From days to minutes, from minutes to milliseconds with SQLAlchemy

Leonardo Rochael Almeida

10-July-2019

Speaker notes

Hi, I’m Leo. I’m a Tech Lead at Geru.

I’m here today to talk to you about ORMs and performance.

I’m by no means an expert in either SQL, SQLAlchemy or ORMs.

But I’d like to pass on lessons learned while optimizing some processes in my company.

Geru

Brazillian Fintech

Backend Stack:

- Others (Celery, MongoDB, Java, ...)

Speaker notes

Our backend stack is almost all Python, with storage mostly in PostgreSQL through SQLAlchemy.

SQLAlchemy

Two aspects:

SQL Expression Language (a Python DSL)

X

Object Relational Mapper (ORM)

Speaker notes

SQLAlchemy has two aspects:

- The SQL Expression Langage, which is a way of mapping SQL constructs into a Pythonic Domain Specific Language (DSL)
- The Object Relational Mapper, which allows mapping Python classes to tables and records of those tables to instances of the respective classes.

The ORM is built upon the DSL, but they can be used without one another.

At Geru we use the ORM almost exclusively.

TODO: Add slides showing code examples contrasting DSL/ORM

SQLAlchemy is Awesome!

However:

Frameworks still require you to make decisions about how to use them, and knowing the underlying patterns is essential if you are to make wise choices.
- Martin Fowler

- *https://martinfowler.com/books/eaa.html*

The ORM Trap

The ORM Trap

- Sensible Python code ➡ Bad SQL access patterns

The ORM Trap

- Sensible Python code ➡ Bad SQL access patterns
- Unnoticeable at low data volumes

The ORM Trap

- Sensible Python code ➡ Bad SQL access patterns
- Unnoticeable at low data volumes
- Like… during development…

The ORM Trap

- Sensible Python code ➡ Bad SQL access patterns
- Unnoticeable at low data volumes
- Like… during development…
- And early production…

Speaker notes

Using a good ORM feels great. Most of the time you forget it’s even there!

And that is actually the problem, because the DB **is** an external system with an API and should be treated as such.

The API just happens to be SQL…

The Fix: Let the DB do its Job

The Fix: Let the DB do its Job

- Be aware of implicit queries.

The Fix: Let the DB do its Job

- Be aware of implicit queries.
    - Specially from relationships.

The Fix: Let the DB do its Job

- Be aware of implicit queries.
  - Specially from relationships.
- Aim for O(1) queries per request/job/activity.

The Fix: Let the DB do its Job

- Be aware of implicit queries.
  - Specially from relationships.
- Aim for O(1) queries per request/job/activity.
  - Avoid looping through model instances

The Fix: Let the DB do its Job

- Be aware of implicit queries.
    - Specially from relationships.
- Aim for O(1) queries per request/job/activity.
    - Avoid looping through model instances
        - Let the DB do it for you

