

PYSPARK CASE STUDY

HARSHINI V

DESCRIPTION:

ONLINE BANKING ANALYSIS:

This is the first project where we worked on apache spark, In this project what we have done is that we downloaded the datasets from KAGGLE where everyone is aware of, we have downloaded loan, customers credit card and transactions datasets . After downloading the datasets we have cleaned the data . Then after by using new tools and technologies like spark, HDFS, Hive and many more we have executed new use cases on the datasets, that we have downloaded from kaggle. As we all know apache spark is a framework that can quickly process the large datasets. So now let me explain the dataflow of how we have done is, first primarily we have ingested the data that is , we retrieved the data and then downloaded the datasets from kaggle and then we stored this datasets in cloud storage and imported from MYSQL to hive by sqoop this is how we have ingested the data , second after ingesting the data we have processed the large datasets in hive and then we have analyzed the data using pyspark in jupyter notebook by implementing several use cases.

LOAD DATASETS INTO SPARK DATAFRAMES:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("OnlineBankingAnalysis").getOrCreate()
loan_df = spark.read.csv("/FileStore/tables/loan.csv", header=True, inferSchema=True)
credit_df = spark.read.csv("/FileStore/tables/credit_card.csv", header=True, inferSchema=True)
txn_df = spark.read.csv("/FileStore/tables/txn.csv", header=True, inferSchema=True)
```

EXPLANATION:

In the above code, I am initializing a Spark session using the SparkSession.builder.appName() method, which is a prerequisite for working with Spark in PySpark. I named the application "OnlineBankingAnalysis." This session allows me to interact with Spark and perform operations on large datasets. After setting up the Spark session, I used the spark.read.csv() method to load three CSV files into Spark DataFrames: loan.csv, credit_card.csv, and txn.csv. By specifying header=True, I ensure that the first row of the CSV file is treated as column headers. Additionally, setting inferSchema=True allows Spark to automatically infer the data types of the columns, making it easier to work with the data without manually specifying types.

```

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("OnlineBankingAnalysis").getOrCreate()
loan_df = spark.read.csv("/FileStore/tables/loan.csv", header=True, inferSchema=True)
credit_df = spark.read.csv("/FileStore/tables/credit_card.csv", header=True, inferSchema=True)
txn_df = spark.read.csv("/FileStore/tables/txn.csv", header=True, inferSchema=True)

```

▶ (6) Spark Jobs

- ▶ loan_df: pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: integer ... 13 more fields]
- ▶ credit_df: pyspark.sql.dataframe.DataFrame = [RowNumber: integer, CustomerId: integer ... 11 more fields]
- ▶ txn_df: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more fields]

IN LOANDATA.CSV FILE:

Number of loans in each category:

```
loan_df.groupBy("Loan Category").count().show()
```

EXPLANATION:

In this code, I am performing a groupBy() operation on the loan_df DataFrame using the column "Loan Category." The groupBy() function is used to group the data based on the distinct values in the "Loan Category" column. After grouping, I apply the count() function, which counts the number of occurrences (or records) in each group (i.e., for each unique loan category). Finally, the show() method displays the result in the console. This allows me to see how many records exist for each loan category in the dataset.

```

# Number of loans in each category
loan_df.groupBy("Loan Category").count().show()

```

▶ (3) Spark Jobs

Loan Category	count
HOUSING	61
TRAVELLING	48
BOOK STORES	7
AGRICULTURE	12
GOLD LOAN	72
EDUCATIONAL LOAN	17
AUTOMOBILE	53
BUSINESS	24
COMPUTER SOFTWARES	25
DINNING	11
SHOPPING	30
RESTAURANTS	37
ELECTRONICS	13
BUILDING	6
RESTAURANT	20
HOME APPLIANCES	13

Number of people who have taken more than 1 lakh loan:

```
loan_df.filter(loan_df["Loan Amount"] > 100000).count()
```

EXPLANATION:

In this code, I am filtering the loan_df DataFrame to identify the records where the "Loan Amount" is greater than 100,000 using the filter() function. The condition loan_df["Loan Amount"] > 100000 checks for all rows where the loan amount exceeds 1 lakh. After applying this filter, I use the count() function to determine how many such records exist, i.e., how many people have taken a loan amount greater than 1 lakh. This gives me the total number of individuals who have loans above the specified threshold.

```
# Number of people who have taken more than 1 lakh loan
loan_df.filter(loan_df["Loan Amount"] > 100000).count()
```

► (2) Spark Jobs

Out[14]: 0

Number of people with income greater than 60,000 rupees:

```
loan_df.filter(loan_df["income"] > 60000).count()
```

EXPLANATION:

In this code, I am filtering the loan_df DataFrame to find the records where the "income" column value is greater than 60,000 rupees. The filter() function is used to apply the condition loan_df["income"] > 60000, which checks for individuals with an income greater than the specified amount. After filtering the data, the count() function is used to calculate the total number of records that meet this condition. This gives the number of people with an income greater than 60,000 rupees, providing insights into this specific segment of the dataset.

