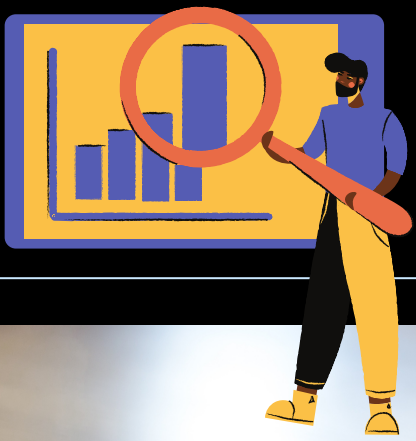
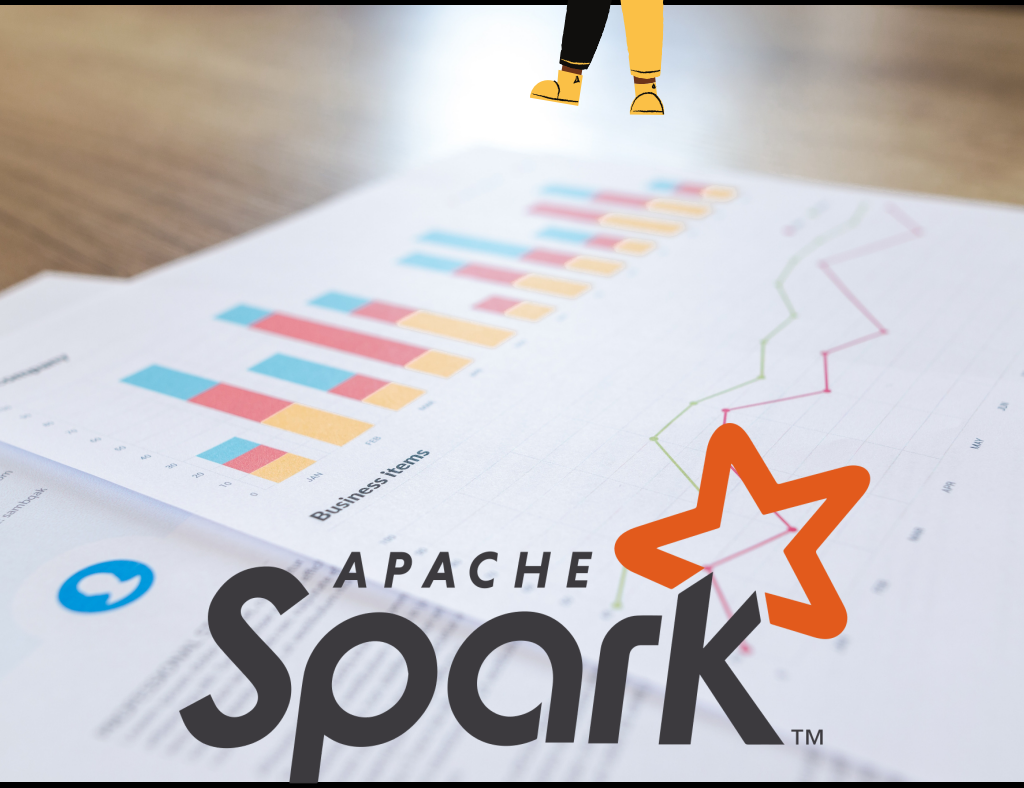
COMPREHENSIVE GUIDE TO INTERVIEWS FOR SPARK FOR BIG DATA





ZEP ANALYTICS

Introduction

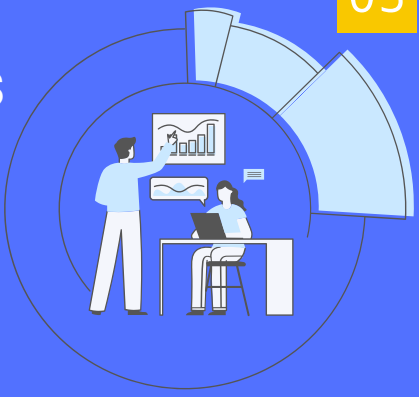
We've curated this series of interview which guides to accelerate your learning and your mastery of data science skills and tools.

From job-specific technical questions to tricky behavioral inquires and unexpected brainteasers and guesstimates, we will prepare you for any job candidacy in the fields of data science, data analytics, or BI analytics and Big Data.

These guides are the result of our data analytics expertise, direct experience interviewing at companies, and countless conversations with job candidates. Its goal is to teach by example - not only by giving you a list of interview questions and their answers, but also by sharing the techniques and thought processes behind each question and the expected answer.

Become a global tech talent and unleash your next, best self with all the knowledge and tools to succeed in a data analytics interview with this series of guides.

COMPREHENSIVE GUIDE TO INTERVIEWS FOR DATA SCIENCE



Data Science interview questions cover a wide scope of multidisciplinary topics. That means you can never be quite sure what challenges the interviewer(s) might send your way.

That being said, being familiar with the type of questions you can encounter is an important aspect of your preparation process.

Below you'll find examples of real-life questions and answers. Reviewing those should help you assess the areas you're confident in and where you should invest additional efforts to improve.

1.Explain spark architecture?

Apache Spark follows a master/slave architecture with two main daemons and a cluster manager -
i. Master Daemon - (Master/Driver Process) ii. Worker Daemon -(Slave Process)
A spark cluster has a single Master and any number of Slaves/Workers. The driver and the executors run their individual Java processes and users can run them on the same horizontal spark cluster or on separate machines i.e. in a vertical spark cluster or in mixed machine configuration.

2. Explain about Spark submission

The spark-submit script in Spark's bin directory is used to launch applications on a cluster. It can use all of Spark's supported cluster managers through a uniform interface so you don't have to configure your application especially for each one.
youtube link: https://youtu.be/t84cxWxiiDg
Example code:

```
./bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://207.184.161.138:7077 \
--deploy-mode cluster \
--supervise \
--executor-memory 20G \
--total-executor-cores 100 \
 /path/to/examples.jar
arguments1
```

3. Difference Between RDD, Dataframe, Dataset?

Resilient Distributed Dataset (RDD)

RDD was the primary user-facing API in Spark since its inception. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.

DataFrames (DF)

Like an RDD, a DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction; it provides a domain specific language API to manipulate your distributed data.

Datasets (DS)

Starting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a strongly-typed API and an untyped API, as shown in the table below. Conceptually, consider DataFrame as an alias for a collection of generic objects Dataset[Row], where a Row is a generic untyped JVM object.

4. When to use RDDs?

Consider these scenarios or common use cases for using RDDs when:

1. you want low-level transformation and actions and control on your dataset; your data is unstructured, such as media streams or streams of text;
2. you want to manipulate your data with functional programming constructs than domain specific expressions;
3. you don't care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; and
4. you can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.

5. What are the various modes in which Spark runs on YARN? (Client vs Cluster Mode)

YARN client mode: The driver runs on the machine from which client is connected

YARN Cluster Mode: The driver runs inside cluster

6. What is DAG - Directed Acyclic Graph?

Directed Acyclic Graph - DAG is a graph data structure which has edge which are directional and does not have any loops or cycles. It is a way of representing dependencies between objects. It is widely used in computing.

7. What is a RDD and How it works internally?

RDD (Resilient Distributed Dataset) is a representation of data located on a networkwhich is Immutable - You can operate on the rdd to produce another rdd but you can't alter it.

