What are the features and characteristics of Apache Spark which make it superior than other Big Data solutions like Hadoop-MapReduce?

Apache Spark is the Next-Gen Big Data tool (considered as future of Big Data and successor of MapReduce), below are the features of Spark:
Speed: Speed always matters for processing data, organizations want to process voluminous data as fast as possible. Spark is Lightening fast processing tool makes it speedier to handle complex processing. As it follows the concept of RDD (resilient distributed dataset) which allows it to store data transparently in memory, which helps in reducing read & write to disc one of the main time consuming factor.
Usability: Ability to support multiple languages makes it dynamic. It allows you quickly write application in Java, Scala ,Python and R.
In Memory Computing: Keeping data in servers' RAM as it makes accessing stored data quickly. In memory analytics accerlates iterative machine learning algoritms as it saves data read and write round trip from/to disk.
Pillar to Sophisticated Analytics: Spark comes with tools for interactive / declarative queries, streaming data, machine learning which are addition to simple map and reduce ,so that users can combine all this into single workflow.
Real Time Stream Processing: Spark streaming can handle real time stream processing along with integration of other frameworks which concludes that sparks streaming ability is easy, fault tolerance and Integrated.
Compatibility with hadoop & existing Hadoop Data: Spark is compatible with both versions of Hadoop ecosystem. Be it YARN (Yet Another Resource Negotiator) or SIMR (Spark in MApReduce). It can read anything existing Hadoop data that's what makes it suitable for migration of pure Hadoop-MapReduce applications. It can run independently too.
Lazy Evaluation: Another outstanding feature of Spark which is call by need or memorization. It waits for instructions before providing final result which saves time significant time.
Active, progressive and expanding community: Built by wide set of developers from over 100 companies. It has active mailing state and JIRA for issue tracking. It is most active component in Apache repository.

How can we split single HDFS block into partitions RDD?

When we create the RDD from a file stored in HDFS
data = context.textFile("/user/dataflair/file-name")
by default one partition is created for one block. ie. if we have a file of size 1280 MB (with 128 MB block size) there will be 10 HDFS blocks, hence similar number of partitions (10) will be created.
If you want to create more partitions than number of blocks, you can specify the same while RDD creation:
data = context.textFile("/user/dataflair/file-name", 20)
It will create 20 partitions for the file. ie for each block 2 partitions will be created.
NOTE: it is often recommended to have more no of partitions than no of block, it improves the performance.

Explain RDD, how it provides abstraction in Spark and make spark operator rich ?

RDD is representation of set of records, immutable collection of objects with distributed computing. RDD is large collection of data or an array of reference of partitioned objects. Each and every datasets in RDD is logically partitioned across many servers so that they can be computed on different nodes of cluster. RDD are fault tolerant i.e. self-recovered / recomputed in case of failure. Dataset could be data loaded externally by the users which can be in the form of json file, csv file, text file or database via JDBC with no specific data structure.
RDD is Lazily Evaluated i.e. it is memorized or called when required or needed, which saves lots of time. RDD is a read only, partitioned collection of data. RDDs can be created through deterministic operations or on stable storage or from other RDDs. It can also be generated by parallelizing an existing collection in your application or referring a dataset in an external storage system. It is cacheable. As it operates on data over multiple jobs in computations such as logistic regression, k-means clustering, PageRank algorithms, which makes it reuse or share data among multiple jobs.

Explain Apache spark eco-system components: Spark SQL, Spark Streaming, Spark MLlib and GraphX.
In which scenarios we can use these components ? what type of problems can be solved using them ?

