Apache Spark is a fast and general-purpose cluster computing system. Spark is a process engine but need a cluster to store data. Apache spark supports in-memory cluster computing (stores the data in RAM).Spark does not have it's own file management system.

Different File Formats supported by spark and ways to load them.
1)sparkContext :- textfile,sequencefile
2)sqlContext :- Paraquet,Avro,Json,ORC.

In master node you have the driver program (The driver program is the process running the main() function of the application) which drives your application. The code you write behaves as a driver program and in the interactive shell,shell behaves as driver program.The first thing to do in driver program is to create a spark context(which is useful to establish a connection to your cluster is an entry point in your spark application).Spark context works with the cluster manager to manage various jobs. A job is splitted into multiple tasks which are distributed across the worker nodes. The driver talks with the cluster manager & negotiates the resources .Cluster manager launches the executors on behalf of the driver. The driver will send tasks to executors and monitors them.

Apache Spark 2 main abstraction are RDD and DAG

1)RDD(Resilient Distributed Dataset) :- it is an immutable distributed collection of data ,where the data in rdd is divided into the logical partitions based on the no of executors,which may be computed on different nodes of cluster.

Resilient :- fault-tolerant with the help of RDD lineage graph(DAG) and so able to recompute missing or damaged partitions due to node failures.
Distributed :- since Data resides on multiple nodes.
Dataset :- represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or
database via JDBC with no specific data structure.

2)Direct Acyclic Graph (DAG):- The driver implictly converts the user code that contains transformations and actions into a logically directed acyclic graph called DAG.After which DAG gets converted into physical execution plan where it creates physical execution units called tasks.DAG is arrangment of edges and vertices ,where vertices indicate RDD and edges indicate operations on rdd.when we call an action DAG is submitted to DAG Scheduler that divides the graph into stages of jobs.

The Spark RDD operations are Transformations and ACtions

TRANSFORMATIONS :- Transformation is a process of forming new RDDs from the existing ones. it is a user specific function.All transformed RDDs are lazy in nature. To trigger the execution, an

action is a must. Up to that action data inside RDD is not transformed or available.
The transformations are of 2 types narrow and wide transformations.

NARROW TRANFORMATIONS :- The data on which transformation needs to be applied is from a single pattiton

WIDE TRANFORMATIONS :- The data on which transformation needs to be applied is from a multiple pattitons of parent RDD.

Actions :- An action is an operation, triggers execution of computations and RDD transformations. Also, returns the result back to the storage or its program

Lazy evaluation :- In accordance with a spark, it does not execute each operation right away, that means it does not start until we trigger any action. Once an action is called all the transformations will execute in one go.

you may also persist an RDD in memory using the persist (or cache) method, in which Spark will keep the elements around on the cluster for much faster access the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

DIfference between persist and cache :- The difference between cache() and persist() is that using cache() the default storage level is MEMORY_ONLY while using persist() we can use various storage levels.

lineLengths.persist() :- We can persist the RDD in memory and use it efficiently across parallel operations.The storage levels for persistance are
MEMORY_ONLY:-
MEMORY_AND_DISK :-
MEMORY_ONLY_SER :-
MEMORY_AND_DISK_SER :-
DISK_ONLY :-

rdd.cache() :- When we use the cache() method we can store all the RDD in-memory.

RDD lineage graph :- While we create a new RDD from an existing Spark RDD, that new RDD also carries a pointer to the parent RDD in Spark. That is the same as all the dependencies between the RDDs those are logged in a graph, rather than the actual data. It is what we call as lineage graph.After an action has been called, this is a graph of what transformations need to be executed.

Spark’s primary abstraction is a distributed collection of items called a Dataset. Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets. Due to Python’s dynamic nature, we don’t need the Dataset to be strongly-typed in Python. As a result, all Datasets in Python are Dataset[Row], and we call it DataFrame .

