.pre-commit-config.yaml	chore: update pre-commit hook versions	6 months ago
CHANGELOG.md	chore: add support for Django 5.2	5 months ago
LICENSE	Update LICENSE	8 years ago
MANIFEST.in	Migrate from setup.py and setup.cfg to pyproject.toml	2 years ago
README.md	Remove codecov integration.	last year
manage.py	Add internal classes and add type hints + mypy.	3 years ago
pyproject.toml	chore: add support for Django 5.2	5 months ago
tox.ini	chore: add support for Django 5.2	5 months ago

README · Apache-2.0 license

About

🚀 Detect backward incompatible migrations for your django project

pypi.python.org/pypi/django-migratio...

mysql · python · django · database · migrations · linter · postgresql

Readme · Apache-2.0 license · Activity · Custom properties · 567 stars · 10 watching · 66 forks · Report repository

Releases 38

5.2.0 Latest on Mar 30

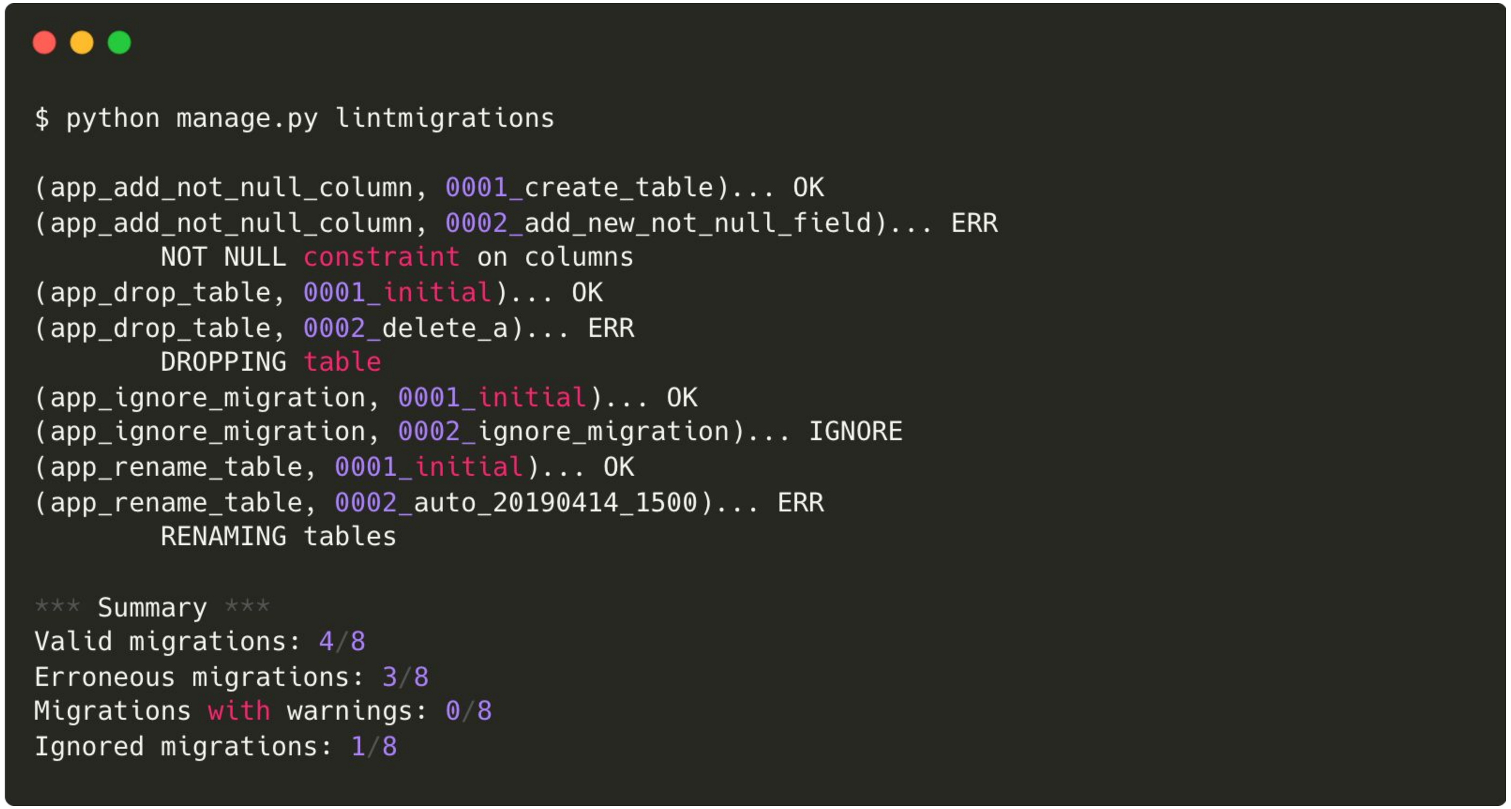
+ 37 releases

Packages

No packages published

```
$ pip install django-migration-linter
```

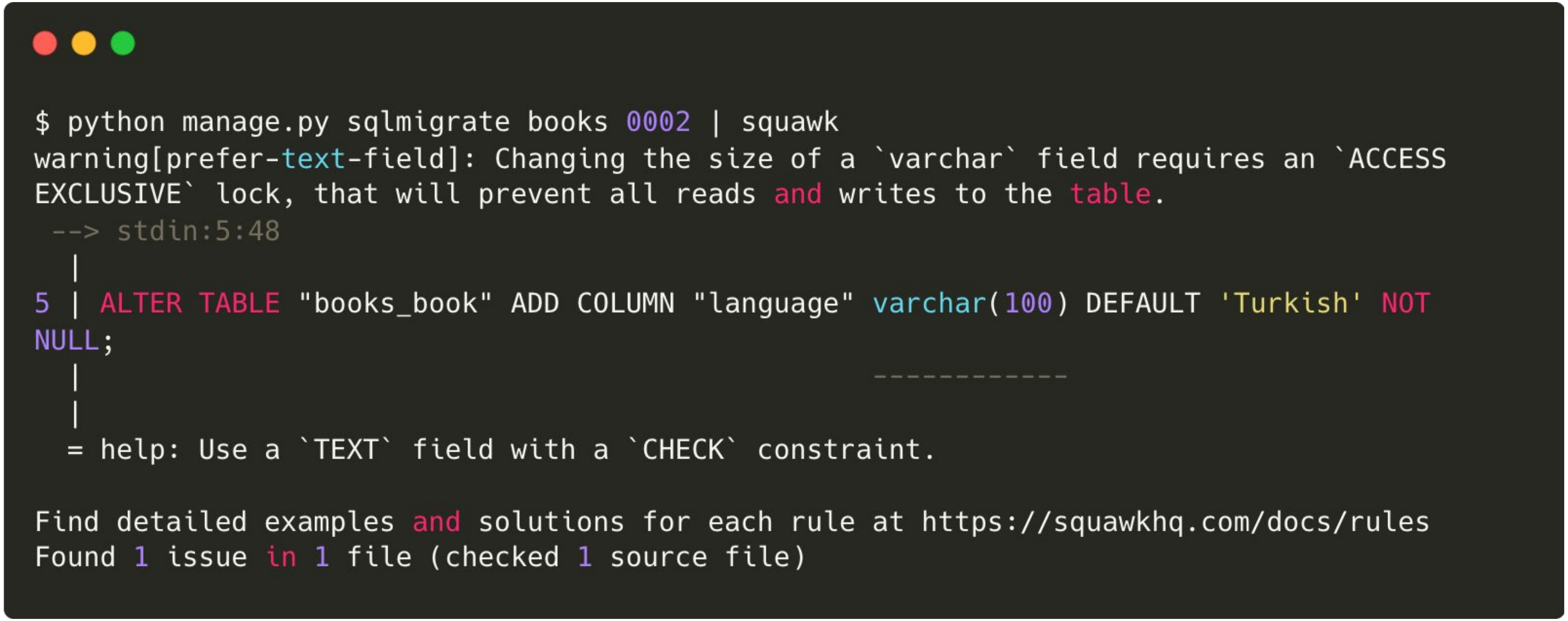
manage.py lintmigrations ✨



```
$ python manage.py lintmigrations

(app_add_not_null_column, 0001_create_table)... OK
(app_add_not_null_column, 0002_add_new_not_null_field)... ERR
        NOT NULL constraint on columns
(app_drop_table, 0001_initial)... OK
(app_drop_table, 0002_delete_a)... ERR
        DROPPING table
(app_ignore_migration, 0001_initial)... OK
(app_ignore_migration, 0002_ignore_migration)... IGNORE
(app_rename_table, 0001_initial)... OK
(app_rename_table, 0002_auto_20190414_1500)... ERR
        RENAMING tables

*** Summary ***
Valid migrations: 4/8
Erroneous migrations: 3/8
Migrations with warnings: 0/8
Ignored migrations: 1/8
```

