INSTITUTE FOR ADVANCED COMPUTING AND SOFTWARE DEVELOPMENT (IACSD) AKURDI, PUNE

Documentation On

" ETL PIPELINE FOR RETAIL DATA VISUALIZATION USING BIG DATA TECHNOLOGIES "

PG-DBDA MARCH 2023

Submitted By:

Group No: 02

Roll No.	Name:
233525	Manish Katekar
233551	Jayesh Tarvatkar

Mr.Abhijit Nagargoje — **Project Guide**

Mr.Rohit Puranik — **Centre Coordinator**

ABSTRACT

This thesis presents the development of an ETL (Extract, Transform, Load) pipeline utilizing Apache Spark, aimed at streamlining the intricate procedure of aggregating, refining, and structuring retail data for subsequent visualization. This comprehensive pipeline efficiently extracts data from diverse origins, encompassing CSV files and relational databases.

Through the utilization of PySpark, the extracted data undergoes meticulous cleansing and transformation, rendering it primed for visualization. The resulting pipeline not only bestows the organization with time and resource efficiency by automating the ETL workflow, but also empowers the extraction of invaluable business insights through the discerning lens of data-driven visualizations.

ACKNOWLEDGEMENT

This project **" ETL Pipeline for Retail Data Visualization Using Big Data Technologies "** was a great learning experience for us and we are submitting this work to INSTITUTE FOR ADVANCED COMPUTING AND SOFTWARE DEVELOPMENT AKURDI (IACSD), PUNE.

We all are very glad to mention the name of **Mr. Abhijit Nagargoje** sir for his valuable guidance to work on this project. His guidance and support helped us to overcome various obstacles and intricacies during the course of project work.
.

We extend our sincere gratitude to **Dr. Shantanu Pathak**, the Course Coordinator of PG-DBDA, **Mrs Priti Take** Madam and **Mrs Priyanka Bhor** Madam, for their invaluable guidance and unwavering support throughout the duration of our Post Graduate Diploma in Big Data Analytics (PG -DBDA) at IACSD, Pune.

We would like to express our deepest gratitude to **Mr. Rohit Puranik** (Centre Coordinator), whose unwavering support and exceptional coordination ensured the provision of all necessary resources, including essential hardware, internet facilities, and additional lab hours. His contributions were pivotal in enabling us to successfully complete our project and navigate through the entire course journey here at IACSD, Pune, until the very last day

Table of Contents

LIST OF FIGURES

INTRODUCTION

1.1 AIMS AND OBJECTIVES

- AIM

The primary aim of this thesis is to develop an automated ETL pipeline using Apache Spark for the purpose of collecting, cleansing, and preparing retail data, ultimately facilitating seamless data visualization.

- OBJECTIVES

1. ETL Pipeline Development: Construct a robust ETL pipeline utilizing Apache Spark that efficiently extracts data from various sources, including .xlsx files and relational databases.

2. Data Cleansing and Preparation: Implement data cleaning and transformation processes using PySpark to ensure the collected data is accurate, consistent, and suitable for visualization.

3. Automation: Design the pipeline to operate autonomously, minimizing manual intervention and expediting the process of data aggregation and preparation.

4. Data Visualization: Integrate the prepared data with data visualization tools to generate insightful graphical representations that aid in comprehending the retail business's performance and trends.

5. Time and Resource Efficiency: Demonstrate that the automated pipeline significantly reduces the time and resources required for data preparation and visualization tasks compared to traditional manual methods.

6. Insight Generation: Showcase how the automated pipeline and resulting visualizations contribute to the identification of actionable insights and informed decision-making within the company's business operations.

7. Scalability and Flexibility: Ensure that the developed pipeline can handle varying data volumes and adapt to evolving data sources and formats, showcasing its scalability and flexibility.

8. Documentation and Guidelines: Provide comprehensive documentation and guidelines for the implemented ETL pipeline, enabling easy replication and adoption in similar contexts.

By accomplishing these objectives, the thesis aims to deliver a highly effective ETL pipeline that not only simplifies the complex data preparation process but also empowers the organization with timely and meaningful insights derived from data visualizations.

Overall Description

2.1 Workflow of Project

The diagram below shows workflow of this project.