Partitioned / Parallel - The data located on RDD is operated in parallel. Any operation on RDD is done using multiple nodes.

Resilience - If one of the node hosting the partition fails, other nodes takes its data.
You can always think of RDD as a big array which is under the hood spread over many computers which is completely abstracted. So, RDD is made up many partitions each partition on different computers.

8. What do we mean by Partitions or slices?

Partitions also known as 'Slice' in HDFS, is a logical chunk of data set which may be in the range of Petabyte, Terabytes and distributed across the cluster.
By Default, Spark creates one Partition for each block of the file (For HDFS)
Default block size for HDFS block is 64 MB (Hadoop Version 1) / 128 MB (Hadoop Version 2) so as the split size.
However, one can explicitly specify the number of partitions to be created. Partitions are basically used to speed up the data processing.
If you are loading data from an existing memory using sc.parallelize(), you can enforce your number of

partitions by passing second argument.
You can change the number of partitions later using repartition().
If you want certain operations to consume the whole partitions at a time, you can use: mappartition().

9. What is the difference between map and flatMap?

Map and flatmap both function are applied on each element of RDD. The only difference is that the function that is applied as part of map must return only one value while flatmap can return a list of values.
So, flatmap can convert one element into multiple elements of RDD while map can only result in equal number of elements.
So, if we are loading rdd from a text file, each element is a sentence. To convert this RDD into an RDD of words, we will have to apply using flatmap a function that would split a string into an array of words. If we have just to cleanup each sentence or change case of each sentence, we would be using map instead of flatmap.

10. How can you minimize data transfers when working with Spark?

The various ways in which data transfers can be minimized when working with Apache Spark are:

1.Broadcast Variable- Broadcast variable enhances the efficiency of joins between small and large RDDs.
2.Accumulators - Accumulators help update the values of variables in parallel while executing.
3.The most common way is to avoid operations ByKey, repartition or any other operations which trigger shuffles.

11. Why is there a need for broadcast variables when working with Apache Spark?

These are read only variables, present in-memory cache on every machine. When working with Spark, usage of broadcast variables eliminates the necessity to ship copies of a variable for every task, so data can be processed faster. Broadcast variables help in storing a lookup table inside the memory which enhances the retrieval efficiency when compared to an RDD lookup ().

12. How can you trigger automatic clean-ups in Spark to handle accumulated metadata?

You can trigger the clean-ups by setting the parameter "spark.cleaner.ttl" or by dividing the long running jobs into different batches and writing the intermediary results to the disk.

13. Why is BlinkDB used?

BlinkDB is a query engine for executing interactive SQL queries on huge volumes of data and renders query results marked with meaningful error bars. BlinkDB helps users balance ‘query accuracy’ with response time.

14. What is Sliding Window operation?

Sliding Window controls transmission of data packets between various computer networks. Spark Streaming library provides windowed computations where the transformations on RDDs are applied over a sliding window of data. Whenever the window slides, the RDDs that fall within the particular window are combined and operated upon to produce new RDDs of the windowed DStream.

15. What is Catalyst Optimiser?

Catalyst Optimizer is a new optimization framework present in Spark SQL. It allows Spark to automatically transform SQL queries by adding new optimizations to build a faster processing system.

16. What do you understand by Pair RDD?

Paired RDD is a distributed collection of data with the key-value pair. It is a subset of Resilient Distributed Dataset. So it has all the feature of RDD and some new feature for the key-value pair. There are many transformation operations available for Paired RDD. These operations on Paired RDD are very useful to solve many use cases that require sorting, grouping, reducing some value/function. Commonly used operations on paired RDD are: groupByKey() reduceByKey() countByKey() join(), etc.

17. What is the difference between persist() and cache()?

persist () allowsthe user tospecify the storagelevel whereas cache () usesthe default storage level(MEMORY_ONLY).

18. What are the various levels of persistence in Apache Spark?

Apache Spark automatically persists the intermediary data from various shuffle operations, however it is often suggested that users call persist () method on the RDD in case they plan to reuse it. Spark has various persistence levels to store the RDDs on disk or in memory or as a combination of both with different replication levels.
The various storage/persistence levels in Spark are

- MEMORY_ONLY
- MEMORY_ONLY_SER
- MEMORY_AND_DISK
- MEMORY_AND_DISK_SER, DISK_ONLY
- OFF_HEAP

19. Does Apache Spark provide check pointing?

Lineage graphs are always useful to recover RDDs from a failure but this is generally time consuming if the RDDs have long lineage chains. Spark has an API for check pointing i.e. a REPLICATE flag to persist. However, the decision on which data to checkpoint - is decided by the user. Checkpoints are useful when the lineage graphs are long and have wide dependencies.

20. What do you understand by Lazy Evaluation?

Spark is intellectual in the manner in which it operates on data. When you tell Spark to operate on a given dataset, it heeds the instructions and makes a note of it, so that it does not forget - but it does nothing, unless asked for the final result. When a transformation like map () is called on a RDD-the operation is not performed immediately. Transformations in Spark are not evaluated till you perform an action. This helps optimize the overall data processing workflow.

21. What do you understand by SchemaRDD?

An RDD that consists of row objects (wrappers around basic string or integer arrays) with schema information about the type of data in each column. Dataframe is an example of SchemaRDD.

22. What are the disadvantages of using Apache Spark over Hadoop MapReduce?

Apache spark does not scale well for compute intensive jobs and consumes large number of system resources. Apache Spark's in-memory capability at times comes a major roadblock for cost efficient processing of big data. Also, Spark does have its own file management system and hence needs to be integrated with other cloud based data platforms or apache hadoop.

23. What is "Lineage Graph" in Spark?

Whenever a series of transformations are performed on an RDD, they are not evaluated immediately, but lazily(Lazy Evaluation). When a new RDD has been created from an existing RDD, that new RDD contains a pointer to the parent RDD. Similarly, all the dependencies between the RDDs will be logged in a graph, rather than the actual data. This graph is called the lineage graph.
Spark does not support data replication in the memory. In the event of any data loss, it is rebuilt using the "RDD Lineage". It is a process that reconstructs lost data partitions.

24. What do you understand by Executor Memory in a Spark application?

Every spark application has same fixed heap size and fixed number of cores for a spark executor. The heap size is what referred to as the Spark executor memory which is controlled with the spark. executor. Memory property of the -executor-memory flag. Every spark application will have one executor on each worker node. The executor memory is basically a measure on how much memory of the worker node will the application utilize.

25. What is an "Accumulator"?

"Accumulators" are Spark's offline debuggers. Similar to "Hadoop Counters", "Accumulators" provide the number of "events" in a program. Accumulators are the variables that can be added through associative operations. Spark natively supports accumulators ofnumeric value typesand standard mutablecollections. "AggregrateByKey()"and "combineByKey()" uses accumulators.

26. What is SparkContext ?

sparkContext was used as a channel to access all spark functionality.
The spark driver program uses spark context to connect to the cluster through a resource manager (YARN or Mesos.). sparkConf is required to create the spark context object, which stores configuration parameter like appName (to identify your spark driver), application, number of core and memory size of executor running on worker node.
In order to use APIs of SQL, HIVE, and Streaming, separate contexts need to be created.
Example:
creating sparkConf :
val conf = new SparkConf().setAppName("Project").setMaster("spark://master:7077") creation of sparkContext: val sc = new SparkContext(conf)

27. What is SparkSession ?

SPARK 2.0.0 onwards, SparkSession provides a single point of entry to interact with underlying Spark functionality and it allows Spark programming with DataFrame and Dataset APIs. All the functionality available with sparkContext are also available in sparkSession.