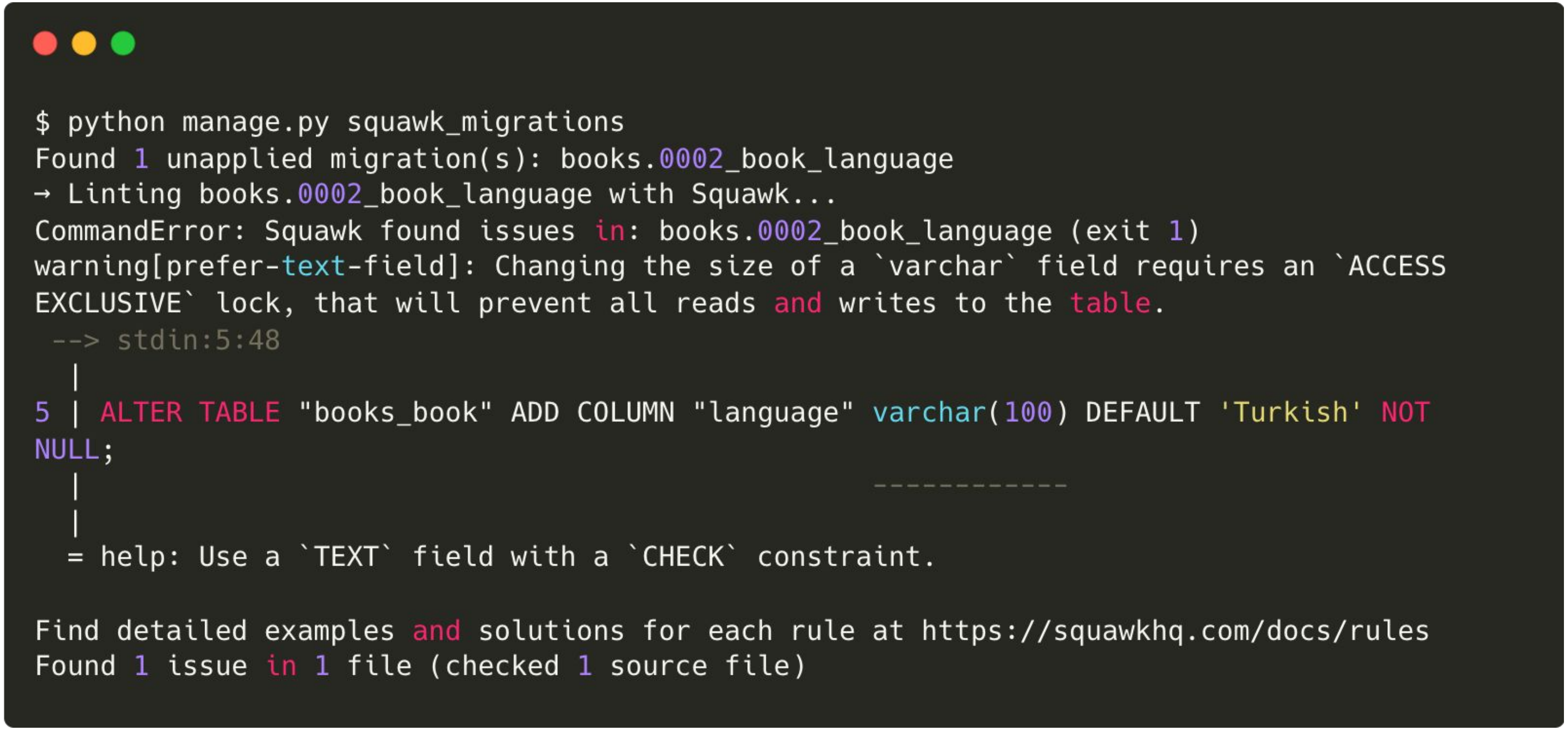
```
$ npm install squawk-cli
```

$ manage.py sqlmigrate books 0002 | squawk



```
$ python manage.py sqlmigrate books 0002 | squawk
warning[prefer-text-field]: Changing the size of a `varchar` field requires an `ACCESS
EXCLUSIVE` lock, that will prevent all reads and writes to the table.
 --> stdin:5:48
  |
5 | ALTER TABLE "books_book" ADD COLUMN "language" varchar(100) DEFAULT 'Turkish' NOT
NULL;
  |                                                ------------
  |
  = help: Use a `TEXT` field with a `CHECK` constraint.

Find detailed examples and solutions for each rule at https://squawkhq.com/docs/rules
Found 1 issue in 1 file (checked 1 source file)
```

$ manage.py squawk_migrations (djangoesque way)



```
$ python manage.py squawk_migrations
Found 1 unapplied migration(s): books.0002_book_language