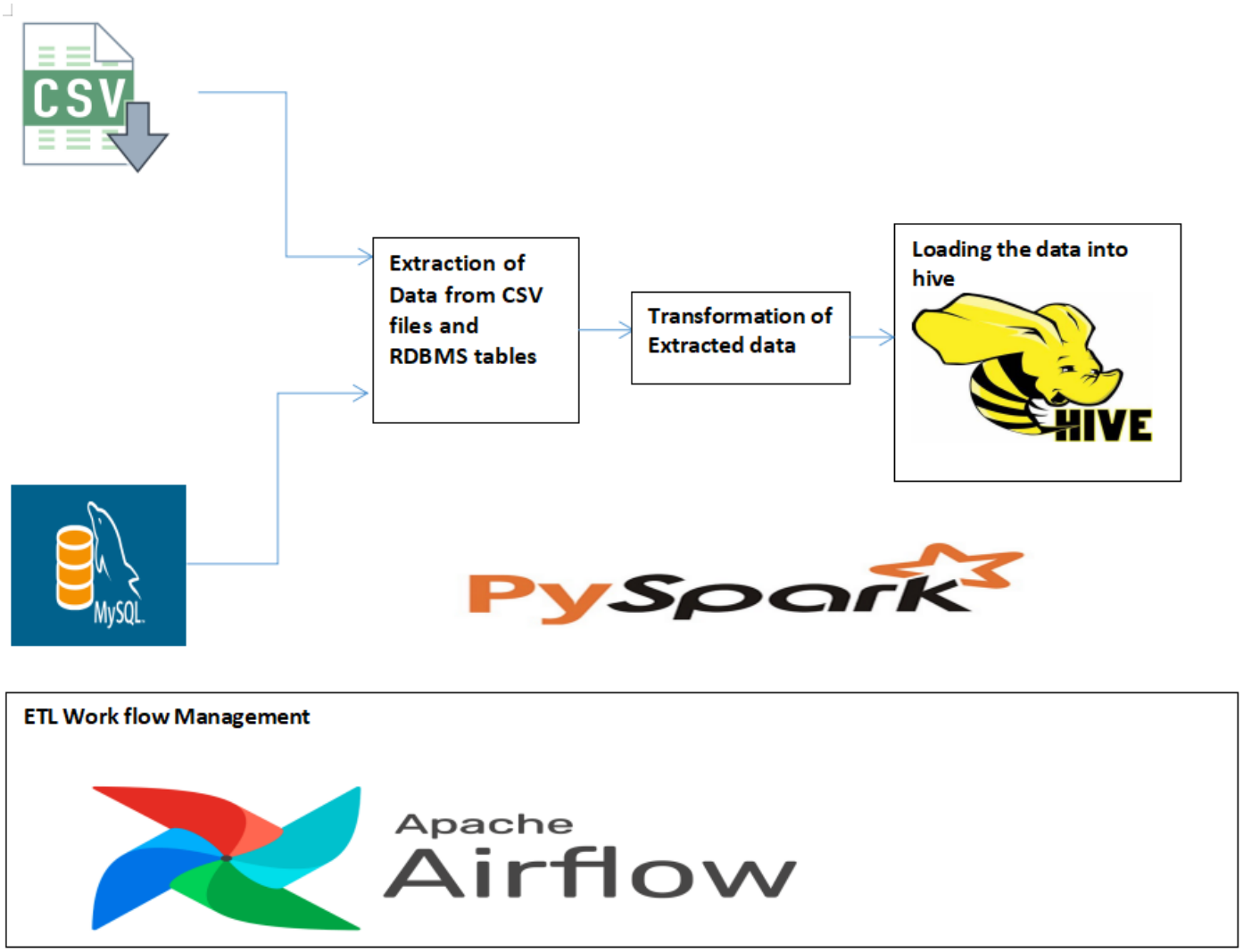

Figure 1: Workflow Diagram

2.2 Setting up the environment and importing the necessary libraries and modules:

```
import os
import sys
os.environ["SPARK_HOME"] = "/home/talentum/spark"
os.environ["PYLIB"] = os.environ["SPARK_HOME"] + "/python/lib"
os.environ["PYSPARK_PYTHON"] = "/usr/bin/python3.6"
os.environ["PYSPARK_DRIVER_PYTHON"] = "/usr/bin/python3"
sys.path.insert(0, os.environ["PYLIB"] +"/py4j-0.10.7-src.zip")
sys.path.insert(0, os.environ["PYLIB"] +"/pyspark.zip")

from pyspark.sql import SparkSession
from pyspark.sql.functions import udf,trim
from pyspark.sql.types import StringType
from pyspark.sql.functions import col,when,create_map, lit
from itertools import chain
```

```
import pyspark.sql.functions as F
```

This code sets up the environment and imports the necessary libraries and modules to work with PySpark and perform various data transformation and analysis tasks. The imported functions, classes, and modules will be used later in your code to perform Spark operations.

Environment Setup:

os.environ["SPARK_HOME"]: This line sets the environment variable SPARK_HOME to the path where Spark is installed on your system. This is used to specify the location of the Spark installation.

os.environ["PYLIB"]: This line sets the environment variable PYLIB to the directory where the Python libraries for Spark are located.

os.environ["PYSPARK_PYTHON"]: This line sets the environment variable PYSPARK_PYTHON to the path of the Python interpreter you want to use for running PySpark.
os.environ["PYSPARK_DRIVER_PYTHON"]: This line sets the environment variable PYSPARK_DRIVER_PYTHON to the path of the Python interpreter you want to use for the PySpark driver program (the program that submits Spark jobs).

sys.path.insert(...): These lines insert paths to the required PySpark libraries into the Python path.

Import Statements:

from pyspark.sql import SparkSession: This imports the SparkSession class, which is the entry point to using Spark's DataFrame and SQL API.

from pyspark.sql.functions import ...: These import various functions and classes from the pyspark.sql.functions module, which provides functions for data transformation and analysis.

from pyspark.sql.types import StringType: This imports the StringType class, which represents the data type of a string column in a Spark DataFrame.

from itertools import chain: This imports the chain function from the itertools module. chain is used to concatenate the contents of multiple iterators.

2.3 Extraction of Data :

We have the data related to products, order_items and orders in comma separated values(.csv) files in local file system. Also the data related to customers and payments resided in Relational database management system(MySQL) tables.
By using PySpark APIs, we are extracting the above data located in different locations in operational systems and converted each one of them into dataframe, so finally we got five dataframes.
Following are the steps to perform extraction :

1) Create SparkSession object :

SparkSession object created which acts as a entry point to programming Spark with the Dataset and DataFrame API. To create a Spark session, we have to use **SparkSession.builder** attribute. Also Sets a name for the application, which will be shown in the Spark web UI.

```
    def create_spark_session(self):
        # A class level variable
```

```
        self.spark = SparkSession.builder \
            .appName("ETL_pipeline") \
            .enableHiveSupport().getOrCreate()
```

2) Extracting data from RDBMS tables :

In the provided code snippet, following variables are assigned as follows :
url: This is the URL that specifies the connection details for the MySQL database. The URL format includes the protocol (jdbc:mysql), the host (127.0.0.1), the port (3306), the database name (test), and some additional connection parameters (useSSL=false and allowPublicKeyRetrieval=true).
driver: This variable holds the fully qualified class name of the MySQL JDBC driver. The JDBC driver is essential for establishing a connection to the MySQL database.
user: This variable stores the username that will be used to authenticate and connect to the MySQL database.
password: This variable stores the password associated with the specified username for authentication.

In remaining code section which is a part of a Python class ‘Ingest’ that interacts with Apache Spark to ingest data from a MySQL database table named "project.customers". The class has a constructor (__init__) that takes a Spark session as an argument. It also contains a method ingest_table_customer that performs the data ingestion from the mentioned table and returns dataframe.

Similarly done for extracting payments table from database.

```
class Ingest:

    url = "jdbc:mysql://127.0.0.1:3306/test?useSSL=false&allowPublicKeyRetrieval=true"
    driver = "com.mysql.jdbc.Driver"
    user = "bigdata"
    password = "Bigdata@123"

    def __init__(self, spark):
        # A class level variable
        self.spark = spark
    def ingest_table_customer(self):
        print("Ingesting from customer table")
        df_customers1 =  self.spark.read\
            .format("jdbc")\
            .option("driver", self.driver)\
            .option("url", self.url)\
            .option("user", self.user)\
            .option("password", self.password)\
            .option("dbtable", "project.customers")\
            .load()
        df_customers=df_customers1.exceptAll(df_customers1.limit(1))
        return df_customers
```

3) Extracting data from csv files :
The method ingest_orders within the same class Ingest is designed to ingest data from a CSV file located at the path "file:///home/talentum/project/orders.csv" into a Spark DataFrame.

self.spark.read: This line starts the process of reading data using the Spark session.
.format("csv"): Specifies the data format to be used for reading, which is CSV in this case.
.option("header", "true"): Specifies that the first row of the CSV file contains the header, indicating column names.
.option("inferSchema", "true"): Specifies that Spark should attempt to infer the schema of the CSV file based on the data types present.

