PySpark Case Study

-Sarthak Niranjan Kulkarni (Maverick)

- sarthakkul2311@gmail.com - (+91) 93256 02791

25/11/2024 (Monday)

1. Loandata.csv file

1. Number of loans in each category:

→ # Group by 'Loan Category' and count the number of loans in each category

loan_category_counts = loan_df.groupBy("Loan Category").count()

Show the result

loan_category_counts.show()

```
loan_category_counts:  pyspark.sql.dataframe.DataFrame = [Loan Category: string, count: long]
+------------------+-----+
|     Loan Category|count|
+------------------+-----+
|           HOUSING|   67|
|        TRAVELLING|   53|
|       BOOK STORES|    7|
|       AGRICULTURE|   12|
|         GOLD LOAN|   77|
|  EDUCATIONAL LOAN|   20|
|        AUTOMOBILE|   60|
|          BUSINESS|   24|
|COMPUTER SOFTWARES|   35|
|           DINNING|   14|
|          SHOPPING|   35|
|       RESTAURANTS|   41|
|       ELECTRONICS|   14|
|          BUILDING|    7|
|        RESTAURANT|   20|
|   HOME APPLIANCES|   14|
+------------------+-----+
```

---

2. Number of people with income greater than 60000 rupees

→ from pyspark.sql.functions import col

Filter the rows where 'Income' is greater than 60,000

people_above_60k = loan_df.filter(col("Income") > 60000)

Count the number of such people

num_people_above_60k = people_above_60k.count()

Display the result

print(f"Number of people with income greater than 60,000 rupees: {num_people_above_60k}")

```
▸ (2) Spark Jobs
  ▸ people_above_60k: pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: string ... 13 more fields]
Number of people with income greater than 60,000 rupees: 198
```

---

3. Number of people with 2 or more returned cheques and income less than 50000

→ from pyspark.sql.functions import col

Remove any leading/trailing spaces from the column names if necessary

loan_df_cleaned = loan_df.select([col(c).alias(c.strip()) for c in loan_df.columns])

Filter the rows where 'Returned Cheque' >= 2 and 'Income' < 50,000

people_filtered = loan_df_cleaned.filter((col("Returned Cheque") >= 2) & (col("Income") < 50000))

Count the number of such people

num_people_filtered = people_filtered.count()

Display the result

print(f"Number of people with 2 or more returned cheques and income less than 50,000 rupees: {num_people_filtered}")

```
  ▸ loan_df_cleaned: pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: string ... 13 more fields]
  ▸ people_filtered: pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: string ... 13 more fields]
Number of people with 2 or more returned cheques and income less than 50,000 rupees: 137
```

---

4. Number of people with 2 or more returned cheques and are single

→ from pyspark.sql.functions import col

Remove leading/trailing spaces from column names if needed

loan_df_cleaned = loan_df.select([col(c).alias(c.strip()) for c in loan_df.columns])

Filter the rows where 'Returned Cheque' >= 2 and 'Marital Status' is 'SINGLE'

people_filtered = loan_df_cleaned.filter((col("Returned Cheque") >= 2) & (col("Marital Status") == "SINGLE"))

Count the number of such people

num_people_filtered = people_filtered.count()

Display the result

print(f"Number of people with 2 or more returned cheques and are single: {num_people_filtered}")

```
▸ loan_df_cleaned:  pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: string ... 13 more fields]
▸ people_filtered:  pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: string ... 13 more fields]
Number of people with 2 or more returned cheques and are single: 111
```

---

5. Number of people with expenditure over 50000 a month

→ from pyspark.sql.functions import col

Filter the rows where 'Expenditure' > 50,000

people_filtered = loan_df.filter(col("Expenditure") > 50000)

Count the number of such people

num_people_filtered = people_filtered.count()

Display the result

print(f"Number of people with expenditure over 50,000 a month: {num_people_filtered}")

```
▸ people_filtered:  pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: string ... 13 more fields]
Number of people with expenditure over 50,000 a month: 6
```

