**Big Data Analytics**

**Journal**

Submitted By

**Tagadghar Shailesh Tagadghar**

**Roll No: 31031523034**

**MSc CS – Part II**

**Department Of Computer Science**

**Somaiya Vidyavihar University**

**SK Somaiya college**

Index :

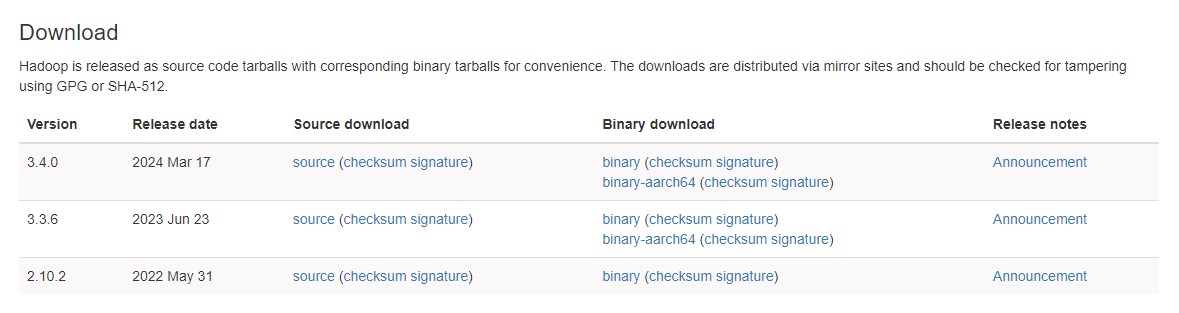
|  |  |
| --- | --- |
| Practical No. | Title |
|  | Installation of Hadoop in Windows. |
|  | Getting Started with Scala. |
|  | GraphX |
|  | PySpark |

**Practical 1**

**Aim :** Installation of Hadoop in Windows

**Steps :**

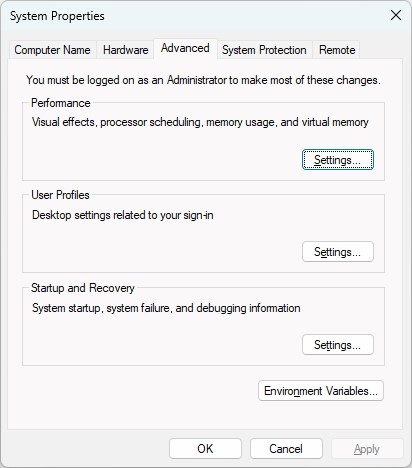
1. Download Binary File for Windows. <https://hadoop.apache.org/releases.html>



<https://dlcdn.apache.org/hadoop/common/hadoop-3.4.0/hadoop-3.4.0.tar.gz>

<https://www.oracle.com/java/technologies/javase-downloads.html>

1. Extract the file using Winrar.
2. Go to “Edit Environment Variables” and Click Environment Variables.

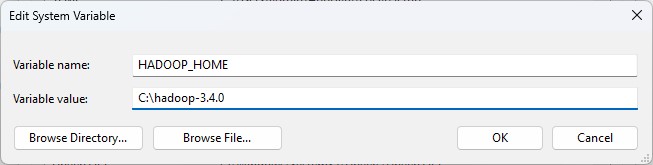
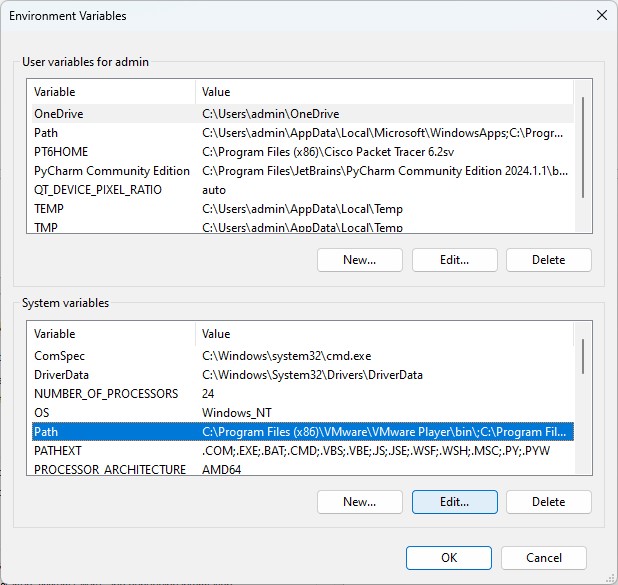


1. Under System Variables click “New” and set “Variable name” as JAVA\_HOME and “Variable value” as the path of your JAVA JDK.

# C:\Program Files\Java\jdk-21

1. Similarly add “HADOOP\_HOME” variable.
2. Download the bin folder from the below link <https://drive.google.com/drive/folders/1iURNbow2IglhAhSy3sfY5xxVfAg33NBW>
3. Extract the bin archive and replace the bin folder in Hadoop folder with the bin folder in this archive.
4. Check if “winutils” is working. If you get any dll error then download that dll and paste in the Windows -> System32 folder.
5. Move the Hadoop folder to C drive.
6. Create a data folder in the hadoop home directory and add the folders datanode and namenode to it.
7. Add the following path to “Path” under “System Variables” in “Edit Environment Variables”

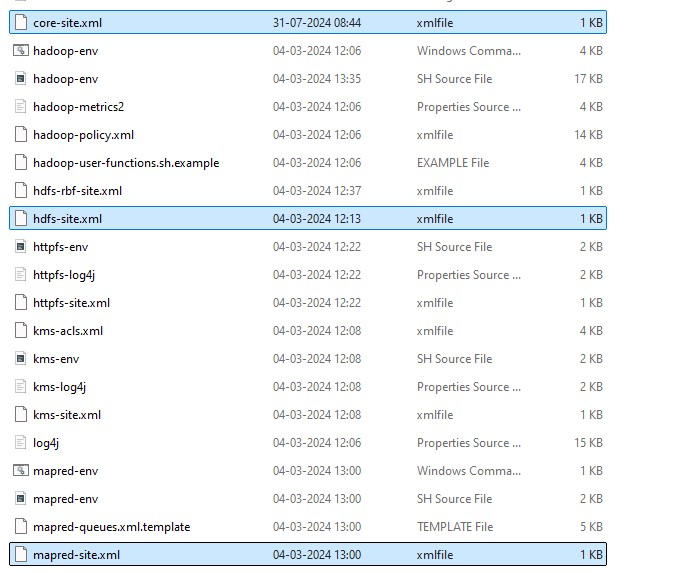
# C:\hadoop-3.4.0\sbin



12. If your PC username has spaces in it then go to hadoop-env.cmd and find this line

# set HADOOP\_IDENT\_STRING=%USERNAME%

1. Change the above line to your PC username instead of %USERNAME% but WITHOUT SPACES set HADOOP\_IDENT\_STRING=DeepShah
2. Make the changes to the following files as given, in “etc/hadoop” folder of hadoop home.