```
# Number of people with income greater than 60,000 rupees
loan_df.filter(loan_df["income"] > 60000).count()
```

► (3) Spark Jobs

Out[15]: 192

Number of people with expenditure over 50,000 a month:

```
loan_df.filter(loan_df["Expenditure"] > 50000).count()
```

EXPLANATION:

In this code, I am filtering the loan_df DataFrame to identify records where the "Expenditure" column value exceeds 50,000. The filter() function is applied with the condition loan_df["Expenditure"] > 50000, which checks for individuals whose monthly expenditure is greater than the specified amount. Once the filter is applied, the count() function is used

to determine how many records match this condition. This gives the total number of people with an expenditure of over 50,000 per month, helping to analyze the financial behavior of individuals in this dataset.

```
# Number of people with expenditure over 50,000 a month
loan_df.filter(loan_df["Expenditure"] > 50000).count()
```

► (3) Spark Jobs

```
Out[19]: 6
```

Number of members eligible for a credit card:

```
loan_df.filter((loan_df["Income"] > 60000) & (loan_df["Loan Amount"] < 100000)).count()
```

EXPLANATION:

This code filters the loan_df DataFrame to find individuals who are eligible for a credit card loan. The conditions are that their "Income" must be greater than 60,000 rupees and their "Loan Amount" must be less than 100,000 rupees. The count() function is then used to determine how many individuals meet these criteria.

```
# Number of members eligible for a credit card
loan_df.filter((loan_df["Income"] > 60000) & (loan_df["Loan Amount"] < 100000)).count()
```

► (2) Spark Jobs

```
Out[20]: 0
```

IN CREDIT.CSV FILE:

Credit card users in Spain:

```
credit_df.filter(credit_df["Geography"] == "Spain").count()
```

EXPLANATION:

This code filters the credit_df DataFrame to find the number of credit card users in Spain. It checks if the "Geography" column is equal to "Spain" using the filter() function. The count() function then returns the total number of users who meet this condition.

```
# Credit card users in Spain
credit_df.filter(credit_df["Geography"] == "Spain").count()
```

► (3) Spark Jobs

```
Out[21]: 2477
```

Number of members who are eligible and active in the bank:

```
credit_df.filter((credit_df["creditscore"] > 650) & (credit_df["isactivemember"] == 1)).count()
```

EXPLANATION:

This code filters the `credit_df` DataFrame to find the number of eligible and active bank members. It checks two conditions: the "creditscore" must be greater than 650, and the "isactivemember" column must be equal to 1 (indicating the person is an active member). The `count()` function then returns the total number of members who meet both conditions.

```
# Number of members who are eligible and active in the bank
credit_df.filter((credit_df["creditscore"] > 650) & (credit_df["isactivemember"] == 1)).count()
```

► (3) Spark Jobs

Out[22]: 2655

IN TRANSACTIONS FILE:

Maximum withdrawal amount in transactions:

```
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" WITHDRAWAL AMT ").isNotNull())
max_withdrawal_row = txn_df_filtered.agg({" WITHDRAWAL AMT ": "max"}).collect()[0]
max_withdrawal_amt = max_withdrawal_row[0]
formatted_max_withdrawal_amt = f"₹{max_withdrawal_amt:,.1f}"
print("Maximum Withdrawal Amount:", formatted_max_withdrawal_amt)
```

EXPLANATION:

This code calculates the maximum withdrawal amount from the `txn_df` DataFrame. First, it filters the DataFrame to remove any rows where the "WITHDRAWAL AMT" is null. Then, it uses the `agg()` function to compute the maximum withdrawal amount. The result is collected using `collect()` and the first row (the maximum value) is accessed. The withdrawal amount is then formatted to include the currency symbol (₹) and is displayed with commas as thousand separators, rounding to one decimal place. Finally, the formatted result is printed as the maximum withdrawal amount.

