Report on Data Pipeline Retail Store

Filippo Pizzo

1 Introduction

1.1 Scenario and Our Solution

For our project, we imagined a chain of retail stores as the client. Every store collects daily transaction data in a CSV file. The client wants the data to be stored for analysis and to support the company's decision making.
To do so, we propose an ETL pipeline, where the data is transferred to a database at night. After that, the data can be analyzed using queries to answer the clients' questions about their transactions and sales.
The ETL pipeline is divided in 3 steps:

- Data Ingestion
- Data Transformation
- Data Storage

In the data ingestion, data from the CSV files are extracted and putted into a data frame so that the transformation can happen.
Then in the data transformation we transform the extracted data to make them consistent and structured for further use.
At the end in the data storage, the data are stored in a local database so they can be analyzed.

1.2 Tools Used

To build this data pipeline, we use Python scripts. In these Python scripts, we use Pandas to read the data from the CSV files and transform it and Psycopg2 to connect to the database and store the data in the database.
For the database management we choose PostgreSQL, that's because it supports SQL, which we are more familiar with, and supports large-scale of workloads that can be useful if the retail company work with big amount of data. We also choose PostgreSQL because we could use pgAdmin with it, which provide a simple interface.
The data can then be visualized with Grafana, a platform for data visualization and analysis. In Grafana we can connect to the database and then use queries to analyze data and report information on graphs.

To orchestrate everything, we use Apache Airflow, which is a workflow management platform that allow us to incorporate python scripts and use batch automation. We choose it because we think that this characteristics suit very well for a retail store and also because we think that we are not working in a super reactive system that needs to analyze data in real time, but manage the data at regular intervals, that in our case is at midnight when the shops are already closed.

1.3 Structure of the Report

The configuration of these tools is described in section 2.
After that we go through different parts of building a data pipeline with:

- Data Ingestion in section 3
- Data Transformation in section 4
- Data Storage in section 5
- Data Visualization in section 6.

How the workflow was managed will be explained in section 7.
Then in section 8 we will show the data pipeline in practice and how it performs with different size of datasets.
After that some possible improvements are discussed in section 9.
Finally we will conclude in section 10.

2 Configuration of the Tools

2.1 Configuration of the virtual environment

Normally this type of pipeline should be implemented in a distributed environment, but because we don't have the resources to do that and this is a project made for didactic purposes, we decide to develop our project locally.
We decide to implement our application on a virtual environment so that we could have an isolated environment that is not influenced by other projects, preventing in this way possible version conflicts. A virtual environment is also needed to configure Apache AirFlow correctly.
To configure it, we first have to install `python3-pip` that is needed for the installation of the venv and also for the configuration of other tools we need to use.
After the correct installation of `python3-pip`, we can then install the virtual environment and then create a virtual environment inside our folder:

```
sudo apt install python3-pip
sudo pip3 install virtualenv
python3 -m venv airflow_env
```

After the creation of the virtual environment, we can initiate it with:

```
source airflow_env/bin/activate
```

2.2 Configuration of Apache Airflow

Now that we have correctly configure our virtual environment, we can then configure Apache Airflow. To do this, we have to first initiate the environment with the above command.
Now we can install Apache AirFlow and create a new user:

```
pip3 install apache-airflow[gcp,sentry]
airflow db init
airflow users create --username admin --password admin --firstname
admin --lastname admin --role Admin --email name.surname@vub.be
```

The creation of the user is needed to access the web server and use the Apache Airflow interface.
Another important step is installing the necessary dependencies for interacting with PostgreSQL databases:

```
pip install apache-airflow-providers-postgres
```

Now that we have configured the necessary things, we can create a folder called `dags` inside the virtual environment, that in our case is `airflow_env`, and create a new dag inside this folder.
Finally, if we want to see the dag in the interface, we have to run:

```
airflow scheduler
airflow webserver -p 8080
```

2.3 Configuration of PostgreSQL and pgAdmin

To configure the connection to the PostgreSQL database, we have create two file:
`database.ini` which contains the configuration settings for the connection to the database.
`config.py` which reads the configuration from `database.ini` and convert them to a dictionary that is needed for instantiating the connection.
We connect to the localhost with Psycopg2 for the transactions. Six tables were created in the database to later store the data: cities, stores, products, transactions, stocks, transaction_products. In section 4 it is described, why these tables are needed. pgAdmin also needed to be connected to the database to be used for the project. When this connection was established it was possible to look into the tables in the database for checking the correctness of the Python scripts.