A Spark DataFrame is a distributed collection of data organized into named columns.Conceptually, it is equivalent to relational tables with good optimization techniques, that provides operations to filter, group, or compute aggregates, and can be used with Spark SQL. DataFrames can be constructed from structured data files, existing RDDs, tables in Hive, or external databases. It is an immutable distributed collection of data. DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction.DataFrame contains rows with Schema. The schema is the illustration of the structure of data.DataFrame in Apache Spark prevails over RDD but contains the features of RDD as well.
COnverting an rdd to dataframe Eg :- 1)val dataFrame = rdd.toDF(). 2)val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema).here schema is

```
val schema =
  StructType(
    schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))
```

READING AND WRITING A DATAFRAME :-

```
val df = sqlContext.read.format("json").load("examples/src/main/resources/people.json")
df.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

AFTER RUNNING AN SQL QUERY THE VALUES OF THE QUERY CAN BE ACESSED THROUGH :-

```
/* row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T] */
teenagers.map(_.getValuesMap[Any](List("name", "age"))).collect().foreach(println)
```

** DATAFRAME CAN BE CONVERTED INTO DATASET BY SPECIFYING THE CLASS .MAPPING WILL BE DONE BY NAME**
EG :-

```
val df = sqlContext.read.json(path)
          val ds = df..as[Person]. /* here Person is a class in scala */
```

Dataset is a data structure in SparkSQL which is strongly typed and is a map to a relational schema. It represents structured queries with encoders. It is an extension to data frame API. Spark Dataset provides both type safety and object-oriented programming interface. We encounter the release of the dataset in Spark 1.6.Datasets are similar to RDDs, however, instead of using Java Serialization or Kryo they use a specialized Encoder to serialize the objects for processing or transmitting over the network. While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.

Resilient Distributed Datasets (RDD) :-it is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.

Resilient :- fault-tolerant with the help of RDD lineage graph(DAG) and so able to recompute missing or damaged partitions due to node failures.
Distributed :- since Data resides on multiple nodes.
Dataset :- represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

HOw to create an rdd :-
1) val rdd = sc.textFile("/user/HAASAAT0876_13877/landing/sparklearning/Data/order_items")
2) val rdd = sc.sequenceFile("/user/HAASAAT0876_13877/landing/sparklearning/Data/order_items_seq")
3)val rdd = df.rdd //here df is converted to rdd.
writing an rdd to text and sequence file :-
1)rdd.saveAsTextFile("/user/HAASAAT0876_13877/landing/sparklearning/Data/oders1")
2)rdd.saveAsSequenceFile("/user/HAASAAT0876_13877/landing/sparklearning/Data/orders1_seq)

TRANSFORMATIONS ON RDD :-

1) Map :- Returns a new rdd by passing each element of source through a function.
Eg :- 1) val rdd1 = rdd.map(x => x.length) //here it takes an rdd and caluculate the no of characters for each line in rdd.
2) val rdd2 = rdd.map(x => x.split(',')).map(a => a(0)) //here it take the rdd and split it by comma and takes the 1st column after separting the records.

3)val rdd3 = rdd.flatMap(line => line.split(","))
val rdd4 = rdd3.map(x => (x,1))
val count = rdd4.reduceByKey((a,b) => a+b) //this is the example of wordcount 1st splited by , then for each word a key value pair is created then for same key values are added.

2)Filter :- Return a new rdd formed by selecting those elements of the source on which func returns true.
Eg :- val rdd2 = rdd.filter(x => x.contains("CLOSED"))

3)Flat Map :- Similar to map, but each input item can be mapped to 0 or more output items
Eg :- val rdd3 = rdd.flatMap(line => line.split(","))

4)Map partition :- Similar to map, but runs separately on each partition (block) of the RDD.It converts each partition of the source RDD into multiple elements of the result

5)Map partition with index :- Similar to mapPartitions, but also provides func with an integer value representing the index of the partition.Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

6)Union :- It contains a new rdd that contains all the elements in the source rdd's
[eg :-val rdd3 = rdd1.union(rdd2)].

7)intersection :- It contains a new rdd that contains matching elements in the source rdd's and second rdd .
Eg :- val low = (1 to 5).toSet
val medium = (3 to 7).toSet
val i = low.intersect(medium)

8)distinct() :- Return a new dataset that contains the distinct elements of the source dataset.
Eg :- val result = d1.distinct().

9)groupByKey() :- When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
Eg :- val rdd1 = rdd.groupByKey().

10)reduceByKey() :- When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func.
Eg :- val rdd2 = rdd1.reduceByKey(_+_) or val count = rdd4.reduceByKey((a,b) => a+b)

11)aggregateByKey :- When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions.aggregateByKey() is almost identical to reduceByKey() (both calling combineByKey() behind the scenes), except you give a starting value for aggregateByKey(). aggregateByKey function in Spark accepts total 3 parameters,

1)Initial value or Zero value
It can be 0 if aggregation is type of sum of all values
We have have this value as Double.MaxValue if aggregation objective is to find minimum value
We can also use Double.MinValue value if aggregation objective is to find maximum value
Or we can also have an empty List or Map object, if we just want a respective collection as an output for each key
2)Sequence operation function which transforms/merges data of one type [V] to another type [U]
3)Combination operation function which merges multiple transformed type [U] to a single type [U]
Eg :- val aggrRDD = studentRDD.map(t => (t._1, (t._2, t._3)))
.aggregateByKey(zeroVal)(seqOp, combOp)
