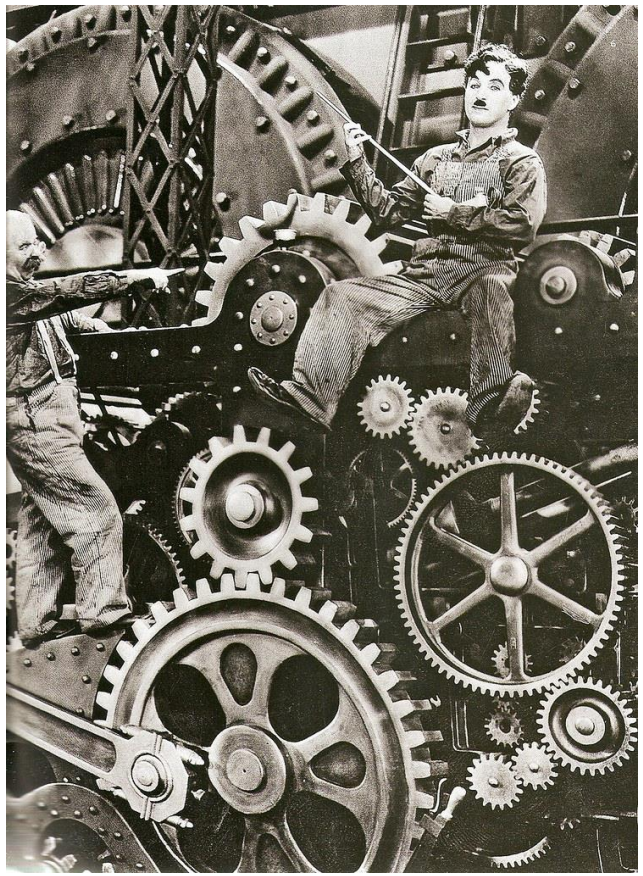
Lessons learned using Postgres in data science projects

PyData 2022, Berlin

Jacopo Farina
jacopofarina.eu
@jacopofar
github.com/jacopofar



How do we process so much data fast enough for it to be useful?

- Lots of data (at peak **>5 tickets per second**)
- Must be done **at least daily**
- Data must be **fresh**

So what was the problem?

Accurate pricing requires real-time data and processing real-time data is hard

1. Large volumes of data takes long time to process
2. Application should remain stable even when schema changes
3. Outages can happen due to peak time activity and upgrades

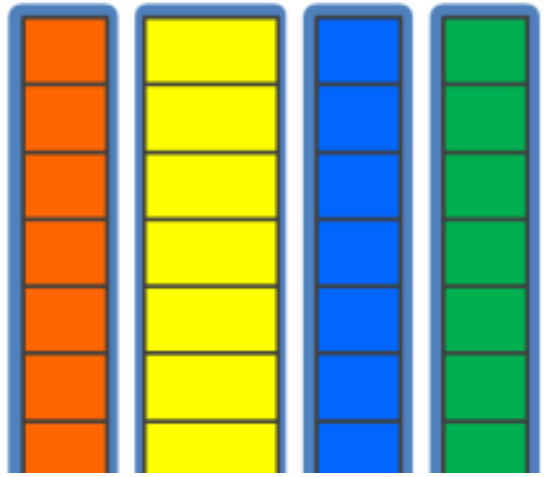
Problems with the database?

- Lots of data, performance issues
- Different Python libraries
- How to keep track of the queries
- How to keep track of schema changes

The problem we are discussing:

