Final 2024-2025

Q1:

Hadoop V1:

MapReduce
Data Procesing
+
Resource Management
HDFS
Storage layer

Hadoop V2:

MapReduce + Spark + ...
Data processing layer
YARN
Resource Managment
HDFS
Data Storage layer

Q2:

1) F, on Spark
2) T
3) F, requires JVM
4) T, frameworks or Graph
5) F, Spark supports SQL, ML, Streaming, graph procesing
6) T
7) F, Spark use lazy evaluation
8) T
9) F, 4 processes: NameNode, DataNode, Resource Manager, Node Manager
10) F, it is Java-based

Q3:

1) To store data on HDFS,
with : data size = 745 MB, Block size = 65 MB
. Nb of blocks needed = $\frac{745}{65} = 11,46 \simeq 12$ block
where:
11 blocks are of size 65 MB and 1 block of size $(745 - 11 \times 65) = 30$ MB

1.1) The file blocks are stored physically on DataNodes, Because DataNodes are responsible for storing the actual data in HDFS, while NameNode only stores metadata to keep track of these blocks locations and access permissions.

2) For this data to be published in a Kafka broker, 5 topics need to be created.
The data comes from 5 different resources (given), and Kafka wer to create one topic for each data source in order to separate and organize data streams.

3) Yes, Because Spark supports graph analytics through GraphX Library and can directly read the streamed data stored on HDFS and analyze it as graph data.

4) Creating an RDD:
- 1st method: Create by loading a data file
This method is common, in which each line in the file becomes a record in the RDD.
→ Scala code: val rdd1 = sc.textFile("data.txt")

- 2nd method: Create a new RDD From existing RDD.
This method creates an new RDD by applying transformation Opr to an existing RDD, where the parent RDD remains intact and can be used in other operations.
→ Scala code: val rdd2 = rdd1.filter(_.startsWith("text"))
- 3rd method: Parallelizing collection.
This method takes a local data collection and distributes it in a parallel manner across the cluster to create RDD.
→ Scala code: val rdd3 = sc.parallelize(List("lorem", "text"))

5) Spark offers several transformations and actions, depending on the application:
- Transformations such as: filter(), map(), flatMap(), join(), groupByKey(), distinct(), sortBy(), reduceByKey() (Uses lazy evaluation, needs an action to be executed).
- Actions such as: collect(), count(), reduce(), save().
→ 2 transformations and 2 action examples:
- 1st example:
```
val result = rdd1.groupByKey()
result.collect().foreach(println)
```
- 2nd example:
```
val lineLengths = rdd1.map(s => s.length)
val totalLength = lineLengths.reduce((a,b) => a+b)
```

Q 4:

1) hdfs namenode -format
2) Starts HDFS services: NameNode, DataNode, SecondaryNameNode
3) Starts YARN services: ResourceManager, NodeManager
4) hdfs dfs -mkdir /word-count
5) hdfs dfs -put word-dataset / word-count
6) hdfs dfs -cat /word-count/word-dataset
   - For 1TB dataset:
   hdfs dfs -head /word-count/word-dataset
7) hadoop jar \
   $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples*.jar \
   wordcount \
   /word-count/word-dataset \
   /word-count/output.

Q 5:

1)

MapReduce 1:	YARN:
. Jobtracker	. Resource Manager, Application Master, timeline server
. tasktracker	. Node Manager
. Slot	. Container

2) YARN is more scalable than MapReduce V1, because it separates resource management from data processing. This improved resource utilization, supports multiple processing engines, and allows Hadoop to scale to larger clusters.