PART C

C.1 What is RDD (Explain the Resilient and Distributed)? How does Spark work with RDD?

An RDD, or Resilient Distributed Dataset, is the fundamental data structure of Apache Spark. It is "resilient" because it has built-in fault tolerance and can rebuild lost data using lineage information. This means that if any partition of an RDD is lost, Spark can recompute it from the original transformations that created it. The "distributed" aspect refers to how RDDs are distributed across the cluster on multiple nodes, which allows for parallel processing on multiple machines. It is immutable object, which means once its created, it cannot be changed. Spark works with RDDs by performing two types of operations: transformations, which create a new RDD from an existing one, and actions, which return a value to the driver program after running a computation on the RDD. Transformations are lazy, meaning they're not computed immediately but rather just add to the lineage of the RDD. Actions, on the other hand, trigger the actual computation.

C.2 Your colleague is trying to understand her Spark code by adding a print statement inside her split_line(..) function, as shown in this code snippet:

```
def split_line(line):
        print('splitting line...')
        return line.split(' ')
lines = spark_context.textFile("hdfs://host:9000/i-have-a-dream.txt")
print(lines.flatMap(split_line).take(10))
```

When she runs this code in her notebook, she sees the following output:

['I', 'am', 'happy', 'to', 'join', 'with', 'you', 'today', 'in','what']

But, she doesn't see the "splitting line…" output in her notebook. Why not?

The "splitting line..." output doesn't appear in notebook because the 'split_line' function is executed on the worker nodes and not on the driver node where the notebook is running. Spark operations that modify data, like 'flatMap', are distributed and performed in parallel across the cluster. The 'print' statements inside such functions will be executed on the worker nodes and the output will typically be written to the worker's logs, not to the standard output of the driver program. If she wants to see the output of 'print' statements during debugging, she would need to check the logs of the worker nodes.

C.3 Calling .collect() on a large dataset may cause driver application to run out of memory Explain why.

Calling '.collect()' on a large dataset may cause the driver application to run out of memory because this action retrieves the entire dataset from the worker nodes and tries to load it into the memory of the driver node. If the dataset is too large, it will exceed the available memory of the driver, potentially causing an out-of-memory error. This is why '.collect()' should be used with caution, especially with large datasets, and operations should be designed to minimize the amount of data that needs to be sent to the driver.

C.4 Are partitions mutable or immutable? Why is this advantageous?

Partitions in Spark are immutable, meaning once a dataset is partitioned, the partitioning cannot be changed. This immutability is advantageous for several reasons. Firstly, it provides fault

tolerance. If a partition is lost, it can be recomputed from the original dataset because the operations that created it are deterministic and do not change. Secondly, it allows for concurrency and parallelism. Multiple tasks can read the data in a partition simultaneously without any risk of data being changed by other tasks, enabling efficient parallel computations. Thirdly, it enhances efficiency by eliminating the need for locks or other synchronization primitives, which can be a significant source of overhead in parallel computations. It also makes it easier to cache partitions in memory, as there's no need to invalidate or update cached data when it changes. Lastly, Spark keeps track of the lineage of each partition (i.e., the sequence of transformations used to create it), which is used for recomputing lost data and for optimizing computations. Therefore, the immutability of partitions in Spark is a key factor in enabling efficient, parallel, and fault-tolerant computations on large datasets.

C.5 What is the difference between DataFrame and RDD? Explain their advantages/disadvantages?

The difference between DataFrames and RDDs lies in their abstraction levels and the operations they support. DataFrames are a higher-level abstraction compared to RDDs and are part of the Spark SQL library. They allow you to work with data in a more structured way, with rows and named columns, similar to tables in relational databases.

Advantages of DataFrames:
- They have a richer optimization engine (Catalyst optimizer) that can perform advanced optimizations (like predicate pushdown, logical plan optimization).
- DataFrames support various data sources and can easily handle different data formats.
- They provide a domain-specific language for structured data manipulation, allowing users to perform SQL-like queries.

Disadvantages of DataFrames:
- They are less flexible than RDDs as they impose a structure on the data.
- Advanced or custom data manipulations might be more complex to implement compared to using RDDs.

Advantages of RDDs:
- They provide fine-grained control over the physical distribution and partitioning of data across the cluster (useful for complex, customized parallel computations).
- They allow for more flexible and complex manipulations since you can apply any function to the data.

Disadvantages of RDDs:
- They lack the built-in optimization engine, so the user must manually optimize their distributed computations.
- They are more low-level, meaning more verbose code for operations that are easily done with DataFrames.

In summary, DataFrames should be used when you can leverage the built-in optimizations and work with structured data, while RDDs are better suited for tasks that require fine-grained control and custom parallel processing.

PART D

A colleague has mentioned her Spark application has poor performance, what is your advice?

Improving the performance of a Spark application can be approached from several angles, focusing on optimization strategies that leverage Spark's architecture and design principles. Here are four recommendations:

1. Avoid User-Defined Functions (UDFs) When Possible:
User-Defined Functions (UDFs) in Spark can significantly reduce performance, especially when they are not properly optimized. UDFs operate on a row-by-row basis, which can lead to increased serialization and deserialization overhead, ultimately slowing down computation. Instead of UDFs, try to leverage Spark's built-in functions as much as possible, as these are optimized to run efficiently on the entire dataset. Spark SQL and the DataFrame API offer a wide range of built-in functions that can cover many use cases that would otherwise require UDFs. When you absolutely must use UDFs, consider using Pandas UDFs (for PySpark) or vectorized UDFs, as they can offer better performance by processing data in batches rather than row by row. This approach can significantly minimize the overhead associated with traditional UDFs and leverage the optimization capabilities of Spark.

2. Tune Spark Configuration Parameters:
Adjusting Spark's configuration parameters can lead to substantial performance improvements. For instance, tuning the ‘spark.executor.memory’ and ‘spark.executor.cores’ parameters allows you to optimize the resource allocation for your application, ensuring that executors have enough memory and CPU cores to process tasks efficiently. Furthermore, configuring the ‘spark.sql.shuffle.partitions’ parameter to match the number of cores in your cluster can optimize the shuffle process, reducing bottlenecks during wide transformations.

3. Use Dataframe/Dataset API Over RDDs:
Whenever possible, use the Dataframe/Dataset API instead of RDDs. In PySpark use, DataFrame over RDD as Dataset’s are not supported in PySpark applications. The Dataframe/Dataset API offers more efficient execution plans through its Catalyst optimizer, which applies advanced optimization techniques such as predicate pushdown and constant folding. These APIs also benefit from Tungsten's off-heap memory management and code generation, making them much faster and less memory-intensive than RDDs for most operations.

4. Minimize Shuffles and Optimize Caching:
Shuffling is a costly operation in Spark that involves redistributing data across different executors or nodes. To minimize shuffles, try to collocate related data and operations that can benefit from data locality. Use narrow transformations (e.g., ‘map’, ‘filter’) as much as possible, and avoid unnecessary wide transformations (e.g., ‘groupBy’, ‘join’). Additionally, caching datasets in

memory with ‘persist()’ or ‘cache()’ can be beneficial for data that is accessed multiple times, reducing the need to recompute or reread data from disk.