PySpark Coding Challenge

Transformations & Actions

-Bavatharani

```
# Import necessary modules
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg, sum, max, row_number
from pyspark.sql.window import Window
# Create Spark Session
spark = SparkSession.builder.appName("PySparkCodingChallenge").getOrCreate()
# Load CSV files
emp_df = spark.read.csv("/content/Employees.csv", header=True, inferSchema=True)
dept_df = spark.read.csv("/content/Department.csv", header=True, inferSchema=True)
# Show data
print("Employee Data:")
emp_df.show()
print("Department Data:")
dept_df.show()
```

```
Employee Data:
+---+-----+-------+------+
| id| name|   dept|salary|
+---+-----+-------+------+
|  1|Alice|     HR|  4000|
|  2|  Bob|     IT|  5000|
|  3|Cathy|     HR|  4200|
|  4|David|     IT|  5500|
|  5|  Eva|Finance|  6000|
|  6|Frank|Finance|  4800|
|  7|Grace|     IT|  5300|
|  8|Helen|     HR|  3900|
|  9|  Ian|     IT|  6100|
| 10| Jane|Finance|  4700|
+---+-----+-------+------+

Department Data:
+---------+---------+
|     dept| location|
+---------+---------+
|       HR| New York|
|       IT|Bangalore|
|  Finance|   London|
|Marketing|Singapore|
+---------+---------+
```

1. Filter employees who earn more than 4500

filtered_df = emp_df.filter(emp_df.salary > 4500)

filtered_df.show()

```
+---+-----+-------+------+
| id| name|   dept|salary|
+---+-----+-------+------+
|  2|  Bob|     IT|  5000|
|  4|David|     IT|  5500|
|  5|  Eva|Finance|  6000|
|  6|Frank|Finance|  4800|
|  7|Grace|     IT|  5300|
|  9|  Ian|     IT|  6100|
| 10| Jane|Finance|  4700|
+---+-----+-------+------+
```

Explanation:

The filter() function is used to extract only those rows that satisfy a condition. Here, it returns only employees whose salary is greater than 4500.

2. Join employee and department dataframes on department column

joined_df = emp_df.join(dept_df, on="dept", how="inner")

joined_df.show()

```
+-------+---+-----+------+---------+
|   dept| id| name|salary| location|
+-------+---+-----+------+---------+
|     HR|  1|Alice|  4000| New York|
|     IT|  2|  Bob|  5000|Bangalore|
|     HR|  3|Cathy|  4200| New York|
|     IT|  4|David|  5500|Bangalore|
|Finance|  5|  Eva|  6000|   London|
|Finance|  6|Frank|  4800|   London|
|     IT|  7|Grace|  5300|Bangalore|
|     HR|  8|Helen|  3900| New York|
|     IT|  9|  Ian|  6100|Bangalore|
|Finance| 10| Jane|  4700|   London|
+-------+---+-----+------+---------+
```

Explanation:

We use an inner join to combine rows where dept matches in both employee and department datasets. This helps enrich employee data with location info.

3. Calculate total and average salary of all employees

from pyspark.sql.functions import sum, avg

agg_df = emp_df.agg(

sum("salary").alias("total_salary"),

avg("salary").alias("average_salary")

)

agg_df.show()

```
+------------+--------------+
|total_salary|average_salary|
+------------+--------------+
|       49500|        4950.0|
+------------+--------------+
```

Explanation:

This is a basic aggregation. sum() gives the total salary payout, and avg() gives the mean salary of all employees — useful for financial summaries.

4. Group by department and find max salary in each

from pyspark.sql.functions import max

grouped_df = emp_df.groupBy("dept").agg(max("salary").alias("max_salary"))

grouped_df.show()

```
+-------+----------+
|   dept|max_salary|
+-------+----------+
|     HR|      4200|
|Finance|      6000|
|     IT|      6100|
+-------+----------+
```

Explanation:

groupBy() is used to group data based on a column — here dept — and apply an aggregation (in this case, maximum salary) for each group.

5. Rank employees within each department by salary

from pyspark.sql.window import W

indow

from pyspark.sql.functions import row_number

window_spec = Window.partitionBy("dept").orderBy(emp_df.salary.desc())

ranked_df = emp_df.withColumn("salary_rank", row_number().over(window_spec)) ranked_df.show()

```
+---+-----+-------+------+-----------+
| id| name|   dept|salary|salary_rank|
+---+-----+-------+------+-----------+
|  5|  Eva|Finance|  6000|          1|
|  6|Frank|Finance|  4800|          2|
| 10| Jane|Finance|  4700|          3|
|  3|Cathy|     HR|  4200|          1|
|  1|Alice|     HR|  4000|          2|
|  8|Helen|     HR|  3900|          3|
|  9|  Ian|     IT|  6100|          1|
|  4|David|     IT|  5500|          2|
|  7|Grace|     IT|  5300|          3|
|  2|  Bob|     IT|  5000|          4|
+---+-----+-------+------+-----------+
```

Explanation:

Window functions allow row-wise operations across grouped data. row_number() assigns a rank to each employee within their department based on descending salary.

6. Display only employee name and department

selected_df = emp_df.select("name", "dept")

selected_df.show()

```
+-----+-------+
| name|   dept|
+-----+-------+
|Alice|     HR|
|  Bob|     IT|
|Cathy|     HR|
|David|     IT|
|  Eva|Finance|
|Frank|Finance|
|Grace|     IT|
|Helen|     HR|
|  Ian|     IT|
| Jane|Finance|
+-----+-------+
```

Explanation:

select() lets you choose specific columns — here we limit output to just name and department, which is helpful for minimal views.

