Introduction

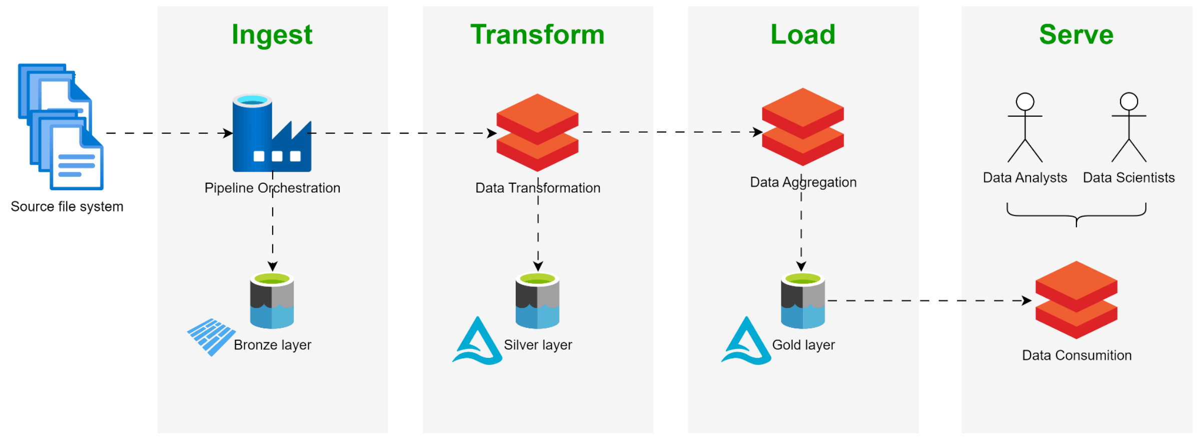
The main purpose of this case study is to create a scalable and easily maintainable data solution that integrates data from multiple sources and delivers it to Data Analysts and Data Scientists teams in a well-structured, readable, and efficient manner.
The solution will focus on building a data lakehouse architecture based on the Medallion data layers Bronze, Silver and Gold, along with implementing a Star Schema in the Gold layer.
The project stack will involve Azure technology, specifically using Azure Data Factory, Azure Data Lake, and Azure Databricks resources.

This solution will include the following resources and corresponding deliverables:

- Azure Data Factory: this resource will be responsible for orchestrating and triggering the process according to the required frequency, in this case, daily. The deliverables will include six JSON files: one for the pipeline, two for the datasets, two for the linked services and one for the trigger, along with screenshots to visually represent them.
- Azure Data Lake: this resource will be responsible for data storage. It will be structured as follows:
    - one container is designated for configuration files, where it will store JSON files containing metadata configurations for data ingestion into Medallion data layers;
    - one container is designated for the Landing zone, where raw CSV format files will be stored to simulate the source system;
    - one container is designated for the Bronze layer, where raw Parquet format files will be stored in folders separated by data source and data ingestion date (e.g., bronze/olist/2024-03-09/customers);
    - one container is designated for the Silver layer, which will store data in delta table format (e.g., silver/customer);
    - one container is designated for the Gold layer, which will store data in delta table format in a star schema (e.g., gold/dim2_customers).

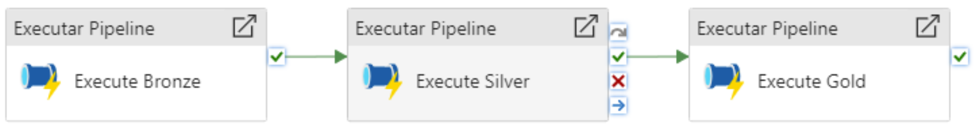
  The deliverables will consist of screenshots representing the file system in each data layer.
- Azure Databricks: this resource will be responsible for data processing across all data layers. The deliverables will be two PySpark scripts: one for cleaning, transforming, and loading data between the Bronze and Silver layers dynamically, and another for aggregating and loading data between the Silver and Gold layers dynamically.