Below are the Spark ecosystem components:
Spark Core: Spark Core is the base for parallel and distributed processing of huge datasets. It is incharge of all the essential I/O functionalities, programing and observance the roles on spark clusters, task dispatching, and networking with different storage systems, fault recovery and economical memory management. It uses special collection referred to as RDD (Resilient Distributed Datasets).
Spark SQL element : SparkSQL is module / component in Apache Spark that is employed to access structured and semi structured information. It is a distributed framework that is tightly integrated with varied spark programming like Scala, Python and Java. It supports relative process in spark programs via RDD further as on external datasource. It is used for manipulating and taking information in varied formats. The means through that {we can|we will|we square measure able to} act with Spark SQL are SQL/HQL, DataFrame API and Datasets API. It provides higher improvement.
The main abstraction in SparkSQL is information sets that acts on structured data. It translate ancient SQL and HiveQL queries into Spark jobs creating Spark accessible wide. It supports real time data analaytics, data streaming SQL.
SparkSQL defines 3 varieties of function:
Built-in perform or user-defined function: Object comes with some functions for column manipulation. Using scala we are able to outlined user outlined perform.
Aggregate Function: Operates on cluster of rows and calculates one come back price per cluster.
Window Aggregate: Operates on cluster of rows and calculates one come back price every row in an exceedingly cluster

Different type of APIs for accessing SparkSQL:
Spark SQL: Executing SQL queries or Hive queries, result are going to be came in variety of DataFrame.
DataFrame: It is similar to relative table in SparkSQL. It is distributed assortment of tabular information having rows and named column. It will perform filter, intersect, join, sort mixture and many more. It powerfully trust options of RDD. As it trust RDD, it is lazy evaluated and immutable in nature. DataFrameAPI is offered in Scala, Java and Python.
Datasets API: Dataset is new API additional to SparkApache to supply benefit of RDD because it is robust written and declarative in nature. Dataset is assortment of object or records with familiar schema. It should be modeled in some data-structre. It offers improvement of DataFrames and static kind safety of Scala. We can convert information-set to Data Frame.
Spark Streaming: Spark Streaming is a light-weight API that permits developers to perform execution and streaming of information application. Discretized Streams kind the bottom abstraction in Spark Streaming. It makes use of endless stream of {input information|input file|computer file} to method data in time period. It leverages the quick programing capability of Apache Spark Core to perform streaming analytics by ingesting information in mini-batches. Information in Spark Streaming is accepted from varied information sources and live streams like Twitter, Apache Kafka, IoT Sensors, Amazon response, Apache Flume, etc. in an event drive, fault-tolerant and type-safe applications.
Spark element Mllib: MLlib in Spark stands for machine learning (ML) library. Its goal is to form sensible machine learning effective, ascendible and straightforward. It consists of some learning algorithms and utilities, as well as classification, regression, clustering, cooperative filtering, spatial property reduction, further as lower-level improvement primitives and higher-level pipeline genus Apis.
GraphX: GraphX is a distributed graph process framework on prime of Apache Spark. because it is predicated on RDDs, that square measure immutable, graphs square measure immutable and so GraphX is unsuitable for graphs that require to be updated, in addition to in an exceedingly transactional manner sort of a graph info.

Can you explain in detail what is Apache Spark? What are the features of Spark, what type of big data problems spark can solve.

Spark is an open source big data framework. It has an expressive development APIs to allow big data professionals to efficiently execute streaming as well as batch and It provides faster and more general data processing platform engine. It is basically designed for fast computation. It was developed at UC Berkeley in 2009. Spark is an Apache project which is also known as "lighting fast cluster computing". It distributes data in file system across the cluster, and process that data in parallel. It covers wide range of workloads like batch applications, iterative algorithms, interactive queries and streaming. It lets you write application in Java, Python or Scala.

It was developed to overcome the limitations of MapReduce cluster computing paradigm. Spark keeps things in memory whereas map reduce keep shuffling things in and out i.e. on disk. It allows to cache data in memory which is beneficial in iterative algorithm those used in machine learning.

Spark is easier to develop as it knows how to operate on data. It supports SQL queries, streaming data and graph data processing. Spark doesn't need Hadoop to run, it can run on its own using other storages like Cassandra, S3 from which spark can read and write. In terms of speed spark run programs up to 100X faster in memory or 10Xfaster on disk than Map Reduce.