```
# Maximum withdrawal amount in transactions
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" WITHDRAWAL AMT ").isNotNull())
max_withdrawal_row = txn_df_filtered.agg({" WITHDRAWAL AMT ": "max"}).collect()[0]
max_withdrawal_amt = max_withdrawal_row[0]
formatted_max_withdrawal_amt = f"₹{max_withdrawal_amt:,.1f}"
print("Maximum Withdrawal Amount:", formatted_max_withdrawal_amt)
```

► (2) Spark Jobs

► txn_df_filtered: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more fields]
Maximum Withdrawal Amount: ₹459,447,546.4

Minimum withdrawal amount of an account:


```
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" WITHDRAWAL AMT ").isNotNull())
min_withdrawal_row= txn_df_filtered.agg({" WITHDRAWAL AMT ": "min"}).collect()[0]
min_withdrawal_amt = min_withdrawal_row[0]
formatted_min_withdrawal_amt = f"₹{min_withdrawal_amt:,.2f}"
print("Minimum Withdrawal Amount:", formatted_min_withdrawal_amt)
```

EXPLANATION:

This code calculates the minimum withdrawal amount from the `txn_df` DataFrame. It starts by filtering the DataFrame to exclude any rows where the "WITHDRAWAL AMT" is null. Next, it uses the `agg()` function to compute the minimum withdrawal amount. The result is collected with `collect()`, and the first row (the minimum value) is extracted. The withdrawal amount is then formatted with the currency symbol (₹), ensuring that commas are used as thousand separators, and rounded to two decimal places. Finally, the formatted result is printed as the minimum withdrawal amount.

```
# Minimum withdrawal amount of an account
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" WITHDRAWAL AMT ").isNotNull())
min_withdrawal_row= txn_df_filtered.agg({" WITHDRAWAL AMT ": "min"}).collect()[0]
min_withdrawal_amt = min_withdrawal_row[0]
formatted_min_withdrawal_amt = f"₹{min_withdrawal_amt:,.2f}"
print("Minimum Withdrawal Amount:", formatted_min_withdrawal_amt)
```

▶ (2) Spark Jobs

▶  txn_df_filtered: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more fields]

Minimum Withdrawal Amount: ₹0.01

Maximum deposit amount of an account:


```
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" DEPOSIT AMT ").isNotNull())
max_deposit_row = txn_df_filtered.agg({" DEPOSIT AMT ": "max"}).collect()[0]
max_deposit_amt = max_deposit_row[0]
formatted_max_deposit_amt = f"₹{max_deposit_amt:,.1f}"
print("Maximum Deposit Amount:", formatted_max_deposit_amt)
```

EXPLANATION:

This code calculates the maximum deposit amount from the `txn_df` DataFrame. It first filters the DataFrame to remove rows where the "DEPOSIT AMT" is null. Then, it uses the `agg()` function to find the maximum deposit amount. The result is retrieved using `collect()`, and the first row (the maximum value) is extracted. The deposit amount is then formatted with the currency symbol (₹), ensuring thousand separators and rounding it to one decimal place. Finally, the formatted maximum deposit amount is printed.

```
# Maximum deposit amount of an account
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" DEPOSIT AMT ").isNotNull())
max_deposit_row = txn_df_filtered.agg({" DEPOSIT AMT ": "max"}).collect()[0]
max_deposit_amt = max_deposit_row[0]
formatted_max_deposit_amt = f"₹{max_deposit_amt:,.1f}"
print("Maximum Deposit Amount:", formatted_max_deposit_amt)
```

▶ (2) Spark Jobs

▶  txn_df_filtered: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more field]
Maximum Deposit Amount: ₹544,800,000.0

Minimum deposit amount of an account:


```
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" DEPOSIT AMT ").isNotNull())
min_deposit_row = txn_df_filtered.agg({" DEPOSIT AMT ": "min"}).collect()[0]
min_deposit_amt = min_deposit_row[0]
formatted_min_deposit_amt = f"₹{min_deposit_amt:,.2f}"
print("Minimum Deposit Amount:", formatted_min_deposit_amt)
```

EXPLANATION:

This code calculates the minimum deposit amount from the txn_df DataFrame. It starts by filtering out rows where the "DEPOSIT AMT" is null. Then, it uses the agg() function to calculate the minimum deposit amount. The result is fetched using collect(), and the first row (which contains the minimum value) is accessed. The deposit amount is then formatted with the currency symbol (₹), ensuring it includes thousand separators and rounds it to two decimal places. Finally, the formatted minimum deposit amount is printed.