2.4 Configuration of Grafana

To configure Grafana, after its installation, we have to start it by executing `grafana-server.exe`, located in the bin directory.
After that, to run Grafana we have to go to the Grafana port that by default is http://localhost:3000 and sign in by entering `admin` as username and password.
Now that we can access to the interface, we have to connect the database by adding the host URL, the name of the database, and the information for the authentication.

3 Data Ingestion

The first part in the data pipeline is the data ingestion.
In the data ingestion we start from CSV files, provided by the retail stores, where there are stored the data about the transactions that happen that day.
The CSV files are composed of 14 columns: transaction_id, date_time, payment_method, product_id, product_description, product_category, unit_price, quantity, discount, store_id, store_city, store_address, store_postalcode, transaction_type (that can be Buy or Sell).
We can notice that every row of CSV file has different type of fields saved in it, indeed in every row we can find:

- information about the transaction
- information about the product
- information about the product in a transaction
- information about the store

To read data from the CSV file, we use Pandas, with who we save the data in a data frame, which is a two dimensional data structures with labeled rows and columns.

4 Data Transformation

After the ingestion of the data, our goal is to transform data to consistent and useful. So from a complicated and unique data frame we want to:

- remove the anomalies creating a log file for re elaboration
- create new useful calculated fields
- partition data into more data frames based on the object they describe

4.1 Cleaning Data

In this step of the data transformation, our goal is to remove the anomalies from the data frame we have saved data in.
For this step, it is difficult to find a specific method that works for every company, since it depends on the needs of the company. In our case, we assume that the data saved by the customer can have errors on the value fields, duplicated rows and inconsistencies between fields of the same object.

For errors on the value fields, we mean that the fields quantity, discount, date and transaction_type, that are easily checkable without knowing other information, can have values that are incorrect.
In fact, quantity cannot have a value that is less than 1, discount can't have a value bigger than 1, that means 100%, the date of the transaction cannot be after the moment we are executing the program and the transaction type can be only buy or sell.
So as the first step of the data cleaning we check every row and if we found a row that has one of the above described error, we add it to a list called removed_rows with the causal of why it has an error. Otherwise we save it in a list called sales_toKeep and then convert it to a data frame.

After checking the correctness of the values inside the data frame, we check if there are duplicated rows. This step consist in using a set to save only the rows that have a pair transaction_id and product_id that has not been already saved.
The rows that have a unique pair that has already been saved, are saved in the removed_rows list with the causal duplicated row.
We decided to do not keep the duplicated rows because we assume that this was agreed with the customer, since his priority is to have data that can be consistently store in the database.
The choice of keeping the first row and not another one is casual since we can't know which one of them is actually the correct one. In the case that the stored row is not the correct one, our idea was that the store manager have the possibility to access to the list of rows that we don't keep and tell us which rows on the log file he see as correct and change them with the one saved in the database.

The last step of the data cleaning is to check the fields related to the same object. We do this because, as already said, the data frame contains fields about different objects, that are: transaction, product, product in a transaction and about the store. In the data frame for every object we have an identifier whose purpose is to uniquely identify a specific instance of an object.
Our goal in this step is to have only one instance of a specific object for every identifier.
To do this, for every object, we check the rows that have the instance that is more frequent in the data frame and we keep it. Meanwhile the rows that have less frequent instances are gonna be collected in the removed_rows list with the

causal that in this case is a string that tells for which object the row has been removed and the object identifier of the conflict row.
We choose this implementation of the problem because we assume that this was agreed with the customer, since his priority is to have consistent data to save. Then in case of some rows that have wrongly not been insert in the database, even in this case the store manager can look at the rows that we didn't keep and decide if they have correctly been removed or modify them to make it possible to insert the rows in the database.
Now that the cleaning step is finished we can save the list of remove_rows in a .txt file with a log function so that they can be modified and rerun or permanently removed.
We choose to create this log file to preserve the existing data, because they are an asset of the company, so we let the store manager see and in case correct them.

