amandarizkipati@gmail.com

HACKTIV8

AUTOMATION PIPELINE

COMPREHENSIVE DATA ANALYTICS PROGRAM

❯ Start Slide

REFERENCE

DATASET SOURCE

[Retail Transaction Dataset - Kaggle]

amandarizkipati@gmail.com

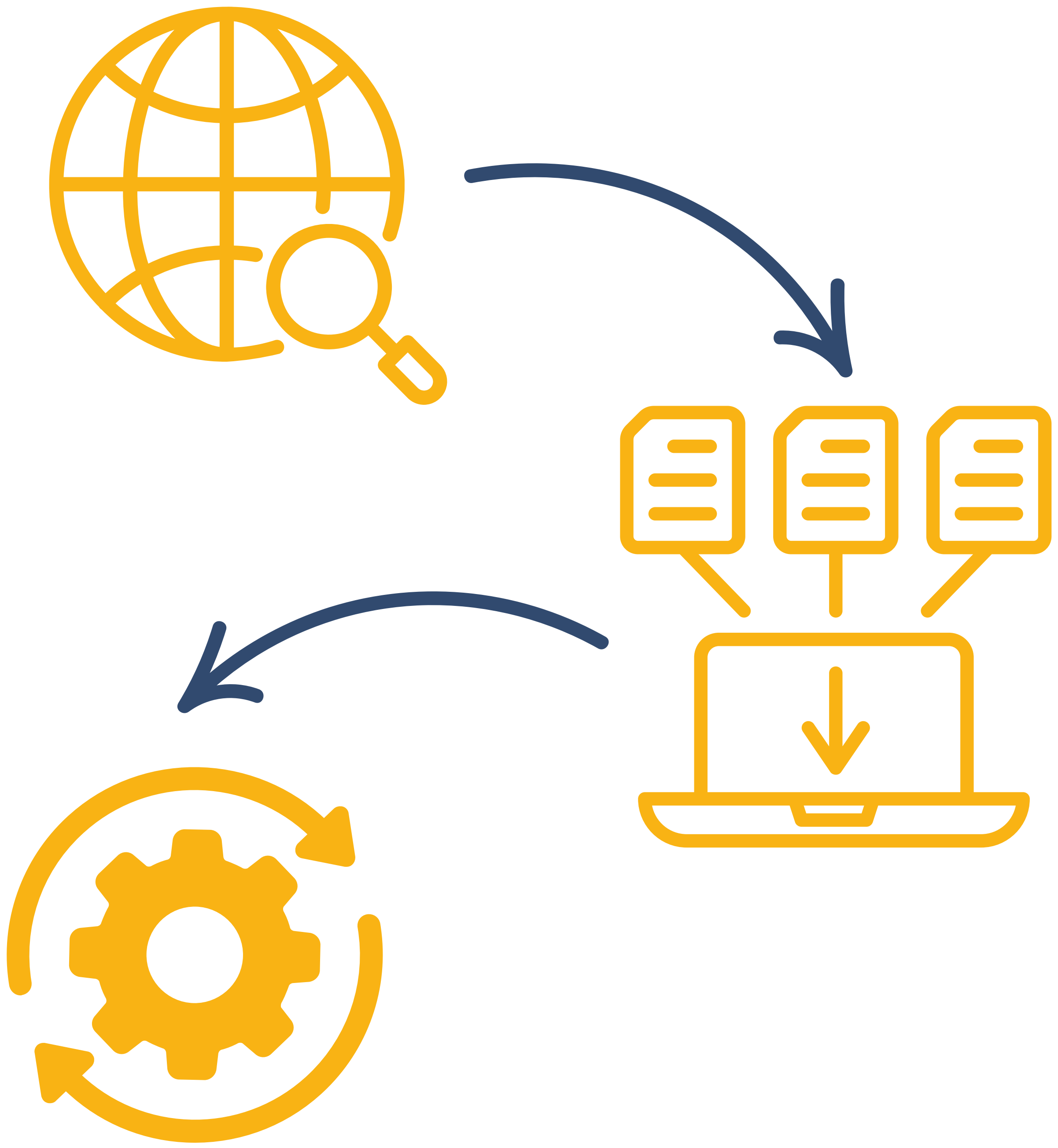
Background of the **Analysis**

Company Profile

RetailX is a multi-location retail company offering fashion, electronics, and daily goods. As the business scales, it requires an automated data system to manage transactions and support strategic insights.