In order to use APIs of SQL, HIVE, and Streaming, no need to create separate contexts as sparkSession includes all the APIs.
Once the SparkSession is instantiated, we can configure Spark's run-time config properties.
Example:Creating Spark session:
val spark = SparkSession.builder.appName("WorldBankIndex").getOrCreate() Configuring properties:
spark.conf.set("spark.sql.shuffle.partitions", 6)
spark.conf.set("spark.executor.memory", "2g")

28. Why RDD is an immutable ?

Immutable data is always safe to share across multiple processes as well as multiple threads. Since RDD is immutable we can recreate the RDD any time. (From lineage graph). If the computation is time-consuming, in that we can cache the RDD which result in performance improvement.

29. What is Partitioner?

A partitioner is an object that defines how the elements in a key-value pair RDD are partitioned by key, maps each key to a partition ID from 0 to numPartitions - 1. It captures the data distribution at the output. With the help of partitioner, the scheduler can optimize the future operations. The contract of partitioner ensures that records for a given key have to reside on a single partition.
We should choose a partitioner to use for a cogroup-like operations. If any of the RDDs already has a partitioner, we should choose that one. Otherwise, we use a default HashPartitioner.

There are three types of partitioners in Spark :
a) Hash Partitioner :- Hash- partitioning attempts to spread the data evenly across various partitions based on the key.
b) Range Partitioner :- In Range- Partitioning method , tuples having keys with same range will appear on the same machine.
c) Custom Partitioner

RDDs can be created with specific partitioning in two ways :
i) Providing explicit partitioner by calling partitionBy method on an RDD
ii) Applying transformations that return RDDs with specific partitioners .

30. What are the benefits of DataFrames ?

1.DataFrame is distributed collection of data. In DataFrames, data is organized in named column.
2. They are conceptually similar to a table in a relational database. Also, have richer optimizations.
3. Data Frames empower SQL queries and the DataFrame API.
4. we can process both structured and unstructured data formats through it. Such as: Avro, CSV, elastic search, and Cassandra. Also, it deals with storage systems HDFS, HIVE tables, MySQL, etc.
5. In Data Frames, Catalyst supports optimization(catalyst Optimizer). There are general libraries available to represent trees. In four phases, DataFrame uses Catalyst tree transformation:
- Analyze logical plan to solve references
- Logical plan optimization
- Physical planning
- Code generation to compile part of a query to Java bytecode.
6. The Data Frame API's are available in various programming languages. For example Java, Scala, Python, and R.
7. It provides Hive compatibility. We can run unmodified Hive queries on existing Hive warehouse.
8. It can scale from kilobytes of data on the single laptop to petabytes of data on a large cluster.
9. DataFrame provides easy integration with Big data tools and framework via Spark core.

31. What is Dataset ?

A Dataset is an immutable collection of objects, those are mapped to a relational schema. They are strongly-typed in nature.
There is an encoder, at the core of the Dataset API. That Encoder is responsible for converting between JVM objects and tabular representation. By using Spark's internal binary format, the tabular representation is stored that allows to carry out operations on serialized data and improves memory utilization. It also supports automatically generating encoders for a wide variety of types, including primitive types (e.g. String, Integer, Long) and Scala case classes. It offers many functional transformations (e.g. map, flatMap, filter).

32. What are the benefits of Datasets?

1. Static typing- With Static typing feature of Dataset, a developer can catch errors at compile time (which saves time and costs).
2. Run-time Safety:- Dataset APIs are all expressed as lambda functions and JVM typed objects, any mismatch of typed-parameters will be detected at compile time. Also, analysis error can be detected at compile time too,when using Datasets, hence saving developer-time and costs.
3. Performance and Optimization- Dataset APIs are built on top of the Spark SQL engine, it uses Catalyst to generate an optimized logical and physical query plan providing the space and speed efficiency.

4. For processing demands like high-level expressions, filters, maps, aggregation, averages, sum,SQL queries, columnar access and also for use of lambda functions on semi-structured data, DataSets are best.
5. Datasets provides rich semantics, high-level abstractions, and domain-specific APIs

33. What is Shared variable in Apache Spark?

Shared variables are nothing but the variables that can be used in parallel operations.
Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only “added” to, such as counters and sums.

34. How to accumulated Metadata in Apache Spark?

Metadata accumulates on the driver as consequence of shuffle operations. It becomes particularly tedious during long-running jobs.
To deal with the issue of accumulating metadata, there are two options:
First, set the spark.cleaner.ttl parameter to trigger automatic cleanups. However, this will vanish any persisted RDDs.
The other solution is to simply split long-running jobs into batches and write intermediate results to disk. This facilitates a fresh environment for every batch and don’t have to worry about metadata build-up.

35. What is the Difference between DSM and RDD?

On the basis of several features, the difference between RDD and DSM is:

i. Read

RDD - The read operation in RDD is either coarse-grained or fine-grained. Coarse-grained meaning we can transform the whole dataset but not an individual element on the dataset. While fine-grained means we can transform individual element on the dataset.

DSM - The read operation in Distributed shared memory is fine-grained.

ii. Write

RDD - The write operation in RDD is coarse-grained.

DSM - The Write operation is fine grained in distributed shared system.

iii. Consistency

RDD - The consistency of RDD is trivial meaning it is immutable in nature. We can not realtor the content of RDD i.e. any changes on RDD is permanent. Hence, The level of consistency is very high.

DSM - The system guarantees that if the programmer follows the rules, the memory will be consistent. Also, the results of memory operations will be predictable.

iv. Fault-Recovery Mechanism

RDD - By using lineage graph at any moment, the lost data can be easily recovered in Spark RDD.

Therefore, for each transformation, new RDD is formed. As RDDs are immutable in nature, hence, it is easy to recover.
DSM - Fault tolerance is achieved by a checkpointing technique which allows applications to roll back to a recent checkpoint rather than restarting.

v. Straggler Mitigation
Stragglers, in general, are those that take more time to complete than their peers. This could happen due to many reasons such as load imbalance, I/O blocks, garbage collections, etc.
An issue with the stragglers is that when the parallel computation is followed by synchronizations such as reductions that causes all the parallel tasks to wait for others.
RDD - It is possible to mitigate stragglers by using backup task, in RDDs. DSM - To achieve straggler mitigation, is quite difficult.

vi. Behavior if not enough RAM
RDD - As there is not enough space to store RDD in RAM, therefore, the RDDs are shifted to disk. DSM - If the RAM runs out of storage, the performance decreases, in this type of systems.

36. What is Speculative Execution in Spark and how to enable it?

One more point is, Speculative execution will not stop the slow running task but it launch the new task in parallel.
Tabular Form :
Spark Property >> Default Value >> Description
spark.speculation >> false >> enables (true) or disables (false) speculative execution of tasks.
spark.speculation.interval >> 100ms >> The time interval to use before checking for speculative tasks.
spark.speculation.multiplier >> 1.5 >> How many times slower a task is than the median to be for speculation.
spark.speculation.quantile >> 0.75 >> The percentage of tasks that has not finished yet at which to start speculation.

