* 1. Which of the following DataFrame operations are wide transformations (that is, they result in a shuffle)?

\*A. repartition()  
B. filter()  
\*C. orderBy()  
\*D. distinct()  
E. drop()  
F. cache()

* 1. Which of the following methods are NOT a DataFrame action?

\*A. limit()  
B. foreach()  
C. first()  
\*D. printSchema()  
E. show()  
\*F. cache()

* 1. Which of the following statements about Spark accumulator variables is NOT true?

\*A. For accumulator updates performed inside actions only, Spark guarantees that each task’s update to the accumulator will be applied only once, meaning that restarted tasks will not update the value. In transformations, each task’s update can be applied more than once if tasks or job stages are re-executed.  
B. Accumulators provide a shared, mutable variable that a Spark cluster can safely update on a per-row basis.  
C. You can define your own custom accumulator class by extending org.apache.spark.util.AccumulatorV2 in Java or Scala or pyspark.AccumulatorParam in Python.  
D. The Spark UI displays all accumulators used by your application.

* 1. Given an instance of SparkSession named spark, review the following code:

import org.apache.spark.sql.functions.\_

val a = Array(1002, 3001, 4002, 2003, 2002, 3004, 1003, 4006)

3,

val b = spark

.createDataset(a)

.withColumn("x", col("value") % 1000)

val c = b

.groupBy(col("x"))

.agg(count("x"), sum("value"))

.drop("x")

.toDF("count", "total")

.orderBy(col("count").desc, col("total"))

.limit(1)

.show()

Which of the following results is correct?

\*A.

+-----+-----+

|count|total|

+-----+-----+

| 3| 7006|

+-----+-----+

B.

+-----+-----+

|count|total|

+-----+-----+

| 1| 3001|

+-----+-----+

C.

+-----+-----+

|count|total|

+-----+-----+

| 2| 8008|

+-----+-----+

D.

+-----+-----+

|count|total|

+-----+-----+

| 8|20023|

+-----+-----+

* 1. Given an instance of SparkSession named spark, which one of the following code fragments executemost quickly and produce a DataFrame with the specified schema? Assume a variable named schema with the correctly structured StructType to represent the DataFrame's schema has already been initialized.

Sample data:

id,firstName,lastName,birthDate,email,country,phoneNumber 1,Pennie,Hirschmann,2017-12-03,[ph123@databricks.com](mailto:ph123@databricks.com),US,+1(123)4567890

Schema:

id: integer

firstName: string

lastName: string

birthDate: timestamp

email: string

county: string

phoneNumber: string

A.

val df = spark.read

.option("inferSchema", "true")

.option("header", "true")

.csv("/data/people.csv")

B.

val df = spark.read

.option("inferSchema", "true")

.schema(schema)

.csv("/data/people.csv")

C.

val df = spark.read

.schema(schema)

.option("sep", ",")

.csv("/data/people.csv")

\*D.

val df = spark.read

.schema(schema)

.option("header", "true")

.csv("/data/people.csv")

* 1. Consider the following DataFrame:

val rawData = Seq(

(1, 1000, "Apple", 0.76),

(2, 1000, "Apple", 0.11),

(1, 2000, "Orange", 0.98),

(1, 3000, "Banana", 0.24),

(2, 3000, "Banana", 0.99)

)

val dfA = spark.createDataFrame(rawData).toDF("UserKey", "ItemKey", "ItemName", "Score")

Select the code fragment that produces the following result:

+-------+-----------------------------------------------------------------+

|UserKey|Collection |

+-------+-----------------------------------------------------------------+

|1 |[[0.98, 2000, Orange], [0.76, 1000, Apple], [0.24, 3000, Banana]]|

|2 |[[0.99, 3000, Banana], [0.11, 1000, Apple]] |

+-------+-----------------------------------------------------------------+

A.

import org.apache.spark.sql.expressions.Window

dfA.withColumn(

"Collection",

collect\_list(struct("Score", "ItemKey", "ItemName")).over(Window.partitionBy("ItemKey"))

)