Data Lakehouse architecture



Overall process is built in Azure technology using Azure Data Factory, Azure Data Lake, and Azure Databricks.
The first step is to ingest the source files into Azure Data Lake through Azure Data Factory pipelines.
The second step is to transform and clean data into delta tables through Azure Databricks notebooks.
The third step is to load data into delta tables through Azure Databricks notebooks.
Finally, the Data Analysts and Data Scientists will be able to access the data using Azure Databricks SQL or notebooks.

The ADF is orchestrating the process in the pipeline *Load_olist.json*:

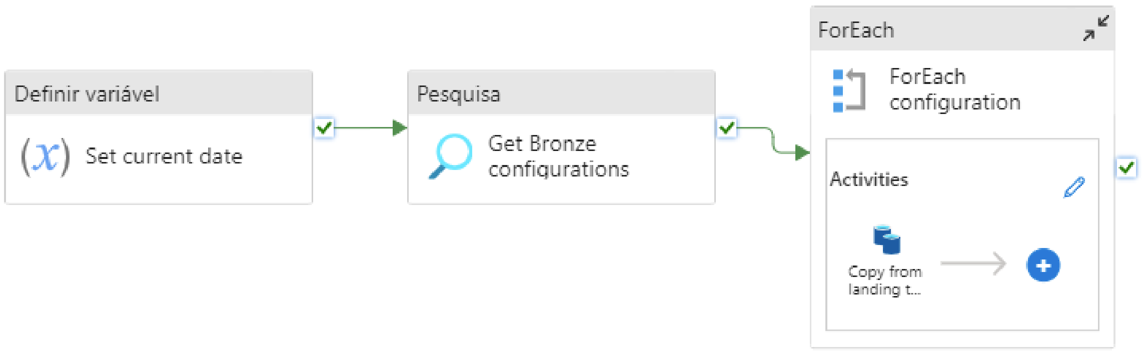


The pipeline executes the main pipelines:
- *Landing2Bronze.json*, where the data is loaded into Bronze layer;
- *Bronze2Silver.json*, where the data is loaded into Silver layer;
- *Silver2Gold.json*, where the data is loaded into Gold layer.

Ingestion

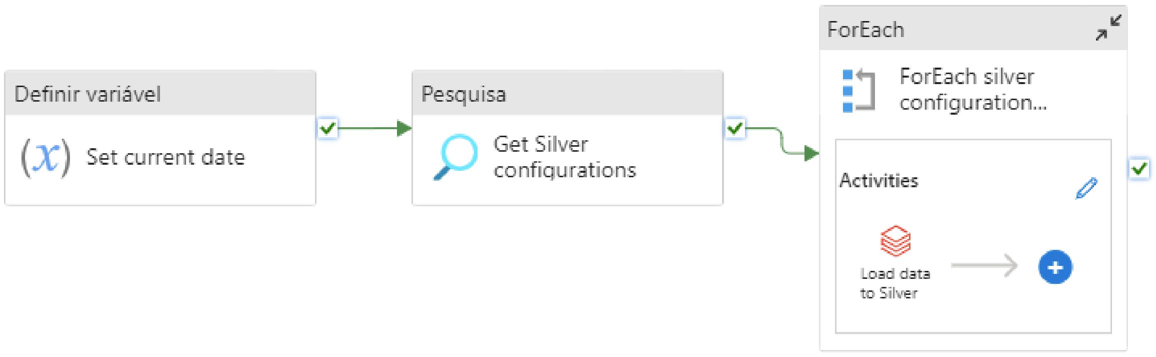
The ingestion process consists of loading data from the Landing layer, which is simulating the source system such as a SFTP, into the Bronze layer.

To support the loading, the configuration file *LoadToBronze.json* contains a mapping between the source file path and file name, and the target file path and file name. This is mainly for the process to know which files need to load and where to load.