37. How is fault tolerance achieved in Apache Spark?

The basic semantics of fault tolerance in Apache Spark is, all the Spark RDDs are immutable. It remembers the dependencies between every RDD involved in the operations, through the lineage graph created in the DAG, and in the event of any failure, Spark refers to the lineage graph to apply the same operations to perform the tasks.
There are two types of failures - Worker or driver failure. In case if the worker fails, the executors in that worker node will be killed, along with the data

in their memory. Using the lineage graph, those tasks will be accomplished in any other worker nodes. The data is also replicated to other worker nodes to achieve fault tolerance. There are two cases:

·1.Data received and replicated - Data is received from the source, and replicated across worker nodes. In the case of any failure, the data replication will help achieve fault tolerance.
·2.Data received but not yet replicated - Data is received from the source but buffered for replication. In the case of any failure, the data needs to be retrieved from the source.

For stream inputs based on receivers, the fault tolerance is based on the type of receiver:

1.Reliable receiver - Once the data is received and replicated, an acknowledgment is sent to the source. In case if the receiver fails, the source will not receive acknowledgment for the received data. When the receiver is restarted, the source will resend the data to achieve fault tolerance.
2.Unreliable receiver - The received data will not be acknowledged to the source. In this case of any failure, the source will not know if the data has been received or not, and it will nor resend the data, so there is data loss.
To overcome this data loss scenario, Write Ahead Logging (WAL) has been introduced in Apache

Spark 1.2. With WAL enabled, the intention of the operation is first noted down in a log file, such that if the driver fails and is restarted, the noted operations in that log file can be applied to the data. For sources that read streaming data, like Kafka or Flume, receivers will be receiving the data, and those will be stored in the executor's memory. With WAL enabled, these received data will also be stored in the log files.
WAL can be enabled by performing the below:

Setting the checkpoint directory, by using streamingContext.checkpoint(path)
Enabling the WAL logging, by setting spark.stream.receiver.WriteAheadLog.enable to True.

38. Explain the difference between reduceByKey, groupByKey, aggregateByKey and combineByKey?

1.groupByKey:
groupByKey can cause out of disk problems as data is sent over the network and collected on the reduce workers.
Example:-

```
sc.textFile("hdfs://").flatMap(line => line.split(" ")).map(word => (word,1)) .groupByKey().map((x,y) => (x,sum(y)) )
```

2.reduceByKey:
Data is combined at each partition , only one output for one key at each partition to send over network. reduceByKey required combining all your values into another value with the exact same type.
Example:-

```
sc.textFile("hdfs://").flatMap(line => line.split(" ")
).map(word => (word,1)) .reduceByKey((x,y)=>
(x+y))
```

3.aggregateByKey:
same as reduceByKey, which takes an initial value.

3 parameters as input 1). initial value 2). Combiner logic function 3).merge Function

Example:-

```
val inp
=Seq("dinesh=70","kumar=60","raja=40","ram=60","di
nesh=50","dinesh=80","kumar=40"
,"raja=40")
val rdd=sc.parallelize(inp,3)
val pairRdd=rdd.map(_.split("=")).map(x=>
(x(0),x(1)))
val initial_val=0
val addOp=(intVal:Int,StrVal: String)=>
intVal+StrVal.toInt val mergeOp=
(p1:Int,p2:Int)=>p1+p2
val out=pairRdd.aggregateByKey(initial_val)
(addOp,mergeOp) out.collect.foreach(println)
```

4.combineByKey:
combineByKey values are merged into one value at each partition then each partition value is merged into a single value. It's worth noting that the type of the combined value does not have to match the type of the original value and often times it won't be.
3 parameters as input

1. create combiner
2. mergeValue
3. mergeCombiners

Example:

```
val inp =Array(("Dinesh", 98.0), ("Kumar", 86.0), ("Kumar", 81.0), ("Dinesh", 92.0), ("Dinesh", 83.0), ("Kumar", 88.0))
val rdd = sc.parallelize(inp,2)

//Create the combiner
val combiner = (inp:Double) => (1,inp)

//Function to merge the values within a partition.Add 1 to the # of entries and inp to the existing inp val mergeValue = (PartVal: (Int,Double),inp:Double) =>{
(PartVal._1 + 1, PartVal._2 + inp)
}

/
```

```
/Function to merge across the partitions
val mergeCombiners = (PartOutput1:(Int, Double) ,
PartOutput2:(Int, Double))=>{
(PartOutput1._1+PartOutput2._1 ,
PartOutput1._2+PartOutput2._2)
}
//Function to calculate the average.Personinps is a
custom type val CalculateAvg = (personinp:(String,
(Int, Double)))=>{
val (name,(numofinps,inp)) = personinp
(name,inp/numofinps)
}

val rdd1=rdd.combineByKey(combiner, mergeValue,
mergeCombiners) rdd1.collect().foreach(println)
val rdd2=rdd.combineByKey(combiner, mergeValue,
mergeCombiners).map( CalculateAvg)
rdd2.collect().foreach(println)
```

39. Explain the mapPartitions() and mapPartitionsWithIndex() ?

mapPartitions() and mapPartitionsWithIndex() are both transformation.

mapPartitions() :

It runs one at a time on each partition or block of the Rdd, so function must be of type iterator<T>. It improves performance by reducing creation of object in map function.

mapPartitions() can be used as an alternative to map() and foreach() .

mapPartitions() can be called for each partitions while map() and foreach() is called for each elements
in an RDD.
Hence one can do the initialization on per-partition basis rather than each element basis
MappartionwithIndex():
It is similar to MapPartition but with one difference that it takes two parameters, the first parameter is the index and second is an iterator through all items within this partition (Int, Iterator<T>).
mapPartitionsWithIndex is similar to mapPartitions() but it provides second parameter index which keeps the track of partition.

40. Explain fold() operation in Spark?

fold() is an action. It is wide operation (i.e. shuffle data across multiple partitions and output a single value)It takes function as an input which has two parameters of the same type and outputs a single value of the input type.
It is similar to reduce but has one more argument 'ZERO VALUE' (say initial value) which will be used in the initial call on each partition.
def fold(zeroValue: T)(op: (T, T) ⇒ T): T
Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value". The function op(t1, t2) is allowed to modify t1 and return it as its result value to avoid object allocation; however, it should not modify t2.

This behaves somewhat differently from fold operations implemented for non-distributed collections in functional languages like Scala. This fold operation may be applied to partitions individually, and then fold those results into the final result, rather than apply the fold to each element sequentially in some defined ordering. For functions that are not commutative, the result may differ from that of a fold applied to a non-distributed
collection.
zeroValue: The initial value for the accumulated result of each partition for the op operator, and also the initial value for the combine results from different partitions for the op operator - this will typically be the neutral element (e.g. Nil for list concatenation or 0 for summation)
Op: an operator used to both accumulate results within a partition and combine results from different partitions
Example :

```
val rdd1 = sc.parallelize(List(1,2,3,4,5),3)
rdd1.fold(5)(_+_)
Output : Int = 35

val rdd1 = sc.parallelize(List(1,2,3,4,5)) rdd1.fold(5)(_+_)
Output : Int = 25

val rdd1 = sc.parallelize(List(1,2,3,4,5),3)
rdd1.fold(3)(_+_)
Output : Int = 27
```

41. Difference between textFile Vs wholeTextFile ?

Both are the method of org.apache.spark.SparkContext.

textFile() :

def textFile(path: String, minPartitions: Int = defaultMinPartitions): RDD[String]

Read a text file from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings

For example sc.textFile("/home/hdadmin/wc-data.txt") so it will create RDD in which each individual line an element.