The Objective

To develop an automated data pipeline for real-time analysis of transactions, customer behavior, and sales trends, enabling data-driven decisions, improving efficiency, and optimizing business performance.



Business Understanding

Retail businesses require daily, weekly, and monthly sales dashboards to:

- Identify top-selling products and dominant product categories.
- Detect top-performing store locations.
- Understand customer responses to discounts and promotions.
- Monitor payment trends (cash, card, digital).
- Recognize seasonal shopping patterns and peak transaction times.

Currently, transactional data exists as raw CSV files that cannot be directly analyzed without preprocessing. An automated ETL pipeline is required to consistently process and store this data into a database.

HACKTIV8

Business Problem

How can we build an automated data pipeline system that not only extracts raw retail transaction data but also performs data cleaning and transformation, calculates essential metrics such as total sales, discounts, and transaction (if needed), and efficiently stores the results in a MongoDB database for timely and reliable analysis?

HACKTIV8

Business Process

The business process involves automatically extracting daily retail transaction data using PySpark, cleaning and transforming it to calculate key metrics such as total sales and discount values, and then loading the structured data into MongoDB. This entire workflow is scheduled and orchestrated using Apache Airflow, ensuring consistent, timely, and accurate data availability for analysis and reporting.

HACKTIV8

Retail Transaction Dataset

▲ 138

Data Card | Code (16) | Discussion (3) | Suggestions (0)

Columns:

1. `CustomerID:` Unique identifier for each customer.
2. `ProductID:` Unique identifier for each product.
3. `Quantity:` The number of units purchased for a particular product.
4. `Price:` The unit price of the product.
5. `TransactionDate:` Date and time when the transaction occurred.
6. `PaymentMethod:` The method used by the customer to make the payment.
7. `StoreLocation:` The location where the transaction took place.
8. `ProductCategory:` Category to which the product belongs.
9. `DiscountApplied(%):` Percentage of the discount applied to the product.
10. `TotalAmount:` Total amount paid for the transaction.

Dataset Overview

The Retail Transaction Dataset consists of 10 key columns that capture essential information from each transaction, including customer and product identifiers, purchase quantity, pricing details, transaction time, payment methods, store locations, product categories, discounts applied, and the total payment made.

Pra-Automation

```
retail_transaction.printSchema()
```

```
root
 |-- CustomerID: integer (nullable = true)
 |-- ProductID: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- Price: double (nullable = true)
 |-- TransactionDate: string (nullable = true)
 |-- PaymentMethod: string (nullable = true)
 |-- StoreLocation: string (nullable = true)
 |-- ProductCategory: string (nullable = true)
 |-- DiscountApplied(%): double (nullable = true)
 |-- TotalAmount: double (nullable = true)
```

```
from pyspark.sql.functions import col, sum

# Create SparkSession
spark = SparkSession.builder \
    .appName("Missing Value Check") \
    .getOrCreate()

# Check missing (null) values per column
missing_values = retail_transaction.select([
    sum(col(c).isNull().cast("int")).alias(c)
    for c in retail_transaction.columns
])

# Show result
missing_values.show()
```

```
+----------+---------+--------+-----+---------------+-------------+-------------+---------------+------------------+-----------+
|CustomerID|ProductID|Quantity|Price|TransactionDate|PaymentMethod|StoreLocation|ProductCategory|DiscountApplied(%)|TotalAmount|
+----------+---------+--------+-----+---------------+-------------+-------------+---------------+------------------+-----------+
|         0|        0|       0|    0|              0|            0|            0|              0|                 0|          0|
+----------+---------+--------+-----+---------------+-------------+-------------+---------------+------------------+-----------+
```

```
# Hitung jumlah baris asli
original_count = retail_transaction.count()

# Hitung jumlah baris setelah dropDuplicates
dedup_count = retail_transaction.dropDuplicates().count()

# Cek apakah ada duplikat
if original_count > dedup_count:
    print("Duplicate rows found.")
else:
    print("No duplicate rows found.")
```

```
No duplicate rows found.
```

HACKTIV8

01 Checking Data Types

Based on the analysis, the data types of all attributes in each DataFrame are already appropriate and correctly assigned. Therefore, no data type conversion or modification is necessary.

