Cooking Python and PostgreSQL

Aleksandr Dinu

January 22, 2026

Goals for this talk

- Revisit common gotchas of Python ORMs usage
- PostgreSQL-specific tips to make devs (and ops) happier

ORMs - why do we even need them?

- ORMs define tables, columns, contraints and foreign keys using Python concepts - classes, attributes and methods
- Operate with data as if they were regular 'objects' from the database
  - not *rows*, but *objects*
  - not *database cursors*, but *seqs of objects*
- Ease database schema management
  - converting changes from the code of models to the DDL statements
  - give CLI for 'migrations' execution
- Ease connection and transaction control
  - give tools to manage transaction boundaries
  - connection pooling

Django

```
class Question(Model):
    question_text = CharField()
    published_at = DateTimeField()

class Choice(Model):
    question = ForeignKey(Question)
    choice_text = CharField()
```

sqlalchemy

```
class Question(Base):
    __tablename__ = 'question'

    id = Column(Integer, primary_key=True)
    question_text = Column(String())
    published_at = Column(DateTime)

class Choice(Base):
    __tablename__ = 'choice'

    id = Column(Integer, primary_key=True)
    question_id = Column(
        Integer, ForeignKey('question.id'))
    question = relationship(Question)
    choice_text = Column(String())
```

ORMs - Object-Relational Mapping

- ORM converts fetched database tuples into objects for us to match application code models
- ORM implements objects in a way that is easy to use for developers
- ORM allows to define relationships between models and use those as simple model instance attributes

ORMs - be aware of when and what you're fetching

```
## BAD
## doing SELECT * FROM questions;
##   and counting 'objects' in Python
# Django ORM
questions_count = len(Question.objects.all())

# SQLAlchemy
questions_count = len(session.query(Question).all())
```

ORMs - be aware of when and what you're fetching

```
## BETTER
## doing SELECT COUNT(*) FROM questions;
##   and getting integer value back
# Django ORM
questions_count = Question.objects.count()

# SQLAlchemy
questions_count = session.query(Question).count()
```

ORMs - be aware of when and what you're fetching

Dummy (and most likely wrong) benchmark:

- sqlite database (no network costs, 100000 entries)
- performing objects fetching and running `len()` on the first 1000 questions
- performing `SELECT COUNT(*)` query on the first 1000 questions

```
Percentiles:    ( 25%,  50%,  75%)
Fetch + len(): (5.94, 6.04, 6.40) (ms)
Count query:   (0.71, 0.72, 0.73) (ms)
```

ORMs - Object-Relational Mapping

- Relational algebra operates with the term 'relations' (tables/views) and defines 'join' - operation that allows to combine 'relations'
- ORMs offer ways to express 1:M, 1:1, M:M relationships between models
- These relationships are later translated in 'join' operations when ORM translates method calls into SQL-queries

ORMs - Object-Relational Mapping - what often goes wrong

Same dummy benchmark:

- sqlite database (no network costs, 100000 entries, 5-10 choices each)
- selecting 100 questions with all related choices + iterating through all choices
- option #1 fetches choices in a 'joined' manner
  - `joinedload()` in SQLA or `.select_related()` in Django
- option #2 fetches choices in a 'lazy' manner
  - classic "N+1 problem"
  - `lazyload()` in SQLA or not using `.select_related()/.prefetch_related()` in Django

```
Percentiles:              (     25%,      50%,      75%)
Joined load + iterate:    (   43.27,    44.12,    44.85) (ms)
Lazy load + iterate:      ( 932.75,   933.53,   933.92) (ms)
```

ORMs - Object-Relational Mapping

Most ORMs offer schema management tooling:

- able to generate DDL statements based on the object model description in application code, aka 'migrations'
- also can apply such 'migrations' to update database schema to the most recent state

ORMs - Object-Relational Mapping - what goes wrong

- Not all SQL-dialect concepts can be expressed in ORM terms
  - think of custom types, extension, triggers, stored procedures
- Altering database schema can be backward-incompatible
  - e.g. removing a column in a table that's still used by some running application
- Ignoring operational semantics of the underlying database engine
  - altering schema may cause table rewrites, performance degradation, extensive locking or other not expected behavior