→ Linting books.0002_book_language with Squawk...
CommandError: Squawk found issues in: books.0002_book_language (exit 1)
warning[prefer-text-field]: Changing the size of a `varchar` field requires an `ACCESS
EXCLUSIVE` lock, that will prevent all reads and writes to the table.
 --> stdin:5:48
  |
5 | ALTER TABLE "books_book" ADD COLUMN "language" varchar(100) DEFAULT 'Turkish' NOT
NULL;
  |                                                    ------------
  |
  = help: Use a `TEXT` field with a `CHECK` constraint.

Find detailed examples and solutions for each rule at https://squawkhq.com/docs/rules
Found 1 issue in 1 file (checked 1 source file)
```

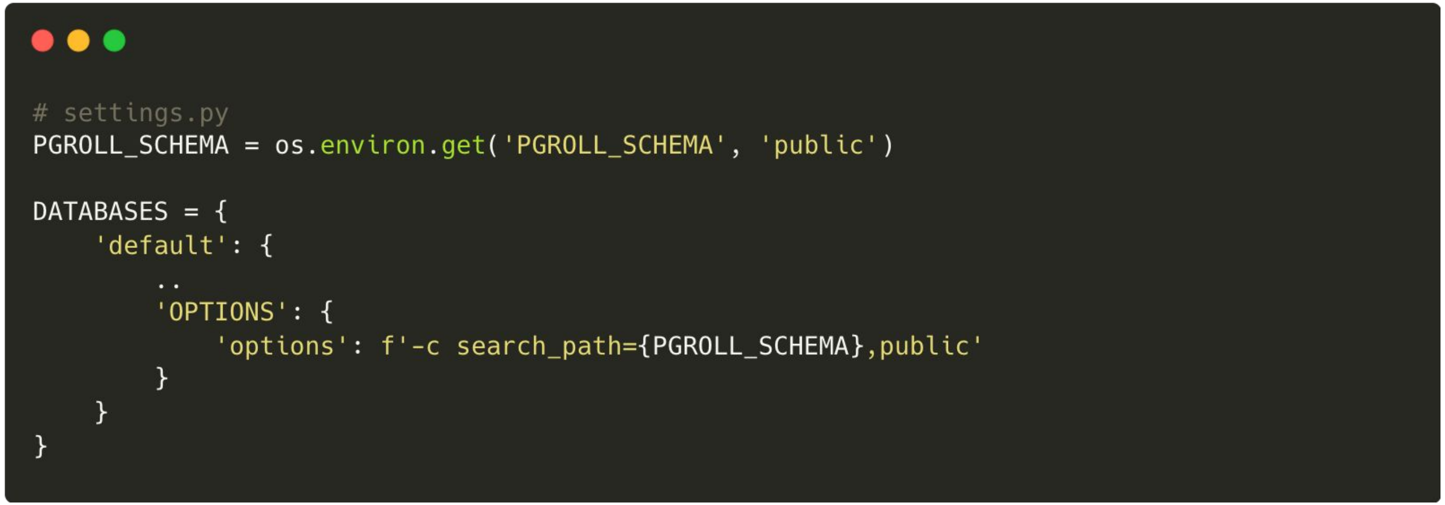
https://tinyurl.com/
squawkmigrations

pgroll

creating temporary shadow columns and views during migrations, allowing both old and new versions of your application to run simultaneously.