.select("UserKey", "Collection")

.show(20, false)

B.

dfA.groupBy("UserKey")

.agg(collect\_list(struct("Score", "ItemKey", "ItemName")))

.toDF("UserKey", "Collection")

.show(20, false)

C.

dfA.groupBy("UserKey", "ItemKey", "ItemName")

.agg(sort\_array(collect\_list(struct("Score", "ItemKey", "ItemName")), false))

.drop("ItemKey", "ItemName")

.toDF("UserKey", "Collection")

.show(20, false)

\*D.

dfA.groupBy("UserKey")

.agg(sort\_array(collect\_list(struct("Score", "ItemKey", "ItemName")), false))

.toDF("UserKey", "Collection")

.show(20, false)

* 1. tableA is a DataFrame consisting of 20 fields and 40 billion rows of data with a surrogate key field. tableB is a DataFrame functioning as a lookup table for the surrogate key consisting of 2 fields and 5,000 rows. If the in-memory size of tableB is 22MB, what occurs when the following code is executed:

val df = tableA.join(broadcast(tableB), Seq("primary\_key"))

A. The broadcast function is non-deterministic, thus a BroadcastHashJoin is likely to occur, but isn't guaranteed to occur.  
B. A normal hash join will be executed with a shuffle phase since the broadcast table is greater than the 10MB default threshold and the broadcast command can be overridden silently by the Catalyst optimizer.  
C. The contents of tableB will be replicated and sent to each executor to eliminate the need for a shuffle stage during the join.  
D. An exception will be thrown due to tableB being greater than the 10MB default threshold for a broadcast join.

* 1. Consider the following DataFrame:

import org.apache.spark.sql.functions.\_

val people = Seq(

("Ali", 0, Seq(100)),

("Barbara", 1, Seq(300, 250, 100)),

("Cesar", 1, Seq(350, 100)),

("Dongmei", 1, Seq(400, 100)),

("Eli", 2, Seq(250)),

("Florita", 2, Seq(500, 300, 100)),

("Gatimu", 3, Seq(300, 100))

)

.toDF("name", "department", "score")

Select the code fragment that produces the following result:

+----------+-------+-------+

|department| name|highest|

+----------+-------+-------+

| 0| Ali| 100|

| 1|Dongmei| 400|

| 2|Florita| 500|

| 3| Gatimu| 300|

+----------+-------+-------+

A.

val maxByDept = people

.withColumn("score", explode(col("score")))

.groupBy("department")

.max("score")

.withColumnRenamed("max(score)", "highest")

maxByDept

.join(people, "department")

.select("department", "name", "highest")

.orderBy("department")

.dropDuplicates("department")

.show()

B.

people

.withColumn("score", explode(col("score")))

.orderBy("department", "score")

.select(col("name"), col("department"), first(col("score")).as("highest"))

.show()

\*C.

import org.apache.spark.sql.expressions.Window

val windowSpec = Window.partitionBy("department").orderBy(col("score").desc)

people

.withColumn("score", explode(col("score")))

.select(

col("department"),

col("name"),

dense\_rank().over(windowSpec).alias("rank"),

max(col("score")).over(windowSpec).alias("highest")

)

.where(col("rank") === 1)

.drop("rank")

.orderBy("department")

.show()

D.

people

.withColumn("score", explode(col("score")))

.groupBy("department")

.max("score")

.withColumnRenamed("max(score)", "highest")

.orderBy("department")

.show()

* 1. Which of the following standard Structured Streaming sink types are idempotent and can provide end-to-end exactly-once semantics in a Structured Streaming job?

A. Console  
\*B. Kafka  
C. File  
D. Memory

* 1. Which of following statements regarding caching are TRUE?

A. The default storage level for a DataFrame is StorageLevel.MEMORY\_AND\_DISK.  
\*B. The uncache() method evicts a DataFrame from cache.  
C. The persist() method immediately loads data from its source to materialize the DataFrame in cache.  
D. Explicit caching can decrease application performance by interfering with the Catalyst optimizer's ability to optimize some queries.