core-site.xml

<configuration>

<property>

<name>fs.default.name</name>

<value>hdfs://localhost:9000</value>

</property> </configuration>

mapred-site.xml

<configuration>

<property>

<name>mapred.framework.name</name>

<value>yarn</value>

</property> </configuration> hdfs-site.xml <configuration>

<property>

<name>dfs.replication</name>

<value>1</value>

</property>

<property>

<name>dfs.namenode.name.dir</name>

<value>C:\hadoop-3.4.0\data\namenode</value>

</property>

<property>

<name>dfs.datanode.data.dir</name>

<value>C:\hadoop-3.4.0\data\datanode</value>

</property> </configuration>

yarn-site.xml <configuration>

<property>

<name>yarn.nodemanager.aux-services</name>

<value>mapreduce\_shuffle</value>

</property> <property>

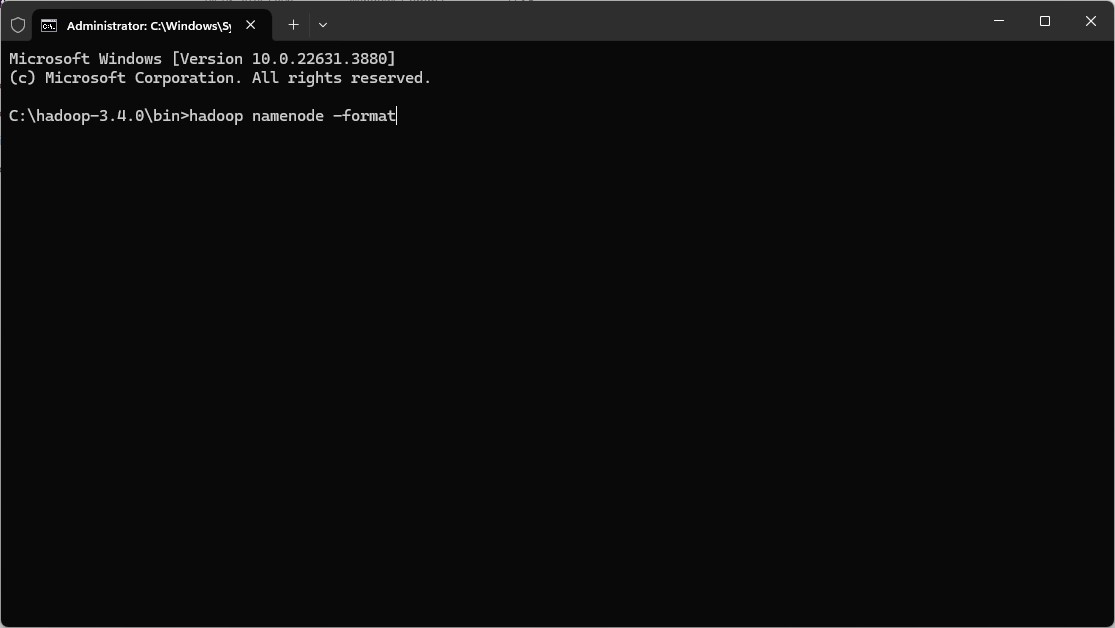
<name>yarn.nodemanager.auxservice.mapreduce.shuffle.class</name>

<value>org.apache.hadoop.mapred.shuffleHandler</value> </property> </configuration>

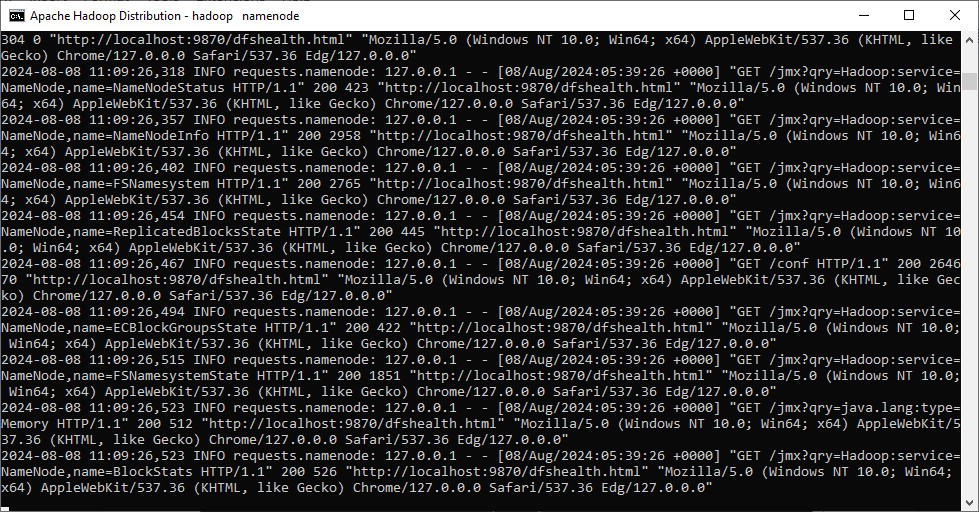
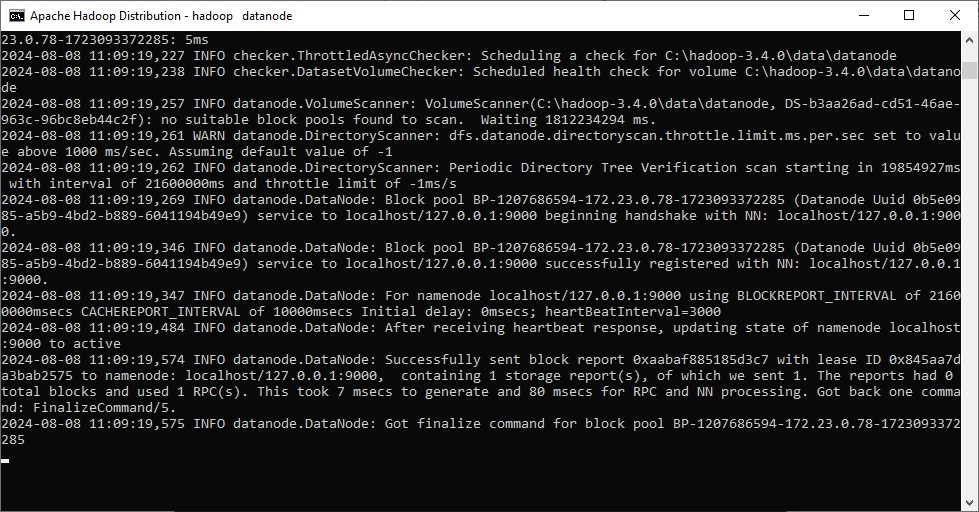
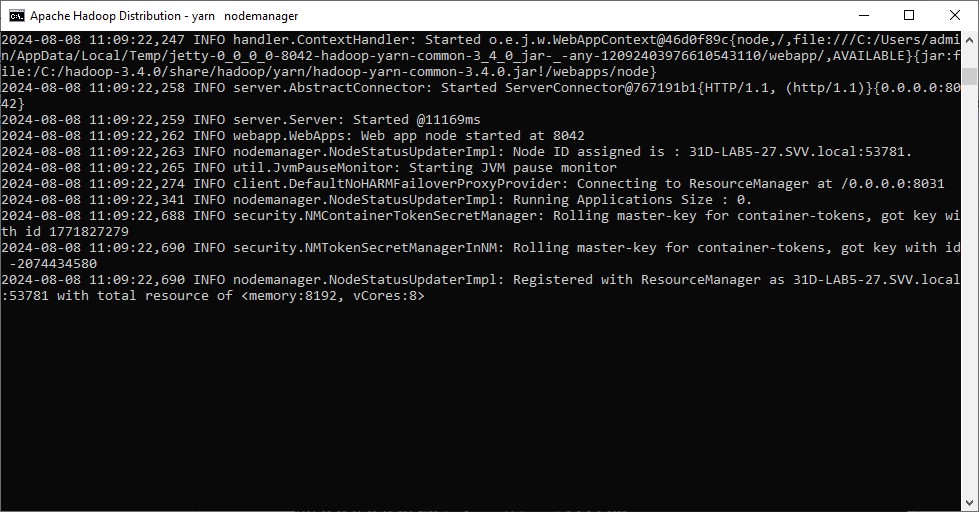
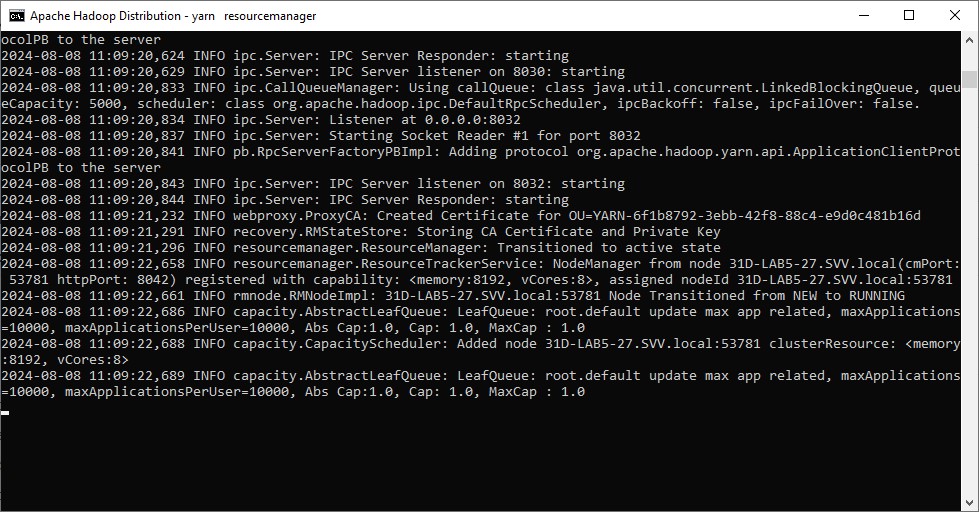
15. Go to hadoop-env.cmd file in /etc/hadoop folder and replace the set JAVA\_HOME=%JAVA\_HOME% line with the following:

# set JAVA\_HOME=C:\Progra~1\Java\jdk-21

1. Restart your PC for the changes to take effect.
2. Go to Admin Command prompt and type “hadoop” to see if the server is recognized.
3. Type “hdfs namenode -format” to format the namenode.