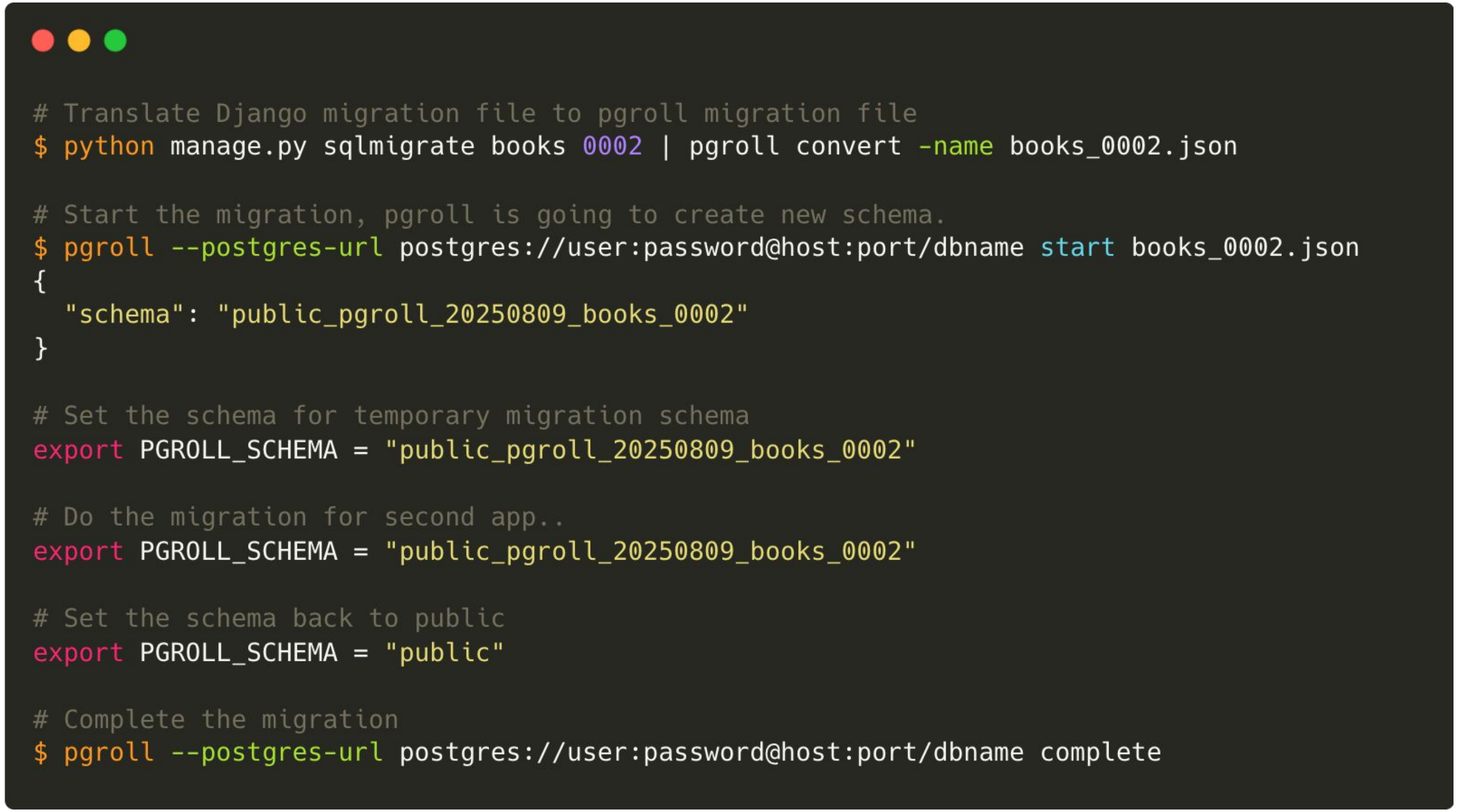
pgroll setup

- pgroll init --postgres-url postgres://user:password@host:port/dbname



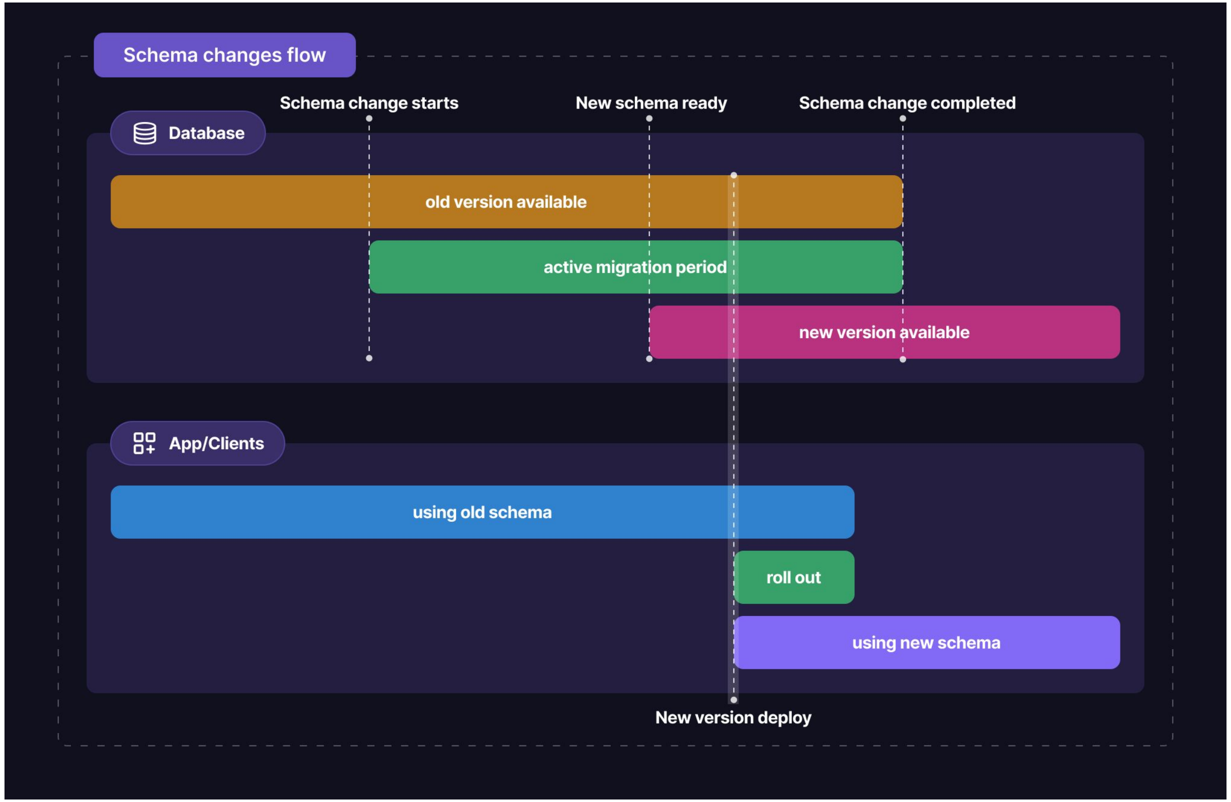
```
# settings.py
PGROLL_SCHEMA = os.environ.get('PGROLL_SCHEMA', 'public')

DATABASES = {
    'default': {
        ..
        'OPTIONS': {
            'options': f'-c search_path={PGROLL_SCHEMA},public'
        }
    }
}
```