.load("file:///home/talentum/project/orders.csv"): Loads the CSV data from the specified file path into a DataFrame named df_orders.
return df_orders: The method returns the DataFrame df_orders after the CSV data has been ingested.

```
    def ingest_orders(self):
        print("Ingesting from orders.csv")
        df_orders = self.spark.read \
            .format("csv") \
            .option("header", "true") \
            .option("inferSchema", "true") \
            .load("file:///home/talentum/project/orders.csv")

        return df_orders
```

Similarly done to extract data from other csv files order_items, products.

2.4 Transformation of Data :
The class Transform have a constructor (__init__) that takes a Spark session as an argument and a method transform_data that performs the data transformation.

1) Trim :
The overall effect of this code is to remove leading and trailing whitespace from the values in the "customer_state" column of the df_customers DataFrame and store the result in a new DataFrame df_c.

```
def trim(s):
    return s.strip()

udf_trim = udf(trim, StringType())
df_c = df_customers.withColumn("customer_state", udf_trim("customer_state"))
```

2) Joining the dataframes : In this transformation, performing a series of joins on different DataFrames that previously obtained and then selecting specific columns to create a new DataFrame named retail_inner_join.
The join Operations:
df_c.alias("t1").join(...): This renames the df_c DataFrame as t1 and performs a join with the df_orders DataFrame on the condition that the "customer_id" column matches between the two DataFrames.
The subsequent .join operations similarly continue chaining joins on different DataFrames (df_payments, df_order_items, df_products) based on certain column conditions.
Column Selection:
After the joins, the .select(...) statement is used to choose specific columns from the resulting joined DataFrame retail_inner_join. The selected columns include customer-related information (customer_id, customer_city, customer_state), order-related information (order_id, order_status, etc.), payment-related information, item-related information, and product category information.

```
retail_inner_join = df_c.alias("t1").join(df_orders.alias("t2"), col("t1.customer_id") == col("t2.customer_id")) \
            .join(df_payments.alias("t3"), col("t2.order_id") == col("t3.order_id")) \
            .join(df_order_items.alias("t4"), col("t3.order_id") == col("t4.order_id")) \
            .join(df_products.alias("t5"), col("t4.product_id") == col("t5.product_id")) \
            .select(
                col("t1.customer_id"),
                col("t1.customer_city"),
                col("t1.customer_state"),
                col("t2.order_id"),
                col("t2.order_status"),
                col("t2.order_approved_at"),
                col("t2.order_delivered_timestamp"),
```

```
                col("t3.payment_sequential"),
                col("t3.payment_type"),
                col("t3.payment_value"),
                col("t4.order_item_id"),
                col("t4.product_id"),
                col("t4.seller_id"),
                col("t5.product_category_name"))
```

3) Translating the columns and dropping unnecessary columns :

```
        # Convert each item of dictionary to map type
        mapping_expr = create_map([lit(x) for x in chain(*self.dictionary.items())])

        # Create a new column by calling the function to map the values
        df_state = retail_inner_join.withColumn("state",mapping_expr[col("customer_state")])

        mapping_expr1 = create_map([lit(y) for y in chain(*self.dictionary_product.items())])
        df_category = df_state.withColumn("category",mapping_expr1[col("product_category_name")])
        df_final=df_category.drop("customer_state", "order_approved_at",
"order_delivered_timestamp","product_category_name")

        return df_final
```

Here's what this code does:

mapping_expr = create_map(...): This creates a mapping expression using the create_map function. The chain(*self.dictionary.items()) part is used to flatten the dictionary and create a list of alternating keys and values. These are then converted to Spark lit literals, which are used as the arguments for the create_map function. The resulting mapping_expr is a map that will be used to replace values in the "customer_state" column.

df_state = retail_inner_join.withColumn(...): This line adds a new column named "state" to the retail_inner_join DataFrame. The new column is created by applying the mapping_expr to the values in the "customer_state" column. The col("customer_state") refers to the column you want to transform. Now we got "state" column having full form of states so will reduce the ambiguity.

mapping_expr1 = create_map(...): Similar to the first mapping expression, this creates a mapping expression for the "product_category_name" column based on the self.dictionary_product dictionary.

df_category = df_state.withColumn(...): This line adds another new column named "category" to the df_state DataFrame (which already includes the "state" column). The new column is created by applying the mapping_expr1 to the values in the "product_category_name" column so as to get reduced the categories that will help in analysis.

df_final = df_category.drop(...): This line creates the final DataFrame by dropping columns "customer_state", "order_approved_at", "order_delivered_timestamp", and "product_category_name" from the df_category DataFrame.

Finally, the method returns the df_final DataFrame, which represents the transformed and cleaned data.

2.5 Loading the Data into Data warehouse:

Here we are defining a class named Persist with a method persist_data that is used to save a DataFrame into a Hive external table named "retail".

```
#Loading into hive
```

```
class Persist:
    def __init__(self, spark):
        self.spark = spark

    def persist_data(self, df_final):
        print("Loading into hive external table - retail")
        df_final.write.mode('overwrite').option("path","/home/talentum/retail_data").saveAsTable("retail")
```

def persist_data(self, df_final): This is a method within the Persist class that is intended to persist a DataFrame into a Hive external table. It takes a DataFrame df_final as a parameter.

df_final.write.mode('overwrite').option("path", "/home/talentum/retail_data").saveAsTable("retail"):