PostgreSQL - improve observability

- Specify application name while connecting to the database
  - `create_engine("postgresql://...", connect_args={"application_name":"myapp"})`
  - `log_line_prefix = '%a %u %d'` in postgresql.conf and you'll see it in PostgreSQL logs
  - (almost) all cloud providers support monitoring based on the supplied app name
- Enable logging of slow queries
  - `log_min_duration_statement = 1000` - log all queries slower that 1000ms
  - `log_lock_waits = on` + `deadlock_timeout = 1s` - log all queries that were waiting for any database locks longer than 1s
- Add metrics around number of queries performed during request handling
  - it would help to identify N+1 queries

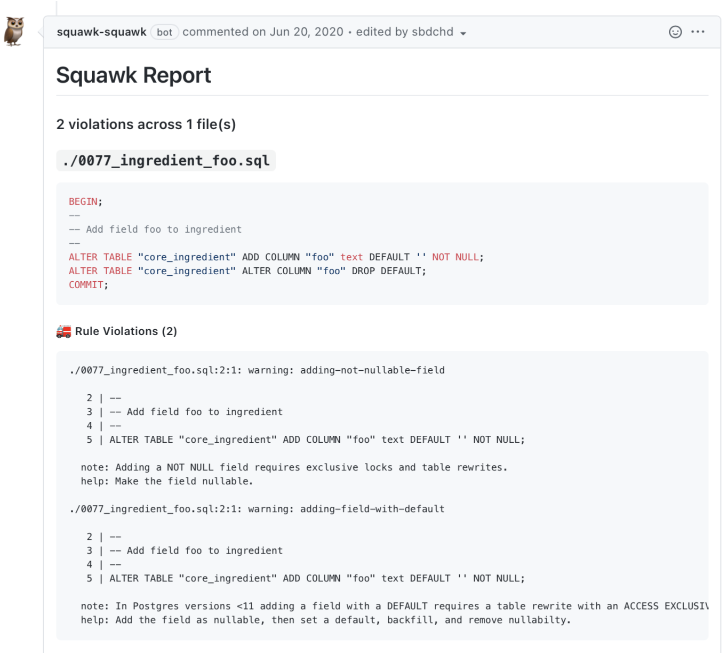
PostgreSQL - know your queries

- If you have caught a slow query in production, pick it
- Run `EXPLAIN (ANALYZE, BUFFERS) <your-query>` to get a query's execution plan
- The `BUFFERS` option tells you how many pages of 8k PostgreSQL used to answer this particular query and in which way:
  - `hit` - number of pages found in the shared buffers
  - `read` - number of pages read from the disk
  - `write` - number of pages written to the disk (e.g. in case of sorting, joins, etc)
  - once pages are read from the disk, they are in shared buffer cache. next run of the same query will be faster because of this.
- `BUFFERS` output is especially relevant in cloud environment
  - if you run PostgreSQL on top of AWS EBS/Azure Managed Disk or Google's Persistent Disk - those `read`'s directly convert to IOPS you use.

PostgreSQL - test more, test early and test often

- Lint your migrations with Squawk
  - let CI tell you which migrations are backward-incompatible or can cause excessive locking
- Add performance regression tests of queries that you executeq with RegreSQL
- HYPE ALERT: take a closer look at branching
  - rather new-ish approach to testing related to databases
  - allows you to have full copy for production database without actually copying the content of it, but rather tracking changes that happen on top of a snapshot of the database state at some point.

PostgreSQL - Squawk



squawk-squawk bot commented on Jun 20, 2020 • edited by sbdchd

Squawk Report

2 violations across 1 file(s)