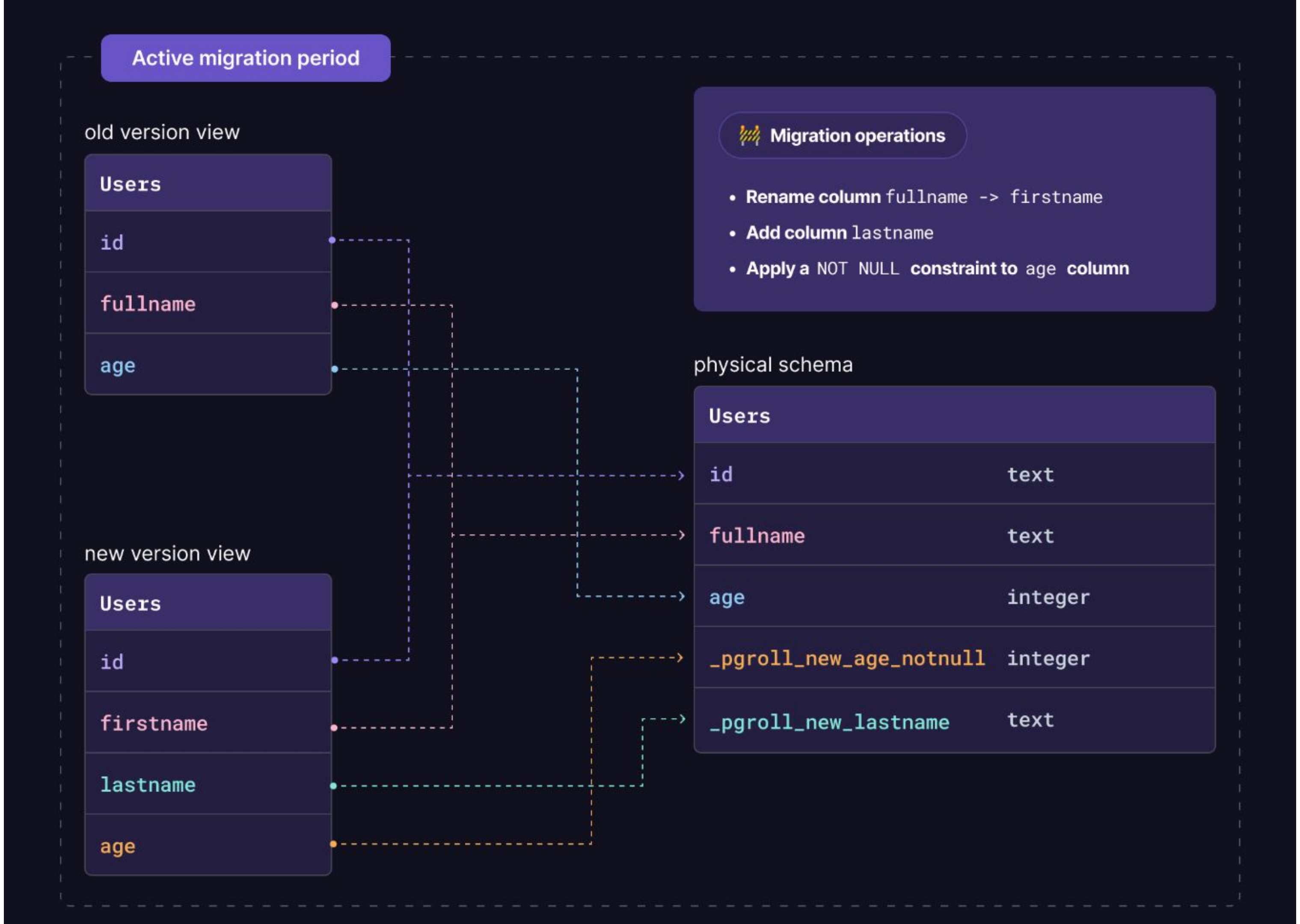
deployment with pgroll



```
# Translate Django migration file to pgroll migration file
$ python manage.py sqlmigrate books 0002 | pgroll convert -name books_0002.json

# Start the migration, pgroll is going to create new schema.
$ pgroll --postgres-url postgres://user:password@host:port/dbname start books_0002.json
{
  "schema": "public_pgroll_20250809_books_0002"
}

# Set the schema for temporary migration schema
export PGROLL_SCHEMA = "public_pgroll_20250809_books_0002"

# Do the migration for second app..
export PGROLL_SCHEMA = "public_pgroll_20250809_books_0002"

# Set the schema back to public
export PGROLL_SCHEMA = "public"

# Complete the migration
$ pgroll --postgres-url postgres://user:password@host:port/dbname complete
```

