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 4 – Working with RDDs and Transformations in Apache Spark (Solution)

Question 1: Explain in your own words what lazy evaluation means in Spark:

Lazy evaluation means that Spark doesn't execute transformations immediately. Instead, it builds a logical plan (DAG) of transformations, and the computation only runs when an action (like collect() or count()) is called.

This allows Spark to optimize the execution before running.

Question 2: Fill in the blanks:

In Spark, transformations are **lazy** (they don't execute immediately), while actions are **eager** (they trigger the execution).

Question 3: If we execute the line *rdd.count()*, what happens internally?

a. Spark immediately creates all partitions.
b. Spark builds a DAG and executes it because *count()* is an action.
c. Spark executes line by line without optimization.

Answer: Spark builds a DAG and executes it because count() is an action. When count() is called, Spark triggers execution by reading the data, applying transformations, and returning the result.

Question 4: Complete the following sentence:

- map() always produces **one** output element(s) per input, while flatMap() can produce **zero, one, or many**.

Explanation:

- *map()* → one-to-one transformation
- *flatMap()* → one-to-many (e.g., splitting sentences into words)

Question 5: When would you not use collect()?

When the dataset is very large, because collect() brings all data to the driver. It should only be used for small datasets or debugging.

For large datasets, use **take(n)** or write results to storage using **saveAsTextFile().**

Mini-Exercise :

- **Step 1**: Create an RDD → rdd = spark.sparkContext.parallelize(grades)
- **Step 2**: Transform to (student, grade) → rdd2 = rdd.map(lambda x: (x[0], x[2]))
- **Step 3**: Compute total and count →
- **totals** = rdd2.combineByKey(lambda v: (v,1), lambda acc,v: (acc[0]+v, acc[1]+1), lambda a,b: (a[0]+b[0], a[1]+b[1]))
- **Step 4**: Compute averages → avg = totals.mapValues(lambda x: x[0]/x[1])

Question 6: If one node fails during computation, how does Spark recover the missing data?

Spark uses the RDD lineage graph. If a partition is lost, Spark recomputes it from its original data using the transformation history. No manual replication is needed because the DAG describes how each partition was created.

Question 7: Explain the difference between RDD persistence and re-computation:

- Persistence: Keeps an RDD in memory or disk after first computation → avoids recomputation next time.
- Re-computation: Spark rebuilds the lost RDD partition from lineage if needed.

Persistence is used to improve performance when the same data is used multiple times.

Question 8: Why is *reduceByKey* preferred over *groupByKey* in most cases?

Because reduceByKey aggregates data locally on each node before shuffling. groupByKey sends all values over the network before aggregation, causing more network I/O.

Hence, reduceByKey is faster and more efficient.

Optional Challenge

Goal: Count frequency of each word.

Solution (conceptual):

1. Create RDD → rdd = sc.parallelize(data)
2. Split words → rdd2 = rdd.flatMap(lambda x: x.split(" "))
3. Map each word → rdd3 = rdd2.map(lambda x: (x, 1))
4. Reduce by key → rdd4 = rdd3.reduceByKey(lambda a,b: a+b)
5. Collect result → rdd4.collect()

Expected output: [("big",2), ("data",3), ("spark",1), ("analytics",1), ("processing",1)]