Everyone knows the use of textFile.

wholeTextFiles() :

def wholeTextFiles(path: String, minPartitions: Int = defaultMinPartitions): RDD[(String, String)]

Read a directory of text files from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI.Rather than create basic RDD, the wholeTextFile() returns pairRDD.

For example, you have few files in a directory so by using wholeTextFile() method,

it creates pair RDD with filename with path as key,and value being the whole file as string.

Example:-

```
val myfilerdd = sc.wholeTextFiles("/home/hdadmin/MyFiles") val keyrdd = myfilerdd.keys
keyrdd.collect
val filerdd = myfilerdd.values
filerdd.collect
```

42. What is cogroup() operation.?

It's a transformation. It's in package org.apache.spark.rdd.PairRDDFunctions
def cogroup[W1, W2, W3](other1: RDD[(K, W1)], other2: RDD[(K, W2)], other3: RDD[(K, W3)]): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2], Iterable[W3]))]
For each key k in this or other1 or other2 or other3, return a resulting RDD that contains a tuple with the list of values for that key in this, other1, other2 and other3.
Example:

```
val myrdd1 = sc.parallelize(List((1,"spark"),(2,"HDFS"),(3,"Hive"),(4,"Flink"),(6,"HBase")))
val myrdd2= sc.parallelize(List((4,"RealTime"),(5,"Kafka"),(6,"NOSQL"),(1,"stream"),(1,"MLlib"))) val result = myrdd1.cogroup(myrdd2)
result.collect
```

Output :

```
Array[(Int, (Iterable[String], Iterable[String]))] = Array((4,(CompactBuffer(Flink),CompactBuffer(RealTime))),(1,(CompactBuffer(spark),CompactBuffer(stream, MLlib))),(6,(CompactBuffer(HBase),CompactBuffer(NOSQL))),(3,(CompactBuffer(Hive),CompactBuffer())),(5,(CompactBuffer(),CompactBuffer(Kafka))),(2,(CompactBuffer(HDFS),CompactBuffer())))
```

43. Explain pipe() operation ?

Return an RDD created by piping elements to a forked external process.
def pipe(command: String): RDD[String]
In general, Spark is using Scala, Java, and Python to write the program. However, if that is not enough, and one want to pipe (inject) the data which written in other languages like 'R', Spark provides general mechanism in the form of pipe() method Spark provides the pipe() method on RDDs.
With Spark's pipe() method, one can write a transformation of an RDD that can read each element in the RDD from standard input as String. It can write the results as String to the standard output.
Example:
test.py
#!/usr/bin/python
import sys
for line in sys.stdin:
print "hello " + line
spark-shell Scala:
val data = List("john","paul","george","ringo") val dataRDD = sc.makeRDD(data)
val scriptPath = "./test.py"
val pipeRDD = dataRDD.pipe(scriptPath)
pipeRDD.foreach(println)

44. Explain coalesce() operation.?

It's in a package org.apache.spark.rdd.ShuffledRDD
def coalesce(numPartitions: Int, shuffle: Boolean = false, partitionCoalescer: Option[PartitionCoalescer] = Option.empty) (implicit ord: Ordering[(K, C)] = null): RDD[(K, C)]
Return a new RDD that is reduced into numPartitions partitions.

Example:
val myrdd1 = sc.parallelize(1 to 1000, 15)
myrdd1.partitions.length
val myrdd2 = myrdd1.coalesce(5,false)
myrdd2.partitions.length
Int = 5

45. Explain the repartition() operation?
> repartition() is a transformation.
> This function changes the number of partitions mentioned in parameter
numPartitions(numPartitions : Int)
> It's in package org.apache.spark.rdd.ShuffledRDD
def repartition(numPartitions: Int)(implicit ord: Ordering[(K, C)] = null): RDD[(K, C)]
Return a new RDD that has exactly numPartitions partitions.
Can increase or decrease the level of parallelism in this RDD. Internally, this uses a shuffle to redistribute data.

If you are decreasing the number of partitions in this RDD, consider using coalesce, which can avoid performing a shuffle.

Example :

```
val rdd1 = sc.parallelize(1 to 100, 3)
rdd1.getNumPartitions
val rdd2 = rdd1.repartition(6)
rdd2.getNumPartitions
```

46. Explain the top() and takeOrdered() operation.?

Both top() and takeOrdered() are actions.

Both returns then elements of RDD based on default ordering or based on custom ordering provided by user.

```
def top(num: Int)(implicit ord: Ordering[T]): Array[T]
```

Returns the top k (largest) elements from this RDD as defined by the specified implicit Ordering[T] and maintains the ordering. This does the opposite of takeOrdered.

```
def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]
```

Returns the first k (smallest) elements from this RDD as defined by the specified implicit Ordering[T] and maintains the ordering. This does the opposite of top.

Example :
```
val myrdd1 = sc.parallelize(List(5,7,9,13,51,89))
myrdd1.top(3) //Array[Int] = Array(89, 51, 13)
myrdd1.takeOrdered(3) //Array[Int] = Array(5, 7, 9)
myrdd1.top(3) //Array[Int] = Array(89, 51, 13)
```

47. Explain the lookup() operation.?

> It is an action
> It returns the list of values in the RDD for key 'key'

Example:
```
val rdd1 = sc.parallelize(Seq(("myspark",78),
("Hive",95),("spark",15),("HBase",25),("spark",39),
("BigData",78),("spark",49)))
rdd1.lookup("spark")
rdd1.lookup("Hive")
rdd1.lookup("BigData")
```

Output:
```
Seq[Int] = WrappedArray(15, 39, 49)
Seq[Int] = WrappedArray(95)
Seq[Int] = WrappedArray(78)
```

48. How to Kill Spark Running Application.?

Get the application Id from the spark scheduler, for instance application_743159779306972_1234 and then, run the command in terminal like below yarn application -kill application_743159779306972_1234

49. How to stop INFO messages displaying on spark console?

Edit spark conf/log4j.properties file and change the following line:
log4j.rootCategory=INFO, console
to
log4j.rootCategory=ERROR, console

for Spark 2.X please import below commands,
import org.apache.log4j.{Logger,Level}
Logger.getLogger("org").setLevel(Level.ERROR)

for Python: spark.sparkContext.setLogLevel("ERROR")

50. Where the logs are available in Spark on YARN?

We can access logs through the command
Syntax:-
yarn logs -applicationId <application ID>

51. How to find out the different values in between two spark dataframes.?

Simply we can achieve by except operation
Example:-

```
scala> customerDF.show
+---+--------+---+------+
|cId|name|age|gender|
+---+--------+---+------+
|  1|  James| 21|     M|
|  2|    Liz| 25|     F|
|  3|   John| 31|     M|
|  4|Jennifer| 45|     F|
```

```
| 5| Robert| 41| M|
| 6| Sandra| 45| F|
+---+--------+---+------+

scala> custDF.show
+---+-----+---+------+
|cId| name|age|gender|
+---+-----+---+------+
| 1|James| 21| M|
| 2| Liz| 25|F|
+---+-----+---+------+

scala> customerDF.except(custDF).show +---+----
----+---+------+
|cId|name|age|gender|
+---+--------+---+------+
| 5| Robert| 41| M|
| 3| John| 31| M|
| 4|Jennifer| 45| F|
| 6| Sandra| 45| F|
+---+--------+---+------+
```

52. What are security options in Apache Spark?

Spark currently supports authentication via a shared secret. Authentication can be configured to be on via the spark.authenticate configuration parameter.This parameter controlswhether the Sparkcommunication protocols do authentication using the shared secret. This authentication is a basic handshake to make sure both sides have the same shared secret and are allowed to

communicate. If the shared secret is not identical they will not be allowed to communicate. The shared secret is created as follows:
For Spark on YARN deployments, configuring spark.authenticate to true will automatically handle generating and distributing the shared secret. Each application will use a unique shared secret.
For other types of Spark deployments, the Spark parameter spark.authenticate.secret should be configured on each of the nodes. This secret will be used by all the Master/Workers and applications.