02 Checking Missing Value

All columns in the dataframe contain no missing values and are ready for analysis.

03 Checking Duplicate Data

All columns in the dataframe are free of duplicate rows, ensuring high data quality and reliable foundation for analysis and modeling.

Great Expectation

```python
# 1. Expect composite key uniqueness (CustomerID + TransactionDate + ProductID)
validator.expect_compound_columns_to_be_unique(
    column_list=["CustomerID", "TransactionDate", "ProductID"],
    ignore_row_if="any_value_is_missing",
    meta={
        "business_rule": "Each product purchase per transaction should be unique"
    }
)
```

```
Calculating Metrics: 100% 7/7 [00:00<00:00, 44.68it/s]
```


```json
{
  "success": true,
  "result": {
    "element_count": 100000,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "partial_unexpected_list": [],
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0
  },
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```


In this validation step using Great Expectations, we applied a rule to ensure that each product purchase within a transaction is unique. This is done by setting a composite key across CustomerID, TransactionDate, and ProductID. The results show that all 100,000 records passed the validation—no duplicates or missing values were found. This confirms the integrity of our transactional data, ensuring that each row represents a unique product purchase as expected.

Great Expectation

```python
# 2. Expect quantity to be minimum 1
validator.expect_column_values_to_be_between("Quantity", min_value = 1)
```

Calculating Metrics: 100% 8/8 [00:00<00:00, 155.41it/s]


```
{
  "success": true,
  "result": {
    "element_count": 100000,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "partial_unexpected_list": [],
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0
  },
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```


In this validation, we checked that all quantity values are at least 1, ensuring there are no transactions with zero or negative quantity. As shown in the results, all 100,000 records passed this expectation with no missing or unexpected values. This confirms that the Quantity column meets the minimum value requirement and the data is clean and reliable for further analysis.

Great Expectation

```python
# 3. Expect PaymentMethod column to contain only a known set of values
valid_payment = ["Cash", "PayPal", "Credit Card", "Debit Card"]
validator.expect_column_values_to_be_in_set("PaymentMethod", valid_payment)
```

Calculating Metrics: 100% 8/8 [00:00<00:00, 159.31it/s]


```
{
  "success": true,
  "result": {
    "element_count": 100000,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "partial_unexpected_list": [],
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0
  },
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```


In this step, we validated the PaymentMethod column to ensure it only contains a predefined set of values—Cash, PayPal, Credit Card, and Debit Card. This helps prevent data quality issues caused by typos or invalid entries. As shown, all 100,000 records met this expectation, with 0% unexpected or missing values. This confirms the consistency and validity of payment method data across the dataset.

Great Expectation

```python
# 4. Validate numeric DiscountApplied(%)
validator.expect_column_values_to_be_of_type(
    column="DiscountApplied(%)",
    type_="float64",
    meta={
        "data_quality": "Must be numeric for calculations"
    }
)
```

Calculating Metrics: 100% 1/1 [00:00<00:00, 107.56it/s]


```
{
  "success": true,
  "result": {
    "observed_value": "float64"
  },
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```


In this expectation, we validated that the DiscountApplied(%) column contains numeric values of type float64. This is crucial for ensuring that discounts can be correctly calculated in further analysis. The validation was successful, confirming that all discount values are indeed in the correct numeric format, which maintains the integrity of any price or discount-related computation.

HACKTIV8

Great Expectation

```python
# 5. Expect TransactionDate to be properly formatted datetime
validator.expect_column_values_to_match_strftime_format(
    column="TransactionDate",
    strftime_format="%m/%d/%Y %H:%M",
)
```

Calculating Metrics: 100% 8/8 [00:00<00:00, 10.96it/s]


```
{
  "success": true,
  "result": {
    "element_count": 100000,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "partial_unexpected_list": [],
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0
  },
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```


This expectation checks whether the TransactionDate column follows the correct datetime format: 'day/month/year, hour:minute'. Ensuring consistent datetime formatting is crucial for time-based analysis and filtering. The validation result shows 100% success, meaning all entries in the column are correctly formatted without any missing or mismatched values.

Great Expectation