Why RDD is used to process the data ? What are the major features/characteristics of RDD (Resilient Distributed Datasets) ?

Properties/Traits of RDD:

Immutable (Read only cant change or modify): Data is safe to share across processes. It can be created or retrieved anytime which makes caching, sharing & replication easy. It is a way to reach consistency in computations.

Partitioned: It is basic unit of parallelism in RDD. Each partition is logical division of data/records.

Coarse gained operations: it's applied to any or all components in datasets through maps or filter or group by operation.

Action/Transformations: All computations in RDDs are actions or transformations.

Fault Tolerant: As the name says or include Resilient which means its capability to reconcile, recover or get back all the data (coarse/fine grained & low overhead) using lineage graph.

Cacheable: It holds data in persistent storage (memory/disk) so that they can be retrieved more quickly on the next request for them.

Persistence: Option of choosing which storage will be used either in-memory or on-disk.

Apache Spark can be run in following three mode :

(1) Local mode (2) Standalone mode (3) Cluster mode

What is the command to start and stop the spark in interactive shell ?

Command to start the interactive shell in Scala :

>>>>bin/spark-shell

First go the spark directory i.e.

hdadmin@ubuntu:~$ cd spark-1.6.1-bin-hadoop2.6/

hdadmin@ubuntu:~/spark-1.6.1-bin-hadoop2.6$ bin/spark-shell

Command to stop the interactive shell in Scala :

scala> Press (Ctrl+D)

One can see following message,

scala> Stopping spark context.

What's Transformation operation, how it processes in apache Spark ?

Transformations are lazy evaluated operations on RDD that create one or many new RDDs, e.g. map, filter, reduceByKey, join, cogroup, randomSplit. Transformations are functions which takes an RDD as the input and produces one or many RDDs as output. They dont change the input RDD as RDDs are immutable and hence cannot be changed or modified, but always produces new RDD by applying the computations operations on them. By applying transformations you incrementally builds an RDD lineage with all the ancestor RDDs of the final RDD(s).
Transformations are lazy, i.e. are not executed immediately. Transformations can be executed only when actions are called. After executing a transformation, the result RDD(s) will always be different from their ancestors RDD and can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap, union, cartesian) or the same size (e.g. map) or it can vary in size.
RDD allows you to create dependencies b/w RDDs. Dependencies are the steps for producing results ie; a program. Each RDD in lineage chain, string of dependencies has a function for operating its data and has a pointer dependency to its ancestor RDD. Spark will divide RDD dependencies into stages and tasks and then send those to workers for execution.

Explain the flatMap() transformation

When one want to produce multiple element (values) for each input element, flatMap() is used.
> As with map(), flatMap() also takes function as an input.
> Output of the function is an List of element through which we can iterate. (i.e. function can return 0 or more element for each input element)
> Simple use of flatMap() is splittin up an input line (string) into words.
Ex. val fm1 = sc.parallelize(List("Good Morning", "Data Flair", "Spark Batch"))
val fm2 = fm1.flatMap(y => y.split(" "))
fm2.foreach{println}
Output is as follows:
Good
Morning
Data
Flair
Spark
Batch

Explain distnct(), union(), intersection() and substract() transformation

---------- distnct() transformation ----------
If one want only unique elements in a RDD in that case one can use d1.distnct() where d1 is RDD
Ex.
val d1 = sc.parallelize(List("c","c","p","m","t"))
val result = d1.distnct()
result.foreach{println}
OutPut:
p
t
m
c

---------- union() transformation ----------

> Its simplest set operation.

> rdd1.union(rdd2) which outputs a RDD which contains the data from both sources.

> If the duplicates are present in the input RDD, output of union() transformation will contain duplicate also which can be fixed using distinct().