This line writes the df_final DataFrame into a Hive external table named "retail". The data is written with an overwrite mode, meaning any existing data in the table will be replaced. The option("path", "/home/talentum/retail_data") specifies the HDFS path where the table data will be stored. Finally, saveAsTable("retail") creates the external table "retail" in Hive with the data from the DataFrame.

```
talentum@talentum-virtual-machine:~$ beeline -u jdbc:hive2://localhost:10000 -n hiveuser -p Hive@123 --incremental=true
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/talentum/hive/lib/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/talentum/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Connecting to jdbc:hive2://localhost:10000
Connected to: Apache Hive (version 2.3.6)
Driver: Hive JDBC (version 2.3.6)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 2.3.6 by Apache Hive
0: jdbc:hive2://localhost:10000>
0: jdbc:hive2://localhost:10000> show tables;
+------------+
|  tab_name  |
+------------+
| retail     |
| test_table |
+------------+
2 rows selected (2.953 seconds)
0: jdbc:hive2://localhost:10000> select * from retail limit 2;
+--------------------------------+----------------------+--------------------------------+--------------------+--------------------------+-------------
---------+----------------------+----------------------+--------------------------------+--------------------------------+-------------+-------------
----+
|       retail.customer_id       | retail.customer_city |         retail.order_id        | retail.order_status | retail.payment_sequential | retail.paym
ent_type | retail.payment_value | retail.order_item_id |       retail.product_id        |        retail.seller_id        | retail.state | retail.catego
ry |
+--------------------------------+----------------------+--------------------------------+--------------------+--------------------------+-------------
---------+----------------------+----------------------+--------------------------------+--------------------------------+-------------+-------------
----+
| e8d87ee946600f7753579a074fbd2d5d | mesquita             | 014405982914c2cde2796ddcf0b8703d | delivered          | 1                        | credit_card
         | 78.43                | 2                    | e95ee6822b66ac6058e2e4aff656071a | a17f621c590ea0fab3d5d883e1630ec6 | NULL         | Toys
    |
| e8d87ee946600f7753579a074fbd2d5d | mesquita             | 014405982914c2cde2796ddcf0b8703d | delivered          | 1                        | credit_card
```

Figure 2: retail table in Hive

2.6 Querying the Hive Table :

Apache Hive is a data warehouse software project built on top of Apache Hadoop for providing data query and analysis.

```
1 row selected (41.425 seconds)
0: jdbc:hive2://localhost:10000> SELECT retail.category, COUNT(DISTINCT retail.order_id) count_orders
. . . . . .> FROM retail
. . . . . .> GROUP BY retail.category;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
+--------------------+--------------+
|   retail.category  | count_orders |
+--------------------+--------------+
| Art                | 16           |
| Arts and Crafts    | 5            |
| Automotive         | 1015         |
| Baby               | 728          |
| Books              | 208          |
| Business           | 108          |
| Construction Tools | 320          |
| Cool Stuff         | 953          |
| Electronics        | 2852         |
| Entertainment      | 226          |
| Fashion            | 536          |
| Food and Drink     | 167          |
| Furniture          | 2422         |
| Garden             | 918          |
| Gifts              | 1490         |
| Health and Beauty  | 3638         |
| Home Appliances    | 574          |
| Home Improvement   | 203          |
| Home and Garden    | 11           |
| Homewares          | 2343         |
| Housewares         | 1477         |
| Musical Instruments| 181          |
| Others             | 478          |
| Pets               | 466          |
| Security           | 43           |
| Sports and Leisure | 2089         |
| Stationery         | 576          |
| Telephony          | 1149         |
| Toys               | 74604        |
| Travel             | 198          |
+--------------------+--------------+
30 rows selected (43.374 seconds)
0: jdbc:hive2://localhost:10000>
```

Figure 3: category wise order counts query

```
0: jdbc:hive2://localhost:10000> select avg(retail.payment_value) Avg_order_value from retail;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or usi
ng Hive 1.X releases.
+-----------------+
| avg_order_value |
+-----------------+
| 121.613777      |
+-----------------+
1 row selected (41.425 seconds)
0: jdbc:hive2://localhost:10000>
```

Figure 4: Average order value query

```
0: jdbc:hive2://localhost:10000> select count(distinct retail.category) total_product_categories from retail ;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or usi
ng Hive 1.X releases.
+-------------------------+
| total_product_categories |
+-------------------------+
| 30                      |
+-------------------------+
1 row selected (35.735 seconds)
0: jdbc:hive2://localhost:10000>
```

Figure 5: Total product categories query

```
0: jdbc:hive2://localhost:10000> select sum(retail.payment_value) Total_revenue from retail;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or usi
ng Hive 1.X releases.
+---------------+
| total_revenue |
+---------------+
| 45884999.72   |
+---------------+
1 row selected (35.178 seconds)
0: jdbc:hive2://localhost:10000>
```

Figure 6: Total revenue query

```
0: jdbc:hive2://localhost:10000> select count(distinct retail.order_id) total_orders from retail ;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or usi
ng Hive 1.X releases.
+--------------+
| total_orders |
+--------------+
| 98666        |
+--------------+
1 row selected (41.97 seconds)
0: jdbc:hive2://localhost:10000>
```

Figure 7: Total no. of orders query

```
0: jdbc:hive2://localhost:10000> select count(distinct retail.customer_city) total_cities from retail ;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or usi
ng Hive 1.X releases.
+--------------+
| total_cities |
+--------------+
| 4110         |
+--------------+
1 row selected (43.629 seconds)
0: jdbc:hive2://localhost:10000>
```

Figure 8: Total no. of cities query

2.7 Tableau Dashboard :

Same transformed dataset imported in BI tool (Tableau) and made a dashboard from it as given below,