4.2 Create Derived Fields

In this step we add new calculated fields to the data frame. Their goal is to extend the data with added values useful for analysis.
The derived fields are:

- total_price: the price of a product in a transaction using quantity, the unit_price and the discount
- total_sold: the total number of product sold for a product
- transaction_price: the total price of a transaction
- stock_level: The stock of a product in a specific store

The derived field total_price is calculated at the moment, this is because we already have the information we need in the row. Meanwhile the others are calculated during the partitioning process of the data frame that contain the product in a transaction. That is because that's the data frame that contains all the correct rows, so for every row that it elaborate, it also update the derived fields.
For calculating total_sold, for every row in the product in a transaction object, we get the row of the product object from the product_id and the row of the transaction object from the transaction_id and if the type of the transaction instance is sell, we update the field total_sold adding the quantity field of that row.
For calculating transaction_price, for every row in the product in a transaction object, we get the row of the transaction object and we update the transaction_price by adding the total_price field of that instance.
For calculating stock_level, for every row in the product in a transaction object, we get the row of the transaction object and the row of the store object and we update the stock_level by checking the transaction_type of the transaction instance:

- if it's buy, then we add the quantity field to the stock_level,
- if it's sell, then we subtract the quantity field from the stock_level.

4.3 Data Partitioning

After cleaning and calculating, the data are still stored all together in one data frame. Therefore, there will be many redundancies in the data and they can cause anomalies. To avoid these anomalies, it was decided to transform the data into a third normal form. We choose the third normal form because it is sufficient for our purpose and higher forms might lead to more complicated database structures.
We split our data into six data frames:

The first data frame is for the cities, that includes store_postalcode as identifier and store_city. For the creation of this data frame, we go through every row of the unique data frame and for every new value of the postal code, we add a new line to the cities data frame.
The second data frame is for the stores, that includes store_id as identifier, store_address and store_postalcode. For the creation of this data frame, we go through every row of the unique data frame and for every new value of the store identifier, we add a new line to the stores data frame.
The third data frame is for the products, that includes product_id as identifier, product_name, product_description, product_category, unit_price and total_sold. For the creation of this data frame, we go through every row of the unique data frame and for every new value of the product identifier, we add a new line to the products data frame.
The fourth data frame is for the transactions, that includes transaction_id as identifier, date_time, payment_method, transaction_type, transaction_price and store_id. For the creation of this data frame, we go through every row of the unique data frame and for every new value of the transaction identifier, we add a new line to the transactions data frame.
The fifth data frame is for the stocks, that includes store_id and product_id used together as identifier and stock_level. For the creation of this data frame, we go through every row of the unique data frame and for every new value of the identifier, we add a new line to the stocks data frame.
The last data frame is for the product in a transaction and it's called transaction_products. It includes transaction_id and product_id used together as identifier and quantity, discount and total_price.

5 Data Storage

After the data is transformed, we can save it in the PostgreSQL database that already includes the tables needed, the structure of the tables can be seen in Figure 1. Therefore a connection with the database is established with Psycopg2 in the Python script thanks to the configuration file.