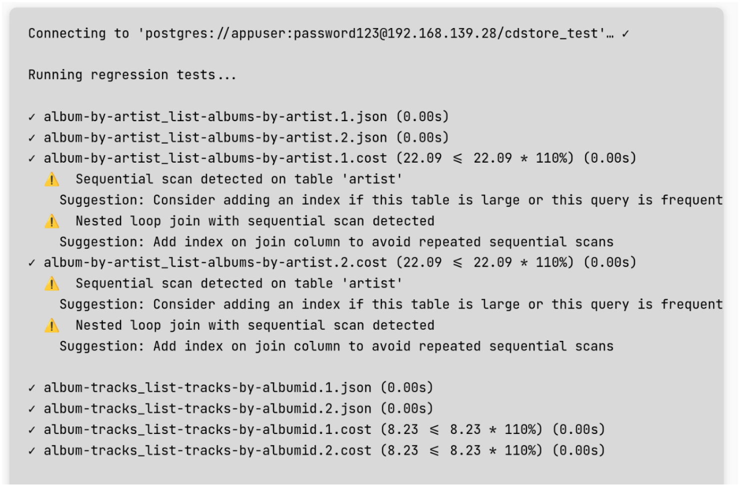
`./0077_ingredient_foo.sql`

```
BEGIN;
--
-- Add field foo to ingredient
--
ALTER TABLE "core_ingredient" ADD COLUMN "foo" text DEFAULT '' NOT NULL;
ALTER TABLE "core_ingredient" ALTER COLUMN "foo" DROP DEFAULT;
COMMIT;
```

🚒 Rule Violations (2)

```
./0077_ingredient_foo.sql:2:1: warning: adding-not-nullable-field

   2 | --
   3 | -- Add field foo to ingredient
   4 | --
   5 | ALTER TABLE "core_ingredient" ADD COLUMN "foo" text DEFAULT '' NOT NULL;

  note: Adding a NOT NULL field requires exclusive locks and table rewrites.
  help: Make the field nullable.

./0077_ingredient_foo.sql:2:1: warning: adding-field-with-default

   2 | --
   3 | -- Add field foo to ingredient
   4 | --
   5 | ALTER TABLE "core_ingredient" ADD COLUMN "foo" text DEFAULT '' NOT NULL;

  note: In Postgres versions <11 adding a field with a DEFAULT requires a table rewrite with an ACCESS EXCLUSIV
  help: Add the field as nullable, then set a default, backfill, and remove nullabilty.
```

PostgreSQL - RegreSQL



```
Connecting to 'postgres://appuser:password123@192.168.139.28/cdstore_test'… ✓

Running regression tests...

✓ album-by-artist_list-albums-by-artist.1.json (0.00s)
✓ album-by-artist_list-albums-by-artist.2.json (0.00s)
✓ album-by-artist_list-albums-by-artist.1.cost (22.09 ⩽ 22.09 * 110%) (0.00s)
  ⚠️  Sequential scan detected on table 'artist'
    Suggestion: Consider adding an index if this table is large or this query is frequent
  ⚠️  Nested loop join with sequential scan detected
    Suggestion: Add index on join column to avoid repeated sequential scans
✓ album-by-artist_list-albums-by-artist.2.cost (22.09 ⩽ 22.09 * 110%) (0.00s)
  ⚠️  Sequential scan detected on table 'artist'
    Suggestion: Consider adding an index if this table is large or this query is frequent
  ⚠️  Nested loop join with sequential scan detected
    Suggestion: Add index on join column to avoid repeated sequential scans

✓ album-tracks_list-tracks-by-albumid.1.json (0.00s)
✓ album-tracks_list-tracks-by-albumid.2.json (0.00s)
✓ album-tracks_list-tracks-by-albumid.1.cost (8.23 ⩽ 8.23 * 110%) (0.00s)
✓ album-tracks_list-tracks-by-albumid.2.cost (8.23 ⩽ 8.23 * 110%) (0.00s)
```

PostgreSQL - branching

Idea is to embed tests against production-sized database into your software delivery pipelines. Think of:

- having preview environment for every pull request based on shared database, but having writes in its own 'database branch'
- running your schema or data migrations against 'database branch' with the same data as in production
- giving access to copies of production dataset to data analytics teams without 2x costs

Many providers:

- `pg_branch`, `pgcow` - extensions/forks of PostgreSQL to work on top of BTRFS and ZFS
- `neon` - fork of PostgreSQL + custom storage layer
- Heroku, Databricks Lakebase, Neon, Postgres.ai - DBaaS products that support branching

Thank you!