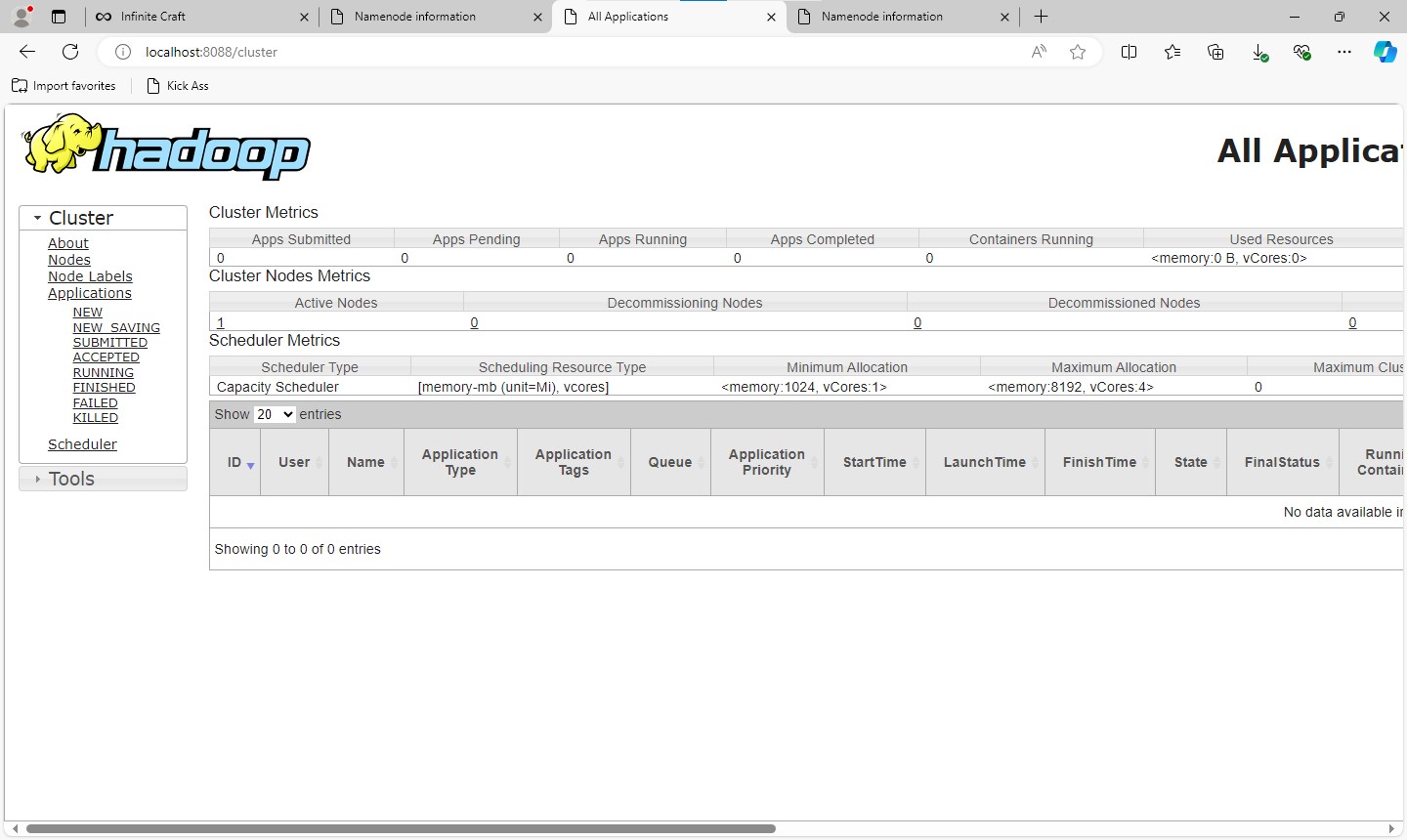
1. Type start-dfs.cmd and start-yarn.cmd to start all hadoop processes



1. Go to your browser and type localhost:9870 to view Hadoop Page.



1. If your yarn process doesnt start then use java version 11.
2. After the yarn process has started visit localhost:8088



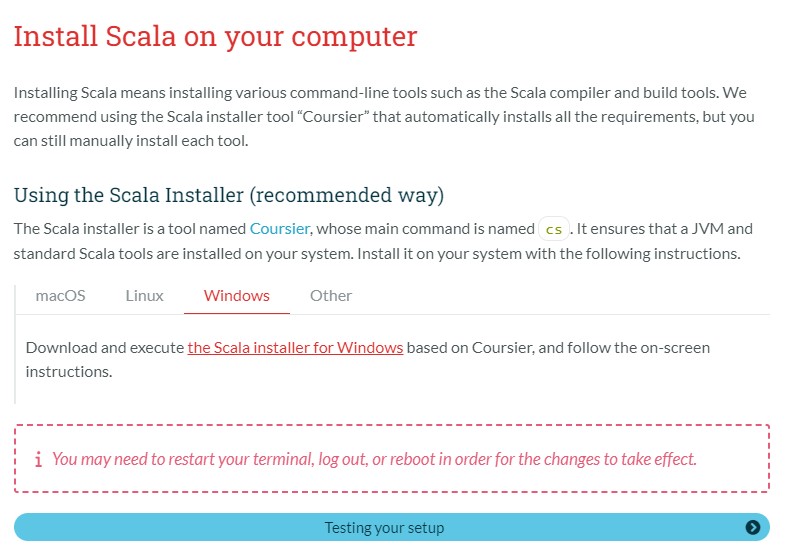
**Practical 2**

**Aim : Getting started with Scala**

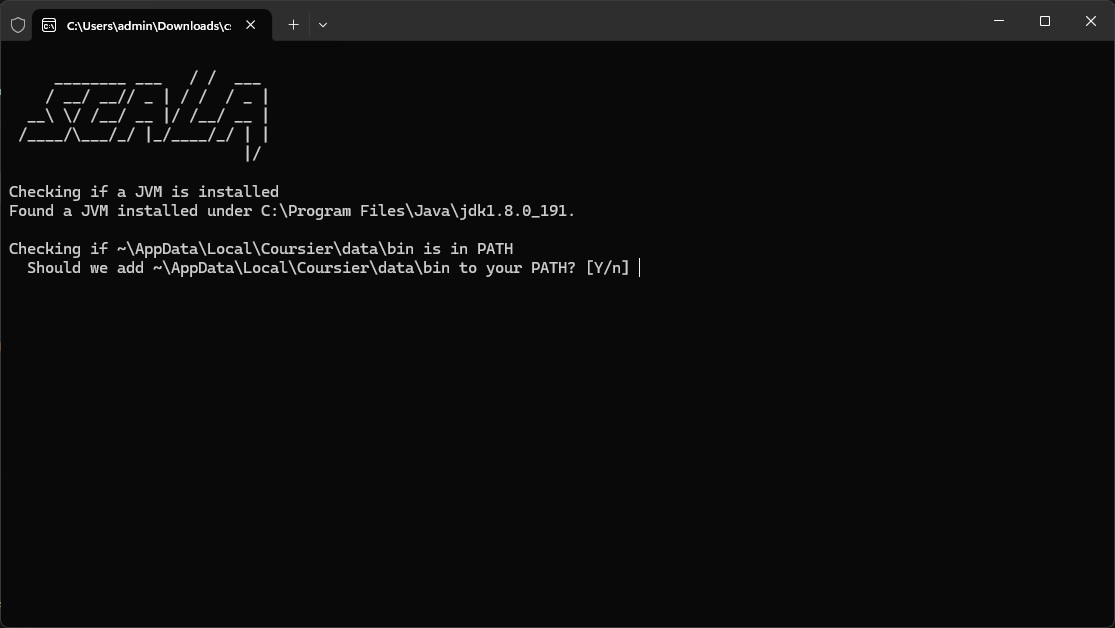
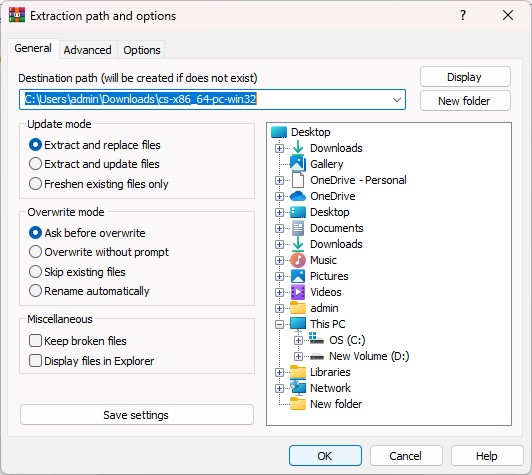
**Steps :**

