

Instructions. Answer the following multiple choice questions by selecting all correct choices. If a question has more than one correct choice, it will say in parentheses how many items you should select. *Select all correct choices to receive full credit!*

1. (6 pts) **Big data properties.**

- (a) “Big Data” concerns which of the following types of data?
☐ structured ☐ semi-structured ☐ unstructured ☒ ***all of these***
- (b) JSON and XML are examples of which type of data?
☐ structured ☐ unstructured ☒ ***semi-structured*** ☐ none of these
- (c) Which of the following statements is true of unstructured data?
☐ It is generally easier to analyze than other types of data.
☐ It fits neatly into a schema.
☒ ***It is the most widespread type of data.***
☐ It is usually found in tables.

Explanation. (a) Big Data is a blanket term for the data that are too large in size and complex in nature, and which may be structured, unstructured, or semi-structured, and may also be arriving at high velocity.

(b) Semi-structured data are that which have a structure but do not fit into the relational database. Semi-structured data are organized, which makes it easier for analysis when compared to unstructured data. JSON and XML are examples of semi-structured data.

2. (6 pts) **Hardware and Architecture.**

- (a) What kind of hardware is typically used for big data applications?
☐ high-performance supercomputers
☒ ***low-cost, commodity hardware***
☐ dumb terminals
☐ floppy disks
- (b) What is “commodity” hardware?
☐ high-performance supercomputers
☐ discarded, second-hand, or recycled hardware
☐ hardware used for trading commodities (e.g., gold, silver, soy-beans)
☒ ***generic, low-specification, industry-grade hardware***
- (c) Which of the following is **not** a drawback of traditional relational database management system (or RDBMS) when used for big data applications?
☐ They do not make it easy to handle massive volumes of unstructured or semi-structured data.
☐ They require more processors and memory to scale up to big data applications.
☒ ***They are relatively slow when used to perform SQL queries on large structured data tables.***
☐ They do not make it easy to capture and process unstructured or semi-structured data arriving at high velocity.

Explanation. (a) Big data uses low-cost commodity hardware to make cost-effective solutions.

(b) Commodity hardware is a low-cost, low performance, and low specification functional hardware with no distinctive features.

3. (10 pts) **Programming Paradigms**

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

- ☐ currying ☒ ***declarative*** ☐ dysfunctional ☒ ***functional*** ☒ ***object-oriented***

(b) Which of the following characteristics are typical of imperative programs. (select two)

- ☒ ***variables are mutable; their values may change or “mutate”***
☒ ***for loops are usually preferred in favor of recursive function calls***
☐ functions are *referentially transparent*
☐ functions are “pure” (do not have *side-effects*)

(c) Which of the following characteristics are typical of functional programs. (select two)

- ☐ variables are *mutable*; their values may change or “mutate”
☐ for loops are usually preferred in favor of recursive function calls
☒ ***functions are referentially transparent***
☒ ***functions are “pure” (do not have side-effects)***

(d) By definition, a *higher-order function* is a function which

- ☐ is passed as an argument to other functions.
☐ is returned as output by other functions.
☐ is called a higher-order number of times in comparison to “lower-order” functions.
☐ requires a higher-order amount of time to compute in comparison to “lower-order” functions.
☒ ***accepts a function (or functions) as input or returns a function (or functions) as output.***

(e) An expression *e* is called *referentially transparent* provided

- ☐ when reduced to “normal form” *e* is obvious or “transparent.”
☐ the values of expressions to which *e* refers are obvious or “transparent.”
☒ ***for all programs p, all occurrences of e in p can be replaced by the result of evaluating e without affecting the meaning of p.***
☐ none of the above

4. (9 pts) **Scala I**

(a) The main programming paradigms of Scala are which of these? (select two)

- ☐ currying ☐ declarative ☐ dysfunctional ☒ **functional** ☒ **object-oriented**

(b) What is the result of the following program?

```
val x = 0
def f(y: Int) = y + 1
val result = {
  val x = f(3)
  x * x
} + x
```

- ☐ 0 ☒ **16** ☐ 32 ☐ it does not type check

(c) Why should we care about writing functions that are “tail-recursive?”