53. What is Scala?

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages

54. What are the Types of Variable Expressions available in Scala?

val (aka Values):
You can name results of expressions with the val keyword.Once refer a value, it does not re-compute it.
Example:

```
val x = 1 + 1
x = 3 // This does not compile.
```

var (aka Variables):

Variables are like values, except you can re-assign them. You can define a variable with the var keyword.
Example:
var x = 1 + 1
x = 3 // This can compile.

55. What is the difference between method and functions in Scala..?

Methods:-
Methods look and behave very similar to functions, but there are a few key differences between them. Methods are defined with the def keyword. def is followed by a name, parameter lists, a return type, and a body.

Example:
def add(x: Int, y: Int): Int = x + y println(add(1, 2)) // 3

Functions:-
Functions are expressions that take parameters.
Bigdata Hadoop: Spark Interview Questions with Answers
You can define an anonymous function (i.e. no name) that returns a given integer plus one: (x: Int) => x + 1
You can also name functions. like
val addOne = (x: Int) => x + 1
println(addOne(1)) // 2

56. What is case classes in Scala?

Scala has a special type of class called a “case” class. By default, case classes are immutable and compared by value. You can define case classes with the case class keywords.

Example:

```
case class Point(x: Int, y: Int)
val point = Point(1, 2)
val anotherPoint = Point(1, 2)
val yetAnotherPoint = Point(2, 2)
```

57. What is Traits in Scala?

Traits are used to share interfaces and fields between classes. They are similar to Java 8’s interfaces. Classes and objects can extend traits but traits cannot be instantiated and therefore have no parameters.

Traits are types containing certain fields and methods. Multiple traits can be combined.

A minimal trait is simply the keyword trait and an identifier:

Example:

```
trait Greeter {def greet(name: String): Unit =
 println("Hello, " + name + "!") }
```

58. What is singleton object in scala?

An object is a class that has exactly one instance is called singleton object. Here’s an example of a singleton object with a method:

```
object Logger {
def info(message: String): Unit = println("Hi i am
Dineshkumar")
}
```

59. What is Companion objects in scala?

An object with the same name as a class is called a companion object. Conversely, the class is the object's companion class. A companion class or object can access the private members of its companion. Use a companion object for methods and values which are not specific to instances of the companion class.

Example:

```
import scala.math._
case class Circle(radius: Double) { import Circle._
def area: Double = calculateArea(radius)
}

object Circle {
private def calculateArea(radius: Double): Double
= Pi * pow(radius, 2.0)
}

val circle1 = new Circle(5.0)
circle1.area
```

60. What are the special datatype available in Scala?

Any:

Any is the supertype of all types, also called the top type. It defines certain universal methods such as equals, hashCode, and toString. Any has two direct subclasses: AnyVal and AnyRef.

Sample Example:

```
val list: List[Any] = List(
"a string",
732, // an integer
'c', // a character
true, // a boolean value
() => "an anonymous function returning a string"
)
list.foreach(element => println(element))
```

AnyVal:
AnyVal represents value types. There are nine predefined value types and they are non-nullable: Double, Float, Long, Int, Short, Byte, Char, Unit, and Boolean. Unit is a value type which carries no meaningful information. There is exactly one instance of Unit which can be declared literally like so: (). All functions must return something so sometimes Unit is a useful return type.

AnyRef:
AnyRef represents reference types. All non-value types are defined as reference types. Every user-defined type in Scala is a subtype of AnyRef. If Scala is used in the context of a Java runtime environment, AnyRef corresponds to java.lang.Object.

Nothing:
Nothing is a subtype of all types, also called the bottom type. There is no value that has type

Nothing. A common use is to signal non-termination such as a thrown exception, program exit, or an infinite loop (i.e., it is the type of an expression which does not evaluate to a value, or a method that does not return normally).

Null:
Null is a subtype of all reference types (i.e. any subtype of AnyRef). It has a single value identified by the keyword literal null. Null is provided mostly for interoperability with other JVM languages and should almost never be used in Scala code. We'll cover alternatives to null later in the tour.

62. What is Currying function or multiple parameter lists in Scala?

Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists,
then this will yield a function taking the missing parameter lists as its arguments. This is formally known as
currying.
Example:

```
def foldLeft[B](z: B)(op: (B, A) => B): B
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val res = numbers.foldLeft(0)((m, n) => m + n)
print(res) // 55
```

63. What is Pattern matching in Scala?

Pattern matching is a mechanism for checking a value against a pattern. A successful match can also deconstruct a value into its constituent parts. It is a more powerful version of the switch statement in Java and it can likewise be used in place of a series of if/else statements.

Syntax:

```
import scala.util.Random

val x: Int = Random.nextInt(10)
x match {
case 0 => "zero"
case 1 => "one"
case 2 => "two"
case _ => "many"
}

def matchTest(x: Int): String = x match {
case 1 => "one"
case 2 => "two"
case _ => "many"
}
matchTest(3) // many
matchTest(1) // one
```

64. What are the basic properties avail in Spark?

It may be useful to provide some simple definitions for the Spark nomenclature:
Worker Node: A server that is part of the cluster and are available to run Spark jobs Master Node: The server that coordinates the Worker nodes.
Executor: A sort of virtual machine inside a node. One Node can have multiple Executors.
Driver Node: The Node that initiates the Spark session. Typically, this will be the server where context is located.
Driver (Executor): The Driver Node will also show up in the Executor list.

65. What are the Configuration properties in Spark?

spark.executor.memory:- The maximum possible is managed by the YARN cluster whichcannot exceed the actual RAM available.
spark.executor.cores:- Number of cores assigned per Executor which cannot be higher than the cores available in each worker.
spark.executor.instances:- Number of executors to start. This property is acknowledged by the cluster if spark.dynamicAllocation.enabled is set to “false”.
spark.memory.fraction:- The default is set to 60% of the requested memory per executor.
spark.dynamicAllocation.enabled:- Overrides the mechanism that Spark provides to dynamically adjust resources. Disabling it provides more control over the number of the Executors that can be

started, which in turn impact the amount of storage available for the session. For more information, please see the Dynamic Resource Allocation page in the official Spark website.

66. What is Sealed classes?

Traits and classes can be marked sealed which means all subtypes must be declared in the same file.
This is useful for pattern matching because we don't need a "catch all" case. This assures that all subtypes are known.
Example:

```
sealed abstract class Furniture
case class Couch() extends Furniture
case class Chair() extends Furniture

def findPlaceToSit(piece: Furniture): String = piece
match { case a: Couch => "Lie on the couch"
case b: Chair => "Sit on the chair"
}
```

67. What is Type Inference?

The Scala compiler can often infer the type of an expression so you don't have to declare it explicitly.
Example:

```
val Name = "Dineshkumar S" // it consider as String
val id = 1234 // considered as int
```

68. When not to rely on default type inference?

The type inferred for obj was Null. Since the only value of that type is null, So it is impossible to assign a different value by default.