* 1. Get Scala for Windows from the below link [Getting](https://docs.scala-lang.org/getting-started/index.html) [Started](https://docs.scala-lang.org/getting-started/index.html) [|](https://docs.scala-lang.org/getting-started/index.html) [Scala](https://docs.scala-lang.org/getting-started/index.html) [Documentation](https://docs.scala-lang.org/getting-started/index.html)

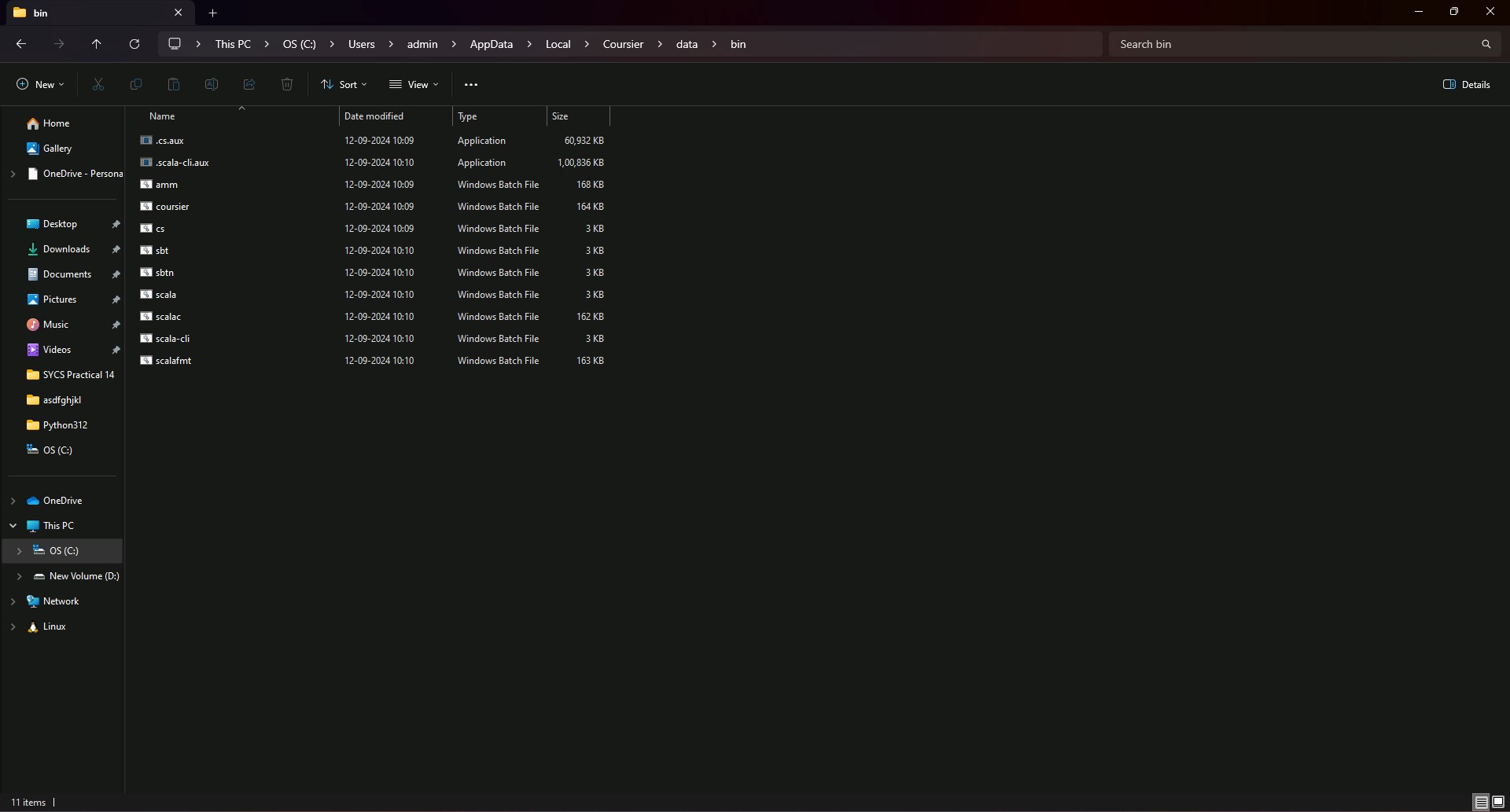
[(scala-lang.org)](https://docs.scala-lang.org/getting-started/index.html)



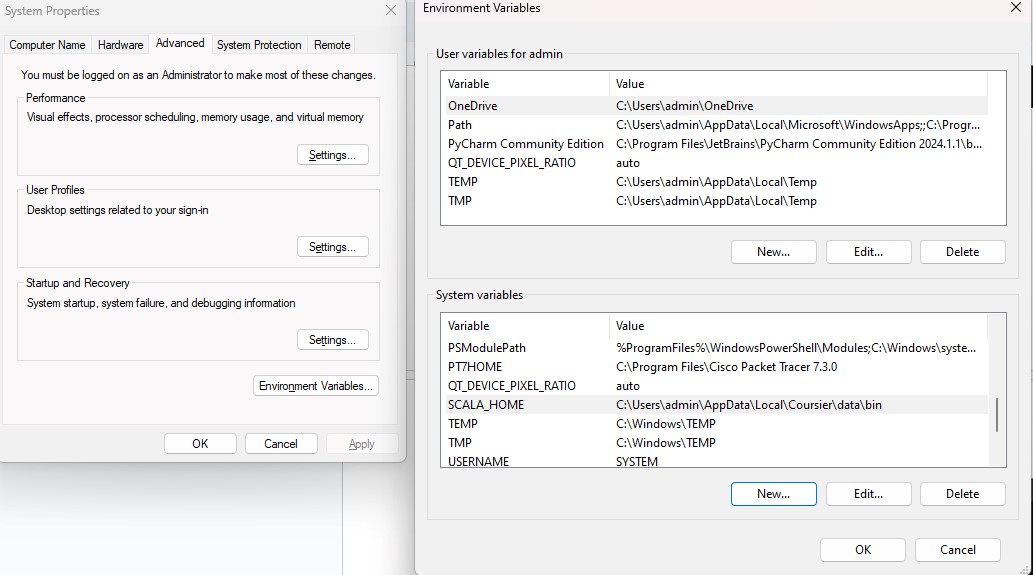
* 1. Extract the given archive and run the executable file.

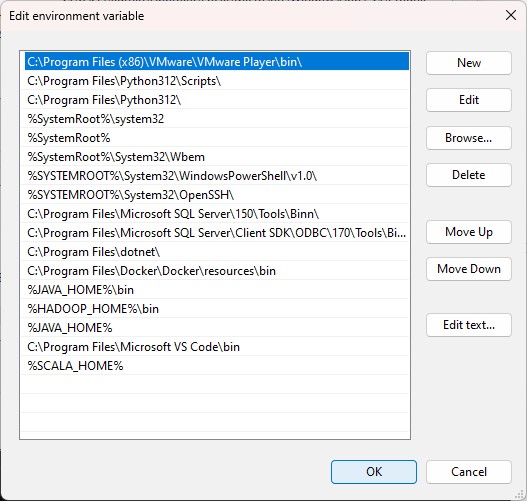


* 1. Go to the following path and then copy it.