```python
# 6. ProductCategory name length should be reasonable
validator.expect_column_value_lengths_to_be_between(
    column="ProductCategory",
    min_value=3,
    max_value=30,
    meta={
        "description": "Category name should have reasonable character length"
    }
)
```

Calculating Metrics: 100% 9/9 [00:00<00:00, 96.11it/s]


```
{
  "success": true,
  "result": {
    "element_count": 100000,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "partial_unexpected_list": [],
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0
  },
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```


This expectation ensures that the length of the values in the ProductCategory column is within a reasonable range, between 3 and 30 characters. This check is useful to avoid entries that are either too short to be meaningful or too long to be practical. The validation passed 100%, indicating that all category names fall within the expected character length range.

Great Expectation

```
# 7. Check if Discount (%) is Logical
validator.expect_column_value_lengths_to_be_between(
    column="DiscountApplied(%)",
    min_value=0,
    max_value=100,
    meta={
        "description": "Discount (%) should be between 0 and 100"
    }
)
```

Calculating Metrics: 100% 9/9 [00:00<00:00, 86.64it/s]


```
{
  "success": true,
  "result": {
    "element_count": 100000,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "partial_unexpected_list": [],
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0
  },
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```


This expectation validates whether the discount percentage in the DiscountApplied(%) column falls within a logical range of 0 to 100. This ensures the data reflects realistic discount values. The validation result shows a 100% success rate, meaning all entries comply with the expected range and there are no anomalies detected.

Great Expectation

```python
# 8. Validate store location format
validator.expect_column_values_to_match_regex(
    column="StoreLocation",
    regex=r"^.+\n.+,\s[A-Z]{2}\s\d{5}$",
    mostly=0.95,
    meta={
        "format": "Address line 1 City, ST ZIPCODE"
    }
)
```

Calculating Metrics: 100% 8/8 [00:00<00:00, 90.76it/s]


```
{
  "success": false,
  "result": {
    "element_count": 100000,
    "unexpected_count": 10806,
    "unexpected_percent": 10.806000000000001,
    "partial_unexpected_list": [
      "USNV Harrell\r\nFPO AA 62814",
      "PSC 1498, Box 4142\r\nAPO AP 10928",
      "Unit 7268 Box 3644\r\nDPO AP 43969",
      "USNS David\r\nFPO AE 12953",
      "Unit 4486 Box 3431\r\nDPO AE 41617",
      "PSC 8454, Box 4823\r\nAPO AE 17356",
      "Unit 5493 Box 4915\r\nDPO AE 46180",
      "Unit 4248 Box 3478\r\nDPO AP 26267",
      "PSC 4308, Box 2125\r\nAPO AE 53765",
      "PSC 3555, Box 8474\r\nAPO AA 67962",
      "Unit 1535 Box 5709\r\nDPO AP 57706",
      "Unit 9800 Box 8766\r\nDPO AE 93292",
      "PSC 9458, Box 9421\r\nAPO AA 84039",
      "USNS Jackson\r\nFPO AA 77311",
      "Unit 4152 Box 6862\r\nDPO AA 32838",
      "PSC 5089, Box 2406\r\nAPO AE 06601",
      "Unit 9796 Box 6648\r\nDPO AA 42931",
      "PSC 5669, Box 2093\r\nAPO AE 56470",
      "Unit 3436 Box 2527\r\nDPO AP 61328",
```


In this step, we applied an expectation to validate the store location format using a regular expression. The expected format is Address line | City, ST ZIPCODE. The result shows that none of the records passed the validation, as indicated by success: false and a high unexpected_count. This means the address values do not match the expected pattern and likely require further cleaning or normalization.

Extract Process

The pipeline begins with the extract, which **reads the raw transaction data from a .csv** file using PySpark. The data is loaded into a DataFrame for scalable processing. After extraction, the raw data is saved as **extracted.csv** for backup and traceability.