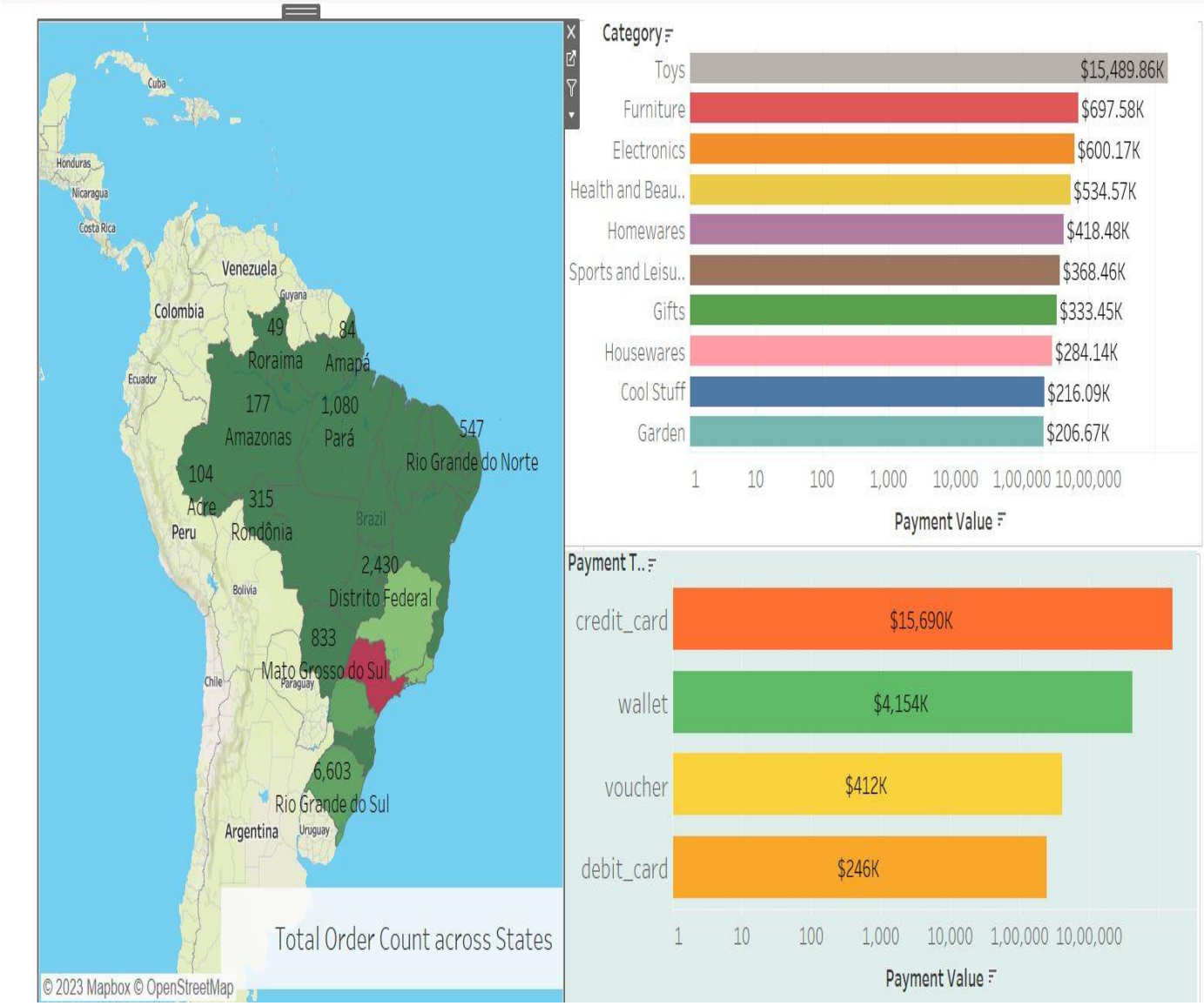

Figure 9: Dashboard showing state wise order count, payment value across payment methods and product category.

2.8 Workflow Orchestration :

To create and run a DAG (Directed Acyclic Graph) in Apache Airflow, which is an open-source platform to programmatically author, schedule, and monitor workflows, you'll need to follow these steps:

1) Install Apache Airflow.

2) Set configuration for the folder where your dag file and main program file lies.

3) Go to that folder

cd /home/talentum/Desktop/airflow-tutorial : Change the current directory to the airflow-tutorial directory on your Desktop.

4) Put your created dag file and main program files there.

5) source ~/unset_jupyter.sh: Source (execute) the script named unset_jupyter.sh to unset certain Jupyter-related environment variables.

6) unset PYTHONPATH: Unset the PYTHONPATH environment variable, which affects how Python searches for modules and packages.

7) conda activate airflow-tutorial: Activate the Conda environment named airflow-tutorial, isolating your Python environment to match the dependencies required for your Airflow setup.

8) airflow initdb: Initialize the Airflow metadata database. This sets up the necessary structure to store information about your DAGs, tasks, and their metadata.

9) airflow scheduler: Start the Airflow Scheduler. The scheduler monitors your DAGs and triggers tasks based on their defined schedules and dependencies.

10) airflow webserver: Start the Airflow Web Server. The web server provides a user interface where you can manage and monitor your DAGs, task runs, and view logs.

11) After running these commands, you should be able to access the Airflow Web UI by opening a web browser and navigating to the appropriate URL (usually http://localhost:8080). From there, you can interact with your DAGs, trigger runs, monitor task statuses, and view logs.

Airflow DAG Code :

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.contrib.operators.spark_submit_operator import SparkSubmitOperator
import airflow
from airflow.models import DAG
from airflow.operators.python_operator import PythonOperator
from data_pipeline1 import *

def etl():
    pipeline = Pipeline()
    pipeline.create_spark_session()
    pipeline.run_pipeline()



dag = DAG(
    dag_id="etl_project",
    description="etl",
    start_date=airflow.utils.dates.days_ago(5),
    schedule_interval=None,
)

bashop = BashOperator(task_id = "print",
                              bash_command="echo 'PGDBDA'",
                              dag=dag
                              )

##define the etl task
etl_task = PythonOperator(task_id="etl_task",
                          python_callable = etl,
                          dag=dag)