Ex.

```
val u1 = sc.parallelize(List("c","c","p","m","t"))
val u2 = sc.parallelize(List("c","m","k"))
val result = u1.union(u2)
result.foreach{println}
```

Output:

```
c
c
p
m
t
c
m
k
```

---------- intersection() transformation ----------

> intersection(anotherrdd) returns the elements which are present in both the RDDs

> intersection(anotherrdd) remove all the duplicate including duplicated in single RDD

```
val is1 = sc.parallelize(List("c","c","p","m","t"))
val is2 = sc.parallelize(List("c","m","k"))
val result = is1.union(is2)
result.foreach{println}
```

Output :

```
m
c
```

---------- subtract() transformation ----------

> Subtract(anotherrdd)

> It returns an RDD that has only value present in the first RDD and not in second RDD.

Ex.

```
val s1 = sc.parallelize(List("c","c","p","m","t"))
val s2 = sc.parallelize(List("c","m","k"))
val result = s1.subtract(s2)
result.foreach{println}
```

Output :

```
t
p
```

Explain the operation reduce()

It takes function with two arguments an accumulator and a value which should be commutative and Associative in mathematical nature. It reduces a list of element s into one as a result. This function produces same result when continuously applied on same set of RDD data with multiple partitions irrespective of elements order. It is wide operation.
> reduce() 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 parameter of the same type and output a single value of the input type.
> i.e. combine the elements of RDD together.

Example 1 :
val rdd1 = sc.parallelize(1 to 100)
val rdd2 = rdd1.reduce((x,y) => x+y)
OR
val rdd2 = rdd1.reduce(_ + _)
Output :
rdd2: Int = 5050
Example 2:
val rdd1 = sc.parallelize(1 to 5)
val rdd2 = rdd1.reduce(_*_)
Output :
rdd2: Int = 120

Explain GroupByKey() operation
> GroupByKey() is transformation which operate on pairRDD (which contains Key/Value).
> PairRDD contains tuple, hence we need to pass the function that operator on tuple instead of each element.
> Its Group values with the same Key in new RDD.
> It is a wide operation because it shuffles data across multiple partition
It works on key value pair, returns a new dataset of grouped items. It will return the new RDD which is made up with key (which is a group) and list of items of that group. Order of elements within group may not be the same when you apply same operation on same RDD over and over. It's a wide operation as it shuffles data from multiple partitions / divisions and create another RDD.
Example :
val rdd1 = sc.parallelize(Seq(5,10),(5,15),(4,8),(4,12),(5,20),(10,50)))
val rdd2 = rdd1.groupByKey()
rdd2.collect()
Output:
Array[(Int, Iterable[Int])] = Array((4,CompactBuffer(8,12)), (10,CompactBuffer(50)), (5,CompactBuffer(10,15,20)))

Explain reduceByKey() operation
> reduceByKey() is transformation which operate on pairRDD (which contains Key/Value)
> PairRDD contains tuple, hence we need to pass the function that operator on tuple instead of each element
> It merges the values with the same key using associative reduce function
> It is wide operation because data shuffles may happen across multiple partitions
> It merges data locally before sending data across partitions for optimize data shuffling
> It takes function as an input which has two parameter of the same type (values associated with same key) and one element output of the input type(value)

> We can say that it has three overloaded functions :|
reduceBykey(function)
reduceByKey(function, numberofpartition)
reduceBykey(partitioner, function)

It uses associative reduce function, where it merges value of each key. It can be used with Rdd only in key value pair. It's wide operation which shuffles data from multiple partitions/divisions and creates another RDD. It merges data locally using associative function for optimized data shuffling. Result of the combination (e.g. a sum) is of the same type that the values, and that the operation when combined from different partitions is also the same as the operation when combining values inside a partition.
Example :
val rdd1 = sc.parallelize(Seq(5,10),(5,15),(4,8),(4,12),(5,20),(10,50)))
val rdd2 = rdd1.reduceByKey((x,y)=>x+y)
or
val rdd2 = rdd1.reduceByKey(_+_)
then
rdd2.collect()
Output:
Array[(Int, Int)] = Array((4,20),(10,50),(5,45))

