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

Practical Works 4 - PySpark Foundations: From Principles to Practice (Solutions)

Setup (for Google Colab)

```
!pip install pyspark
```

Part 1: Creating the Spark Context

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("PySpark Foundations") \
    .getOrCreate()

print("Spark version:", spark.version)
print("App name:", spark.sparkContext.appName)
print("Master:", spark.sparkContext.master)
```

Answers:

- The driver is the process controlling execution — here, the Colab notebook.
- Executors run on worker nodes in a cluster to perform distributed computations.

Part 2: From Code to DAG (Transformations vs Actions)

```
data = [("Ali", "CS", 16), ("Sara", "Math", 12),
        ("Rachid", "CS", 18), ("Nada", "Bio", 14)]

df = spark.createDataFrame(data, ["Name", "Department", "GPA"])

# Transformations – not executed yet
df_filtered = df.filter(df.GPA >= 15)
df_mapped = df_filtered.select("Name", "Department")

# Action – triggers DAG execution
df_mapped.show()
```

Answers:

- Transformations: `filter()`, `select()`
- Action: `show()` (it triggers actual computation)

Part 3: Inspecting Schema and Types

```
df.printSchema()
df.describe().show()
df.dtypes
```

Then, to enforce integer type:

```
from pyspark.sql.functions import col
df2 = df.withColumn("GPA", col("GPA").cast("integer"))
```

Answer:

Schema enforcement allows Spark to optimize execution plans and avoid unnecessary type conversions.

Part 4: Wide vs Narrow Transformations

```
from pyspark.sql.functions import avg

df_filtered = df.filter(df.GPA > 10)
df_avg = df.groupBy("Department").agg(avg("GPA").alias("AvgGPA"))
df_avg.show()
```

```
Answers:
filter() → narrow (no shuffle, operates within partitions)
groupBy() → wide (requires data shuffle across nodes)
```

Part 5: Lazy Evaluation in Practice

```
from pyspark.sql.functions import when, col

df3 = df.withColumn("Performance",
                    when(col("GPA") >= 16, "Excellent")
                    .when(col("GPA") >= 12, "Good")
                    .otherwise("Weak"))

print("Before action – nothing executed yet!")

df3.cache()        # Cache in memory
df3.show()         # First action triggers execution
df3.show()         # Second action retrieves from cache
```

Answers:

- The first show() triggers computation.
- The second show() is faster because Spark uses the cached data (no re-execution).

Part 6: Fault Tolerance Demonstration (RDD Lineage)

```
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
rdd2 = rdd.map(lambda x: x * 2)
print(rdd2.collect())
```

Answer:

If a partition is lost, Spark recomputes it using the lineage graph (the recorded sequence of transformations).

Part 7: End-to-End Example – "City Sales Analysis"

```
%%writefile sales.csv
Product,Category,Price,Quantity,City
Apple,Fruit,2.5,10,Oran
Banana,Fruit,1.2,20,Oran
TV,Electronics,500,3,Algiers
Laptop,Electronics,900,2,Algiers
Orange,Fruit,1.8,15,Constantine
Camera,Electronics,300,4,Oran
```

Now the code:

```
from pyspark.sql.functions import col, sum, avg, round

# 1. Load the data
df = spark.read.option("header", True).csv("sales.csv", inferSchema=True)

# 2. Compute revenue
df = df.withColumn("Revenue", col("Price") * col("Quantity"))

# 3. Total revenue per city
city_revenue = df.groupBy("City").agg(sum("Revenue").alias("TotalRevenue"))
city_revenue.show()

# 4. Average revenue per category per city
category_avg = df.groupBy("City", "Category").agg(round(avg("Revenue"), 2).alias("AvgRevenue"))
category_avg.show()

# 5. Top city by total revenue
top_city = city_revenue.orderBy(col("TotalRevenue").desc()).limit(1)
top_city.show()
```

Answers:

- Transformations: `withColumn`, `groupBy`, `agg`, `orderBy`
- Actions: `show()`, `limit()`
- Shuffle operations: `groupBy`, `orderBy`

- Caching benefit: Avoids re-reading/recomputing intermediate results for multiple analyses.

Part 8: Reflection

- Lazy Evaluation: Spark builds the DAG and executes only when an action is called.
- Fault Tolerance: Spark recomputes lost partitions using RDD lineage.
- Optimization: Catalyst optimizer rearranges and simplifies query plans before execution.
- Performance vs MapReduce: Spark keeps intermediate data in memory, reducing disk I/O and improving speed dramatically.