Coding Challenge-3

Name: Harish Er

Q1) Explain ETL (Extract, Transform, Load) with PySpark(in your own words).

Ans: ETL (Extract, Transform, Load) is a process in data engineering where data is taken from a source system, processed or transformed to fit analytical or operational needs, and then loaded into a target system like a database or data warehouse. Using PySpark, which is a Python interface for Apache Spark, this process becomes scalable and efficient for handling large datasets. Here's how ETL works in PySpark:

1. Extract

This phase involves reading data from different sources like CSV files, JSON files, databases, or even real-time streams. PySpark uses its DataFrame API to simplify this step. For example, you can use spark.read to load data into a DataFrame:

Extract data from a CSV file

df = spark.read.csv("path/to/data.csv", header=True, inferSchema=True)

2. Transform

Once the data is extracted, it often requires cleaning, filtering, aggregating, or reshaping. PySpark provides a rich set of functions for data manipulation through its SQL-like operations and built-in methods. Common transformations include:

- Filtering rows: df.filter(df['column'] > value)
- Adding or modifying columns: df.withColumn('new_col', df['existing_col'] * 2)
- Aggregations: df.groupBy('column').agg({'value': 'sum'})
- Joining datasets: dfl.join(df2, 'common column', 'inner')

An example of transformation:

transformed_df = df.withColumn("total", df["quantity"] * df["price"])

3. Load

In this phase, the transformed data is saved into a target system like a database, a data warehouse, or another file format. PySpark's write APIs allow saving data in various formats like CSV, Parquet, or directly to databases.

Save data to a Parquet file

transformed_df.write.parquet("path/to/output.parquet")

Advantages of PySpark for ETL

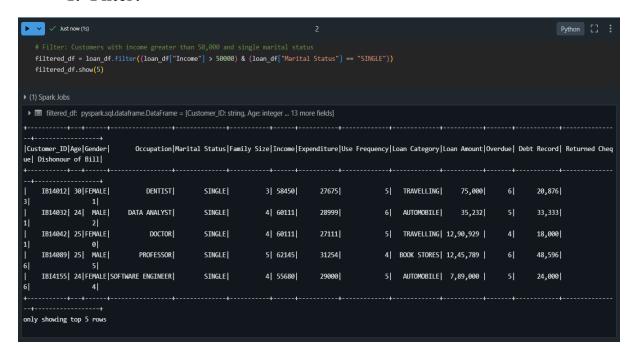
- 1. **Scalability:** Processes large datasets distributed across clusters.
- 2. **Efficiency:** Optimized for high-performance batch and stream processing.
- 3. Flexibility: Supports various data formats and transformation operations.
- 4. **Ease of Use:** Python-friendly API makes it accessible for Python developers.

PySpark's distributed nature makes it ideal for ETL workflows involving massive data volumes, ensuring faster and more efficient processing.

Q2) Using Spark SQL - Transformations such as Filter, Join, Simple Aggregations, GroupBy on the case study dataset.

1. Pyspark:

1. Filter:

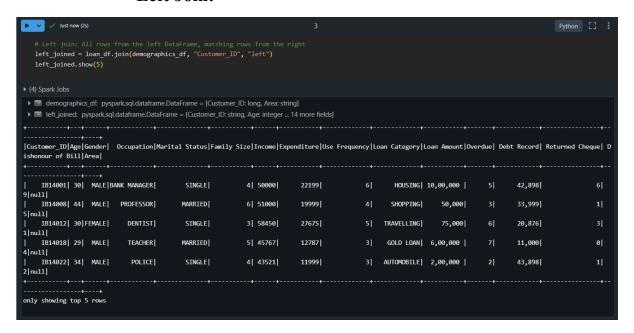


Explanation: Filtering retrieves specific rows based on conditions. In this case, we filter customers with an income greater than 50,000 and marital status as 'SINGLE'.

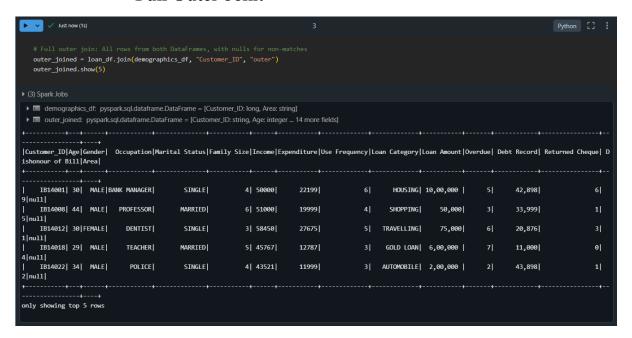
2. Join:

• Inner Join:

• Left Join:



• Full Outer Join:



Explanation:

- Inner Join: Rows are included only if Customer_ID exists in both DataFrames.
- Left Join: Preserves all rows from the loan dataset (loan_data), with nulls for missing matches in demographics.
- o **Full Outer Join**: Combines rows from both DataFrames, filling nulls where no match exists.

3. Simple Aggregations



Explanation: Aggregations calculate metrics like sum, average, or count. Here, we compute the total loan amount.