Explain the mapPartitions() and mapPartitionsWithIndex()

mapPartitions() and mapPartitionsWithIndex() are both transformation.
mapPartitions() :
> 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
mapPartitions() :
> mapPartitionsWithIndex is similar to mapPartitions() but it provides second parameter index which keeps the track of partition.
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.
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>).
mapPartitions() syntax
def mapPartitions[U](f: (Iterator[T]) ⇒ Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]
Permalink
Return a new RDD by applying a function to each partition of this RDD.
mapPartitionsWithIndex syntax
def mapPartitionsWithIndex[U](f: (Int, Iterator[T]) ⇒ Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]
Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

Explain pipe() operation

It is a transformation

def pipe(command: String): RDD[String]

Return an RDD created by piping elements to a forked external process.

>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 language 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 writes the results as String to the standard output.

Explain 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())))
```

By Default, how many partitions are created ?

> 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)

> However, one can explicitly specify the number of partitions to be create

Example1:

> No Partition is not specified

```
val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt")
```

Example2:

> Following code create the RDD of 10 partitions, since we specify the no. of partitions.

```
val rdd1 = sc.textFile("/home/hdadmin/wc-data.txt", 10)
```

One can query about the number of partitions in following way :

```
rdd1.partitions.length
```

OR

rdd1.getNumPartitions
> Best case Scenario is that, we should make RDD in following way :
numbers of cores in Cluster = no. of partitions

Explain values() operation
> values() is a transformation
> It returns an RDD of values only
val rdd1 = sc.parallelize(Seq((2,4),(3,6),(4,8),(5,10),(6,12),(7,14),(8,16),(9,18),(10,20)))
val rdd2 = rdd1.values
rdd2.collect
Output:
Array[Int] = Array(4, 6, 8, 10, 12, 14, 16, 18, 20)
Example2 : Values are duplicate in data set
val rdd1 = sc.parallelize(Seq((2,4),(3,6),(4,8),(2,6),(4,12),(5,10),(5,40),(10,40)))
val rdd2 = rdd1.keys
rdd2.collect
val rdd3 = rdd1.values
rdd3.collect
Output :
Array[Int] = Array(2, 3, 4, 2, 4, 5, 5, 10)
Array[Int] = Array(4, 6, 8, 6, 12, 10, 40, 40)

Explain foreach() operation
foreach() operation is an action.
> It do not return no value.
> It executes input function on each element of an RDD.
It executes the function on each item in RDD. It is good for writing database or publishing to a web services. It executes parameter less function for each data items.
Example :
val mydata = Array(1,2,3,4,5,6,7,8,9,10)
val rdd1 = sc.parallelize(mydata)
rdd1.foreach{x=>println(x)}
OR
rdd1.foreach{println}
Output:
1
2
3
4
5
6
7
8
9
10

Explain the terms 'Partitions' and 'Partitioners

PARTITIONS :
Partitions also known as 'Split' 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 create.
Partitions are basically used to speed up the data processing.
PARTITIONER :
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 (number of partitions - 1)
Partitioner captures the data distribution at output. A scheduler can optimize future operation based on type of partitioner. (i.e. if we perform any operation say transformation or action which require shuffling across nodes in that we may need the partitioner. Please refer reduceByKey() transformation in the forum)
Basically there are three types of partitioners in Spark :
(1) Hash-Partitioner (2) Range-Partitioner (3) One can make its Custom Partitioner
Property Name : spark.default.parallelism
Default Value : For distributed shuffle operations like reduceByKey and join, the largest number of partitions in a parent RDD. For operations like parallelize with no parent RDDs, it depends on the cluster manager:
•Local mode: number of cores on the local machine
•Mesos fine grained mode: 8
•Others: total number of cores on all executor nodes or 2, whichever is larger
Meaning : Default number of partitions in RDDs returned by transformations like join

What is role of Driver program in Spark Application ?
> Driver program is responsible to launch various parallel operations on the cluster.
> Driver program contains application's main() function.
> It is the process which is running the user code which in turn create the SparkContext object, create RDDs and performs transformation and action operation on RDD.
> Driver program access Spark through a SparkContext object which represents a connection to computing cluster (From Spark 2.0 on-wards we can access SparkContext object through SparkSession).
> Driver program is responsible for converting user program into the unit of physical execution called taks.
> It also defines distributed datasets on the cluster and we can apply different operations on dataset (transformation and action).
> Spark program creates logical plan called Directed Acyclic graph which is converted to physical execution plan by driver when driver program runs.

Explain Accumulator in detail.
> Accumulator is shared variables in Apache Spark, used to aggregating information across the cluster.
> In other words, aggregating information / values from worker nodes back to the driver program. (How we will see in below session)
> When we use function inside the operation like map(), filter() etc these functions can use the variables which defined outside these function scope in the driver program.
> When we submit the task to cluster, each task running on the cluster gets a new copy of these variables and updates from these variable do not propagated back to the driver program.
> Accumulator lowers this restriction.

> One of the most common use of accumulator is count the events that occur during job execution for debugging purpose.
> i.e. count the no. of blank lines from the input file, no. of bad packets from network during session, during Olympic data analysis we have to find age where we said (age != 'NA') in SQL query in short finding bad / corrupted records.
Examples :