Transform

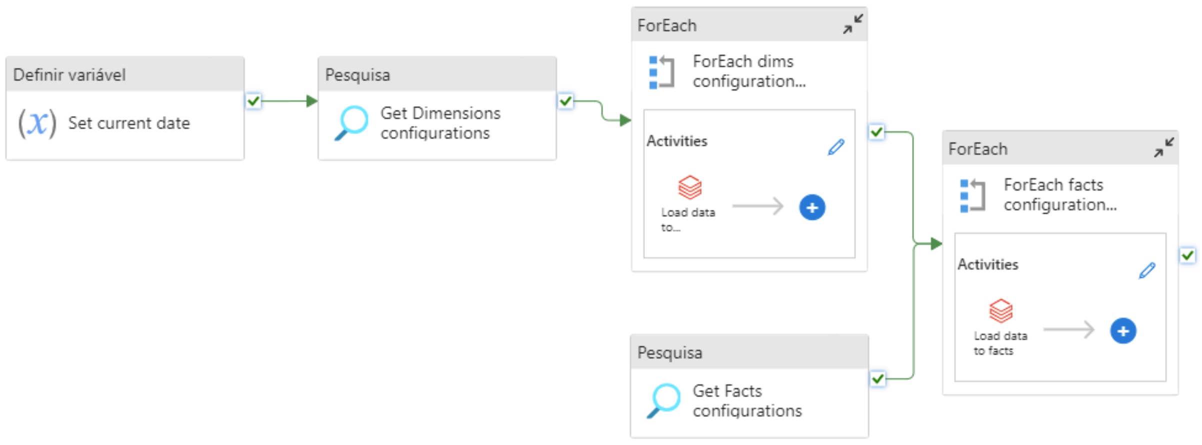
The transformation process is where the data is clean, data types are aligned, and record duplications are discarded. This step has the help of the configuration file *LoadToSilver.json* where it contains the notebook's name to be executed. In this case, we loaded every file into Bronze layer, but since the team only needs a set of those files, Silver layer only contains the necessary tables.