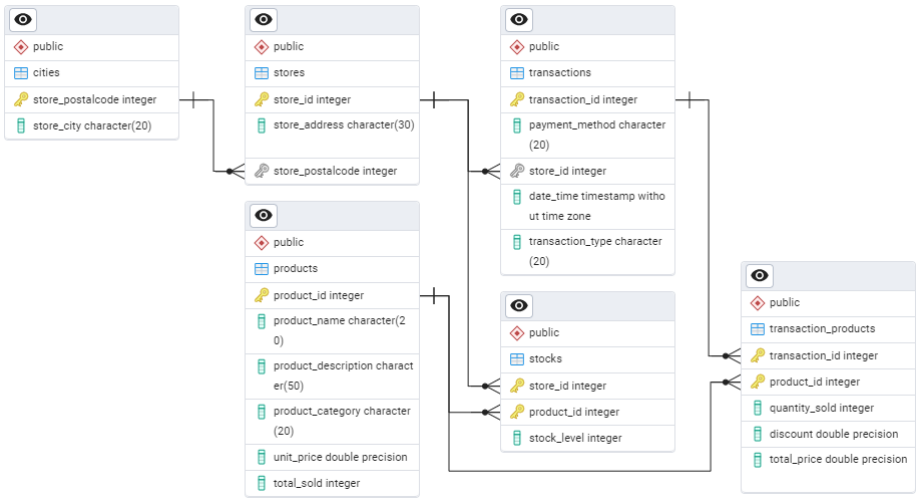


Figure 1: diagram with overview of the tables in the database

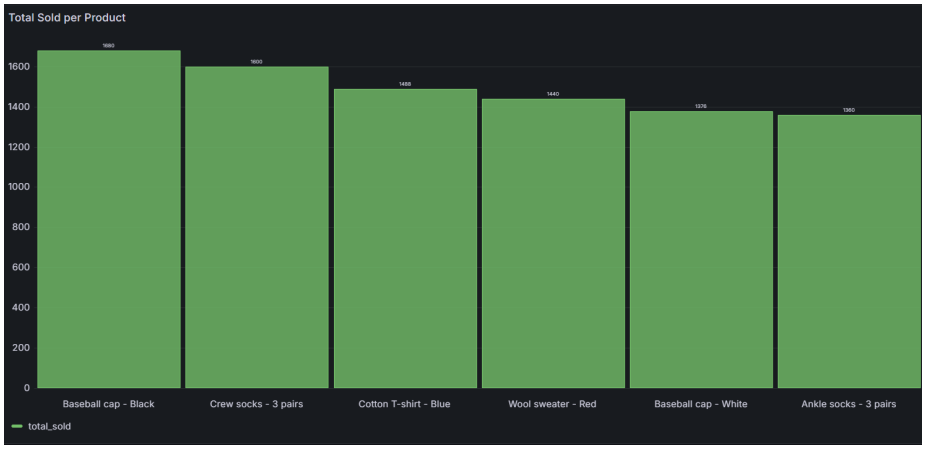
The data are then inserted into the tables through some SQL statements. These statements are written as strings in the Python script and then executed through Psycopg2 as well as committed by it. After that the connection can be closed.
The table transaction_products, need to be updated for every row of new information by adding the new rows, since every row of the CSV file is about a product in a transaction, so a new instance for this object.
In the other tables, we will add information only if there isn't that information saved. Even if new rows will not be inserted, some tables need to be updated for every row of that object. This tables are the one that need to update their derived fields, that in our case are products and stocks. The table transactions doesn't need to be updated because an instance of transaction can appear more times, but only in one CSV file. While for products and stocks, it's normal to have the same instance in more CSV files because people can buy a specific t-shirt on different days or stores.
The others table, just insert new rows if there is a new instance that didn't appear before, otherwise they do nothing.

6 Data Visualization

Now that we have the data consistently stored in the database, we can use them to give our customer, that in our case could be the store manager, insight into the data.
To do this as already mentioned we use Grafana, which is an application for data visualization and analysis, that with queries can represent in graphs information useful for seeing the situation of the stores.
To do this graphs we can use the calculated fields or we can do complex queries

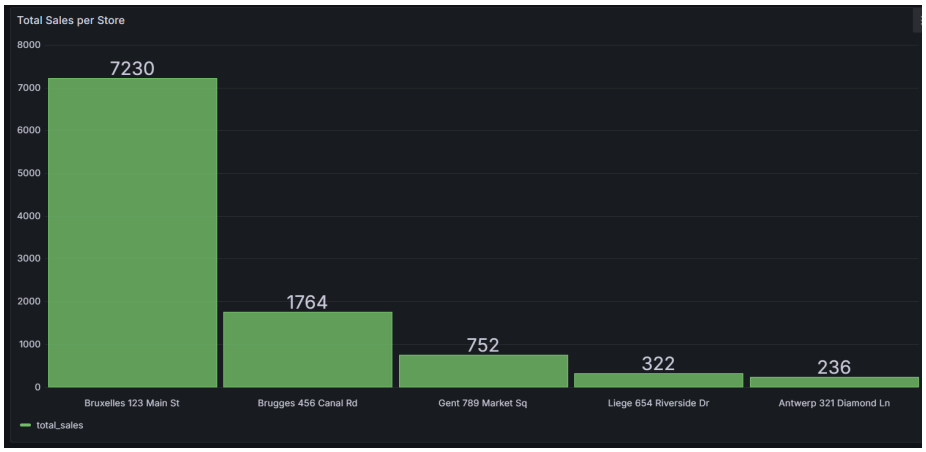
to have more type of insights.
We will now explain some queries made for data analysis:
Starting with a simple query, we made a graph to see the best seller of all time. To do it we use the derived field total_sold in the products table:

```
SELECT product_description, total_sold
FROM products
ORDER BY total_sold DESC
LIMIT 6
```