//here seqOp &combOp are functions for the sequence and combination.

12)sortByKey () :- When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys

in ascending or descending order, as specified in the boolean ascending argument.
Eg :- val rdd1 = rdd.sortByKey()

Join () :- When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
Eg :- val rdd3 = rdd2.join(rdd1).

13)cogroup () :- When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples.
Eg :- var rdd3 = rdd1.cogroup(rdd2).

14)cartesian :- When called on datasets of types T and U, returns a dataset of (T, U) pairs .Cartesian transformation generates a cartesian product of two RDDs. Each element of our first RDD is paired with each element of the second RDD.
Eg :- val joined = customers.cartesian(products).

15)Pipe :- Sometimes in data analysis, we need to use an external library which may not be written using Java/Scala. Ex: Fortran math libraries. In that case, spark's pipe operator allows us to send the RDD data to the external application.
Eg :- val pipeRDD = dataRDD.pipe(scriptPath) /* here the script path is the path to the any of external scripts */.

16)coalesce(numPartitions) :- Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
Eg :- val rdd1 = rdd2.coalsec(5,true). //here the first parameter is the num of partitions you want to reduce to and there is no need for the shuffle if you are chaniging the partition from 1000 to 100 and if there is huge change in partiton from 1000 to 1 then this may result in your computation taking place on fewer nodes than you like .Then to avoid this, you can pass shuffle = true

17)repartition :- Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.its recommended to use repartition while increasing no of partitions, because it involve shuffling of all the data.
Eg :- rdd.repartition(10).

18)repartitionAndSortWithinPartitions(partitioner) :- Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. It first repartitions the pairRDD based on the given partitioner and sorts each partition by the key of pairRDD.

Paired RDD :- Spark Paired RDDs are nothing but RDDs containing a key-value pair. Basically, key-value pair (KVP) consists of a two linked data item in it. Here, the key is the identifier, whereas value is the data corresponding to the key value.

ACTIONS ON RDD :-

1)Reduce () :- Aggregate the elements of the dataset using a function func. This function initialize accum variable with default integer value 0, adds up an element every when reduce method is called and returns final value when all elements of RDD X are processed.
Eg :- val y = x.reduce((accum,n) => (accum + n))

2)Collect () :- Return all the elements of the dataset as an array at the driver program.

3)Count() :- Return the number of elements in the dataset.

4)First() :- Return the first element of the dataset.

5)take(n) :- Return an array with the first n elements of the dataset.

6)takeSample(withReplacement, num, [seed]) :- Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.def takeSample(withReplacement: Boolean, num: Int, seed: Long = Utils.random.nextLong): Array[T]
Return a fixed-size sampled subset of this RDD in an array
withReplacement whether sampling is done with replacement
num size of the returned sample
seed seed for the random number generator
returns sample of specified size in an array

Eg :- inputrdd.takeSample(false, 3, System.nanoTime.toInt)

7)takeOrdered(n, [ordering]) :- Return the first n elements of the RDD using either their natural order or a custom comparator.takeOrdered(n,key=func)

Takeordered is an action that returns n elements ordered in ascending order as specified by the optional key function:

If key function returns a negative value (-1), the order is a descending order.
Eg :- rdd.takeOrdered(3, s => (-1) *s). or myrdd1.takeOrdered(3).

8)saveAsTextFile(path) :- Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem.

9)saveAsSequenceFile(path) :- Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem.

10)saveAsObjectFile(path) :- Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().

11)countByKey() :- Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. It actually counts the number of elements for each key and return the result to the master as lists of (key, count) pairs.
Eg :- rdd.countByKey().

12)foreach(func) :- Run a function func on each element of the dataset.

SPARK CONF :-
It contains information about your application and it is used to create spark context.
SPARK CONF():- SparkConf object contains information about your application.
Eg :- 1)val conf = new SparkConf()
.setAppName("simpleSparkApp")
.setMaster("yarn-client")
2)var conf = new SparkConf().setAppName("MI_KITCHEN_Inventory")
conf.set("spark.yarn.maxAppAttempts", "1");
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
conf.set("spark.kryoserializer.buffer.max", "1024m") .

SPARK CONTEXT :-
Spark context sets up internal services and establishes a connection to a Spark execution environment.It basically act as master to your spark application.

Once a SparkContext is created you can use it to create RDDs, accumulators and broadcast variables, access Spark services and run jobs (until SparkContext is stopped).

A Spark context is essentially a client of Spark's execution environment and acts as the master of your Spark application (don't get confused with the other meaning of Master in Spark, though).

SC.PARALLELIZE() :- The sc.parallelize() method is the SparkContext's parallelize method to create a parallelized collection. This allows Spark to distribute the data across multiple nodes, instead of depending on a single node to process the data.

Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. 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.

Broadcast variables are used to save the copy of data across all nodes. This variable is cached on all the machines and not sent on machines with tasks. A Broadcast variable has an attribute called value, which stores the data and is used to return a broadcasted value.