69. How can we debug spark application locally?

Actually we can doing that in local debugging, setting break points, inspecting variables, etc. set spark submission in deploy mode like below -
spark-submit --name CodeTestDinesh --class DineshMainClass --master local[2] DineshApps.jar
then spark driver to pause and wait for a connection from a debugger when it starts up, by adding an option like -
the following:
--conf spark.driver.extraJavaOptions=-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005
where agentlib:jdwp is the Java Debug Wire Protocol option, followed by a comma-separated list of sub-
options:
1. transport: defines the connection protocol used between debugger and debuggee -- either socket or "shared
2. memory" -- you almost always want socket (dt_socket) except I believe in some cases on Microsoft Windows

3. server: whether this process should be the server when talking to the debugger (or conversely, the client) -- you always need one server and one client. In this case, we're going to be the server and wait for a connection from the debugger
4. suspend: whether to pause execution until a debugger has successfully connected. We turn this on so the driver won't start until the debugger connects
5. address: here, this is the port to listen on (for incoming debugger connection requests). You can set it to any available port (you just have to make sure the debugger is configured to connect to this same port)

70. Map collection has Key and value then Key should be mutable or immutable?

Behavior of a Map is not specified if value of an object is changed in a manner that affects equals comparison while object with the key. So Key should be an immutable.

71. What is OFF_HEAP persistence in spark?

One of the most important capabilities in Spark is persisting (or caching) datasets in memory across operations. Each persisted RDD can be stored using a different storage level. One of the possibilities is to store RDDs in serialized format off-heap. Compared to storing data in the Spark JVM, off-heap storage reduces garbage collection overhead and allows executors to be smaller and to share a

pool of memory. This makes it attractive in environments with large heaps or multiple concurrent applications.

74. What is the difference between Apache Spark and Apache Flink?

1.Stream Processing:
While Spark is a batch oriented system that operates on chunks of data, called RDDs, Apache Flink is a stream processing system able to process row after row in real time.

2.Iterations:
By exploiting its streaming architecture, Flink allows you to natively iterate over data, something Spark also supports only as batches

3.Memory Management:
Spark jobs have to be optimized and adapted to specific datasets because you need to manually control partitioning and caching if you want to get it right

4.Maturity:
Flink is still in its infancy and has but a few production deployments

5.Data Flow:
In contrast to the procedural programming paradigm Flink follows a distributed data flow approach. For data set operations where

intermediate results are required in addition to the regular input of an operation, broadcast variables are used to distribute the pre calculated results to all worker nodes.

73. How do we Measuring the impact of Garbage Collection?

GC has happened due to use too much of memory on a driver or some executors or it might be where garbage collection becomes extremely costly and slow as large numbers of objects are created in the JVM. You can do by this validation '-verbose:gc -XX:+PrintGCDetails-XX:+PrintGCTimeStamps' to Spark's JVM options using the `spark.executor.extraJavaOptions`configuration parameter.

74. Apache Spark vs. Apache Storm?

Apache Spark is an in-memory distributed data analysis platform-- primarily targeted at speeding up batch analysis jobs, iterative machine learning jobs, interactive query and graph processing.
One of Spark's primary distinctions is its use of RDDs or Resilient Distributed Datasets. RDDs are great for pipelining parallel operators for computation and are, by definition, immutable, which allows Spark a unique form of fault tolerance based on lineage information. If you are interested in, for example, executing a Hadoop
MapReduce job much faster, Spark is a great option (although memory requirements must be

considered).
Apache Storm is focused on stream processing or what some call complex event processing. Storm implements a fault tolerant method for performing a computation or pipelining multiple computations on an event as it flows into a system. One might use Storm to transform unstructured data as it flows into a system into a desired format.
Storm and Spark are focused on fairly different use cases. The more "apples-to-apples" comparison would be between Storm Trident and Spark Streaming. Since Spark's RDDs are inherently immutable, Spark Streaming implements a method for "batching" incoming updates in user-defined time intervals that get transformed into their own RDDs. Spark's parallel operators can then perform computations on these RDDs. This is different from Storm which deals with each event individually.
One key difference between these two technologies is that Spark performs Data-Parallel computations while Storm performs Task-Parallel computations. Either design makes trade offs that are worth knowing.

75. How to overwrite the output directory in spark?

refer below command using Dataframes,
df.write.mode(SaveMode.Overwrite).parquet(path)

76. How to read multiple text files into a single RDD?

You can specify whole directories, use wildcards and even CSV of directories and wildcards like below.
Eg.:
val rdd = sc.textFile("file:///D:/Dinesh.txt, file:///D:/Dineshnew.txt")

77. Can we run SPARK without base of HDFS?

Apache Spark is a fast and general-purpose cluster computing system. It is not a data storage system. It uses external storage system for storing and reading data. So we can run Spark without HDFS in distributed mode using any HDFS compatible file systems like S3, GPFS, GlusterFs, Cassandra, and etc. There is another file system called Tachyon. It is a in memory file system to run spark in distributed mode.

78. Define about generic classes in scala?

Generic classes are classes which take a type as a parameter. They are particularly useful for collection classes.
Generic classes take a type as a parameter within square brackets [ ]. One convention is to use the letter A as type parameter identifier, though any parameter name may be used.

Example: The instance stack can only take Int values. val stack = new Stack[Int]
stack.push(1)
stack.push(2)
println(stack.pop) // prints 2
println(stack.pop) // prints 1

79. How to enable tungsten sort shuffle in Spark 2.x?

SortShuffleManager is the one and only ShuffleManager in Spark with the short name sort or tungsten-sort.
In other words, there's no way you could use any other ShuffleManager but SortShuffleManager (unless you enabled one using spark.shuffle.manager property).

80. How to prevent Spark Executors from getting Lost when using YARN client mode?

The solution if you're using yarn was to set --conf spark.yarn.executor.memoryOverhead=600, alternatively if your cluster uses mesos you can try --conf spark.mesos.executor.memoryOverhead=600 instead.

81. What is the relationship between the YARN Containers and the Spark Executors.?

First important thing is this fact that the number of containers will always be the same as the executors created by a Spark application e.g. via --num-executors parameter in spark-submit.

Set by the yarn.scheduler.minimum-allocation-mb every container always allocates at least this amount of memory.This means ifparameter --executor-memoryis set toe.g. only 1g but yarn.scheduler.minimum-allocation-mb is e.g. 6g, the container is much bigger than needed by the Spark application.
The other way round, if the parameter --executor-memory is set to somthing higher than the yarn.scheduler.minimum-allocation-mb value, e.g. 12g, the Container will allocate more memory dynamically,but only ifthe requested amountof memory issmaller or equalto yarn.scheduler.maximum-allocation-mb value.
The value ofyarn.nodemanager.resource.memory-mbdetermines, how muchmemory can be allocated in sum by all containers of one host!

So setting yarn.scheduler.minimum-allocation-mb allows you to run smaller containers e.g. for smaller executors (else it would be waste of memory).
Settingyarn.scheduler.maximum-allocation-mbto the maximumvalue (e.g. equalto yarn.nodemanager.resource.memory-mb) allows you to define bigger executors (more memory is allocated if needed, e.g. by --executor-memory parameter).