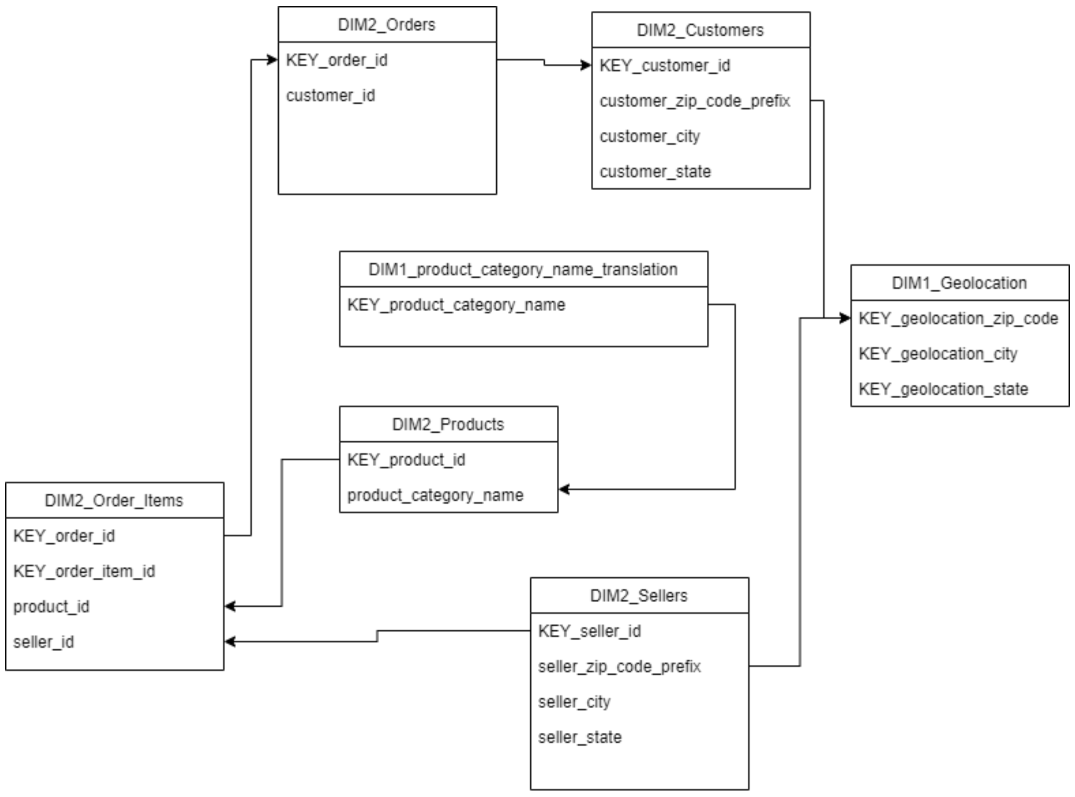
Load

The loading process is where the data is arranged according to dimensional data modeling so the team can access it the most clean and fast way.

Vera Martins, 11/03/2024



Currently, the data model has dimensions, which can be connected as follows:



Questions