```
Eg :- words_new = sc.broadcast(["scala", "java", "hadoop", "spark", "akka"])
 data = words_new.value
print "Stored data -> %s" % (data)
```

ACCUMLATOR:-
Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster Accumulator variables are used for aggregating the information through associative and commutative operations.Accumulators are created from an initial value v; i.e. `SparkContext.accumulator(v)`. Then the tasks running in the cluster can be added to it using the known "add method" or += operator in python. They cannot, however, read the value of it. The driver program has the ability to read the value of the accumulator, using the `value` method

```
Eg :- val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3)).foreach(x => accum += x)
data = accum.value
print "the value is -> %i" % (data).
```

The first thing a Spark program must do is to create a SparkContext object, which tells Spark how to access a cluster. To create a SparkContext you first need to build a SparkConf object that contains information about your application.

SPARK CONTEXT() :- Main entry point for Spark functionality. A SparkContext represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.

```
Eg :-  val sc = new SparkContext(conf).
```

SPARK CONF():- SparkConf object contains information about your application.

```
Eg :-  val conf = new SparkConf()
       .setAppName("simpleSparkApp")
       .setMaster("yarn-client")
```

SPARK SQL :- Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine. The entry point into all functionality in Spark SQL is the SQLContext.

```
val sqlc = new SQLContext(sc).
val hiveContext = new HiveContext(sc).
```

SparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file.

Deploy mode :- Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.

EXPLODE :- The explode function takes a column that consists of arrays and creates one row (with the rest of the

values duplicated) per value in the array
Eg :- select explode(col2) as mycol from table1;
/* here the col2 is the name of columns which contain the array of the values in the row so by applying explode we will seperate each value in array as seperate rows */.

LATERAL VIEW :- lateral view will create a join on the base table and the virtual table created by the explode function and gives the output containing base columns of base table and exploded columns of the virtual table.
Eg :- select author_name,dummy_bookname from table2 lateral view explode(boks_name) virtual as dummy_bookname
/* here the explode function on the column will give a virtual table named virtual and in that we created an alias name to the column dummy_bookname and used that with base columns of the base table */.

Another use of lateral view is it will take the column of the map types and gives output with key and value columns seperately.
Eg:- select key,value from table3 lateral view explode(col1) virtual_table as key,value.

DIFFERENT METHODS TO CREATE A DATAFRAME :-

1)CONVERTING AN RDD TO DATAFRAME :-

```
val df1 = rdd.toDF() //here no schema is applied

//With schema
val schema = new StructType()

.add(StructField("eno",StringType,true))
.add(StructField("salary",StringType,true))
.add(StructField("pname",StringType,true))

val df2 = sqlContext.createDataFrame(rdd,schema)
```

2)SPARK READ FORMAT :-

```
Eg :- val df = sqlContext
    .read.format("com.databricks.spark.csv")
    .option("header", "true")
    .option("inferschema", "true")
    .option("mode", "DROPMALFORMED")
    .load("some_input_file.csv"))
                    (or)

      val df =
sqlContext.read.option("header","true").option("inferschema","true").
json("/user/HAASAAT0876_13877/landing/sparklearning/Data/orders.json"
).
```

3)FROM THE DATABASE TABLES OR FROM THE HIVE TABLES.

