DUMPSBOSS.

Cloudera Certified Developer for Apache Hadoop (CCDH)

Cloudera CCD-410

Version Demo

Total Demo Questions: 8

Total Premium Questions: 80

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Data Ingestion	22
Topic 2, Data Transformation	40
Topic 3, Data Analysis	3
Topic 4, Workflow	14
Topic 5, Mix Questions	1
Total	80

QUESTION NO: 1

MapReduce v2 (MRv2/YARN) is designed to address which two issues?

A. Single point of failure in the NameNode.

B. Resource pressure on the JobTracker.

C. HDFS latency.

D. Ability to run frameworks other than MapReduce, such as MPI.

E. Reduce complexity of the MapReduce APIs.

F. Standardize on a single MapReduce API.

ANSWER: B D

Explanation:

MRv2, built on YARN, separates cluster-wide resource management from the per-application execution logic that had both been concentrated in the MapReduce v1 JobTracker. This redesign directly addresses *Resource pressure on the JobTracker.* In YARN, the ResourceManager performs global resource scheduling, while an ApplicationMaster manages the lifecycle and task coordination for each individual application. Distributing application control this way removes the JobTracker scalability bottleneck and enables the cluster to support many more concurrent applications.

YARN was also designed as a general-purpose resource-management platform rather than a service restricted to MapReduce workloads. Therefore, the *Ability to run frameworks other than MapReduce, such as MPI* is a core YARN objective. Frameworks can implement their own ApplicationMaster and request containers from YARN, allowing diverse distributed processing engines to share the same Hadoop cluster. Apache's YARN documentation describes this split between the ResourceManager and application-specific ApplicationMasters, as well as YARN's role in supporting multiple processing frameworks. See the Apache Hadoop YARN documentation and the Apache Hadoop MapReduce tutorial.

QUESTION NO: 2

Which statements about a MapReduce combiner are correct? Select two.

A. Hadoop may invoke a combiner zero, one, or multiple times.

B. A combiner must produce correct results when applied repeatedly to partial results.

C. A combiner always executes once for every intermediate key emitted by a mapper.

D. A combiner receives all values for a key from every mapper in the job.

E. A combiner eliminates the need for a reducer.

ANSWER: A B

Explanation:

A combiner is an optional local aggregation operation that Hadoop can execute on map output before it is transferred to reducers. Hadoop is permitted to invoke a combiner zero, one, or multiple times, so a job must remain correct if the combiner is never run. Combiner logic must also be safe for repeated partial aggregation; operations such as sum, minimum, and maximum are common examples.

A combiner does not replace the reducer, because it runs independently on each mapper's local output and cannot combine values produced by other mappers. See the Apache Hadoop MapReduce Tutorial.

QUESTION NO: 3

You are using Apache Sqoop to perform recurring imports from a relational database into HDFS. Which options can be used to support an incremental import? Select two.

A. `--incremental append`

B. `--incremental lastmodified`

C. `--incremental replace`

D. `--incremental merge`

E. `--incremental partition`

ANSWER: A B

Explanation:

`--incremental append` is correct because it imports rows whose check-column value is greater than the previously imported value. It is appropriate when new source rows are appended and the selected check column increases for later rows.

`--incremental lastmodified` is also correct because it imports rows that have been created or modified after the previously recorded check-column value. This mode is useful when existing source rows can be updated. Both modes use a check column and a last-value to determine the source records to import. See the Apache Sqoop User Guide: Incremental Imports.

QUESTION NO: 4

What is the disadvantage of using multiple reducers with the default HashPartitioner and distributing your workload across you cluster?

A. You will not be able to compress the intermediate data.

B. You will longer be able to take advantage of a Combiner.

C. By using multiple reducers with the default HashPartitioner, output files may not be in globally sorted order.

D. There are no concerns with this approach. It is always advisable to use multiple reduces.

ANSWER: C

Explanation:

By using multiple reducers with the default HashPartitioner, output files may not be in globally sorted order. Hadoop's default `HashPartitioner` assigns each key to a reducer according to that key's hash value. Each reducer receives its own partition of the key space, and the framework sorts the keys delivered to that reducer before invoking the reduce function. Consequently, each individual reducer output file is sorted by key. However, hash-based partition assignment does not ensure that all keys assigned to one reducer precede all keys assigned to another reducer. Concatenating the part files therefore does not produce one globally sorted result set. This matters when a downstream process requires total key ordering, or when output is expected to form ordered key ranges across files. For globally ordered reducer files, a range-based partitioning strategy, commonly implemented with Hadoop's `TotalOrderPartitioner` and a partition file, is required. Hadoop documents that `HashPartitioner` partitions based on the hash code of the key, while the MapReduce tutorial describes using `TotalOrderPartitioner` to achieve total ordering across reducer outputs. See the HashPartitioner API documentation and the Hadoop MapReduce tutorial.