Our use case, your mileage may vary

- Use of historical data
- Mostly batch, but some web apps
- Python and Postgres



The common way to prepare data for data science

Fast and small operations, access mostly recent data

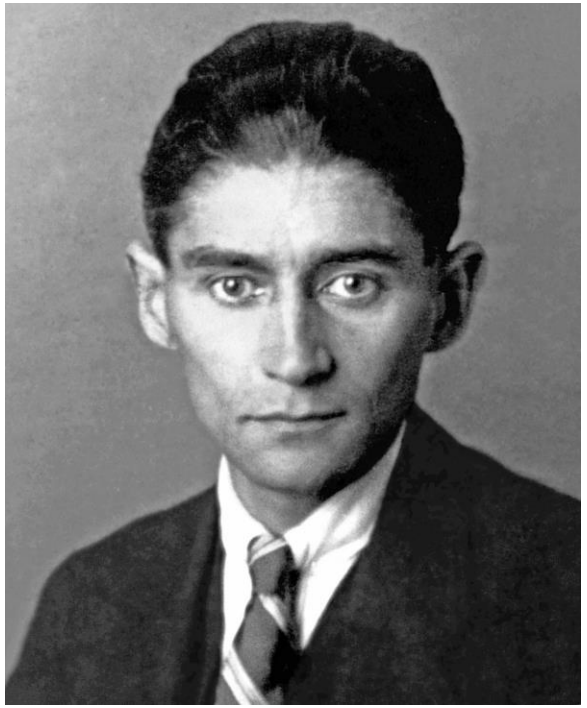
ETL = **E**xtract **T**ransform **L**oad

Massive read of historical data, freshness not essential

Our solution: Kafka as the main (only) source

Events from Kafka are continuously persisted in DB tables using Debezium, a CDC (Change Data Capture).

The data is fresh, but it's not an ETL (**no guarantees on the order**).



ETL steps using materialized views (based on normal views)

A cronjob can refresh a materialized view, the code is on the schema and easy to inspect.

When based on a view it's trivial to switch to have fresh data and benchmark

```
CREATE MATERIALIZED VIEW ext.distances_mview AS
SELECT *
    FROM ext.distances_view;
```

Question

Who can easily say how long does a specific query take **on average**?

A SQL file for each query

Each project has a folder with all the queries as SQL files to simplify searching, editing and running them by hand.

Our trick: a single helper for all queries

```
popularity_df = execute_query(
    'get_relations_popularity',
    stop_relations=stop_relations,
    ride_departure_date_start=start_date,
    ride_departure_date_end=end_date,
)
```

Three advantages of having a single helper:

- **retrying -** the database can be restarted without breaking the pipeline
- **transforming** when needed (e.g., change the datatype)
- **profiling** the execution time - logged and examined as part of the whole application profile

An overview of how the application spends time

What about insertion?

Ways to insert a lot of data quickly

- **execute_many** is the bare minimum
- **COPY** is the best way when upsert is not needed
  - **Unlogged tables** are faster to create for temporary data
- Psycopg2 has the (ugly) **execute_values**
- **Prepared statements** incoming with Psycopg3

detailed comparison: https://jacopofarina.eu/posts/ingest-data-into-postgres-fast/

Preprocessing data and storing as a file

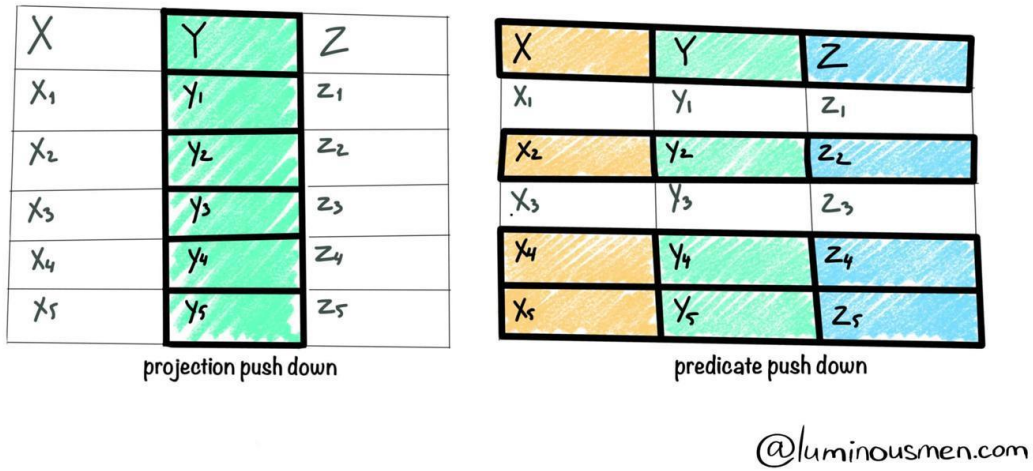
Static or historical data that is queried the same way over and over can be persisted in a file.

Parquet allows for column access

Arrow efficiently represents data as binary

(.. and **Snappy** for compression)

Pandas and Spark can easily handle these files



Handling the schema changes

We want the schema to be handled as code, with all the advantages (single source, pull requests, versioning).