```
Eg :-      val
data=sqlContext.read.format("jdbc").options().map("url"-
>"jdbc:mysql://localhost:3306/education","driver"-
```

```
>"com.mysql.jdbc.Driver","table"->"emp","user"->"root","password"-
>"inetsolve").load())
                                                (or)
      val dataframe = sqlContext.read.format("jdbc").option("url",
"jdbc:db2://localhost/sparksql").option("driver",
"com.mysql.jdbc.Driver").option("dbtable", "table").option("user",
"root").option("password", "root").load().
                                                (or)
       val df4 = sqlContext.sql("select * from
HAASAAT0876_13877.mcrh_acceptance").
```

TO WRITE A DATA FRAME DIRECTLY INTO A HIVE TABLE :-
1)df.write().mode(Overwrite).saveAsTable("dbName.tableName");
2)df.write().mode("overwrite").saveAsTable("schemaName.tableName");

TO WRITE A DATA FRAME DIRECTLY INTO A DATABASE TABLE :-
df.write.mode(Overwrite).jdbc(url,tablename,properties) // here the url if the jdbc connection url for the relational database and properties are the connection properties to connect to the database such as username,driver etc.

TO SAVE A TABLE DIRECTLY AS A CSV FILE :-
df.write.csv("/your/location/data.csv")

To get the schema of the dataframe use " df.printSchema()"

To creaete a temporary table on the dataframe
"df1.registerTempTable("sampletable")"

To create a view on the dataframe "df1.createTempView" and the view is available unitll the session is present.
To create a globaltemporary view on the dataFrame "df1.creaeteGlobalTampView" and this view is available across all the session and Global temporary view is tied to a system preserved database global_temp.

COVERTING THE RDD TO DATAFRAME WITH THE HEADER :-
This can be done by in 2 ways 1)case class and 2)split method.3)By defining it while creating a dataframe in option("header",true).

1)Case class :- A Scala Case Class is like a regular class, except it is good for modeling immutable data,Scala case classes hold all vals, and this makes them immutable.There is a default apply() method in scala case class which takes care of the object creation.

```
Eg :-  case class Demo(id : String,date : String)
           val df12 = df11.as[Demo]
                              (or)
      val rdd8 = rdd.map( p => Demo(p(0).toString,p(2).toString))
       val dataframe = rdd8.toDF()
       dataframe.printSchema().
                              (or)

       case class Demo1(id : String ,date : String,cid : String,status
: String)
      val df11
=rdd.map(_.split(",")).map(r=>Demo(r(0),r(1),r(2),r(3))).toDF
```

```
      val newDf = df.withColumn("D", when($"B".isNull or $"B" === "",
0).otherwise(1))
      rawDF.withColumn("FullName",concat_ws("
",rawDF("FirstName"),rawDF("SurName")))
```

ADDING NEW COLUMN TO THE TABLE :-

1)With Expression :-

```
val df = Seq((1,2)).toDF("x","y") //here we are adding header to
table where column names as x& Y.

val myExpression = "x+y"
df.withColumn("z",expr(myExpression)).show()  //here we are creating
a new column z with expression on the previous columns.
```

2)With String Concatination

```
val df3 = df2.withColumn("fullname",concat_ws("
",df2("name"),df2("surname"))) //here in contcat_ws we will concat 2
columns for the value of 3rd column (fullname which is the new
column) with space in between 2 column values in 3rd column.

 val df4 =
df2.withColumn("fullname",concat(df2("name"),df2("surname"))) //here
in contcat_ws we will concat 2 columns for the value of 3rd column
(fullname which is the new column) continously without any space.
```

FILTERING THE NULL VALUES FROM THE DATAFRAME :-

```
val df2 = df.filter($"location" === "null" || $"location" === "") or
val df2 = df.filter(col("location") === "null" || col("location") ===
"")
 //here the column with null or empty string is filtered .
```

DROPING A ROW THAT CONTAINS EMPTY OR NULL VALUE :-

```
 val df2 = df.filter($"location" === "null" || $"location" ===
"").drop("location") //here row containing empty and null values in
"location" column are removed .
```

REPLACING A COLUMN VALUE OF NULL STRING TO NULL VALUE :-

```
 val df1 = df.withColumn("location", when(col("location") ===
"null","" ).otherwise(col("location")))
                                                  or
  val nullDF=df.withColumn("area", when(col("location") === "",
lit(null)).otherwise(col("location")))
 //here a location column contaning a null string is replaced with
null.

 nullDF.na.drop().show() or nullDF.na.drop("any").show()  //here the
rows which contain either the one column with one null value or the
entire row is null are droped.
```

```
 nullDF.na.drop("all").show() //here the rows which are entirely null
are droped from the dataFrame.

  nullDF.na.fill("aaaa").show()  //here the columns which contain the
null values are replaced by the string in the paranthesis.

  nullDF.na.replace("location",Map("" -> "UNKNOWN")).show()  //here
the DataFrame(nullDF) has the column location and if there is a null
value it is replaced with the string.