```
from pyspark.sql import SparkSession

def load_data(file_path):
    spark = SparkSession.builder \
        .appName("ETL_Extract_PySpark") \
        .getOrCreate()

    df = spark.read.csv(
        file_path,
        header=True,
        inferSchema=True,
        sep=",",
        multiLine=True,
        escape='"'
    )
    return df

if __name__ == "__main__":
    input_path = "opt/airflow/data/data_raw.csv"
    output_path = "opt/airflow/data/extracted.csv"

    df = load_data(input_path)

    # Show 5 rows
    df.show(5)

    # Save as CSV
    df.write \
        .option("header", True) \
        .mode("overwrite") \
        .csv(output_path)
```

Transform Process

In the Transform step, we performed **essential data cleaning and enrichment using PySpark.** A unique **TransactionID** was added using monotonically_increasing_id() to ensure that each record can be uniquely identified. Additionally, we **cleaned newline characters** from the **StoreLocation** column using regexp_replace() to maintain consistency in location data.

Although not fully implemented in this step, it is also **recommended to handle missing values and duplicate entries** at this stage to ensure the reliability and quality of the data before further analysis. The cleaned dataset is saved as **transformed.csv** for the next pipeline step.

```python
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import regexp_replace, monotonically_increasing_id
import shutil
import os

def transform(df: DataFrame) -> DataFrame:
    """
    Transformation steps:
    1. Add a unique TransactionID column.
    2. Clean newline characters in the StoreLocation column.
    """
    df = df.withColumn("TransactionID", monotonically_increasing_id())
    df = df.withColumn("StoreLocation", regexp_replace("StoreLocation", r"\n", " "))
    return df

if __name__ == "__main__":
    # Initialize Spark session
    spark = SparkSession.builder \
        .appName("Transform ETL") \
        .getOrCreate()

    # Load extracted data
    input_path = "opt/airflow/data/extracted.csv"  # Change this if your file is in a different location
    df_raw = spark.read.csv(input_path, header=True, inferSchema=True)

    # Apply transformations
    df_transformed = transform(df_raw)

    # Write the transformed DataFrame to a temporary folder
    temp_output_path = "opt/airflow/data/transformed_temp"
    df_transformed.coalesce(1).write \
        .option("header", True) \
        .mode("overwrite") \
        .csv(temp_output_path)

    # Rename the generated part file to transformed.csv
    final_output_path = "opt/airflow/data/transformed.csv"
    for file_name in os.listdir(temp_output_path):
        if file_name.endswith(".csv"):
            shutil.move(os.path.join(temp_output_path, file_name), final_output_path)
            break

    # Remove the temporary folder
    shutil.rmtree(temp_output_path)

    print(f"[Transform] Transformed data successfully saved to: {final_output_path}")
```

Load Process

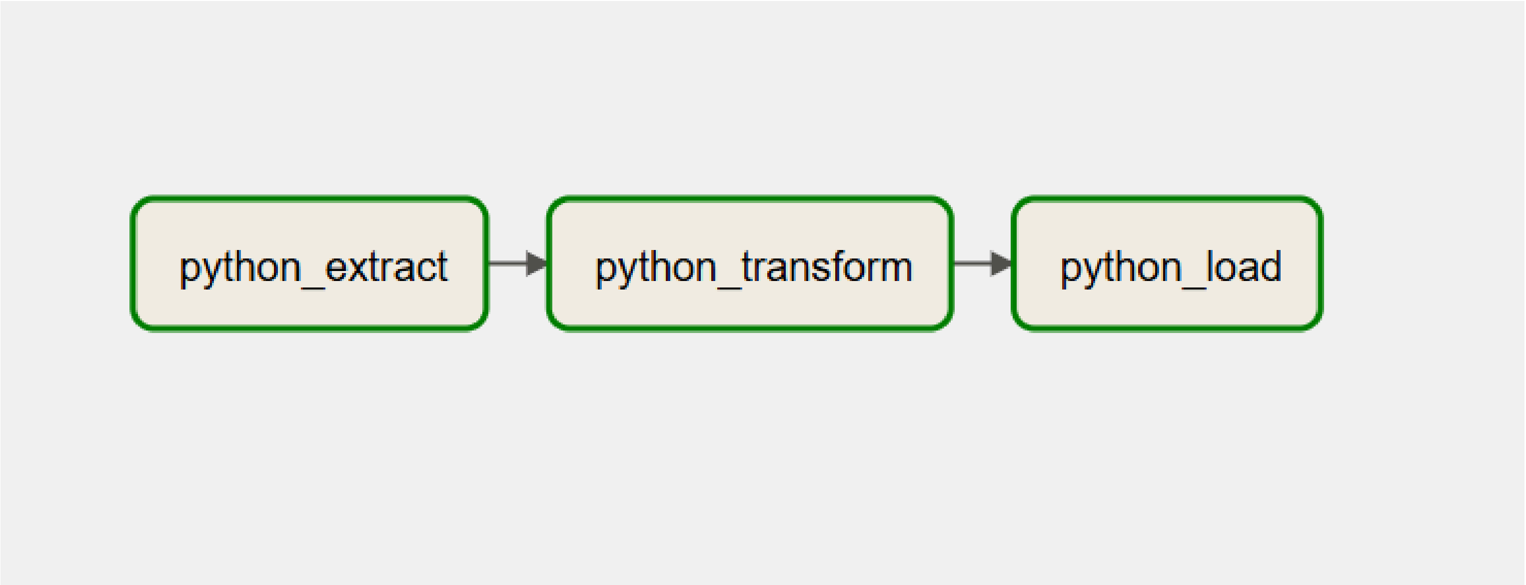
In the Load stage, we **imported the transformed retail transaction data into MongoDB** for centralized and scalable storage. The PySpark DataFrame was first converted to a Pandas DataFrame and then stored into the **P2M3_Database** under the **TransactionData collection**. This approach enables easy integration with analytics tools and supports real-time querying.