bashop >> etl_task
```

Figure 10: DAG Code

import airflow.utils.dates:
This imports the dates module from the Airflow library, which provides functions for working with dates and times.

from airflow import DAG:
This imports the DAG class from the Airflow library, which is used to define DAGs.

from airflow.operators.bash_operator import BashOperator:
This imports the BashOperator class from the Airflow library, which is used to create BashOperator tasks.

from airflow.operators.python_operator import PythonOperator:
This imports the PythonOperator class from the Airflow library, which is used to create PythonOperator tasks.

from data_pipeline1 import *: This imports all of the functions from the data_pipeline1 module.

dag = DAG(dag_id="etl_project", description="etl", start_date=airflow.utils.dates.days_ago(5), schedule_interval=None):
This defines the dag object. The dag_id parameter specifies the unique identifier for the DAG. The description parameter provides a brief description of the DAG. The start_date parameter specifies the date

on which the DAG should start executing. The schedule_interval parameter specifies the frequency at which the DAG should be executed.

bashop = BashOperator(task_id = "print", bash_command="echo 'PGDBDA'", dag=dag):
This defines the bashop task. The task_id parameter specifies the unique identifier for the task. The bash_command parameter specifies the Bash command that should be executed by the task. The dag parameter specifies the DAG to which the task belongs.

etl_task = PythonOperator(task_id="etl_task", python_callable = etl, dag=dag): This defines the etl_task task. The task_id parameter specifies the unique identifier for the task. The python_callable parameter specifies the Python function that should be executed by the task. The dag parameter specifies the DAG to which the task belongs.

bashop >> etl_task:
This specifies the dependency between the bashop and etl_task tasks. The bashop task must be completed successfully before the etl_task can be executed.

Airflow UI :

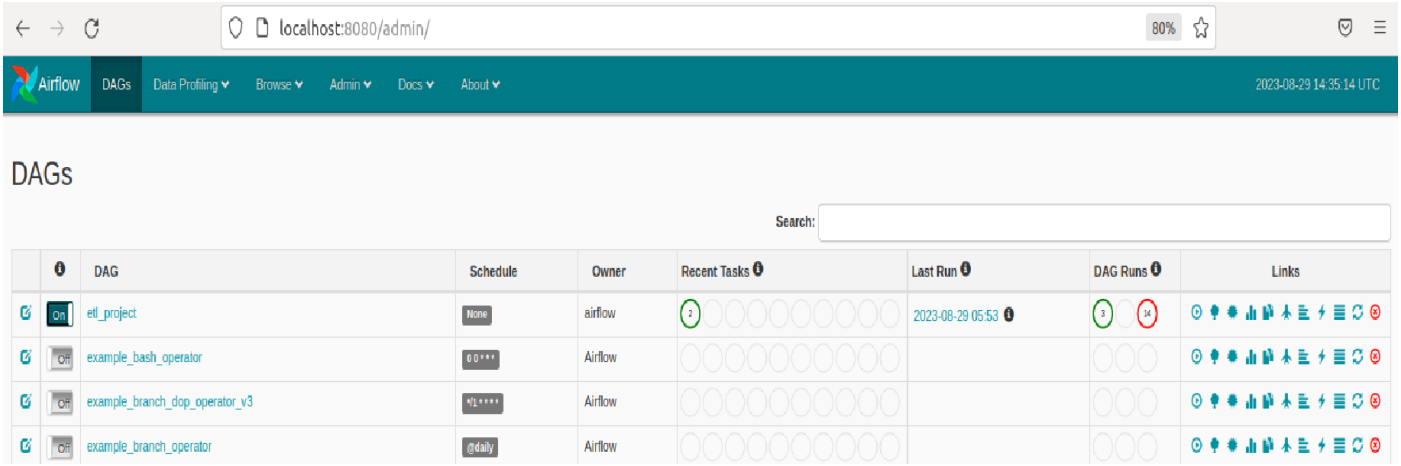

Figure 11: On the DAG

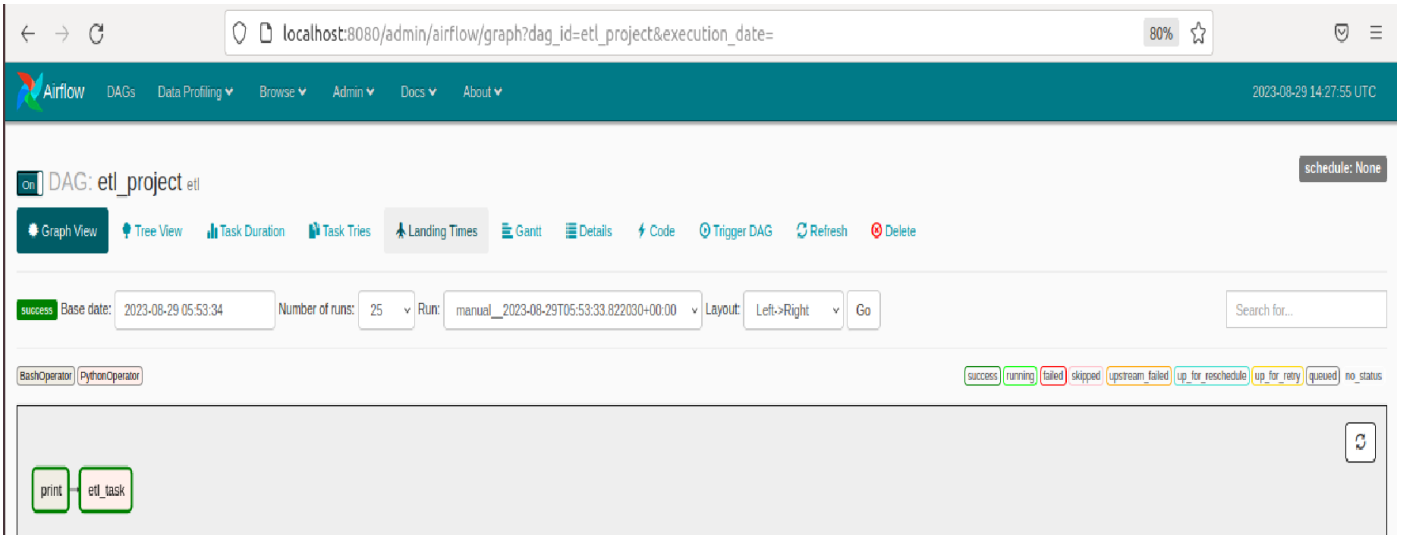

Figure 12: Graph View

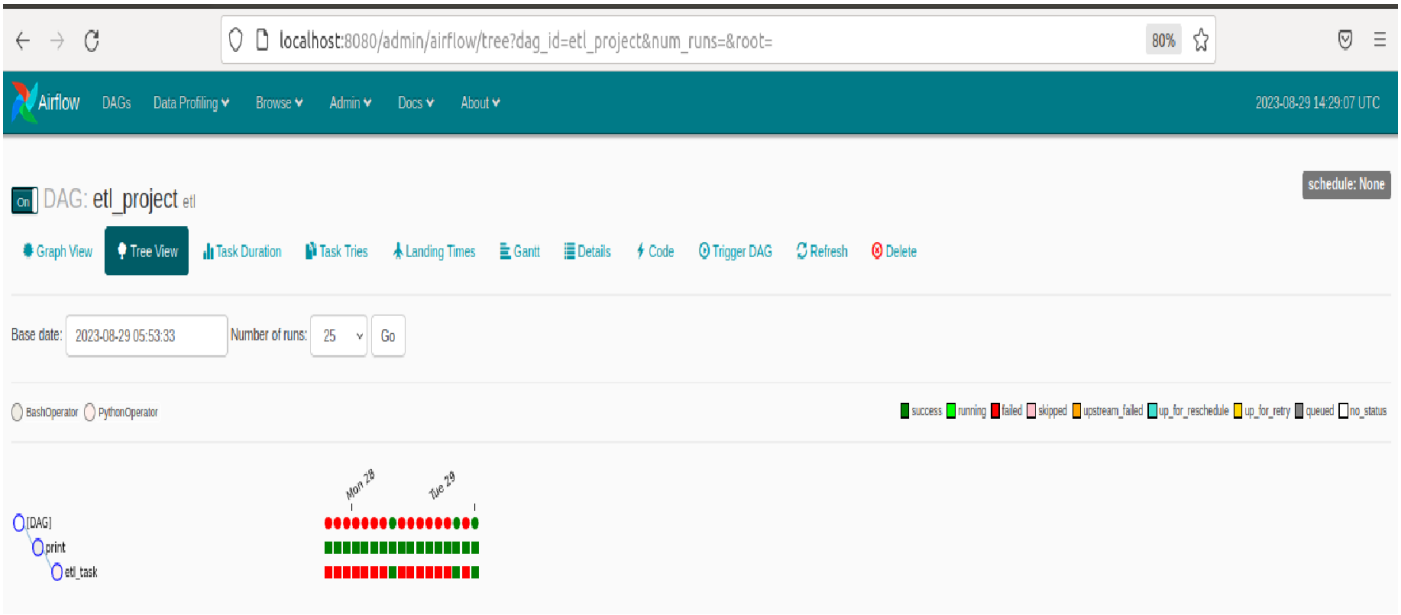

Figure 13: Tree View

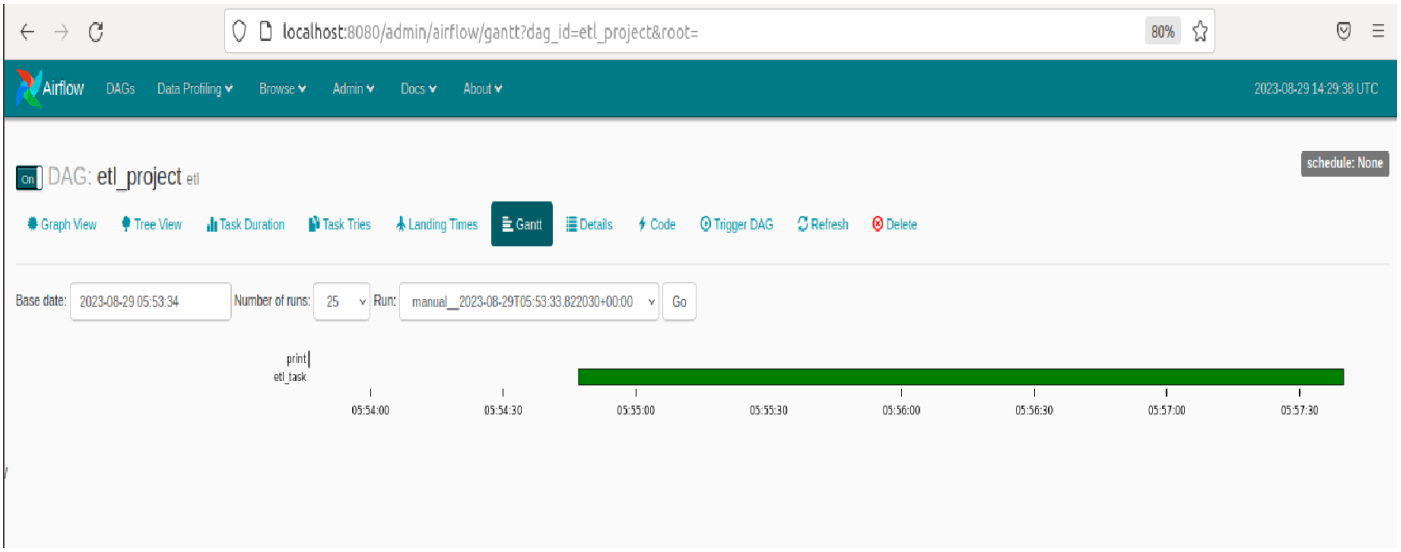

Figure 14: Gantt Chart