1. Does pyspark use lazy or eager loading?

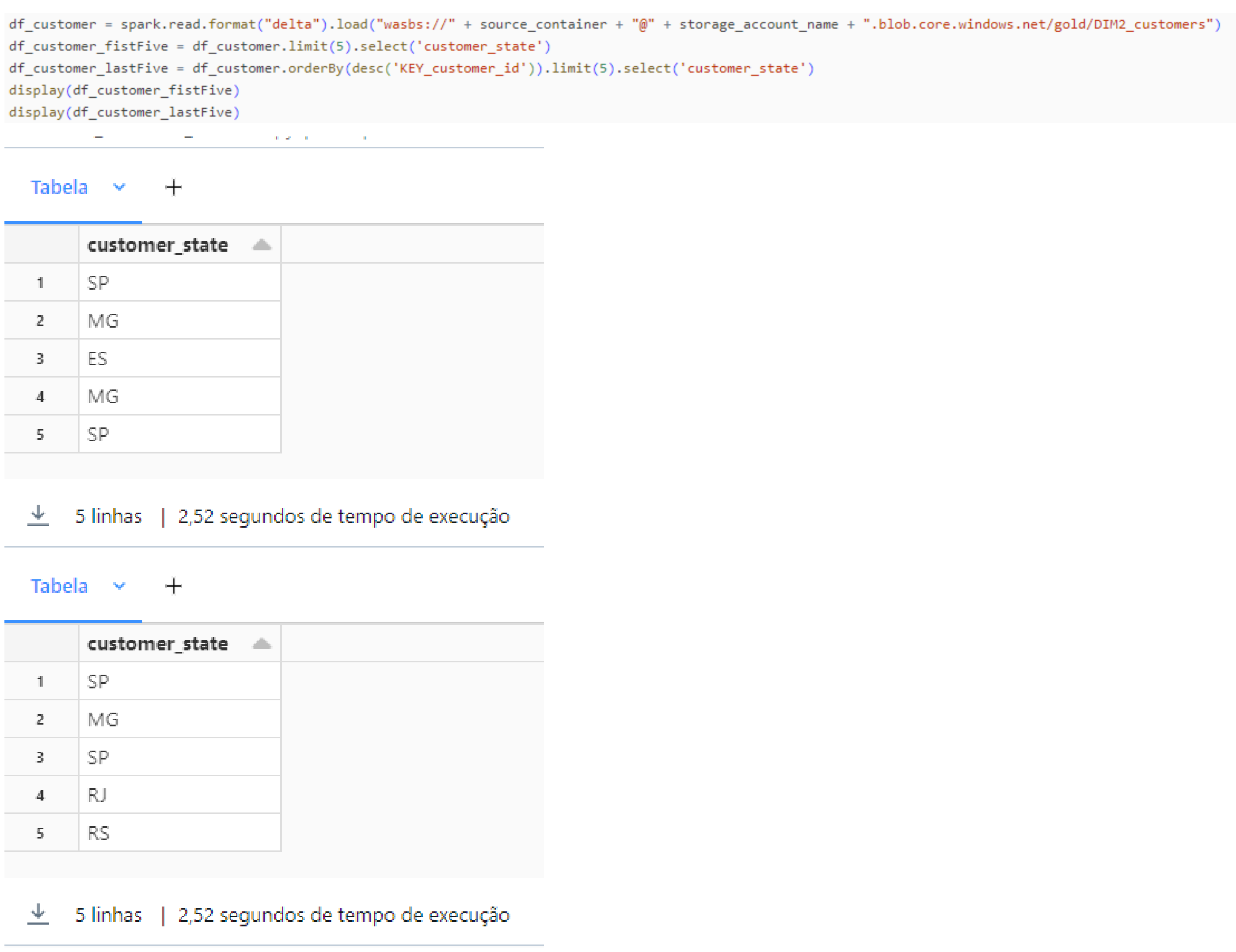
PySpark adopts lazy loading, meaning the transformations are not immediately executed. Instead, the transformations are recorded and executed only when an action is called. This optimizes performance by minimizing unnecessary computations.