```
scala> val record = spark.read.textFile("/home/hdadmin/wc-data-blanklines.txt")
record: org.apache.spark.sql.Dataset[String] = [value: string]
scala> val emptylines = sc.accumulator(0)
warning: there were two deprecation warnings; re-run with -deprecation for details
emptylines: org.apache.spark.Accumulator[Int] = 0
scala> val processdata = record.flatMap(x =>
{
if(x == "")
emptylines += 1
x.split(" ")
})
processdata: org.apache.spark.sql.Dataset[String] = [value: string]
scala> processdata.collect
16/12/02 20:55:15 WARN SizeEstimator: Failed to check whether UseCompressedOops is set; assuming yes
```

Output :

```
res0: Array[String] = Array(DataFlair, provides, training, on, cutting, edge, technologies., "", DataFlair, is, the, leading, training, provider,, we, have, trained, 1000s, of, candidates., Training, focues, on, practical, aspects, which, industy, needs, rather, than, theoretical, knowledge., "", DataFlair, helps, the, organizations, to, solve, BigData, Problems., "", Javadoc, is, a, tool, for, generating, API, documentation, in, HTML, format, from, doc, comments, in, source, code., It, can, be, downloaded, only, as, part, of, the, Java, 2, SDK., To, see, documentation, generated, by, the, Javadoc, tool,, go, to, J2SE, 1.5.0, API, Documentation., "", Javadoc, FAQ, -, This, FAQ, covers, where, to, download, the, Javadoc, tool,, how, to, find, a, list, of, known, bugs, and, feature, reque...
scala> println("No. of Empty Lines : " + emptylines.value)
No. of Empty Lines : 10
```

Explanation and Conclusion of Program :
> In above example, we create an Accumulator[Int] 'emptylines'.
> Here, we want to find the no. of blank lines during our processing.
> After that, we applied flatMap() transformation to process our data but we want to find out no. of empty lines (blank lines) so in flatMap() function if we encounter any blank line, accumulator emptylines increase by 1 otherwise we splite the line by space.
> After that, we check the output as well as no. of blank lines.
> We create the accumulator with initial value in driver program, by calling sc.accumulator(0) i.e. spark Context.accumulator(initial Value) where the return type of type initalValue {org.apache.spark.Accumulator[T] where T is initalValue]
> At the end we call the value() property on the accumulator to access its value.
> Please note that, task(s) on worker nodes can not access the value property of accumulator so for in context of task(s), accumulator is write-only variable.
> The value() property of accumulator is available only in the driver program.

> We can also count the no. of blank lines with the help of transformation / actions but for that we need extra operation but with the help of accumulator we can count the no. of blank lines (or events in broader terms) as we load /process our data.