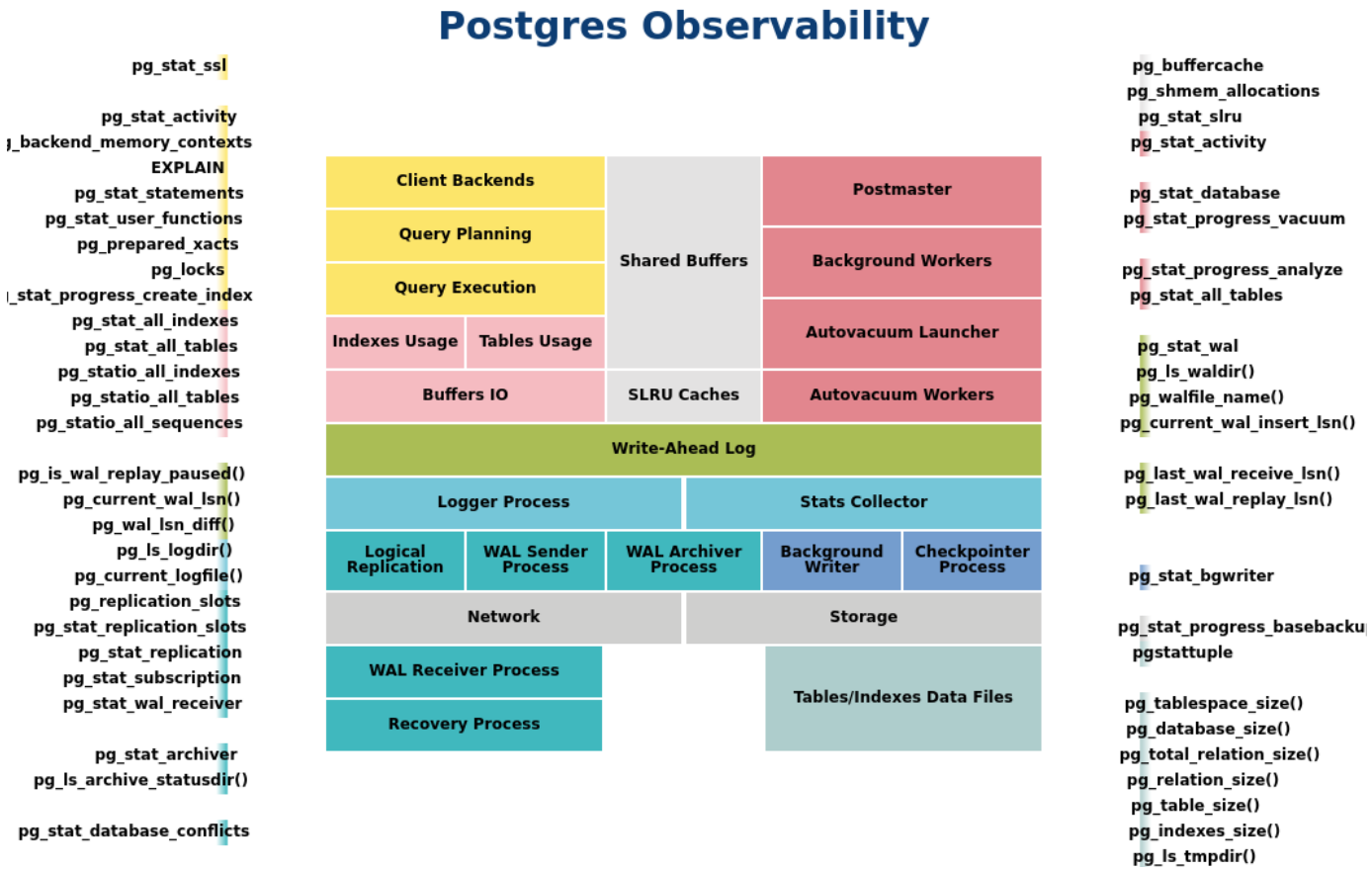
- **SQLAlchemy** supports migration but it's very limited
- **Sqitch**, very powerful but requires a complex config
- For us, a repository with a *schema.sql* file is enough

```
24  dump_schema:
25      # if the file does not exist, the volume mount will create a directory
26      # owned by root m(
27      touch schema_local.sql
28      docker run \
29          --rm \
30          -v ${PWD}/schema.sql:/srv/schema.sql \
31          -v ${PWD}/schema_local.sql:/srv/schema_local.sql \
32          $(POSTGRES_IMAGE) \
33          pg_virtualenv -t sh -c \
34              'psql -d $$PGDATABASE -f /srv/schema.sql \
35              && pg_dump $(PG_DUMP_OPTIONS) -d $$PGDATABASE | sed -e '/^--/d' > /srv/schema_local.sql'
36  .PHONY: dump_schema
37
38  dump_remote_schema:
39      docker run \
40          --rm \
41          $(POSTGRES_IMAGE) \
42          pg_dump $(PG_DUMP_OPTIONS) $(POSTGRES_CONN_STR) | sed -e '/^--/d' > schema_upstream.sql
43  .PHONY: dump_remote_schema
```