- ☐ Recursion should be carried out on the tail, not the head.
☐ Recursion should be carried out on the head, not the tail.
☒ **They are “stack-safe”—they help us avoid stack overflows.**
☐ They are “disk-safe”—they help us avoid network storage leaks.

5. (6 pts) Let `val X = List(1, 2, 3)`
and `val Y = List(1, 2, 3)`.

(a) To what does the expression `X.map(x => Y.map(y => y - x))` evaluate?

- ☐ `List(0, 0, 0, 0, 0, 0, 0, 0, 0)`
☐ `List(List(0, 0, 0), List(0, 0, 0), List(0, 0, 0))`
☐ `List(0, -1, -2, 1, 0, -1, 2, 1, 0)`
☐ `List(0, 1, 2, -1, 0, 1, -2, -1, 0)`
☒ `List(List(0, 1, 2), List(-1, 0, 1), List(-2, -1, 0))`

(b) To what does the expression `X.flatMap(x => Y.map(y => y - x))` evaluate?

- ☐ `List(0, 0, 0, 0, 0, 0, 0, 0, 0)`
☐ `List(List(0, 0, 0), List(0, 0, 0), List(0, 0, 0))`
☐ `List(List(0, -1, -2), List(1, 0, -1), List(2, 1, 0))`
☒ `List(0, 1, 2, -1, 0, 1, -2, -1, 0)`
☐ `List(List(0, 1, 2), List(-1, 0, 1), List(-2, -1, 0))`

6. (12 pts) **Scala II.** The parts below refer to the function

```
def test(x:Bool, y:Int) = if (x) (y + 2)/y else 0
```

Let CBN = call-by-name

and CBV = call-by-value.

(a) Which strategy evaluates `test(true, 2)` most efficiently (in the fewest steps)?

- ☐ CBN ☐ CBV ☒ ***CBN and CBV require the same number of steps***

Explanation. They both perform one addition ($2 + 2$) and one division ($(4/2)$).

(b) Which strategy evaluates `test(true, 1+1)` most efficiently?

- ☐ CBN ☒ ***CBV*** ☐ CBN and CBV require the same number of steps

Explanation. CBV performs two additions ($1 + 1$ and $2 + 2$) and one division ($(4/2)$), while CBN performs three additions ($1 + 1$ and $2 + 2$ and $1 + 1$ again) and one division ($(4/2)$).

(c) Which strategy evaluates `test(false, 2)` most efficiently?

- ☐ CBV ☐ CBN ☒ ***CBN and CBV require the same number of steps***

(d) Which strategy evaluates `test(false, 1+1)` most efficiently?

- ☒ ***CBN*** ☐ CBV ☐ CBN and CBV require the same number of steps

Explanation. CBV performs one addition ($1 + 1$), while CBN performs no operations.

7. (9 pts) **Latency and fault-tolerance.**

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

- ☐ 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***
☐ 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... (select three)

- ☐ 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.***

8. (12 pts) **Transformations and actions.**

(a) A **transformation** on an RDD... (select two)

- ☒ *does not immediately compute a result.*
- ☐ immediately computes and returns a result.
- ☒ *is lazily evaluated.*
- ☐ is eagerly evaluated.

(b) An **action** on an RDD... (select two)

- ☐ does not immediately compute a result.
- ☒ *immediately computes and returns a result.*
- ☐ is lazily evaluated.
- ☒ *is eagerly evaluated.*

(c) After performing a series of transformations on an RDD, which of the following methods would ensure that Spark actually carries out the transformations.

- ☐ `mapValues()` ☒ *`collect()`* ☐ `groupByKey()` ☐ none of these

(d) 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 unnecessarily?

- ☐ `save()` ☒ *`persist()`* ☐ `collect()` ☐ `parallelize()`

(e) Why does the RDD class have no `foldLeft` method?