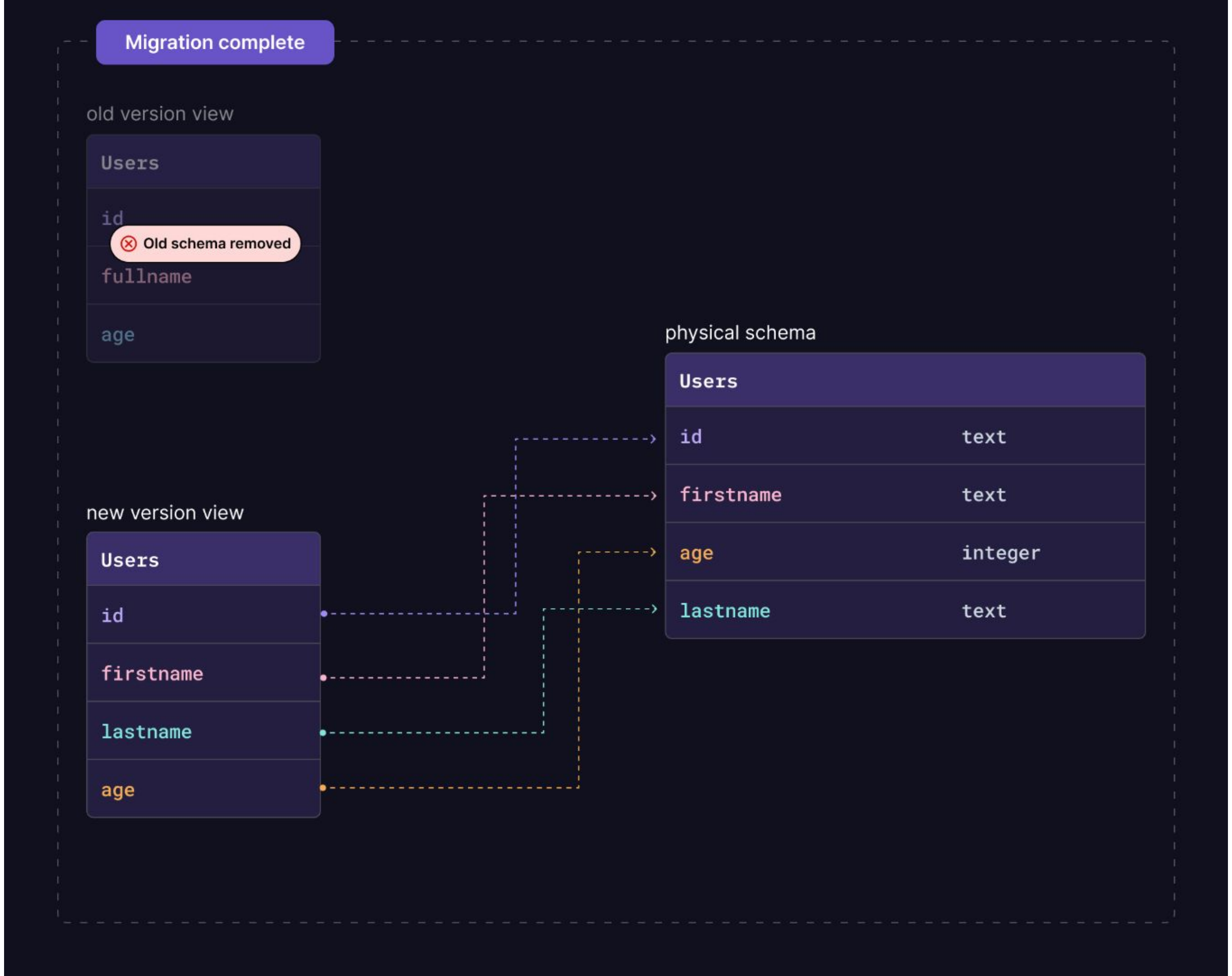
deployment with pgroll





🚧 **Migration operations**

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


Migration complete
old version view
Users
id
Old schema removed
fullname
age
new version view
Users
id
firstname
lastname
age
physical schema
Users
id text
firstname text
age integer
lastname text

How to predict locks?

django-migration-linter

Analyzes Django migration SQL statements using pattern matching to detect operations that could cause table locks or backward incompatibility issues.

squawk

Parses PostgreSQL SQL files using an AST to identify DDL operations that require dangerous lock levels and could block database reads/writes.

pgroll

prevents locks entirely by using virtual schemas and the expand/contract pattern to perform zero-downtime migrations with automatic rollback capabilities.

An alternative to Django Migrations.

Migration is completed!

Questions?

Peaceful Django Migrations

Efe Öge