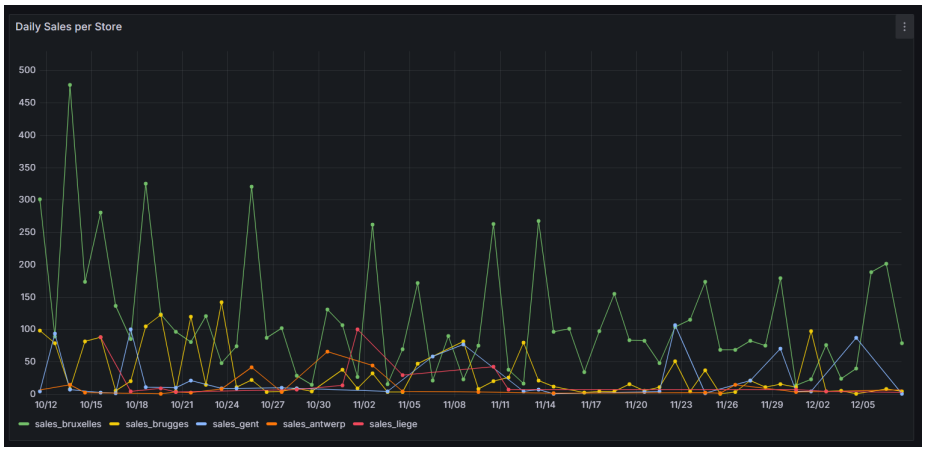
Another graph we made is about the total sales for every store, to do it we sum the quantity_sold field for every store:

```
select CONCAT(store_city, ' ', store_address) AS store_location,
SUM(quantity_sold) AS total_sales
from cities c join stores s
    on c.store_postalcode = s.store_postalcode
join transactions t
    on s.store_id = t.store_id
join transaction_products tp
    on t.transactions.transaction_id = tp.transaction_id
group by store_address, store_city
order by SUM(quantity_sold) desc
```



A more complex graph that we made is about the sales of every store compared for each day, to do it we used quantity_sold that we sum for every day for each store. In fact we did a different query for every store to then compare them in a single graph:

```
select date_time, sum(quantity_sold) as sales_bruxelles
from stores s join transactions t
    on s.store_id = t.store_id and s.store_id = 1
join transaction_products tp
    on t.transaction_id = tp.transaction_id
group by date_time
order by date_time
```



Other graphs that could be useful for data analysis and decision making are:

- average money spent for a transaction for every store / day
- most sold products for each store / day
- less sold products in total

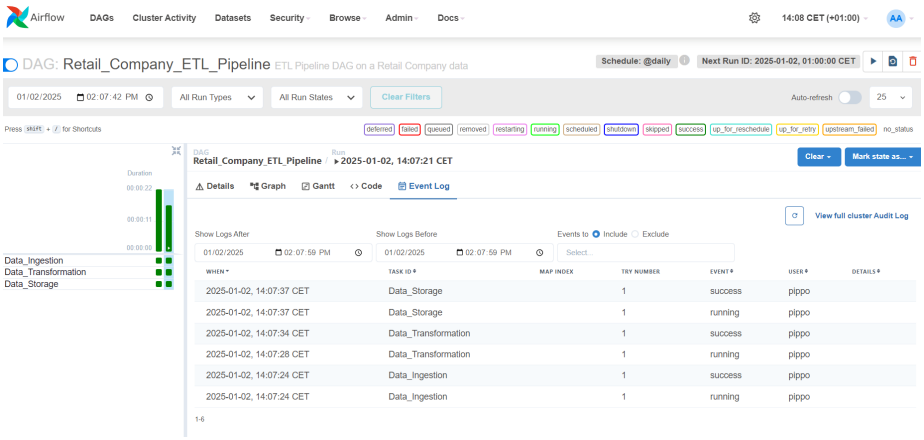
- less sold products for each store / day