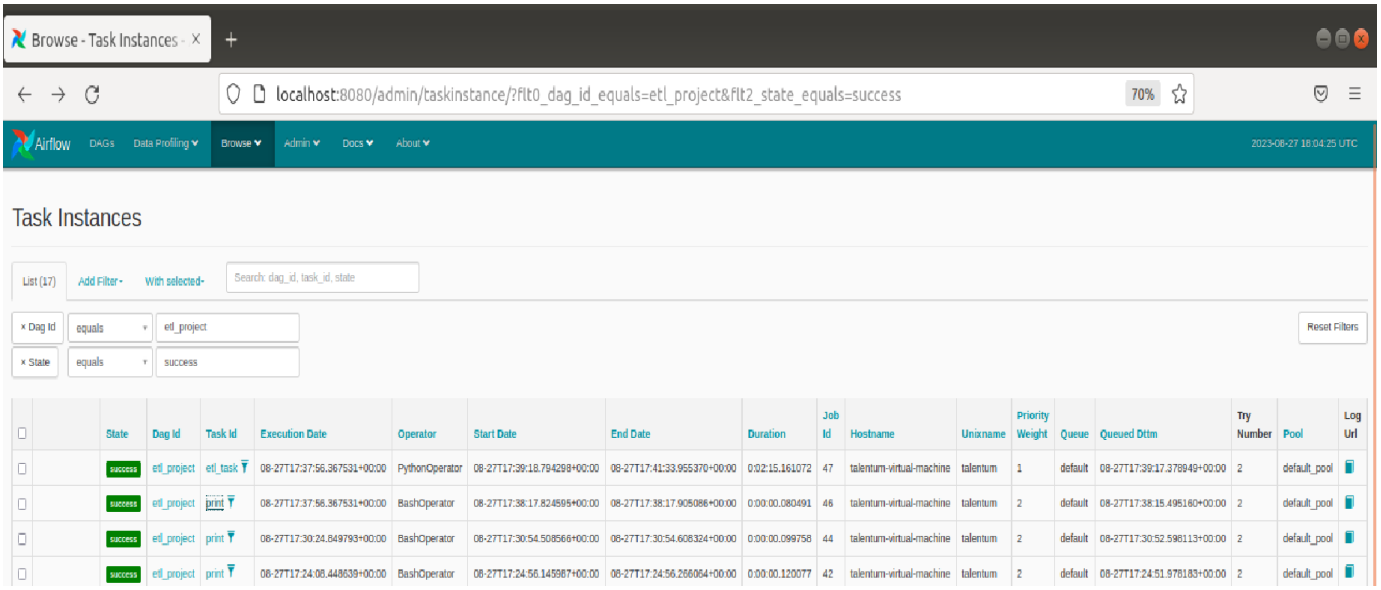

Figure 15: Task Instances

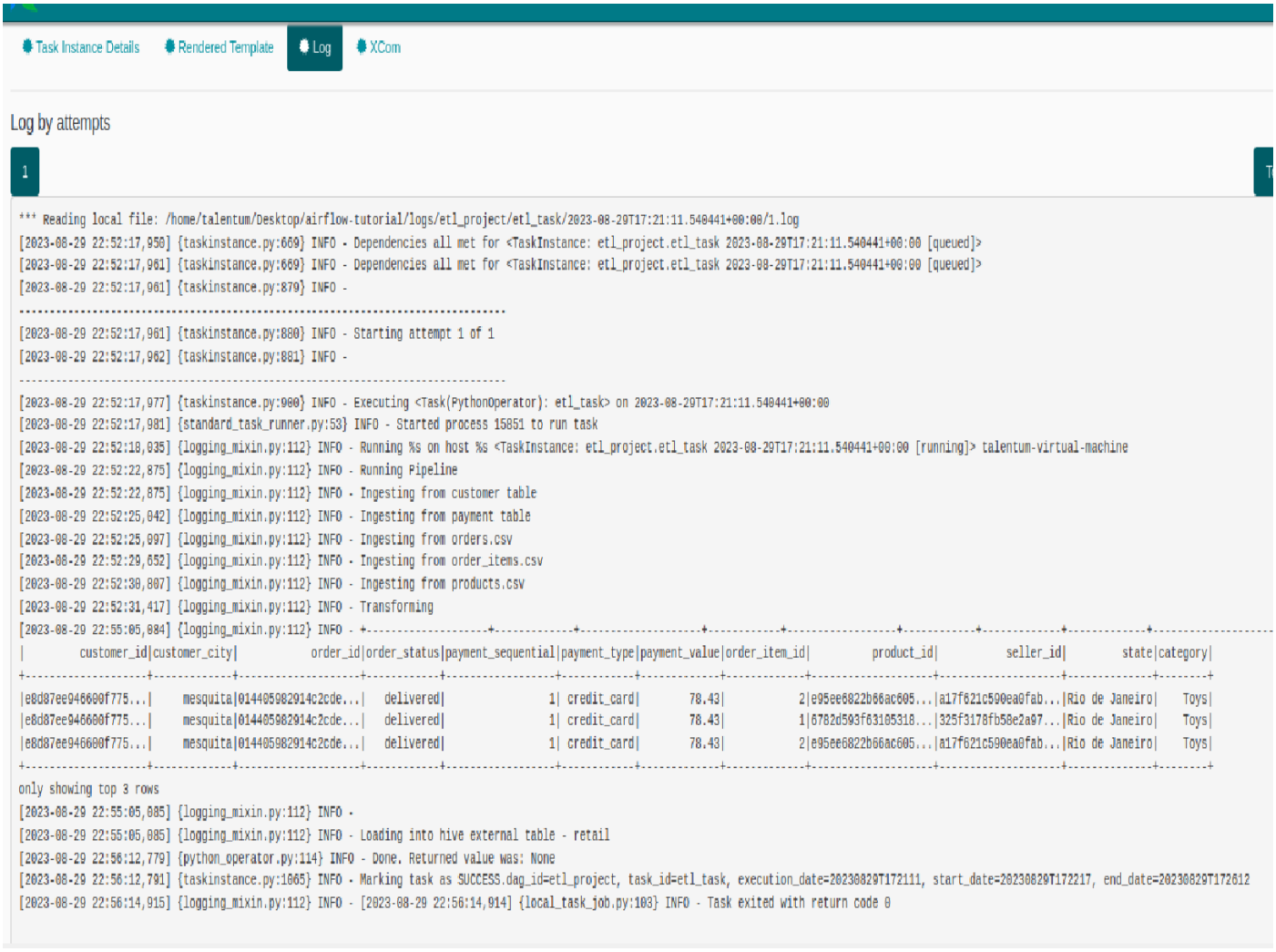

Figure 16: DAG Logs

CONCLUSION

1. The development of the automated ETL pipeline using Apache Spark has successfully addressed the challenges in collecting, cleansing, and preparing retail data for visualization.

2. The integration of diverse data sources, including .csv files and relational databases, showcased the pipeline's versatility and efficiency in data extraction.

3. The data cleansing and transformation processes using PySpark ensured the accuracy and consistency of the collected data, making it suitable for visualization.

4. The pipeline's automation significantly reduced manual intervention, resulting in a streamlined and expedited data preparation workflow.

5. The integration of data visualization tools with the prepared data facilitated the creation of impactful graphical representations, shedding light on business performance and trends.

6. The visualizations provided a bridge between raw data and actionable insights, aiding decision-makers in making informed and strategic choices.

7. The pipeline's scalability and adaptability to varying data volumes and sources underline its potential for application in diverse contexts.

8. The comprehensive documentation and guidelines ensure the replicability of the pipeline in similar scenarios, promoting efficiency and informed decision-making.

In conclusion, the automated ETL pipeline represents a significant step forward in data management, offering businesses the means to leverage data-driven insights for a more efficient and informed future.

FUTURE SCOPE

1. Advanced Analytics Integration: The ETL pipeline can be extended to include advanced analytics techniques such as predictive modeling and machine learning. This would enable the organization to make data-driven predictions and optimize business strategies.

2. Real-time Data Processing: Exploring real-time data processing capabilities can enhance decision-making by providing up-to-the-minute insights. Integrating technologies like Apache Kafka and Spark Streaming can facilitate real-time data ingestion and processing.

3. Big Data Scaling: As data volumes continue to grow, the pipeline can be adapted to handle big data scenarios by leveraging distributed computing frameworks and cloud-based solutions.