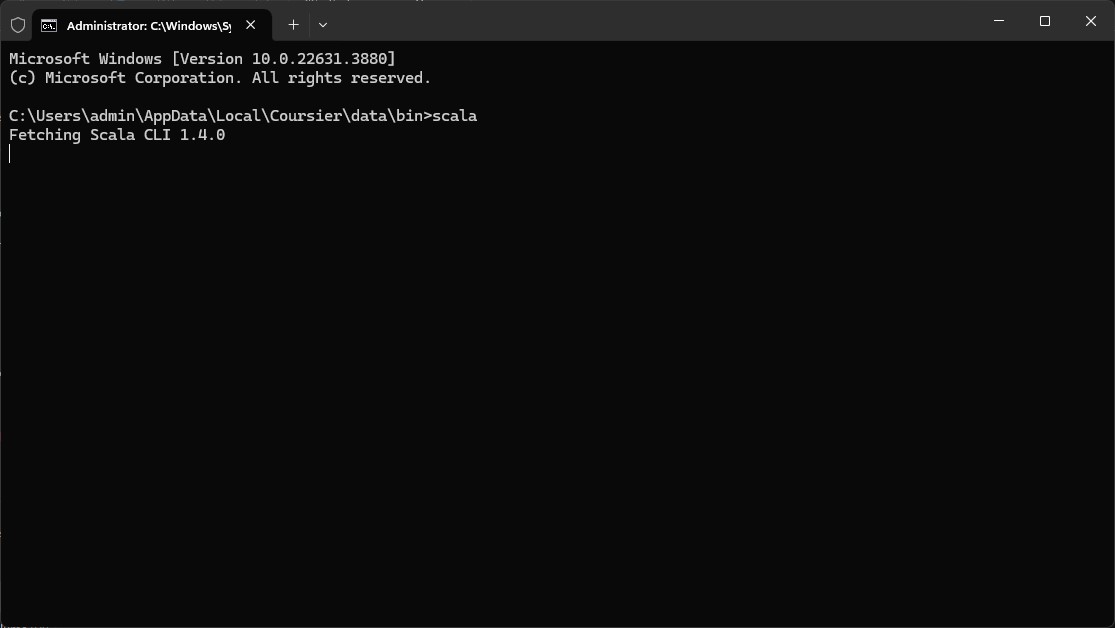
* 1. Set the SCALA\_HOME environment variable.



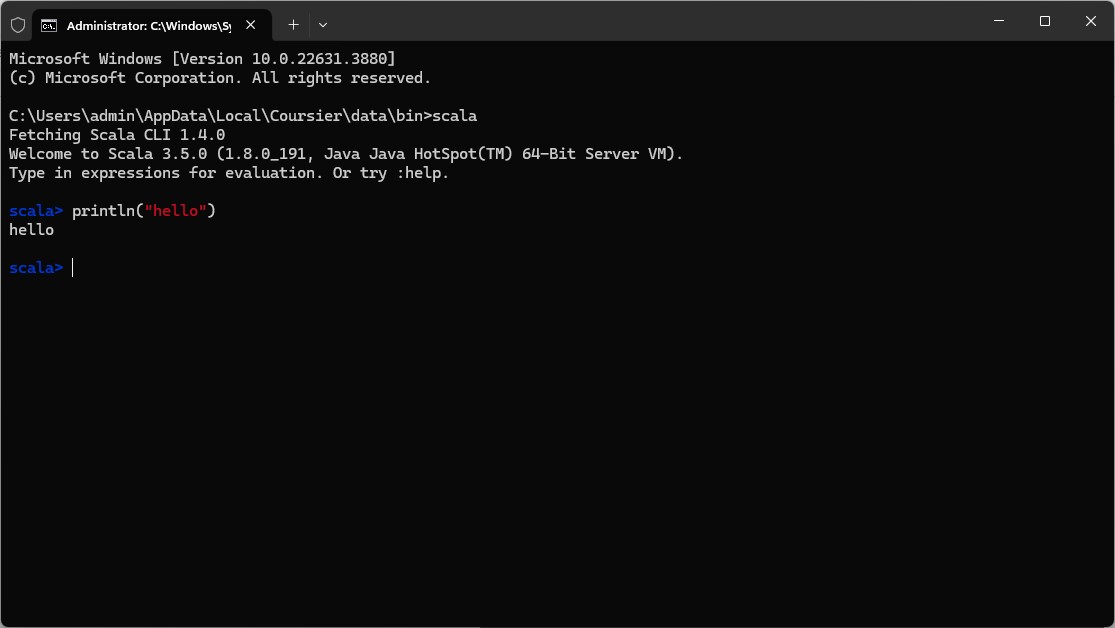


* 1. Open a command prompt on this path and type “scala”. It will download the Scala CLI.

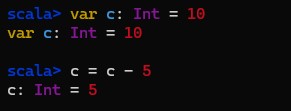
C:\Users\admin\AppData\Local\Coursier\data\bin

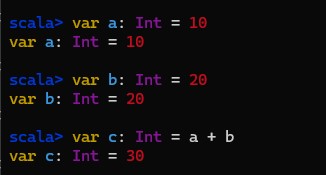


* 1. Test the working with a command or prompt ‘println(“Hello”)’.



‘Var’ before a variable name is used to make the variable mutable.





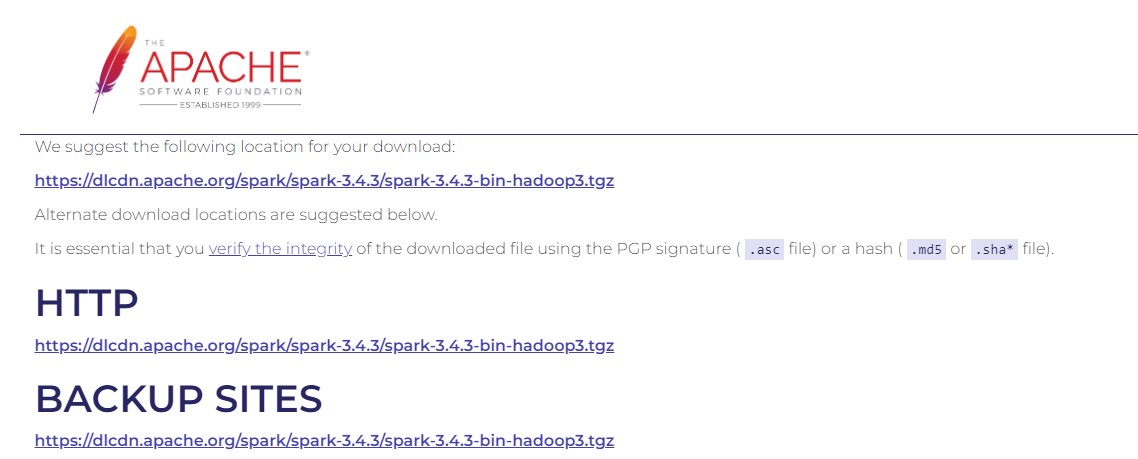
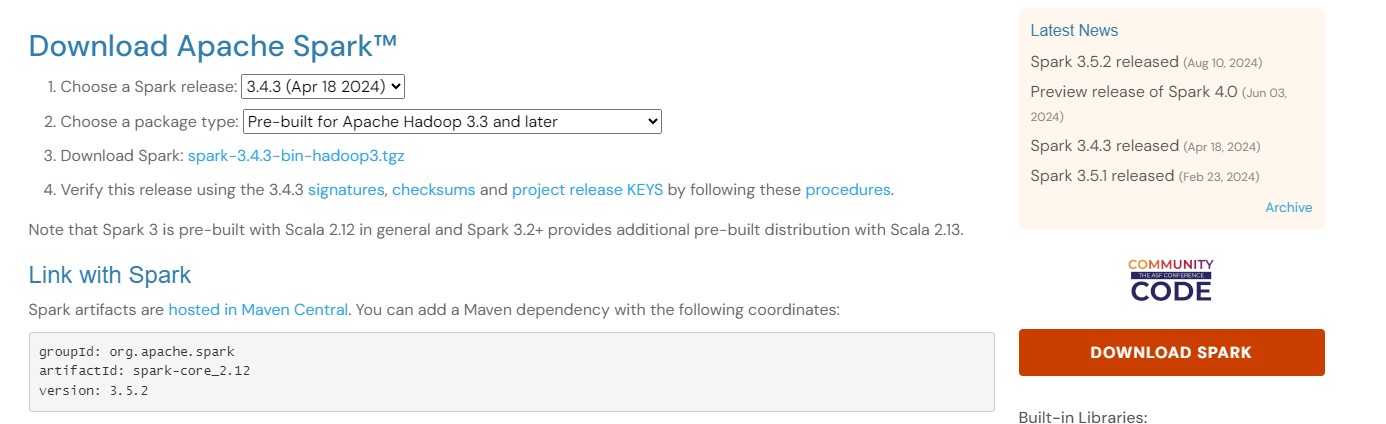
‘Val’ is used to make the variable immutable