7 Implementation of the Workflow

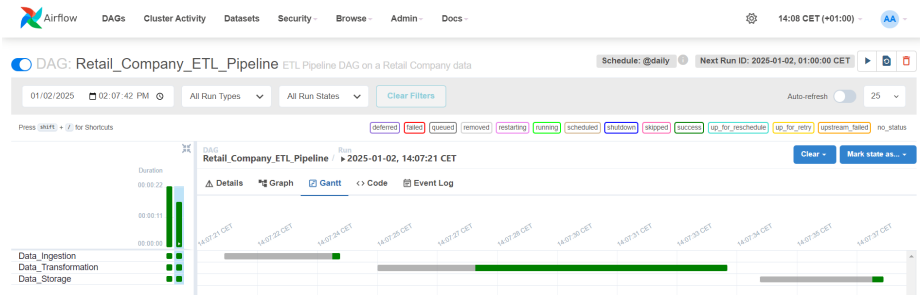
As already mentioned, we use Apache Airflow to orchestrate the workflow. To do that, it works with a DAG that stands for directed acyclic graph. In a DAG, nodes represent the tasks that we have to make, that in this case are data ingestion, data transformation and data storage. Edges represent the dependencies between the nodes, in our case we have that the data transformation depends on the data ingestion and that the data storage depends on the data transformation.
When a dag is triggered, the tasks are gonna be executed one after another, based on their dependencies. Each task has different states in its life cycle, that can be monitored from the Apache Airflow web server to see if the execution is working well and in case of a problem, the cause of it.
To create a DAG we first have to declare its default arguments that in our case are the number of retries, that we set at 3, and the delay between each retry, that in our case is 5 minutes.
After declaring the default arguments we can create an instance of our DAG. To do that we have to specify some important parameter like the start time of the DAG that we set as midnight of the 01/01/2025 and the schedule interval, that we set as daily, which means that from the 1st of December 2024, the dag will automatically run every day at midnight.
Now that we have specify the parameters of the DAG, we can create its tasks. The tasks in this case are data_ingestion, data_transformation and data_storage, which are all PythonOperator since inside every task all we have to do is call a function for the respective task. After the declaration of the task we have to describe in which order we should run the tasks, that in our case is:
`data_ingestion >> data_transformation >> data_storage`
To share information between the tasks, we use Airflow Xcoms: the task that share information push them to Xcoms, while the task that need them pull from them from it.

8 The Data Pipeline in Practice

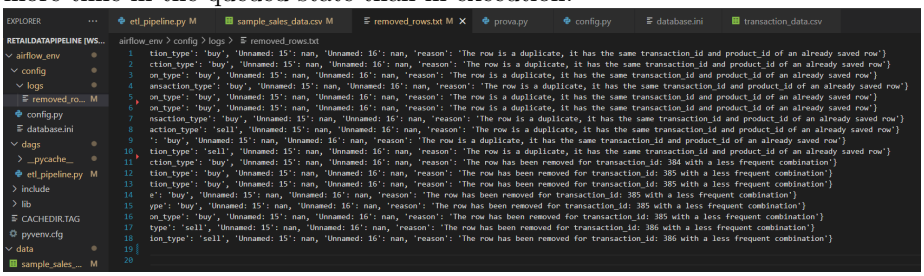
In this chapter we will show you the execution of the data pipeline with different examples of data sets.
We will start with a data set of 1000 rows to see if the program work correctly without giving problems:



We can see that in this case it took 13 seconds to execute the program and that everything worked fine.

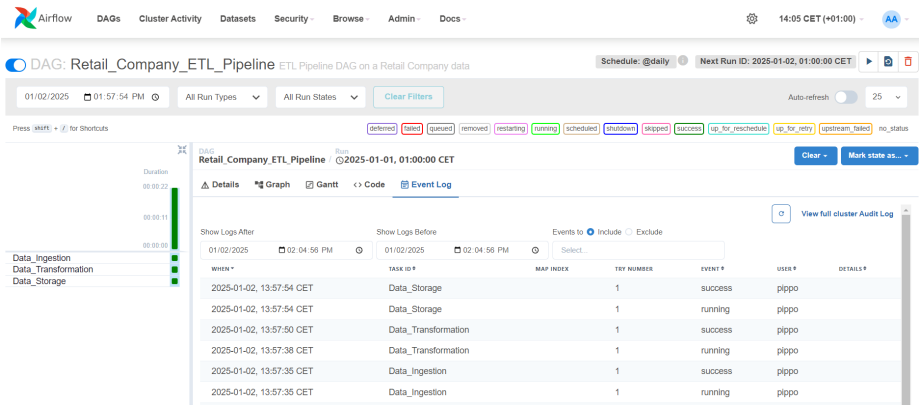


Another thing we can see is that for this amount of data, the tasks where for more time in the queued state than in execution.