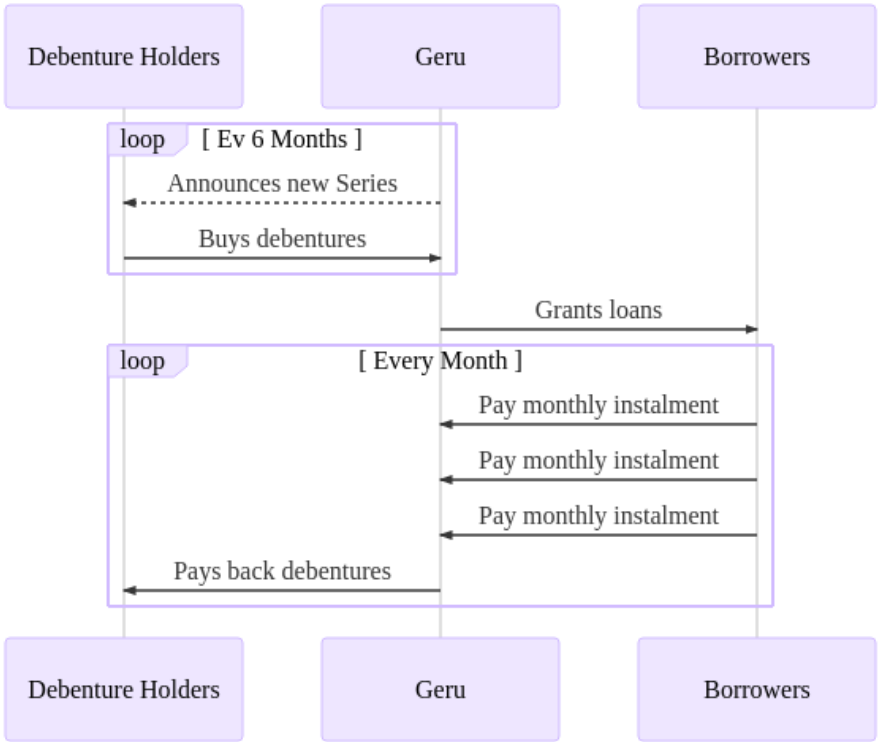
Speaker notes

- Be mindful of the work that the database is doing
- Specially the amount of DB round-trips
- But also the amount of data traffic (row count)

Geru Case 1: The 24+ hour reports

Now it takes minutes

Geru Funding Model



Speaker notes

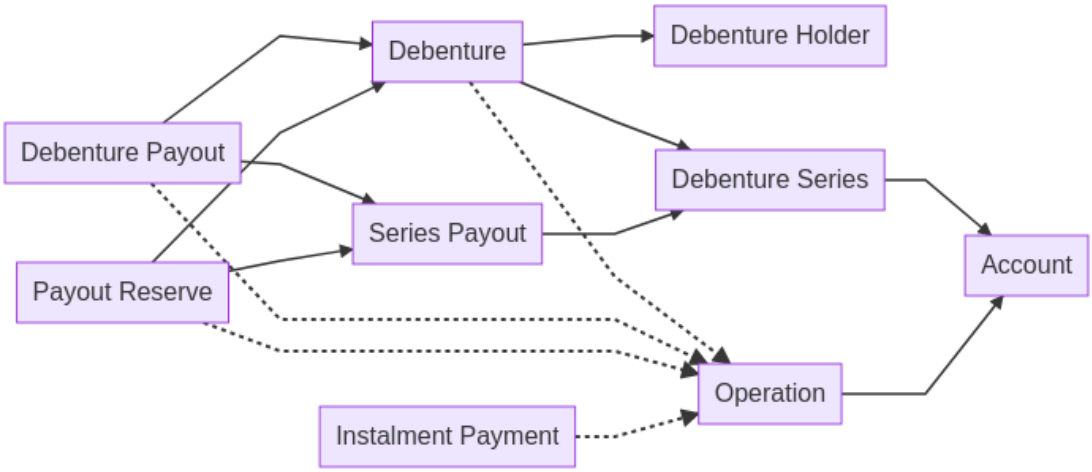
Geru is a Fintech that lends money at rates much lower than the mainstream banks in Brazil. We work online exclusively.

During each month, borrowers pay their monthly instalments, and at the beginning of every month Geru pays back the Debenture Holders.

This is very simplified of course, there are lots of details on top of that:

- Debentures bought later “cost” more but are “worth” the same
- Debenture remuneration is complicated by tax details like
    - Amortization paid back doesn’t pay taxes but the premium on top does pay
    - Amount of time invested reduce taxes
- Different series have different payback rules

Entities and Relationships



ORM Declaration

```
DBSession = scoped_session(sessionmaker(...))

class ORMClass(object):
    """Base class for all models"""
    @classproperty
    def query(cls):
        """
        Convenient query for records of a model, like:

            query = MyModel.query.filter(...).order_by(...)
        """
        return DBSession.query(cls)

Base = declarative_base(cls=ORMClass)
```

Model Declaration

```
class Debenture(Base):

    id = Column(Integer, primary_key=True)
    series_number = Column(Integer, nullable=False)

    sale_price = Column(Numeric, nullable=True)

    sale_date = Column(Date, nullable=True)

    # cont ...
```

Model Declaration

```
class Debenture(Base):
    # cont ...
    holder_id = Column(
        Integer, ForeignKey('debenture_holder.id'),
        nullable=True, index=True,
    )
    holder = relationship(
        'DebentureHolder', backref=backref(
            'debentures', lazy='dynamic',
        ),
        foreign_keys=[holder_id],
    )
    # cont ...
```

Model Declaration

```
class Debenture(Base):
    # cont ...
    series_id = Column(
        Integer, ForeignKey('debenture_series.id'),
        nullable=False, index=True,
    )

    series = relationship(
        'DebentureSeries', backref=backref(
            'debentures', lazy='dynamic',
        ),
        foreign_keys=[series_id],
    )
```

First things first: logging

```
# development.ini
[loggers]
keys = sqlalchemy

[logger_sqlalchemy]
qualname = sqlalchemy.engine
level = INFO
# "level = INFO" logs SQL queries.
# "level = DEBUG" logs SQL queries and results.
# "level = WARN" logs neither (in production).
```

Speaker notes

So I had to debug an issue in the distribution code, but it was taking way too long at each run.