---

6. Number of members who are elgible for credit card

→ from pyspark.sql.functions import col

Filter the rows where 'Income' > 30,000 and 'Overdue' == 0 (eligible for credit card)

eligible_members = loan_df.filter((col("Income") > 30000) & (col("Overdue") == 0))

Count the number of eligible members

num_eligible_members = eligible_members.count()

Display the result

print(f"Number of members eligible for a credit card: {num_eligible_members}")

```
▸ eligible_members: pyspark.sql.dataframe.DataFrame = [Customer_ID: string, Age: string ... 13 more fields]
Number of members eligible for a credit card: 0
```

---

2. Credit.csv File

1. **Credit card users in Spain**

→ from pyspark.sql.functions import col

Filter the DataFrame for users in Spain with at least one product (credit card users)

credit_card_users_spain = credit_df.filter((col("Geography") == "Spain") & (col("NumOfProducts") > 0))

Count the number of credit card users in Spain

num_credit_card_users_spain = credit_card_users_spain.count()

Display the result

print(f"Number of credit card users in Spain: {num_credit_card_users_spain}")

```
▸ credit_card_users_spain: pyspark.sql.dataframe.DataFrame = [RowNumber: string, CustomerId: string ... 11 more fields]
Number of credit card users in Spain: 2477
```

---

2. **Number of members who are eligible and active in the bank**

→ from pyspark.sql.functions import col

Filter the DataFrame for eligible and active members

eligible_active_members = credit_df.filter((col("NumOfProducts") > 0) & (col("IsActiveMember") == 1))

Count the number of eligible and active members

num_eligible_active_members = eligible_active_members.count()

Display the result

print(f"Number of eligible and active members: {num_eligible_active_members}")

```
eligible_active_members: pyspark.sql.dataframe.DataFrame = [RowNumber: string, CustomerId: string ... 11 more fields]
Number of eligible and active members: 5151
```

---

3. Txn.csv File: -

1. **Maximum withdrawal amount in transactions**

→ from pyspark.sql.functions import col

Find the maximum withdrawal amount

max_withdrawal = txn_df.agg({" WITHDRAWAL AMT ": "max"}).collect()[0][0]

Show the result

print("Maximum Withdrawal Amount:", max_withdrawal)

```
Maximum Withdrawal Amount: 9999
```

---

2. **Minimum withdrawal amount of an account in txn.csv**

→ from pyspark.sql.functions import col, min

Find the minimum withdrawal amount grouped by Account No

min_withdrawal = txn_df.groupBy("Account No").agg(min(" WITHDRAWAL AMT ").alias("Min Withdrawal")).show()

```
▸ (2) Spark Jobs
+-------------+--------------+
|   Account No|Min Withdrawal|
+-------------+--------------+
|     1196428'|          0.25|
|     1196711'|          0.25|
|409000362497'|          0.97|
|409000405747'|       1000000|
|409000425051'|          1.25|
|409000438611'|           0.2|
|409000438620'|          0.34|
|409000493201'|       1000000|
|409000493210'|          0.01|
|409000611074'|         10000|
+-------------+--------------+
```

---

3. Maximum deposit amount of an account

→ from pyspark.sql.functions import col, max

Find the maximum deposit amount grouped by Account No

max_deposit = txn_df.groupBy("Account No").agg(max(" DEPOSIT AMT ").alias("Max Deposit")).show()

```
▸ (2) Spark Jobs
+-------------+-----------+
|   Account No|Max Deposit|
+-------------+-----------+
|     1196428'|    9999999|
|     1196711'|  999467.62|
|409000362497'|   99977.78|
|409000405747'|   80408.93|
|409000425051'|       8500|
|409000438611'|   99999.48|
|409000438620'|     9993.8|
|409000493201'|   94982.32|
|409000493210'|      99.02|
|409000611074'|     500000|
+-------------+-----------+
```