2. Is the schema of the tables equal to the one described in Kaggle?

Unfortunately, only the table regarding geolocation has a different schema. To guarantee consistency and compatibility between process and source data, the majority of the tables have the same schema.

3. In which state do the first five and last customers on the customer table live?

Executing the following commands, we have the following result:



```
df_customer = spark.read.format("delta").load("wasbs://" + source_container + "@" + storage_account_name + ".blob.core.windows.net/gold/DIM2_customers")
df_customer_fistFive = df_customer.limit(5).select('customer_state')
df_customer_lastFive = df_customer.orderBy(desc('KEY_customer_id')).limit(5).select('customer_state')
display(df_customer_fistFive)
display(df_customer_lastFive)
```

Tabela

	customer_state
1	SP
2	MG
3	ES
4	MG
5	SP

5 linhas | 2,52 segundos de tempo de execução

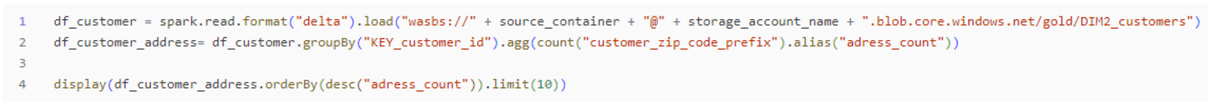
Tabela

	customer_state
1	SP
2	MG
3	SP
4	RJ
5	RS

5 linhas | 2,52 segundos de tempo de execução

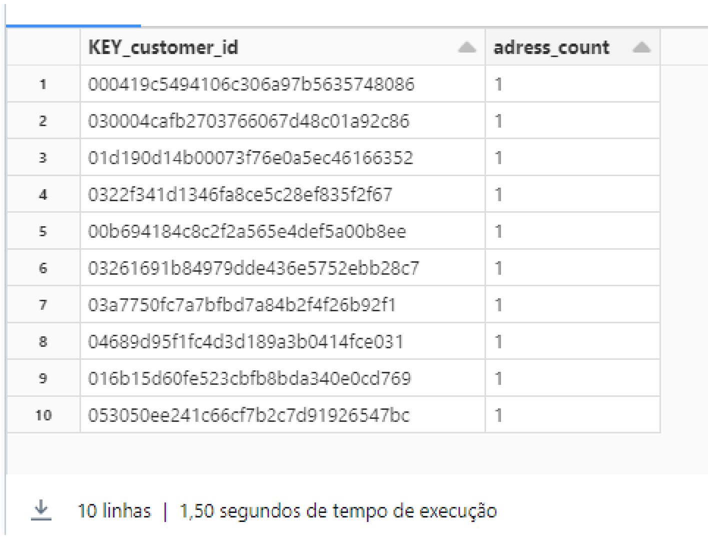
Vera Martins, 11/03/2024

4. Can customer have more than one address? If yes compute the top 10 customers with more addresses.
According to the following command, every customer has one address:



```
df_customer = spark.read.format("delta").load("wasbs://" + source_container + "@" + storage_account_name + ".blob.core.windows.net/gold/DIM2_customers")
df_customer_address= df_customer.groupBy("KEY_customer_id").agg(count("customer_zip_code_prefix").alias("adress_count"))

display(df_customer_address.orderBy(desc("adress_count")).limit(10))
```

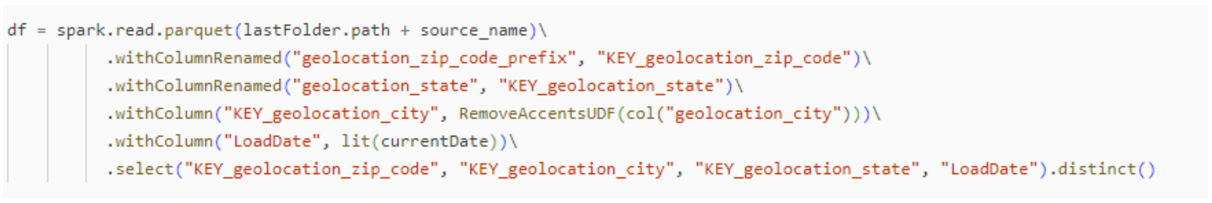


	KEY_customer_id	adress_count
1	000419c5494106c306a97b5635748086	1
2	030004cafb2703766067d48c01a92c86	1
3	01d190d14b00073f76e0a5ec46166352	1
4	0322f341d1346fa8ce5c28ef835f2f67	1
5	00b694184c8c2f2a565e4def5a00b8ee	1
6	03261691b84979dde436e5752ebb28c7	1
7	03a7750fc7a7bfbd7a84b2f4f26b92f1	1
8	04689d95f1fc4d3d189a3b0414fce031	1
9	016b15d60fe523cbfb8bda340e0cd769	1
10	053050ee241c66cf7b2c7d91926547bc	1

10 linhas | 1,50 segundos de tempo de execução

5. How did you clean the addresses table (geolocation) to compute one row per unique combination of postal code, city, state?

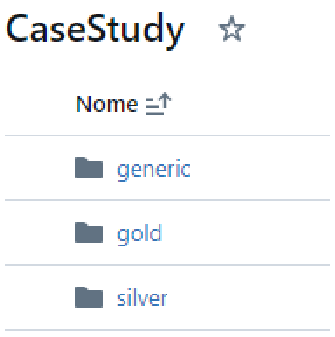
As shown in the following code snip, it was used a user defined function to create a function to clean the geolocation city since it had accents for the the word, for example 'sao paulo' and 'são paulo'.



```
df = spark.read.parquet(lastFolder.path + source_name)\
            .withColumnRenamed("geolocation_zip_code_prefix", "KEY_geolocation_zip_code")\
            .withColumnRenamed("geolocation_state", "KEY_geolocation_state")\
            .withColumn("KEY_geolocation_city", RemoveAccentsUDF(col("geolocation_city")))\
            .withColumn("LoadDate", lit(currentDate))\
            .select("KEY_geolocation_zip_code", "KEY_geolocation_city", "KEY_geolocation_state", "LoadDate").distinct()
```

6. Describe the folder structure you created for this project.

In the databricks side, the structure is:



CaseStudy

Nome

generic

gold

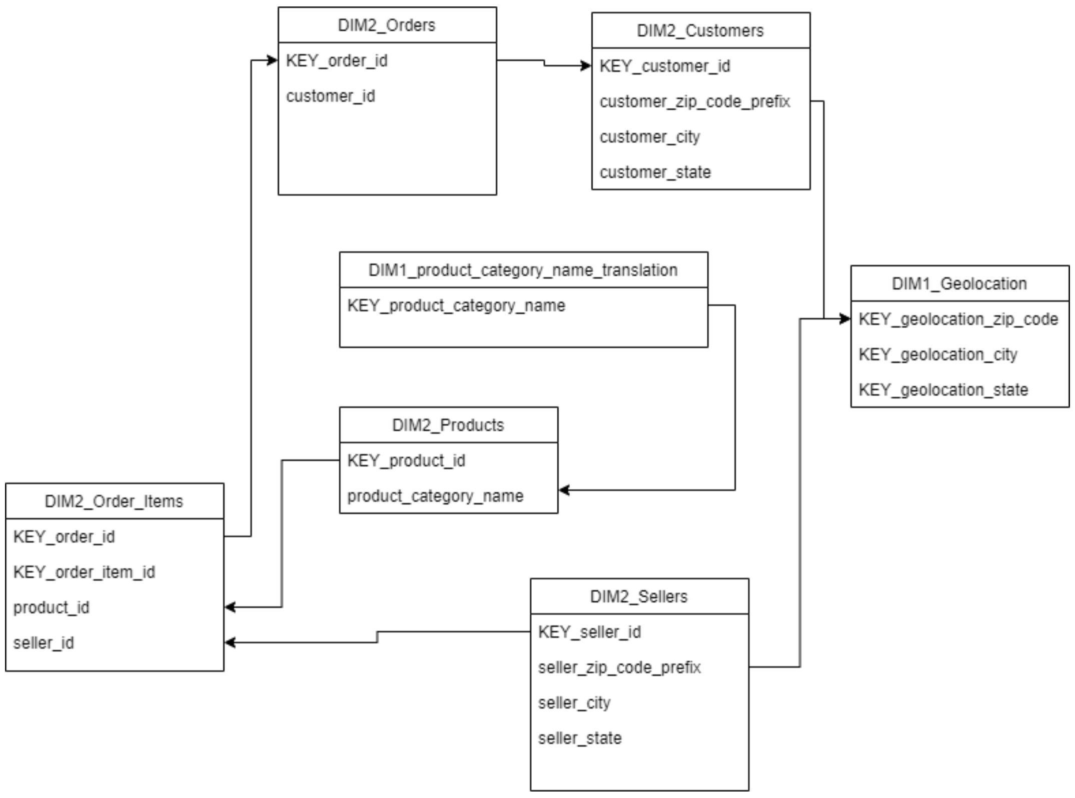
silver

Vera Martins, 11/03/2024

The generic folder contains the scripts generic across the process, such as the functions to load data into dimensions table type 2.
The gold folder contains the script that has the logic to load dynamically the dimensions tables to Gold layer.
The silver folder contains the scripts that has the logic to load tables into Silver layer.

7. What is the schema of your “database” after the ETL pipeline? Make a diagram.

In the follow diagram there is the essential columns for the tables.



8. In terms of CI/CD what would you do to move your code from a development environment to a production environment?

For this process it’s needed a CI/CD pipeline for each resource, Azure Data Factory, Azure Data Lake and Azure Databricks.
For the ADF, after the pull request to the main branch, the ARM templates need to be published in the main branch. After the successful publish, the CI/CD pipeline can be run.
For the Azure Data Lake and Azure Databricks, after the pull request to the main branch, the CI/CD pipeline can be run.

Conclusion

Overall the process is ready to dynamically load a CSV file into a delta table, making the process robust and efficient.
Also, by making the process dynamic, it makes a smooth transition to production.

Some improvements came be made:

- Add logging process tracking, to ensure that the process runned successfully;
- Add a fact table.

Vera Martins, 11/03/2024