```python
from pyspark.sql import SparkSession, DataFrame
from pymongo import MongoClient

def save_to_mongodb(df: DataFrame, db_name: str, collection_name: str):
    """
    Save a Spark DataFrame to MongoDB.
    The DataFrame will be converted to a Pandas DataFrame before inserting.
    """
    pandas_df = df.toPandas()
    client = MongoClient("mongodb+srv://amandarizki:mypassword@amandas-cluster.afffjfs.mongodb.net/")
    db = client[db_name]
    collection = db[collection_name]
    collection.insert_many(pandas_df.to_dict(orient='records'))
    print(f"[Load] Data successfully saved to MongoDB in database: {db_name}, collection: {collection_name}")

if __name__ == "__main__":
    # Initialize Spark Session
    spark = SparkSession.builder \
        .appName("Load ETL to MongoDB") \
        .getOrCreate()

    # Load transformed data from CSV
    input_path = "opt/airflow/data/transformed.csv"
    df_transformed = spark.read.csv(input_path, header=True, inferSchema=True)

    # Save the DataFrame to MongoDB
    save_to_mongodb(df_transformed, "P2M3_Database", "TransactionData")
```

Workflow Orchestration

```python
import datetime as dt
from datetime import timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator

default_args = {
    'owner': 'amanda',
    'start_date': dt.datetime(2024, 11, 1),
    'retries': 1,
    'retry_delay': timedelta(minutes=10),
}

with DAG(
    dag_id='milestone3_schedule',
    default_args=default_args,
    description='ETL pipeline for crypto data every Saturday from 09:10 AM to 09:30 AM',
    schedule_interval='10,20,30 9 * * 6',
    catchup=False
) as dag:

    # Task 1: Extract
    python_extract = BashOperator(
        task_id='python_extract',
        bash_command='sudo -u airflow python /opt/airflow/scripts/extract.py'
    )

    # Task 2: Transform
    python_transform = BashOperator(
        task_id='python_transform',
        bash_command='sudo -u airflow python /opt/airflow/scripts/transform.py'
    )

    # Task 3: Load
    python_load = BashOperator(
        task_id='python_load',
        bash_command='sudo -u airflow python /opt/airflow/scripts/load.py'
    )