---

4. Minimum deposit amount of an account

→ from pyspark.sql.functions import col, min

Find the minimum deposit amount grouped by Account No

min_deposit = txn_df.groupBy("Account No").agg(min(" DEPOSIT AMT ").alias("Min Deposit")).show()

```
▸ (2) Spark Jobs
+-------------+-----------+
|   Account No|Min Deposit|
+-------------+-----------+
|     1196428'|          1|
|     1196711'|       1.01|
|409000362497'|       0.03|
|409000405747'|      10000|
|409000425051'|          1|
|409000438611'|       0.03|
|409000438620'|       0.07|
|409000493201'|        0.9|
|409000493210'|       0.01|
|409000611074'|    1000000|
+-------------+-----------+
```

---

5. Sum of balance in every bank account

→ from pyspark.sql.functions import sum

Find the sum of balance grouped by Account No

sum_balance = txn_df.groupBy("Account No").agg(sum("BALANCE AMT").alias("Total Balance")).show()

```
+-------------+--------------------+
|   Account No|       Total Balance|
+-------------+--------------------+
|409000438611'|-2.49486577068339...|
|     1196711'|-1.60476498101275E13|
|     1196428'| -8.1418498130721E13|
|409000493210'|-3.27584952132095...|
|409000611074'|        1.615533622E9|
|409000425051'|-3.77211841164998...|
|409000405747'|-2.43108047067000...|
|409000493201'|1.0420831829499985E9|
|409000438620'|-7.12291867951358...|
|409000362497'| -5.2860004792808E13|
+-------------+--------------------+
```

---

6. Number of transaction on each date

→ from pyspark.sql.functions import col

Find the number of transactions grouped by VALUE DATE

transaction_count_per_date = txn_df.groupBy("VALUE DATE").count().alias("Number of Transactions").show()

```
+----------+-----+
|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|
```

---

7. List of customers with withdrawal amount more than 1 lakh

→ from pyspark.sql.functions import col

Filter the data for withdrawal amount greater than 1 lakh

customers_with_high_withdrawal = txn_df.filter(col(" WITHDRAWAL AMT ") > 100000).select("Account No", "TRANSACTION DETAILS", "VALUE DATE", " WITHDRAWAL AMT ").show()

```
+-------------+--------------------+----------+----------------+
|   Account No| TRANSACTION DETAILS|VALUE DATE| WITHDRAWAL AMT |
+-------------+--------------------+----------+----------------+
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          133900|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          195800|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          143800|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          331650|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          129000|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          230013|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          367900|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          108000|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          141000|
|409000611074'|INDO GIBL Indiafo...| 16-Aug-17|          206000|
|409000611074'|INDO GIBL Indiafo...|  6-Sep-17|          242300|
|409000611074'|INDO GIBL Indiafo...|  6-Sep-17|          113250|
|409000611074'|INDO GIBL Indiafo...|  6-Sep-17|          206900|
|409000611074'|INDO GIBL Indiafo...|  6-Sep-17|          276000|
|409000611074'|INDO GIBL Indiafo...|  6-Sep-17|          171000|
|409000611074'|INDO GIBL Indiafo...|  6-Sep-17|          189800|
|409000611074'|INDO GIBL Indiafo...|  6-Sep-17|          271323|
```

---

4. Summary of Case Study: -

1. **Reading Data**:

    - PySpark's spark.read.option is used to load CSV data into a DataFrame with headers enabled.

2. **Grouping and Aggregation**:

    - groupBy and count functions are applied to analyze loan data by categories and transaction counts by dates.

3. **Filtering Data**:

    - The filter function is extensively used to extract subsets of data based on conditions, such as income thresholds, marital status, and overdue payments.

4. **Column Operations**:

    - The col function allows dynamic column references, while column names are cleaned using .alias() for consistent processing.

5. **Aggregation Functions**:

    - Functions like max, min, sum, and agg are used to calculate metrics for transactional data (e.g., maximum withdrawals and total balances).

6. **Efficient Counting**:

    - count() is applied to count rows meeting specific criteria, such as active members, high expenditures, or high-value transactions.

7. **DataFrames and SQL Integration**:

    - PySpark seamlessly combines SQL-like operations (select, groupBy, and filtering) with DataFrame-based data manipulation for scalable analysis.