

Lessons learned using Postgres in data science projects Jacop

Jacopo Farina

jacopofarina.eu @jacopofar github.com/jacopofar

PyData 2022, Berlin



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



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

Operational systems

Fast and small operations, access mostly recent data



ETL = Extract Transform Load ML / Business Intelligence

> 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

get_rides_in_lines_and_dates.sql ride_relation_min_price.sql get_ride_relation_min_price get_mappings.sql holidays_data.sql get_holidays_data competing_rides.sql get_inner_competition_data orders_by_relation.sql get_order_items OnlyNominalMinPrices.transform AddPredictions.transform get_alphas_from_ride_relations.sql store_predictions_in_db send_predictions_to_kafka get_prices.sql old_send_predictions_to_kafka predict

Tasks execution time: manual-ufc-pred-wrk-4mz8h.log

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: <u>https://jacopofarina.eu/posts/ingest-data-into-postgres-fast/</u>

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).

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

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

pg_virtualenv to create an ephemeral Postgres instance

FLiX

See: <u>https://pgstats.dev/</u>

pg_stat_database_conflicts

pg stat archiver pg ls archive statusdir()

pg is wal replay paused() pg current wal lsn() pg wal lsn diff() pg_ls_logdir() pg_current_logfile() pg replication slots pg_stat_replication slots pg stat replication pg stat subscription pg stat wal receiver

pg stat activity) backend memory contexts EXPLAIN pg stat statements pg stat user functions pg prepared xacts pg_locks pg stat all indexes pg stat all tables pg_statio_all_indexes pg statio all tables

pg statio all sequences

_stat_progress_create_index

pg_stat_ssl

Client Backends Query Planning Query Execution Indexes Usage Tables Usage

Postgres Observability

Shared Buffers

SLRU Caches

Write-Ahead Log

WAL Archiver

Process

Postmaster

Background Workers

Autovacuum Launcher

Autovacuum Workers

Checkpointer Process

Stats Collector

Storage

Tables/Indexes Data Files

Background Writer

Use Postgres system tables to profile the overall usage

pg buffercache pg shmem allocations pg stat slru pg_stat_activity

> pg stat database pg stat progress vacuum

pg_stat_progress_analyze pg_stat_all_tables

pg stat wal pg_ls_waldir() pg walfile name() pg current wal insert lsn()

pg last wal receive lsn() pg last wal replay lsn()

pg_stat_bgwriter

pg_stat_progress_basebacku pgstattuple

pg tablespace size() pg database size() pg total relation size() pg relation size() pg_table_size() pg_indexes_size() pg ls tmpdir()

WAL Receiver Process **Recovery Process**

Logger Process

Network

WAL Sender

Process

Buffers IO

Logical Replication

Join the ride...



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

Do you have any questions?

Thank you!

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 producttable 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 <u>can</u> 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