82. How to allocate the memory sizes for the spark jobs in cluster?

Before we are answering this question we have to concentrate the 3 main features.
which are –
1. NoOfExecutors,
2. Executor-memory and
3. Number of executor-cores

Lets go with example now, let's imagine we have a cluster with six nodes running NodeManagers, each with 16 coresand 64GB RAM. TheNodeManager sizes, yarn.nodemanager.resource.memory-mb and yarn.nodemanager.resource.cpu-vcores, should be set to 63 * 1024 = 64512 (megabytes) and 15 respectively. We never provide 100% allocation of each resources to YARN containers because the node needs some resources to run the OS processes and Hadoop. In this case, we leave a gigabyte and a core for these system processes. Cloudera Manager helps by accounting for these and configuring these YARN properties automatically. So the allocation likely matched as --num-executors 6 --executor-cores 15 --executor-memory 63G.

However, this is the wrong approach because: 63GB more on the executor memory overhead won't fit within the 63GB RAM of the NodeManagers. The application master will cover up a core on one of the nodes, meaning that there won't be room for a

15-core executor on that node. 15 cores per executor can lead to bad HDFS I/O throughput. So the best option would be to use --num-executors 17 --executor-cores 5 --executor-memory 19G.
This configuration results in three executors on all nodes except for the one with the Application Master, which
will have two executors. --executor-memory was derived as (63/3 executors per node) = 21. 21 * 0.07 = 1.47. 21 -
1.47 ~ 19.

83. How autocompletion tab can enable in pyspark.?

Please import the below libraries in pyspark shell
import rlcompleter, readline
readline.parse_and_bind("tab: complete")

84. Can we execute two transformations on the same RDD in parallel in Apache Spark?

All standard RDD methods are blocking (with exception to AsyncRDDActions) so actions will be evaluated sequentially.
It is possible to execute multiple actions concurrently using non-blocking submission (threads, Futures) with correct configuration of in-application scheduler or explicitly limited resources for each action.

Example:

```
val df = spark.range(100000)
val df1= df.filter('id < 1000)
val df2= df.filter('id >= 1000)
print(df1.count() + df2.count()) //100000
```

Regarding cache it is impossible to answer without knowing the context. Depending on the cluster configuration, storage, and data locality it might be cheaper to load data from disk again, especially when resources are limited, and subsequent actions might trigger cache cleaner.

85. Which cluster type should I choose for Spark?

·Standalone - meaning Spark will manage its own cluster
·YARN - using Hadoop's YARN resource manager
·Mesos - Apache's dedicated resource manager project
Start with a standalone cluster if this is a new deployment. Standalone mode is the easiest to set up and will provide almost all the same features as the other cluster managers if you are only running Spark.
If you would like to run Spark alongside other applications, or to use richer resource scheduling capabilities (e.g. queues), both YARN and Mesos provide these features. Of these, YARN will likely be preinstalled in many

Hadoop distributions.
One advantage of Mesos over both YARN and standalone mode is its fine-grained sharing option, which lets interactive applications such as the Spark shell scale down their CPU allocation between commands. This makes it attractive in environments where multiple users are running interactive shells.
In all cases, it is best to run Spark on the same nodes as HDFS for fast access to storage. You can install Mesos or the standalone cluster manager on the same nodes manually, or most Hadoop distributions already install YARN and HDFS together.

86. What is DStreams in Spark Streaming?

Spark streaming uses a micro batch architecture where the incoming data is grouped into micro batches called Discretized Streams (DStreams) which also serves as the basic programming abstraction.
The DStreams internally have Resilient Distributed Datasets (RDD) and as a result of this standard RDD transformations and actions can be done.

87. What is Stateless Transformation ?

The processing of each batch has no dependency on the data of previous batches called Stateless Transformation. Stateless transformations are simple RDD transformations. It applies on every batch meaning every RDD in a DStream. It includes common RDD transformations like map(), filter(), reduceByKey() etc.

88. What is Stateful Transformation ?

The uses data or intermediate results from previous batches and computes the result of the current batch called Stateful Transformation. Stateful transformations are operations on DStreams that track data across time. Thus it makes use of some data from previous batches to generate the results for a new batch.
In streaming if we have a use case to track data across batches then we need state-ful DStreams. For example we may track a user's interaction in a website during the user session or we may track a particular twitter hashtag across time and see which users across the globe is talking about it.
Types of state-ful transformation.

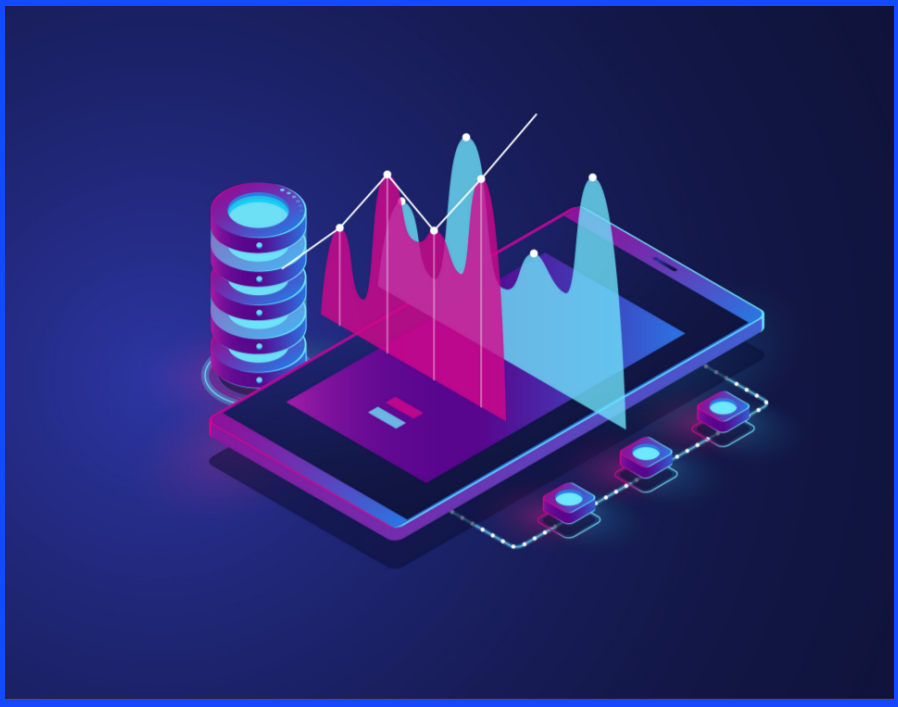
1.State-ful DStreams are of two types - window based tracking and full session tracking.
For stateful tracking all incoming data should be transformed to key-value pairs such that the key states can be tracked across batches. This is a precondition.

2.Window based tracking
In window based tracking the incoming batches are grouped in time intervals, i.e. group batches every 'x' seconds. Further computations on these batches are done using slide intervals.

This brings our list of 80+ SPARK for Big Data interview questions to an end.

We believe this concise guide will help you “expect the unexpected” and enter your first data analytics interview with confidence

We, at Zep provide a platform for Education, where your demand gets fulfilled. You demand we fulfil all your learning needs without costing you extra.



ZEP ANALYTICS

Ready to take the next steps?

Zep offers a platform for education to learn, grow & earn.

Become a Tech Blogger at Zep!!

Why don't you start your journey as a blogger and enjoy unlimited free perks and cash prizes every month.

Explore