So the first thing I did was to enable sqlalchemy statement logging in my development instance, and what I saw was gobs of repeated statements, all alike, just rolling through the logs.

Understanding the cache optimization

See diff and Jupyter

Speaker notes

It’s perfectly reasonable in pure Python to `sum()` over an iteration of attribute accessess in generator comprehension.

But if the generator comprehension is looping over a query then a lot of data is being fetched from the database so that in the end the Python programmer could calculate do what the database could reply with a single line of SQL.

Understanding the insert/update optimization

See diff and Jupyter

When the optimization backfires

See diff and Jupyter

Speaker notes

Unfortunately at some point the complex query that allowed to fetch all information of each integralization started taking hours.

It was a single query taking many hours to execute.

Fortunately it was easy to locate as it was a single query envelopped by logging calls.

The query was then broken into two parts, the second of which was executed in a loop for each integralization. Since the amount of data transmitted was small, and only a single query per loop was added inside a loop that already contained multiple other slower queries, it had no negative impact, and the outer query ran again at the same 2 minutes timeframe as in the beginning.

Geru Case 2: The 1+minute page

Now renders in less than a second.

First things first: slowlog

```
[app:main]
pyramid.includes =
    pyramid_tm
    [...]
    slowlog

# Slowlog configuration:
slowlog = true
slowlog_file = logs/slow.log
```

Speaker notes

A complete understanding of what slowlog does is out of scope for this talk (there is talk by me at PyConUS 2013 about it on YouTube), but basically slowlog watches for wsgi requests that take too long and starts dumping periodic stack traces of the thread handling the slow requests.

Makes it ease to see which point of the code is responsible for the performance issues.

Understanding the authorization optimization

See diff

Conclusions

Understand SQL

Conclusions

Understand SQL

- `SELECT` Documentation

Conclusions

Understand SQL

- `SELECT` Documentation
- `GROUP BY` vs aggregation functions

Conclusions

Understand SQL

- `SELECT` Documentation
- `GROUP BY` vs aggregation functions
- aggregation function w/ filters

Conclusions

Understand SQL

- `SELECT` Documentation
- `GROUP BY` vs aggregation functions
- aggregation function w/ filters
- `DISTINCT ON`

Conclusions

Understand SQL

- `SELECT` Documentation
- `GROUP BY` vs aggregation functions
- aggregation function w/ filters
- `DISTINCT ON`
- window expressions

Speaker notes

Study SQL:

- Read the `SELECT` documentation of your database. Understand how aggregation functions (`sum()`, `array_agg()`) interfere with the cardinality (number of rows) of the result and how they interact with `GROUP BY`.
    - Understand `DISTINCT` and specially `DISTINCT ON`
    - Understand "window" expressions
        - `sum(X) OVER (PARTITION BY ... ORDER BY)`
    - Read about CTEs (`WITH` statement) and subqueries, and how/where you can use them.
    - Understand how to insert and update rows that match the result of other queries.

Conclusions

THEN Study SQLAlchemy

Conclusions

THEN Study SQLAlchemy

- Be aware of the underlying queries

Conclusions

THEN Study SQLAlchemy

- Be aware of the underlying queries
- Push work to the DB

Conclusions

THEN Study SQLAlchemy

- Be aware of the underlying queries
- Push work to the DB
    - As much as possible

Conclusions

THEN Study SQLAlchemy

- Be aware of the underlying queries
- Push work to the DB
  - As much as possible
  - But not too much

- THEN Study SQLAlchemy
    - Learn how to produce in SQLAlchemy the same optimized access patterns you know are possible in SQL
- Be aware of the underlying queries
    - All attribute accesses could represent a roundtrip. Atomic values usually don’t (though they can if you ask SQLAlchemy to defer loading columns). But all relatioship attributes do, unless they’re not dynamic and have been previously accessed in the same session.

The End

Thank you!

- LeoRochael@geru.com.br
- LeoRochael@gmail.com
- @LeoRochael