    python_extract >> python_transform >> python_load
```

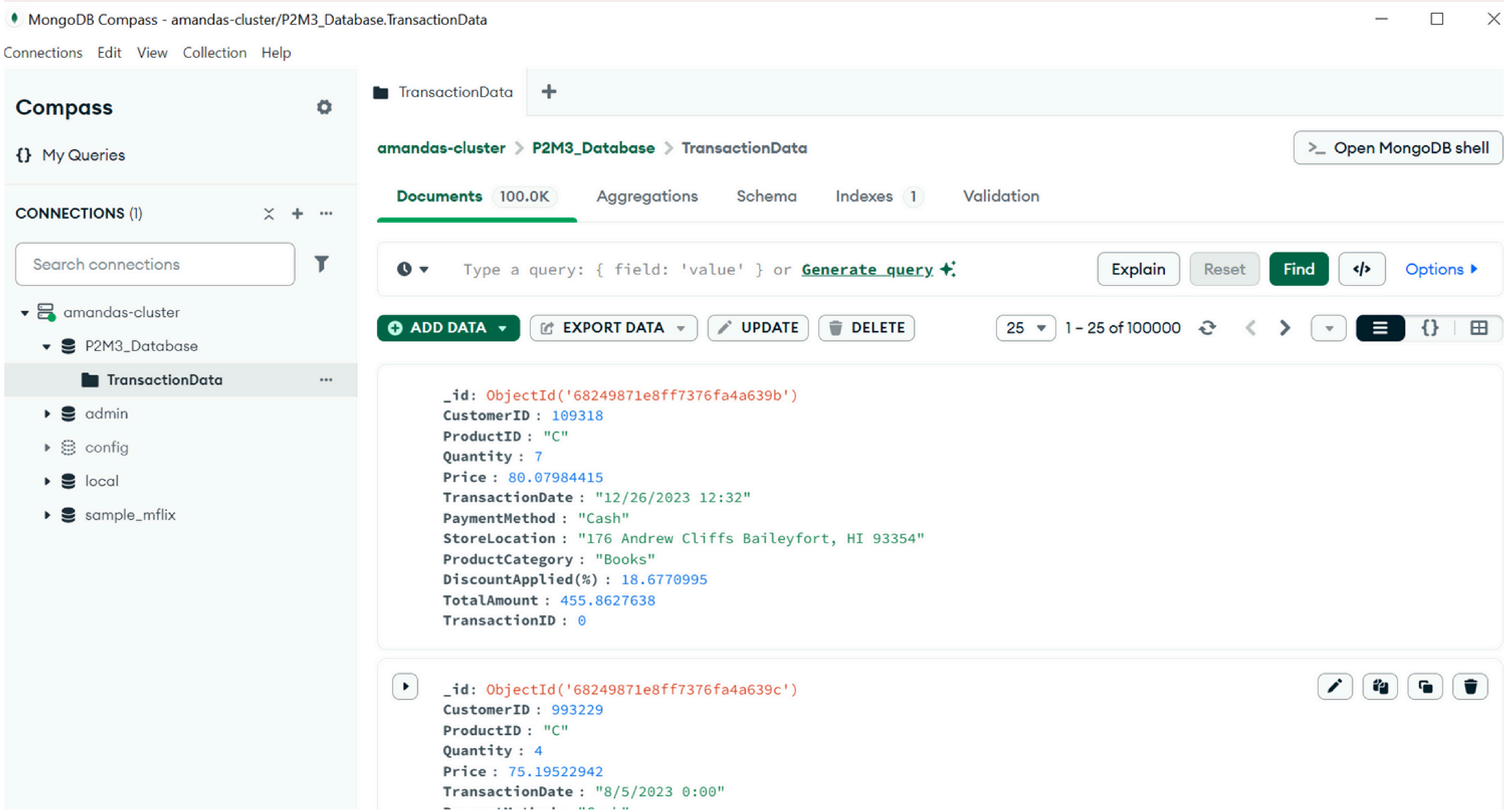
All scripts, **extract.py, transform.py, and load.py** are scheduled to run automatically using **Apache Airflow**. A Directed Acyclic Graph (DAG) orchestrates the entire ETL workflow, triggering each task in sequence every Saturday at **09:10 AM, 09:20 AM, and 09:30 AM**, starting from **November 1, 2024.** This automated scheduling ensures the consistency, reliability, and timeliness of the data pipeline, minimizing manual intervention and reducing the risk of errors.



The ETL pipeline runs in sequence: extract.py loads raw data, transform.py cleans and adds features, and load.py inserts the final data into MongoDB.

The Result

The result is a structured and clean dataset stored in MongoDB, ready for visualization and advanced analytics. Analysts can now access updated data daily without manual processing delays.



THANK YOU

AMANDA RIZKI - CODA RMT 006

amandarizkipati@gmail.com

❯ End Slide