QUESTION NO: 5

You have a directory named jobdata in HDFS that contains four files: _first.txt, second.txt, .third.txt and #data.txt. How many files will be processed by the FileInputFormat.setInputPaths () command when it's given a path object representing this directory?

A. Four, all files will be processed

B. Three, the pound sign is an invalid character for HDFS file names

C. Two, file names with a leading period or underscore are ignored

D. None, the directory cannot be named jobdata

E. One, no special characters can prefix the name of an input file

ANSWER: C

Explanation:

`FileInputFormat.setInputPaths()` registers the supplied directory as an input path; when Hadoop later enumerates that directory's input files, the standard hidden-file filter excludes names beginning with an underscore (_) or a period (.). Consequently, `_first.txt` and `.third.txt` are not considered input files. The remaining names, `second.txt` and `#data.txt`, are eligible for processing, so the job receives two input files. The `#` character does not make an HDFS file name invalid: Hadoop path names may use characters such as `#`, subject to general path and URI handling rules. This filtering behavior is intentional: Hadoop commonly creates underscore-prefixed control and metadata files, including `_SUCCESS`, which should not be treated as data input. Hadoop's API documentation describes the default hidden-file filtering used by `FileInputFormat`, and the implementation uses a filter that rejects path names starting with _ or .. See the FileInputFormat API documentation and the HiddenFileFilter API documentation.

QUESTION NO: 6

You are developing a combiner that takes as input Text keys, IntWritable values, and emits Text keys, IntWritable values. Which interface should your class implement?

A. Combiner

B. Mapper

C. Reducer

D. Reducer

E. Combiner

ANSWER: D

Explanation:

`Reducer<Text, IntWritable, Text, IntWritable>` is correct because, in the Hadoop MapReduce API, a combiner is implemented as a reducer-class implementation and is registered on the job with `Job.setCombinerClass(...)`. The combiner receives each intermediate key together with the iterable of values associated with that key, just as a reducer does. Its generic type parameters therefore identify the intermediate input key type, intermediate input value type, combiner output key type, and combiner output value type. For this case, those types are respectively `Text`, `IntWritable`, `Text`, and `IntWritable`.

A combiner is an optional local aggregation step that can reduce the amount of intermediate data transferred from mapper tasks to reducer tasks. To remain valid in this role, its output must be compatible with the reducer's expected input types. A typical implementation extends Hadoop's `Reducer` class and overrides its `reduce(Text key, Iterable<IntWritable> values, Context context)` method, writing `Text`/`IntWritable` pairs to the context. Hadoop's API documents that the combiner class is a `Reducer` class, and the Reducer API defines these four generic parameters. See Job.setCombinerClass and Reducer API.

QUESTION NO: 7

Which statements about YARN containers are correct? Select two.

A. A container represents an allocation of resources on a node.

B. An ApplicationMaster requests containers from the ResourceManager.

C. A container is an HDFS directory that holds one input split.

D. The NameNode launches application containers.

E. A single container is permanently assigned to one application for the life of the cluster.

ANSWER: A B

Explanation:

A container represents an allocation of resources on a node is correct. A container is YARN's unit of resource allocation and execution, with resource capabilities such as memory and CPU assigned on a particular NodeManager host.

An ApplicationMaster requests containers from the ResourceManager is also correct. The ApplicationMaster negotiates resources with the ResourceManager and then works with the relevant NodeManagers to launch work in allocated containers. See the Apache Hadoop YARN documentation.

QUESTION NO: 8

In a MapReduce job, you want each of your input files processed by a single map task. How do you configure a MapReduce job so that a single map task processes each input file regardless of how many blocks the input file occupies?

A. Increase the parameter that controls minimum split size in the job configuration.

B. Write a custom MapRunner that iterates over all key-value pairs in the entire file.

C. Set the number of mappers equal to the number of input files you want to process.

D. Write a custom FileInputFormat and override the method isSplitable to always return false.

ANSWER: D

Explanation:

`Write a custom FileInputFormat and override the method isSplitable to always return false.` is correct because map task creation is driven by the input splits produced by the job's `InputFormat`, rather than directly by the number of HDFS files or blocks. For file-based input, `FileInputFormat` normally divides splittable files into one or more logical splits, often aligned with HDFS block boundaries to enable parallel processing and data locality. A mapper is assigned one input split, so a multi-block file can otherwise result in multiple mapper tasks.

Overriding `isSplitable(Path)` to return `false` instructs the input format not to divide a file into multiple splits. The input-format split calculation then creates a single split covering the complete file, which results in one map task reading that file. This behavior is independent of the file's HDFS block count, although the mapper may read data from multiple block locations. This is the standard pattern for formats that cannot be safely divided at arbitrary byte offsets, such as certain compressed or record-oriented formats. Hadoop documents `FileInputFormat` and its split-generation behavior in the FileInputFormat API; the TextInputFormat API also shows that splitability is an input-format responsibility.