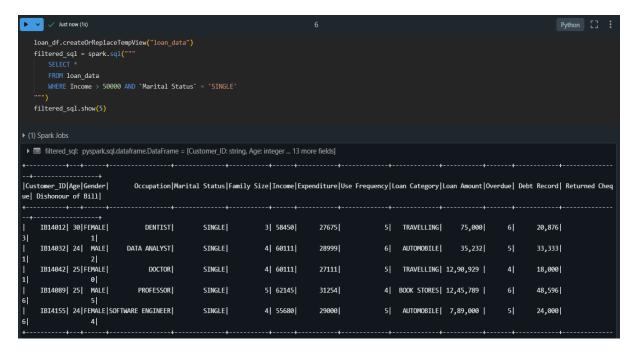
4. GroupBy



Explanation: Grouping organizes data into categories and applies aggregate functions (e.g., average, sum). Here, we calculate the average loan amount for each loan category.

2. Spark SQL:

1. Filter:

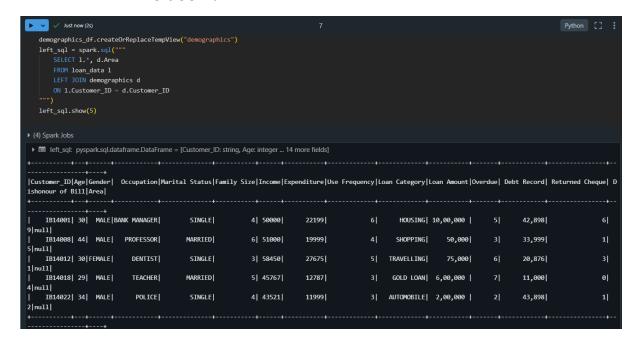


Explanation: Filtering retrieves specific rows based on conditions. In this case, we filter customers with an income greater than 50,000 and marital status as 'SINGLE'.

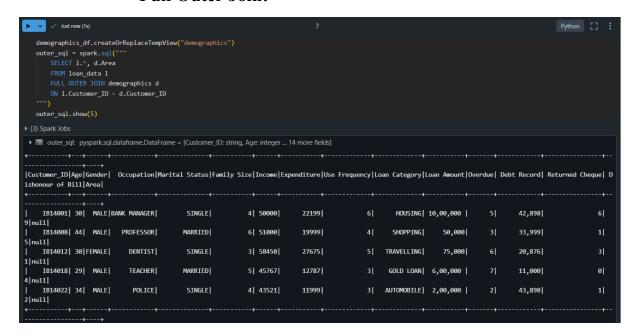
2. Join:

• Inner Join:

• Left Join:



• Full Outer Join:



Explanation:

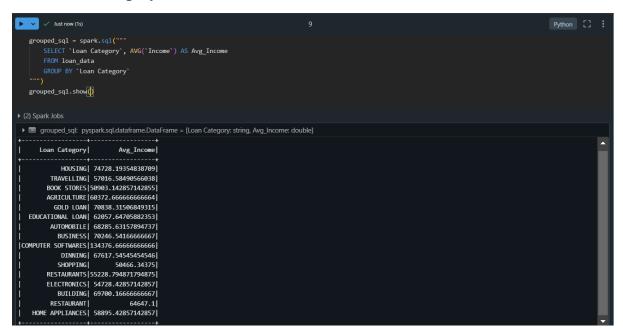
- o **Inner Join:** Includes rows where keys match in both datasets.
- Left Join: Includes all rows from the left dataset and matching rows from the right.
- o **Full Outer Join:** Includes all rows from both datasets, with nulls where no match is found.

3. Simple Aggregations:



Explanation: Aggregations calculate metrics like sum, average, or count. Here, we compute the total loan amount. SQL uses the SUM() function.

4. GroupBy:



Explanation: Grouping organizes data into categories and applies aggregate functions (e.g., average, sum). Here, we calculate the average loan amount for each loan category. SQL groups data with GROUP BY and calculates the average with AVG().

5. Calculate Average Income by Gender:

```
Python []
  ✓ ✓ Just now (11s)
   from pyspark.sql import SparkSession
   spark = SparkSession.builder.appName("Local File Loading").getOrCreate()
   loan_df = spark.read.csv("/FileStore/tables/loan.csv", header=True, inferSchema=True)
   loan_df.createOrReplaceTempView("loan_data")
   average_loan_by_gender_sql = spark.sql("
      SELECT Gender, AVG(`Income`) AS Avg_Income
      FROM loan data
      GROUP BY Gender
      ORDER BY Avg_Income DESC
   average_loan_by_gender_sql.show()
▶ ■ loan_df: pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: integer ... 13 more fields]
▶ ■ average_loan_by_gender_sql: pyspark.sql.dataframe.DataFrame = [Gender: string, Avg_Income: double]
             Avg_Income|
| FFMAI F | 76242 - 93779904307 |
MALE 61961.80694980695
```

Explanation: This query calculates the average Income for each gender and sorts the result in descending order of the average Income. Here's what each part does:

- o SELECT Gender, AVG('Income'): Retrieves the gender and the average loan amount.
- FROM loan data: Specifies the dataset to query.
- GROUP BY Gender: Groups the rows by gender to compute averages for each group.
- ORDER BY Avg_Income DESC: Sorts the results in descending order based on the calculated average.