4. Automated Data Quality Checks: Implementing automated data quality checks within the pipeline can ensure the integrity and accuracy of the collected data. This would further enhance the reliability of the insights generated.

5. User-friendly Dashboards: Developing user-friendly dashboards using visualization tools like Tableau or Power BI can empower non-technical stakeholders to explore and interact with data insights.

6. Natural Language Processing (NLP): Integrating NLP techniques can enable sentiment analysis and text mining, offering a deeper understanding of customer feedback and market trends.

7. Integration with External Data Sources: Incorporating external data sources, such as social media or market trends, can provide a holistic view of the industry landscape and facilitate more informed decision-making.

8. Data Security and Compliance: Enhancing data security measures and ensuring compliance with data regulations will become increasingly important as data privacy concerns continue to evolve.

9. Continuous Improvement: Regularly monitoring and optimizing the pipeline's performance will ensure that it remains efficient and aligned with changing business needs.

8. Collaborative Data Insights: Creating a collaborative environment where different teams can share and collaborate on data insights can lead to cross-functional insights and innovative solutions.

9. Emerging Technologies: As new technologies emerge, such as quantum computing or more advanced AI algorithms, exploring their applicability to data processing and analysis could open up novel opportunities.

In essence, the future scope of the ETL pipeline is vast and promising. By embracing emerging technologies, expanding data sources, and focusing on continuous enhancement, organizations can unlock deeper insights and drive more informed decisions in the dynamic world of data analytics.

REFERENCES

Apache Spark Documentation:

https://spark.apache.org/docs/latest/api/python/reference/index.html

Apache Hive Documentation:

https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients

Apache Airflow Documentation:

https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/index.html

Python Documentation:

https://docs.python.org/3/library/index.html