STRING MATCHING CONDITION ON DATAFRAME WITH CONTAINS AND ==== :-
df1.filter(col("name") === "abbulu").show()  //Here the o/p is row
that contains the name column as "abbulu".

df1.filter(col("name").contains("abb")).show()  //here the o/p is row
that matches the substring condition "abb" in the name column.

RENAMING A COLUMN NAME :-
 df.withColumnRenamed("location" , "area") //here we renamed a column
location to area.
 or
val df = rdd.toDF(newnames) //here we rename the old columns to new
column names .

DSL COMMANDS(Dataframe . withcolumn,filter,contains etc ) domain
specific language.


SUBSTRING OPERATIONS :-

 val result = df.withColumn("cutted", expr("substring(value, 1,
length(value)-1)"))  //Here we created a new column from the old
column (value) by using the substring function and cutting upto
particular length.

                                              or
val df1 = df.withColumn("middlename",substring(col("fullname"),6,5))
//here we are creating new column middlename from the old column
fullname starting at the position 6 to 5 places from 6 .

df.withColumn("new_name",when(substring(col("name"),1,3).contains(sub
string(col("fullname"),1,3)),true).otherwise(false)).show()  //here
the substrings of the 2 columns are compared if both the columns have
same value it prints true else false in the new column.


REGULAR EXPRESIONS ON DATAFRAME :-
df.filter(col("name") rlike ".*r.*").show()  //here in filter
condition we are looking for the match of the regular expresions
which has r in between.

df.filter(col("name") rlike "^a").show()  //here in filter condition
we are looking for the match of the regular expresions which has a in
start.
```

```
 df.filter(col("name") rlike "a$").show() //here in filter condition
we are looking for the match of the regular expresions which has a in
end.

 df.filter(col("location") rlike "mumba[iI]").show()  //here matches
either mumbai or mumbaI.

 df.filter(col("id") rlike "\\d").show()   //here matches a digit

  df.filter(col("id") rlike "[0-2]").show() //here matches a digit in
braces

 df.filter(col("id") rlike "\\D").show() //here matches a nondigit.

  df.filter(col("id") rlike "\\w").show() //here matcehs a single
word characters.

  df.filter(col("name") rlike "kiran?").show()  //here matches kira
or kiran.

  df.filter(col("name") rlike "kiran*").show() or
df.filter(col("name") rlike "kiran+").show() //here matches kira plus
one or more n.

  df.filter(col("id") rlike "\\d{1}").show() //matches exactly one
digit.

  df.filter(col("id") rlike "\\d{1,}").show()  //matches exactly one
or more digits.

  df.filter(col("id") rlike "\\d{1,2,3}").show()//matches one or 2 or
3 digits.


df.withColumn("area",regexp_replace(col("location"),"null","XXX")).sh
ow()  //here the column with null is replaced by the string.


df.withColumn("area",regexp_replace(col("fullname"),pattern,change)).
show()  //here the row with certain value is replaced by someother
value where the variable pattern has the value of matching string and
change variable has the newvalue of the string.


     Translate function :-  here the translate function replaces the
particular character index inside a string .
     df.withColumn("area",translate(col("fullname"),"a","e")).show()
//here the character "a" is replaced with character "e" in the new
column.


Can any plz answer the below hadoop interview questions

1, If 100 table out of 200 table has to be loaded in to HDFS, how to
specify the no of tables in SQoop scripts
```

2,what default compression algorithm for ORC file

3,what is Rank, widow, vectorization, parallelism n HIVE

4, how to optimize HIVE except partion, clustering, ORC

5, how to optimize spark job except narrow transformation

6,how to avoid group by stuffing in hive and spark. How it can be replaced in HIVE and spark

7, During the process of creating a data frame from a RDD, which package need to be imported

7,how to handle unstructured data in spark

8,how to check the process consume time for spark jobs stage by stage