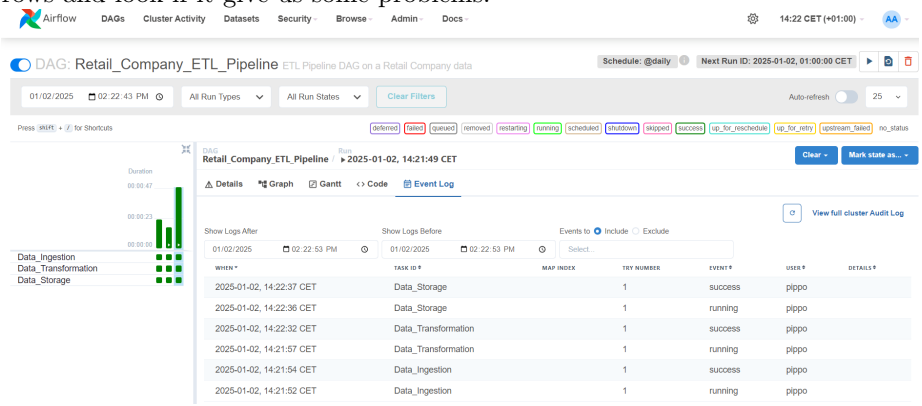
We can also see that in the log file that is called removed_rows.txt, the rows with problems were correctly included with the causal of why they were discarded.

Now we will try execute the program with a data set of 10.000 rows to see if the program can work under higher load of data:

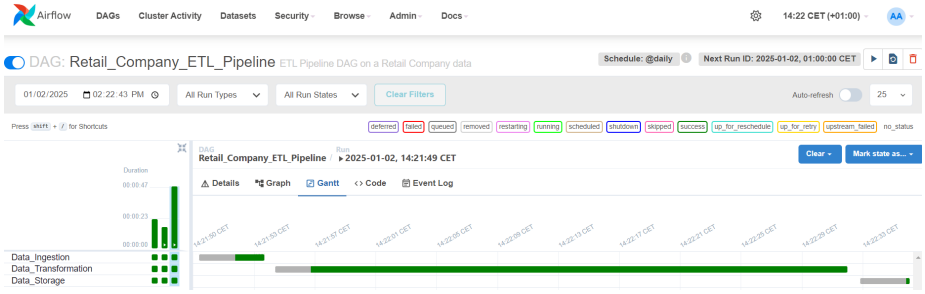


We can see that even this time the execution worked fine with the difference that it took 23 seconds.

As last sample, we will try to execute the program with a data set of 100.000 rows and look if it give us some problems:



We can see that the execution was successful and that this time, it took 47 seconds, that considering it's not executed at the moment but at midnight when the shop is close, we consider it a good time.



We can also see that in this case the time that the tasks spent in the queued state is low compared to the execution time.

9 Possible Improvements

While writing and thinking about how to present the data pipeline some possible improvements became clear. One thing was already mentioned. Right now everything is done locally, for the reason, which were also described before. For a real retail company we would not do that. Since they should have the monetary resources a solution that suits the company would be found for example setting up their own server or paying a hoster.
Right now the database only stores the transaction data, which means that the data like the sold products, the amount that was paid and how it was paid are stored. One next step could be to also store the data of the customer like a customer id, their name, their address and what else could be important. This way it would be possible to categorize customers and do user profiling so the company could send them personalized offer and know their habits.

10 Conclusions

The goal of this project was to build a data pipeline for a chain of retail stores that wants to store their transaction data and analyse it for the company's decision making. This was done by writing Python scripts. In the first step the Data Ingestion was done with Pandas and the data was saved to a data frame. After that the data was transformed by cleaning it, calculating some new fields and seperating it into six data frames to achieve a third normal form. These data frames were then stored in a PostgreSQL database using Psycopg2 to connect to the database, execute and commit SQL statements. After the data was stored the data visualization was possible by using Grafana. Apache Airflow was used to manage the workflow of the Python scripts. Later the data pipeline was tested for some bigger data sets. And in the end some possible improvements were discussed that we would consider in future jobs. We also realized that there are many parts in a data pipeline that should be discussed with the client like how the data cleaning should be done, what kind of setup they want and what type of analysis they need. In conclusion we would build a similar data pipeline again, but with some improvements and the knowledge we gained from this project.