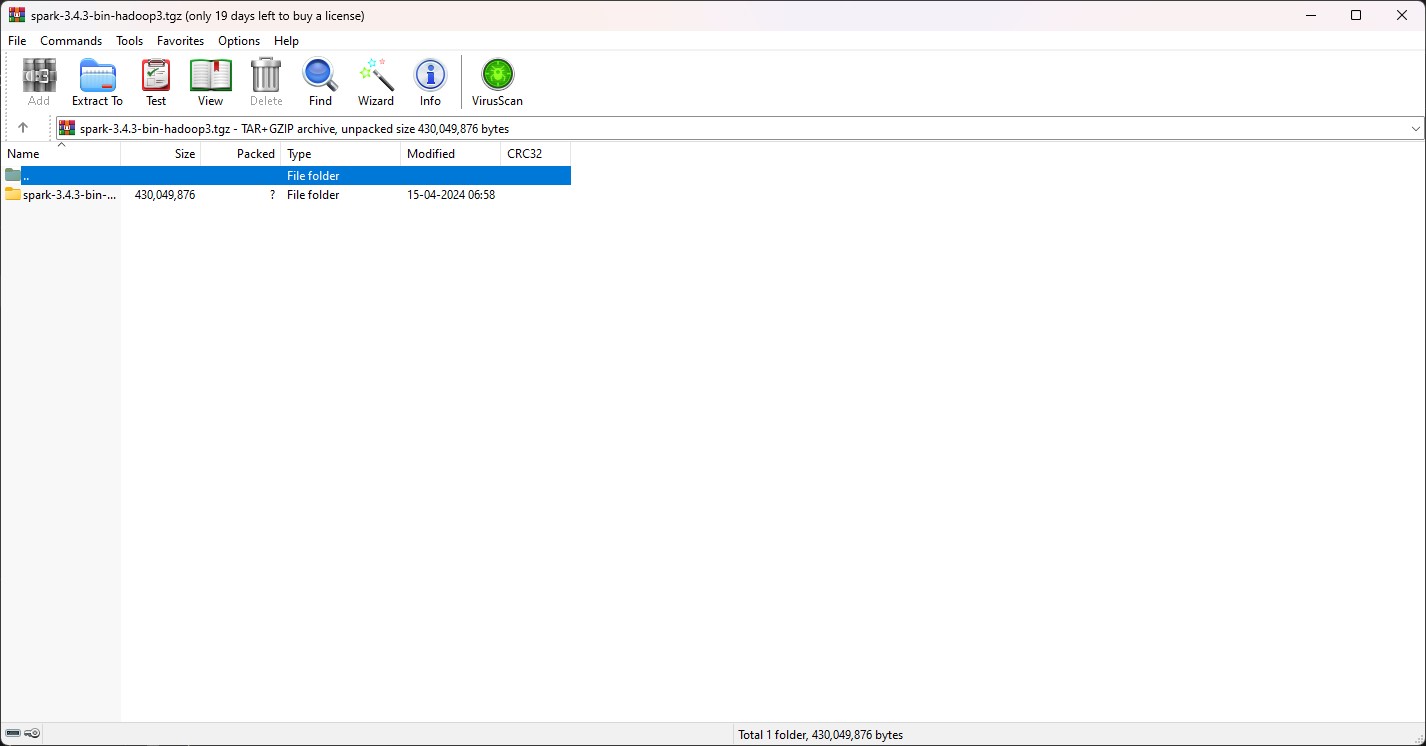
Installing Spark

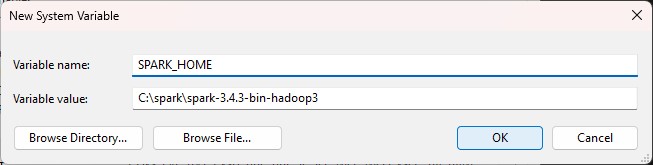
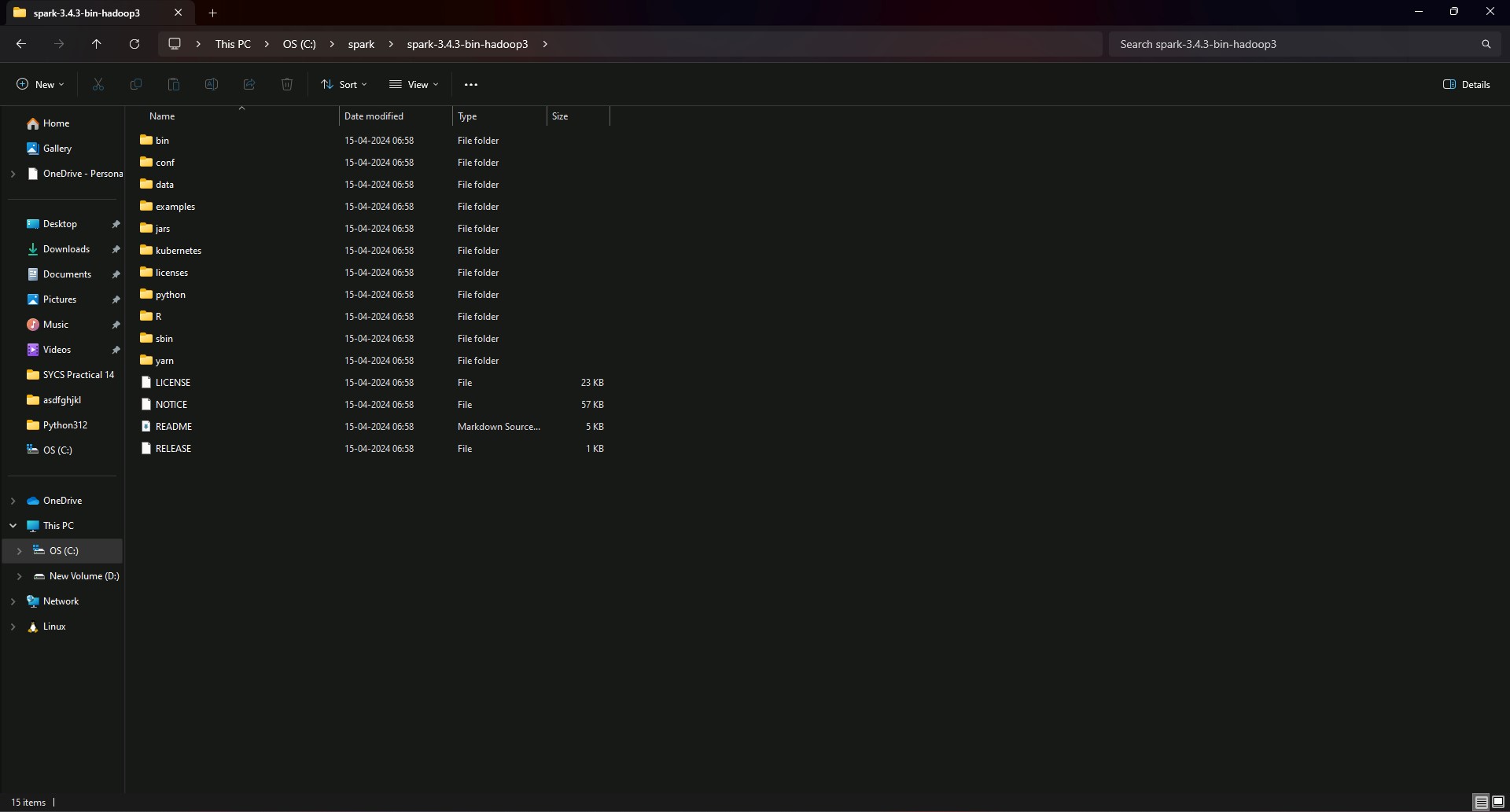
Steps

* 1. Download Spark from the following link [Downloads](https://spark.apache.org/downloads.html) [|](https://spark.apache.org/downloads.html) [Apache](https://spark.apache.org/downloads.html) [Spark](https://spark.apache.org/downloads.html)

