Ministry of Higher Education and Scientific Research
University of Abdelhafid Boussouf - Mila
Institute of Mathematics and Computer Science
Department of Computer Science
Master 2 I2A – Big Data
2025/2026

Directed Works TD 5 – Working with DataFrames and Spark SQL (Solution)

Exercise 1: Creating DataFrames

Question 1:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("SalesAnalysis").getOrCreate()
# TODO: Load the CSV file into a DataFrame
df = spark.read.option("header", True).csv("sales.csv")
# Display the first few rows
df.show()
```

Question 2: *(Assuming Spark infers numeric types correctly.)*

Column	**Type**
Product	String
Category	String
Price	Double
Quantity	Integer
Region	String

Exercise 2: Basic Transformations

Context: We now want to find the total revenue per product (Revenue = Price × Quantity).

Question 1: Fill in the blanks to create a new column called *"Revenue"*:

```
from pyspark.sql.functions import col
df2 = df.withColumn("Revenue", col("Price") * col("Quantity"))
```

Question 2: Write the DataFrame code to select only the columns: *Product, Category, and Revenue*, and show the first 10 rows.

```
df2.select("Product", "Category", "Revenue").show(10)
```

Question 3: If the DataFrame has 100,000 rows, and we call df2.show(10),

- Action. Because it triggers Spark to execute the DAG and display actual results on screen (collecting data from the cluster to the driver).

Exercise 3: Aggregations

Question 1: Complete this code fragment:

```
from pyspark.sql.functions import sum
category_revenue = df2.groupBy("Category").agg(sum("Revenue").alias("TotalRevenue"))
```

Question 2: If Spark executes this, will it trigger a shuffle operation between nodes? Explain your reasoning in one or two sentences.

- Yes, Spark performs a shuffle.
- groupBy() requires moving records with the same key ("Category") to the same node to compute the sum : this redistribution is called shuffle.

Exercise 4: Using Spark SQL

Context: The same DataFrame df2 is registered as a temporary SQL table.

```
df2.createOrReplaceTempView("sales")
```

Question 1: Write the SQL query to find the top 3 categories by total revenue.

```
SELECT Category, SUM(Revenue) AS TotalRevenue
FROM sales
GROUP BY Category
ORDER BY TotalRevenue DESC
LIMIT 3;
```

Question 2: Translate your SQL query into equivalent DataFrame API code.

```
df2.groupBy("Category") \
   .agg(sum("Revenue").alias("TotalRevenue")) \
   .orderBy(col("TotalRevenue").desc()) \
   .limit(3)
```

Exercise 5: Logical Reasoning

For each statement below, mark whether it is **TRUE** or **FALSE** and justify your answer:

Statement	**True/False**	**Explanation**
a) filter() and where() perform the same operation.	**True**	where() is an alias of filter() in DataFrame API.
b) collect() is safe to use on very large datasets.	**False**	It brings all data to the driver; may cause memory overflow.
c) Transformations in Spark are executed immediately.	**False**	Transformations are **lazy**; executed only when an action runs.
d) persist() and cache() store intermediate results in memory.	**True**	Both keep results in memory (cache = persist with MEMORY_ONLY).
e) Spark SQL queries are converted internally into DataFrame transformations.	**True**	SQL queries compile into the same logical plan as DataFrame operations.

Exercise 6: Mini Case Study

Q1. Average score per city (descending):

```
SELECT City, AVG(Score) AS AverageScore
FROM students
GROUP BY City
ORDER BY AverageScore DESC;
```

Q2. Department with highest average per city:

```
SELECT City, Department, AVG(Score) AS AverageScore
FROM students
GROUP BY City, Department;
```

(Optionally, to get only the top department per city, use a window function.)

Q3. Feature to avoid re-reading data:

- Use **caching** or **persistence** (df.cache() / df.persist()),
  so Spark keeps the dataset in memory and reuses it across queries.

Q4. Fault tolerance:

- Spark reconstructs lost data using **RDD lineage**, it knows how each partition was derived and re-executes only the lost tasks on another node.

Q5. MapReduce comparison:

- Needs to write/read from disk between each phase (slow).
- No built-in SQL or DataFrame abstraction (complex code).
- High latency for iterative or interactive tasks.
- Lacks caching and in-memory computation.