pg_dump --schema-only to get the schema from a DB

pg_virtualenv to create an ephemeral Postgres instance

Use Postgres system tables to profile the overall usage



See: https://pgstats.dev/

Join the ride…

We have open positions for data scientists and engineers, discover your new opportunity here!

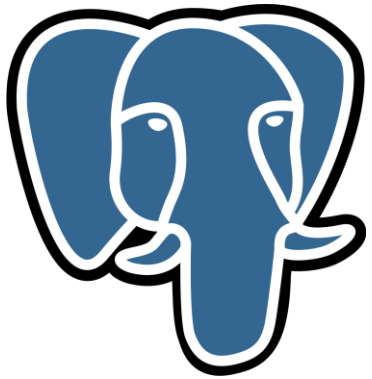
Do you have any questions?


Thank you!
FLIXBUS
SETRA
FLIXBUS

Accessing the data from Python

(essentially, accessing from Pandas)

- **Psycopg2**: the one supported by SQLAlchemy
- **Asyncpg**: binary protocol, nice API. Async only, which means no Pandas
- **Psycopg(3)**: new kid on the block, binary protocol and optional async, SQLAlchemy support on the way



Keep everything running as decoupling/migration goes on

We need to know **who** reads **what** to plan migrations.

E.g., line variations are going to disappear

As the complexity grows, we started to document the product-table matrix, but it's a manual process

Use permissions to track usage metadata

Assigning users to the different components of the pipeline with minimal access to every table, so that permissions track who uses what and not who *can* use what.

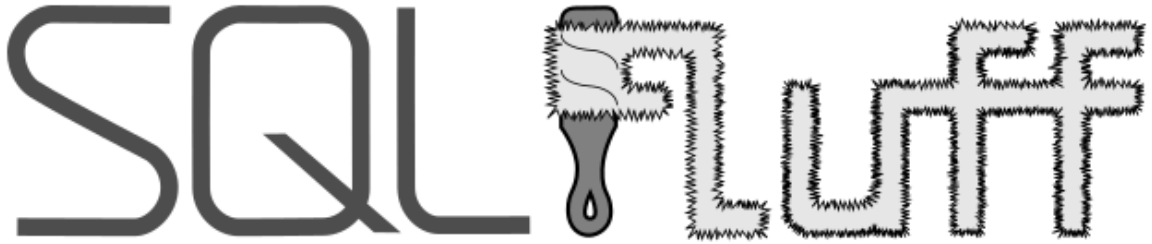
The who-accesses-what becomes explicit part of the schema, easy to check and enforce

Static analysis

Since all the queries are SQL files, we can parse them from the code and infer this information.

Not a trivial task:

- SQL has many dialects and versions and has extensive syntax
- SQLAlchemy and others have different placeholders
- Python's DBAPI has three different parameter passing styles



The SQL Linter for Humans

SQLFluff is a linter (and parser) for many SQL dialects including Postgres, written in Python.

Supports pretty much every parameter passing style (we did a PR).

Allows programmatic access to the parser, still WIP

Currently we can use **sqllineage** to extract this mapping. It's essential, but it works. A few edge cases remains:

- Generated queries
  - Jinja generation can be covered by SQLFluff
- Pandas / SQLAlchemy core operations – static analysis can be extended to Python code, but probably is not worth it