DS 644: Homework 3

Instructions. Answer the following multiple choice questions by selecting the correct choices.

1. **Principles of (functional) programming**

   (a) Which of the following are programming paradigms? (Select three.)

   □ Declarative □ Functional □ Hadoop □ Imperative □ Scala

   (b) What three concepts characterize a purely functional programming language?

   □ referential transparency □ input/output □ no side effects □ procedural
   □ immutability

2. **Big data properties**.

   (a) In lecture we discussed the meaning of the term "Big Data." We decided that, for simplicity, we will call data "big" when it is

   □ at least 1Gb
   □ too big to fit in fast memory (cpu cache + ram) on a single compute node
   □ too big to fit in all computer memory (whether fast or slow)
   □ too big to be dealt with by traditional data-processing software

   (b) "Big Data" concerns which of the following types of data?

   □ structured □ semi-structured □ unstructured □ all of these

   (c) JSON and XML are examples of which type of data?

   □ structured □ unstructured □ semi-structured □ none of these

   (d) Which two of the following statements are true of unstructured data?

   □ It is generally easier to analyze than other types of data.
   □ It is often referred to as "messy" data.
   □ It fits neatly into a schema.
   □ It is the most widespread type of data.
   □ It is usually found in tables.

3. **Latency and fault-tolerance**.

   (a) *Latency* is degradation in performance due to...

   □ a small number of cores in the central processing unit
   □ slow data transfer across the network or cluster
   □ shuffling data between different nodes in a cluster
   □ failure of one or more nodes in the cluster
   □ stack overflow caused by recursion

   (b) Hadoop achieves fault-tolerance by...

   □ using lazy evaluation and garbage collection.
   □ writing intermediate computations to disk.
   □ keeping all data immutable and in-memory.
   □ replaying functional transformations over the original (immutable) dataset.

(c) Spark decreases latency while remaining fault-tolerant by...

- □ using ideas from imperative programming.
- □ using ideas from functional programming.
- □ discarding data when it's no longer needed.
- □ keeping all data immutable and in-memory.
- □ replaying functional transformations over the original (immutable) dataset.

4. **Transformations and actions**.

(a) In Spark a **transformation** on an RDD...

- □ is eagerly evaluated.
- □ is lazily evaluated.
- □ immediately computes and returns a result.
- □ does not immediately compute a result.
- □ usually returns another RDD (once it's evaluated).

(b) In Spark an **action** on an RDD...

- □ is eagerly evaluated.
- □ is lazily evaluated.
- □ immediately computes and returns a result.
- □ does not immediately compute a result.
- □ always returns another RDD (once it's evaluated).

(c) After performing a series of transformations on an `RDD`, which of the following methods could you use to make sure those transformations are not repeated (e.g., on each iteration of an algorithm)?

- □ `save`
- □ `persist`
- □ `memoize`
- □ There is no such method because of the JVM's garbage collection mechanism.

(d) Why does Spark's `RDD` class not have a `foldLeft` method?

- □ `foldLeft` can only be performed on lists of Boolean values.
- □ `foldLeft` doesn't work on immutable collections.
- □ `foldLeft` is not stack-safe.
- □ `foldLeft` is not fault-tolerant.
- □ `foldLeft` is not parallelizable.

(e) Why is available in Spark's `RDD` class that overcomes the limitation of `foldLeft` mentioned in the previous part of this exercise?

□ `aggregate`   □ `fold`   □ `foldLeft`   □ `join`   □ `leftOuterJoin`

5. **Read the docs**. Navigate to the Spark API documentation at

   `https://spark.apache.org/docs/3.3.1/api/scala/org/apache/spark/index.html`

   Enter "RDD" in the search box and select `RDD` from the results that appear on the left.

   (a) Scroll down the resulting `RDD` API documentation page and find the `cache()` method. What does it say?

   □ Persist this RDD with the default storage level (`MEMORY_ONLY`).

   □ Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

   □ Set this RDD's storage level to persist its values across operations after the first time it is computed.

   □ Save this RDD as a SequenceFile of serialized objects.

   (b) On the `RDD` API doc page, find the version of `persist` that takes an argument: `def persist(newLevel: StorageLevel)`. What does it say?

   □ Persist this RDD with the default storage level (`MEMORY_ONLY`).

   □ Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

   □ Set this RDD's storage level to persist its values across operations after the first time it is computed.

   □ Save this RDD as a SequenceFile of serialized objects.

   (c) On the `RDD` API doc page, find the `unpersist` method. What does it say?

   □ Persist this RDD with the default storage level (`MEMORY_ONLY`).

   □ Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

   □ Set this RDD's storage level to persist its values across operations after the first time it is computed.

   □ Save this RDD as a SequenceFile of serialized objects.

   (d) What's the difference between the `sample` and `takeSample` methods of the `RDD` class?

   □ `sample` always uses a with-replacement sampling method, while `takeSample` always samples without replacement.

   □ `sample` returns an `RDD`, while `takeSample` returns an `Array`.

   □ The second argument specifies the number of samples desired either as a fraction of the size of the RDD (`sample`) or as an absolute number (`takeSample`).

   □ There is no difference; they are just two different names one can use to invoke the same function.