7. Add a new column for salary after 15% hike

hike_df = emp_df.withColumn("new_salary", emp_df.salary * 1.15)

hike_df.show()

```
+---+-----+-------+------+-----------------+
| id| name|   dept|salary|       new_salary|
+---+-----+-------+------+-----------------+
|  1|Alice|     HR|  4000|           4600.0|
|  2|  Bob|     IT|  5000|           5750.0|
|  3|Cathy|     HR|  4200|           4830.0|
|  4|David|     IT|  5500|6324.999999999999|
|  5|  Eva|Finance|  6000|6899.999999999999|
|  6|Frank|Finance|  4800|           5520.0|
|  7|Grace|     IT|  5300|6094.999999999999|
|  8|Helen|     HR|  3900|           4485.0|
|  9|  Ian|     IT|  6100|7014.999999999999|
| 10| Jane|Finance|  4700|           5405.0|
+---+-----+-------+------+-----------------+
```

Explanation:

withColumn() adds a new column or modifies an existing one. Here, we compute the increased salary (15% hike) for all employees.

8. Sort employees alphabetically by name

sorted_df = emp_df.orderBy("name")

sorted_df.show()

```
+---+-----+-------+------+
| id| name|   dept|salary|
+---+-----+-------+------+
|  1|Alice|     HR|  4000|
|  2|  Bob|     IT|  5000|
|  3|Cathy|     HR|  4200|
|  4|David|     IT|  5500|
|  5|  Eva|Finance|  6000|
|  6|Frank|Finance|  4800|
|  7|Grace|     IT|  5300|
|  8|Helen|     HR|  3900|
|  9|  Ian|     IT|  6100|
| 10| Jane|Finance|  4700|
+---+-----+-------+------+
```

Explanation:

orderBy() sorts the DataFrame. Sorting alphabetically makes it easier to search or display names in a readable format.

9. Find distinct departments present in employee data

distinct_dept_df = emp_df.select("dept").distinct()

distinct_dept_df.show()

```
+-------+
|   dept|
+-------+
|     HR|
|Finance|
|     IT|
+-------+
```

Explanation:

distinct() removes duplicate rows. This is useful to identify how many unique departments are represented.

10. Drop the salary column from the dataset

dropped_df = emp_df.drop("salary")

dropped_df.show()

```
+---+-----+-------+
| id| name|   dept|
+---+-----+-------+
|  1|Alice|     HR|
|  2|  Bob|     IT|
|  3|Cathy|     HR|
|  4|David|     IT|
|  5|  Eva|Finance|
|  6|Frank|Finance|
|  7|Grace|     IT|
|  8|Helen|     HR|
|  9|  Ian|     IT|
| 10| Jane|Finance|
+---+-----+-------+
```

Explanation:

drop() is used to remove unnecessary columns. This can reduce data size and make views simpler when salary is not needed.

Actions

1. Show all records in a DataFrame

emp_df.show()

```
+---+-----+-------+------+
| id| name|   dept|salary|
+---+-----+-------+------+
|  1|Alice|     HR|  4000|
|  2|  Bob|     IT|  5000|
|  3|Cathy|     HR|  4200|
|  4|David|     IT|  5500|
|  5|  Eva|Finance|  6000|
|  6|Frank|Finance|  4800|
|  7|Grace|     IT|  5300|
|  8|Helen|     HR|  3900|
|  9|  Ian|     IT|  6100|
| 10| Jane|Finance|  4700|
+---+-----+-------+------+
```

Explanation:

This is a basic action that prints rows of the DataFrame to the console.

2. Collect all rows into Python list

rows = emp_df.collect()

for row in rows:

    print(row.name, row.salary)

```
Alice 4000
Bob 5000
Cathy 4200
David 5500
Eva 6000
Frank 4800
Grace 5300
Helen 3900
Ian 6100
Jane 4700
```

Explanation:

collect() brings the entire DataFrame to the driver as a list. Be careful with big data!

3. Count total employees

total = emp_df.count()

print("Total employees:", total)

```
Total employees: 10
```

Explanation:

count() is an action to get the number of rows — useful for record counting.

4. Get the first record

first = emp_df.first()

print(first)

```
Row(id=1, name='Alice', dept='HR', salary=4000)
```

Explanation:

first() returns the top row. Often used for previewing data structure.

5. Convert PySpark DataFrame to Pandas

pandas_df = emp_df.toPandas()

print(pandas_df.head())

```
   id   name     dept  salary
0   1  Alice       HR    4000
1   2    Bob       IT    5000
2   3  Cathy       HR    4200
3   4  David       IT    5500
4   5    Eva  Finance    6000
```

Explanation:

This action moves data to Pandas format — useful for small data and plotting.

6. Describe dataset (summary stats)

emp_df.describe().show()

```
+-------+-----------------+-----+-------+-----------------+
|summary|               id| name|   dept|           salary|
+-------+-----------------+-----+-------+-----------------+
|  count|               10|   10|     10|               10|
|   mean|              5.5| NULL|   NULL|           4950.0|
| stddev|3.0276503540974917| NULL|   NULL|782.0912137766711|
|    min|                1|Alice|Finance|             3900|
|    max|               10| Jane|     IT|             6100|
+-------+-----------------+-----+-------+-----------------+
```

Explanation:

describe() gives count, mean, stddev, min, and max for numeric columns.

7. Print schema of the DataFrame

emp_df.printSchema()

```
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- dept: string (nullable = true)
 |-- salary: integer (nullable = true)
```

Explanation:

printSchema() reveals column names, types, and nullability. It's useful for understanding structure before processing.