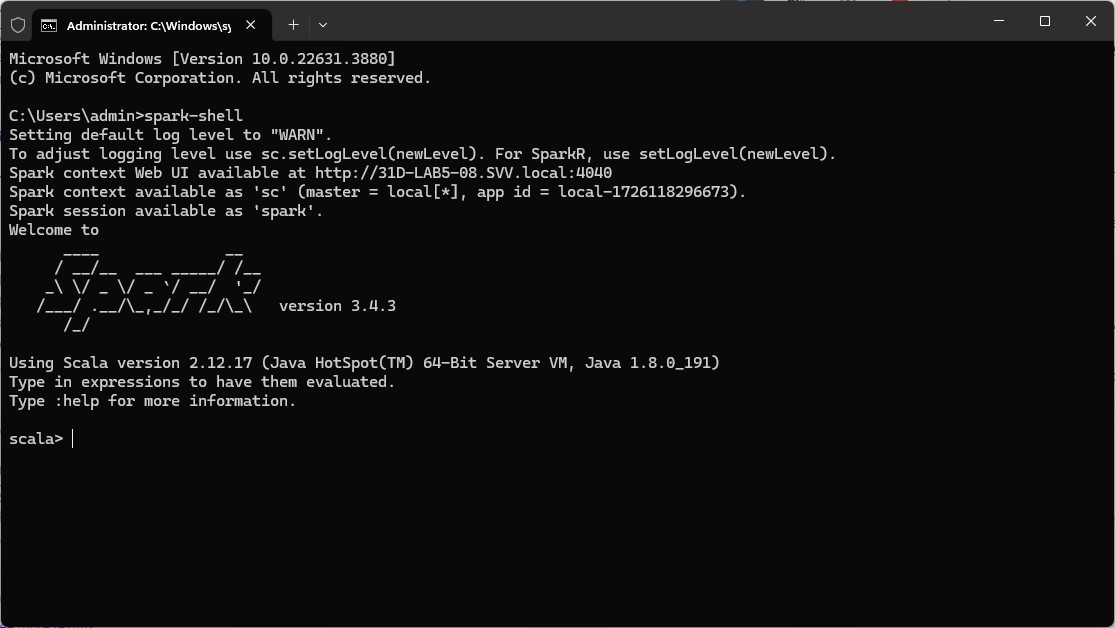


* 1. Extract the files and set the path for its home directory



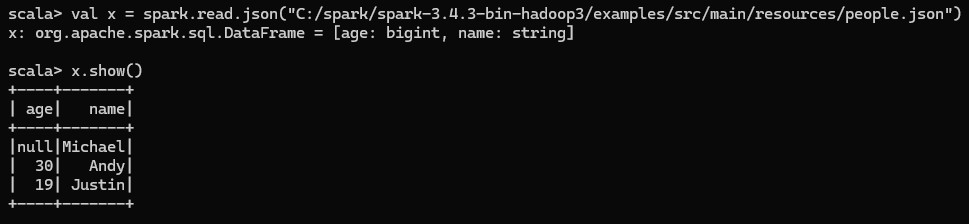


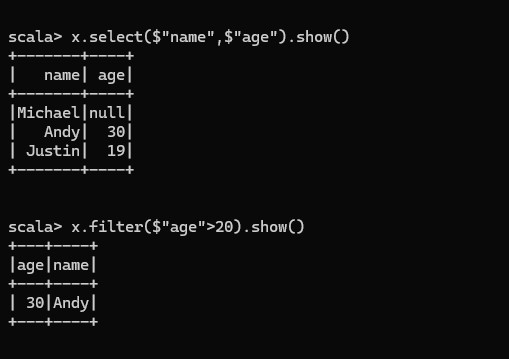
* 1. Check the installation with ‘spark-shell’



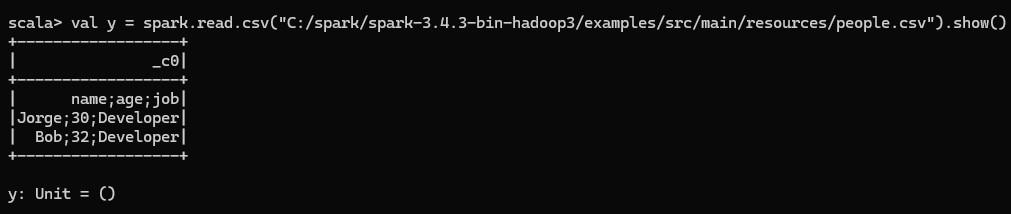
Lets try to show a sample data from this path

C:\spark\spark-3.4.3-bin-hadoop3\examples\src\main\resources\people.json

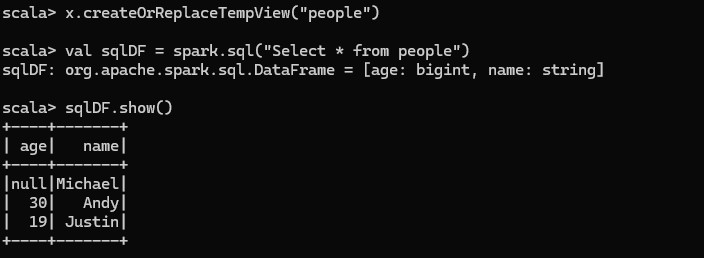




Reading CSV/Excel File

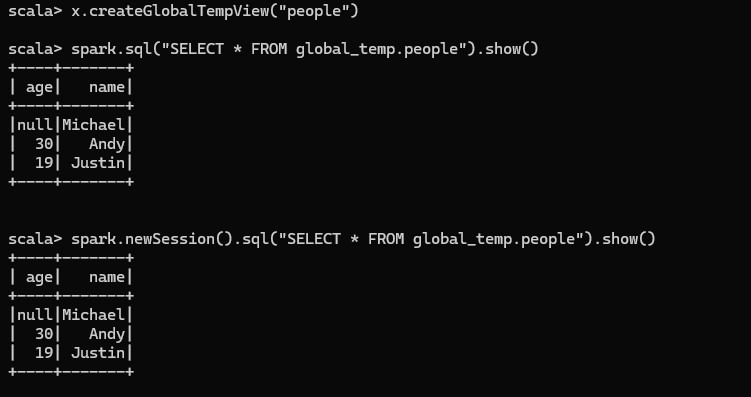


Creating an SQL Tempory View



Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view.

Global temporary view is tied to a system preserved database global\_temp, and we must use the qualified name to refer it, e.g. SELECT \* FROM global\_temp.view.



Creating Datasets

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.

case class Person(name: String, age: Long)

// Encoders are created for case classes val caseClassDS = Seq(Person("Andy", 32)).toDS() caseClassDS.show()

// +----+---+

// |name|age|

// +----+---+

// |Andy| 32|

// +----+---+

// Encoders for most common types are automatically provided by importing spark.implicits.\_ val primitiveDS = Seq(1, 2, 3).toDS() primitiveDS.map(\_ + 1).collect() // Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name

val path = "examples/src/main/resources/people.json" val peopleDS = spark.read.json(path).as[Person] peopleDS.show()

// +----+-------+

// | age| name|

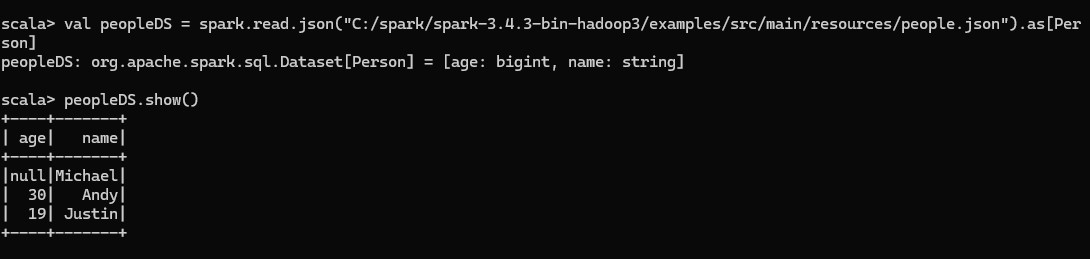
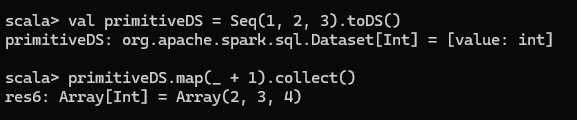
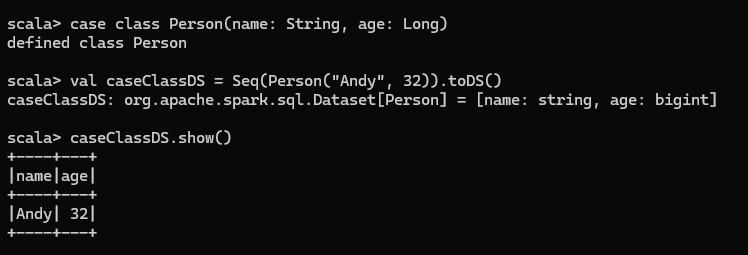
// +----+-------+

// |null|Michael|

// | 30| Andy|

// | 19| Justin|

// +----+-------+



Inferring the Schema Using Reflection

The Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame. The case class defines the schema of the table. The names of the arguments to the case class are read using reflection and become the names of the columns.

Case classes can also be nested or contain complex types such as Seqs or Arrays. This RDD can be implicitly converted to a DataFrame and then be registered as a table. Tables can be used in subsequent SQL statements.