- ☐ `foldLeft` is not stack-safe.
- ☐ `foldLeft` is not fault-tolerant.
- ☐ `foldLeft` only works on PairRDDs.
- ☒ *`foldLeft` is not parallelizable.*
- ☐ It's not true; the RDD class *does* have a `foldLeft` method.

(f) Which method of the RDD class has the same effect as `foldLeft` and overcomes limitations of the latter?

- ☒ *`aggregate`* ☐ `foldRight` ☐ `join` ☐ `leftOuterJoin` ☐ `collect`

9. (12 pts) *Everyday I'm shufflin'.*

(a) What is shuffling?

- ☐ a method for recovering data after hardware failure
- ☐ the method used to ensure a random number generator is unbiased
- ☐ any movement of data
- ☐ moving data from memory to disk, usually caused by insufficient fast memory
- ✓ ***transferring data between nodes in a cluster, usually in order to complete a computation***

(b) What Spark feature or method can be used to reduce or eliminate shuffling?

- ☐ *fault-tolerance*: use higher quality, fault-tolerant hardware
- ✓ ***partitioning: partition an RDD before applying transformations or actions***
- ☐ *pre-shuffling*: use a pre-shuffled random number generator
- ☐ *data siloing*: keep the entire RDD on a single node of the cluster
- ☐ *caching*: keep the entire RDD in fast memory on a single node

(c) Which transformations on an RDD of type `RDD[Int]` have we learned about or used in this class?

- ☐ `curry` ☐ `foldLeft` ☐ `groupByKey` ✓ ☒ `map` ☐ `mapValues`

(d) Which transformations on a *pair* RDD of type `RDD[(Int, String)]` have we learned about or used in this class? (select three)

- ☐ `curry` ☐ `foldLeft` ✓ ☒ `groupByKey` ✓ ☒ `map` ✓ ☒ `mapValues`

10. (6 pts) **Joins.** Let V and W be types. Suppose `rdd1: RDD[(Int, V)]` and `rdd2: RDD[(Int, W)]` are pair RDDs.

(a) What is the type of `rdd1 join rdd2`.

- ☐ `RDD[(Int, V)]`
- ☐ `RDD[(Int, V \cap W)]`
- ☐ `RDD[(Int, V)]` if `rdd1` is larger than `rdd2`; otherwise, `RDD[(Int, W)]`
- ✓ ***`RDD[(Int, (V, W))]`***
- ☐ `RDD[(Int, (V, Option[W]))]`
- ☐ `RDD[(Int, Option[(V, W)])]`

(b) What is the type of `rdd1 leftOuterJoin rdd2`.

- ☐ `RDD[(Int, V)]`
- ☐ `RDD[(Int, V \cap W)]`
- ☐ `RDD[(Int, V)]` if `rdd1` is larger than `rdd2`; otherwise, `RDD[(Int, W)]`
- ☐ `RDD[(Int, (V, W))]`
- ✓ ***`RDD[(Int, (V, Option[W]))]`***
- ☐ `RDD[(Int, Option[(V, W)])]`

11. (12 pts) **Partitioning.**

Consider a Pair RDD with the following keys: 5, 10, 15, 20, 25, 30, 35, 40, 45. Suppose we partition this RDD into 3 blocks.

- (a) If we use hash partitioning, and if `hashCode()` is the identity (`n.hashCode() == n`), then how many key-value pairs will end up in the first block (block 0) of the partition.

☐ 0 ☐ 1 ☐ 2 ☒ 3 ☐ 4 ☐ 5

- (b) If we use range partitioning with ranges [0, 20], [21, 40], [41, 60], then how many key-value pairs will end up in the first block (block 0) of the partition.

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☒ 4 ☐ 5

- (c) Which strategy results in a more even distribution of the RDD?

☒ (a)'s *hash partitioning* ☐ (b)'s range partitioning ☐ they're equivalent

- (d) Which of the following transformations, when performed on a partitioned RDD, will result in an RDD with the same partitioning scheme?

☐ `map` ☒ `mapValues` ☐ `foldLeft` ☐ `repartition`

FOR PERSONAL USE ONLY - DO NOT DISTRIBUTE