```
# Minimum deposit amount of an account
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" DEPOSIT AMT ").isNotNull())
min_deposit_row = txn_df_filtered.agg({" DEPOSIT AMT ": "min"}).collect()[0]
min_deposit_amt = min_deposit_row[0]
formatted_min_deposit_amt = f"₹{min_deposit_amt:,.2f}"
print("Minimum Deposit Amount:", formatted_min_deposit_amt)
```

▶ (2) Spark Jobs

▶  txn_df_filtered: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more fields]
Minimum Deposit Amount: ₹0.01

Sum of balance in every bank account:

```
from pyspark.sql.functions import col, format_number
txn_df = txn_df.withColumn("BALANCE AMT", col("BALANCE AMT").cast("double"))
txn_df_filtered = txn_df.filter((col("BALANCE AMT") > 0) & (col("BALANCE AMT").isNotNull()))
result_df = txn_df_filtered.groupBy("Account no").agg({"BALANCE AMT": "sum"})
result_df = result_df.withColumnRenamed("sum(BALANCE AMT)", "Total Balance")
result_df = result_df.withColumn("Formatted Balance", format_number("Total Balance", 2))
result_df.show(truncate=False)
```

EXPLANATION:

This code calculates the sum of the balance for each bank account in the `txn_df` DataFrame. First, it converts the "BALANCE AMT" column to a double data type using `cast()`. Next, the code filters out rows where the balance is either null or less than or equal to zero. Then, it groups the data by "Account no" and calculates the sum of "BALANCE AMT" for each account using the `groupBy()` and `agg()` functions. The column name for the sum is renamed to "Total Balance." To present the balance in a formatted way, the `format_number()` function is used, ensuring the balance has two decimal places. Finally, the result is displayed using `show()`, without truncating the values.

```
# Sum of balance in every bank account
from pyspark.sql.functions import col, format_number
txn_df = txn_df.withColumn("BALANCE AMT", col("BALANCE AMT").cast("double"))
txn_df_filtered = txn_df.filter((col("BALANCE AMT") > 0) & (col("BALANCE AMT").isNotNull()))
result_df = txn_df_filtered.groupBy("Account no").agg({"BALANCE AMT": "sum"})
result_df = result_df.withColumnRenamed("sum(BALANCE AMT)", "Total Balance")
result_df = result_df.withColumn("Formatted Balance", format_number("Total Balance", 2))
result_df.show(truncate=False)
```

▶ (2) Spark Jobs

```
▶ txn_df: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more fields]
▶ txn_df_filtered: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more fields]
▶ result_df: pyspark.sql.dataframe.DataFrame = [Account no: string, Total Balance: double ... 1 more field]
```

Account no	Total Balance	Formatted Balance
409000611074	1.615533622E9	1,615,533,622.00
409000425051	8.649102501000117E8	864,910,250.10
409000493201	1.0420831829499985E9	1,042,083,182.95

Number of transactions on each date:

```
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col("VALUE DATE").isNotNull())
txn_df_filtered.groupBy("VALUE DATE").count().show()
```

EXPLANATION:

This code calculates the number of transactions for each unique date in the `txn_df` DataFrame. First, it filters out rows where the "VALUE DATE" is null using the `filter()` function. Then, it groups the data by the "VALUE DATE" column using the `groupBy()` function. The `count()` function is applied to count the number of transactions for each date. Finally, the result is displayed using `show()`, which shows the count of transactions per unique value date.

```
# Number of transactions on each date
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col("VALUE DATE").isNotNull())
txn_df_filtered.groupBy("VALUE DATE").count().show()
```

▶ (2) Spark Jobs

```
▶ txn_df_filtered: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more fields]
```

VALUE DATE	count
23-Dec-16	143
7-Feb-19	98
21-Jul-15	80
9-Sep-15	91
17-Jan-15	16
18-Nov-17	53
21-Feb-18	77
20-Mar-18	71
19-Apr-18	71
21-Jun-16	97
17-Oct-17	101
3-Jan-18	70
8-Jun-18	223
15-Dec-18	62
8-Aug-16	97
17-Dec-16	74
3-Sep-15	83
21-Jan-16	76

List of customers with withdrawal amount more than 1 lakh:

```
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" WITHDRAWAL AMT ").isNotNull())
txn_df_filtered.filter(txn_df_filtered[" WITHDRAWAL AMT " ] > 100000).select("Account No").show()
```

EXPLANATION:

This code retrieves a list of customers who have made withdrawal transactions greater than 1 lakh. First, it filters the `txn_df` DataFrame to exclude any rows where the "WITHDRAWAL AMT" is null. Then, it applies another filter to find transactions where the "WITHDRAWAL AMT" is greater than 100,000. The `select("Account No")` part selects the "Account No" column to display only the account numbers of customers who meet the condition. Finally, the `show()` function is used to display the result.

```
# List of customers with withdrawal amount more than 1 lakh
from pyspark.sql.functions import col
txn_df_filtered = txn_df.filter(col(" WITHDRAWAL AMT ").isNotNull())
txn_df_filtered.filter(txn_df_filtered[" WITHDRAWAL AMT "] > 100000).select("Account No").show()
```

- ▶ (1) Spark Jobs

```
▶ txn_df_filtered: pyspark.sql.dataframe.DataFrame = [Account No: string, TRANSACTION DETAILS: string ... 4 more fields]
```

[illegible]