// For implicit conversions from RDDs to DataFrames import spark.implicits.\_

// Create an RDD of Person objects from a text file, convert it to a Dataframe val peopleDF = spark.sparkContext

.textFile("examples/src/main/resources/people.txt")

.map(\_.split(","))

.map(attributes => Person(attributes(0), attributes(1).trim.toInt)) .toDF()

// Register the DataFrame as a temporary view peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by Spark

val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")

// The columns of a row in the result can be accessed by field index teenagersDF.map(teenager => "Name: " + teenager(0)).show()

// +------------+

// | value|

// +------------+

// |Name: Justin|

// +------------+

// or by field name teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()

// +------------+

// | value|

// +------------+

// |Name: Justin|

// +------------+

// No pre-defined encoders for Dataset[Map[K,V]], define explicitly

implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]

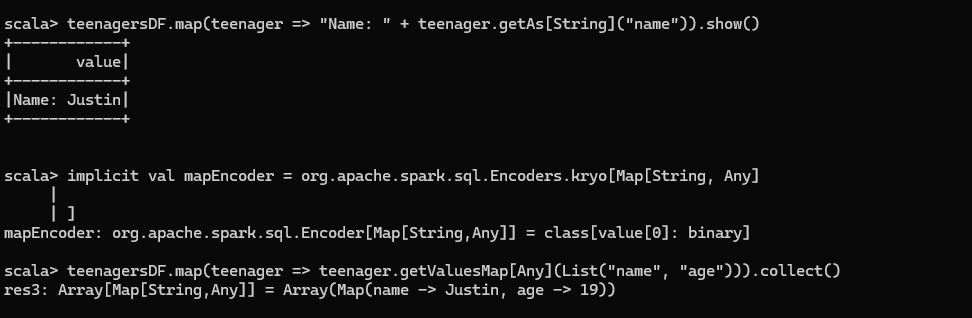
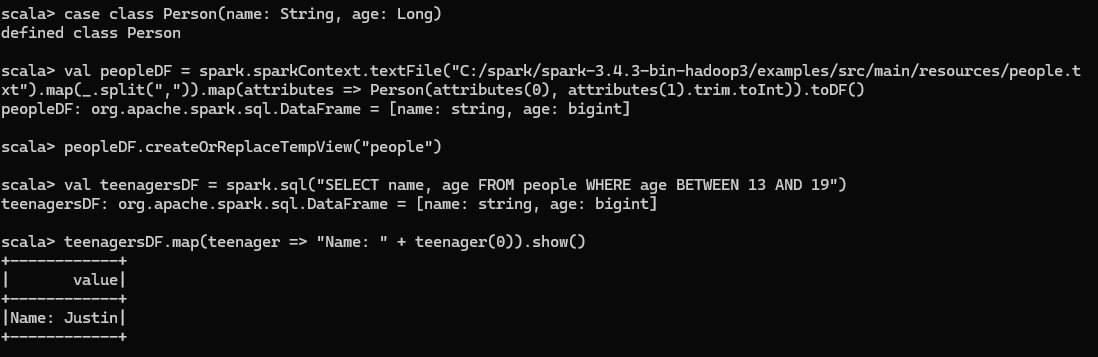
// Primitive types and case classes can be also defined as

// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()

// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]

teenagersDF.map(teenager => teenager.getValuesMap[Any](List("name", "age"))).collect()

// Array(Map("name" -> "Justin", "age" -> 19))



Programmatically Specifying the Schema

When case classes cannot be defined ahead of time (for example, the structure of records is encoded in a string, or a text dataset will be parsed and fields will be projected differently for different users), a DataFrame can be created programmatically with three steps.

Create an RDD of Rows from the original RDD;

Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.

Apply the schema to the RDD of Rows via createDataFrame method provided by SparkSession. import org.apache.spark.sql.Row import org.apache.spark.sql.types.\_

// Create an RDD

val peopleRDD = spark.sparkContext.textFile("examples/src/main/resources/people.txt")

// The schema is encoded in a string val schemaString = "name age"

// Generate the schema based on the string of schema val fields = schemaString.split(" ")

.map(fieldName => StructField(fieldName, StringType, nullable = true))

schema = StructType(fields)

// Convert records of the RDD (people) to Rows val rowRDD = peopleRDD

.map(\_.split(","))

.map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD val peopleDF = spark.createDataFrame(rowRDD, schema)

// Creates a temporary view using the DataFrame peopleDF.createOrReplaceTempView("people")

// SQL can be run over a temporary view created using DataFrames val results = spark.sql("SELECT name FROM people")

// The results of SQL queries are DataFrames and support all the normal RDD operations // The columns of a row in the result can be accessed by field index or by field name results.map(attributes => "Name: " + attributes(0)).show()

// +-------------+

// | value|

// +-------------+

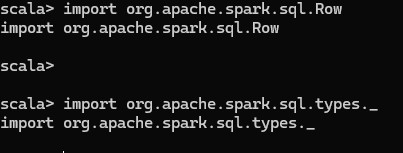
// |Name: Michael|

// | Name: Andy|

// | Name: Justin|

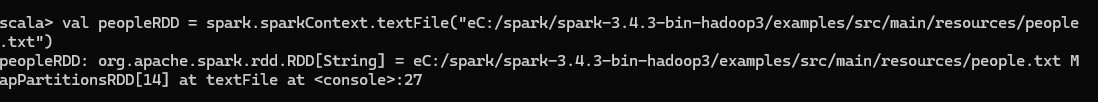
// +-------------+

import org.apache.spark.sql.Row Import org.apache.spark.sql.types.\_



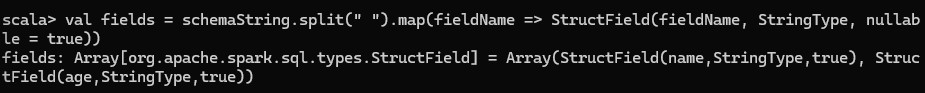
val peopleRDD =

spark.sparkContext.textFile(“C:/spark/spark-3.4.3-bin/hadoop/examples/src/main/resources/peo ple.txt”)

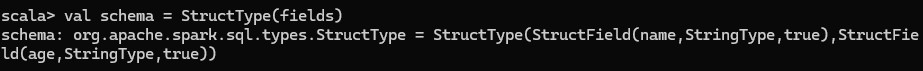


fields = schemaString.split(“ “).map(fieldName => StructField(fieldName, StringType, nullable

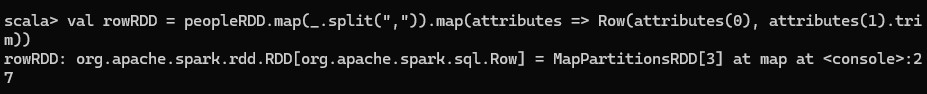
= true))



val schema = StructType(fields)



val rowRDD = peopleRDD.map(\_.split(“,”)).map(attributed => Row(attributes(0), attributes(1).trim))



val peopleDF = spark.createDataFrame(rowRDD, schema)



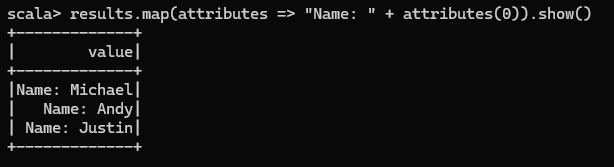
peopleDF.createOrReplaceTempView(“people”)



val results = spark.sql(“SELECT name FROM people”)



results.map(attributes => “Name: “ + attributes(0)).show()



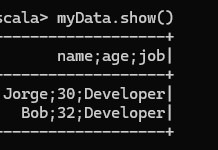
Basic Operations with csv file

myData = spark.read.format(“csv”).option(“inferSchema”,”true”).option(“header”,”true”).option(“delimeter”,”:

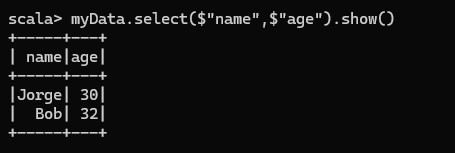
”).load(“C:/spark/spark-3.4.3-hadoop3/examples/src/main/resources/people.csv”)



myData.show()



myData.select($”name”,”$age”).show()



myData.count()



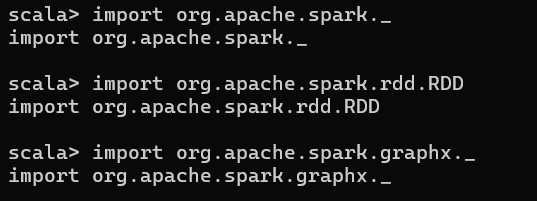
myData.count().toDouble



**Practical 03**

**Aim :** GraphX

import org.apache.spark.\_ import org.apache.spark.rdd.RDD import org.apache.spark.graphx.\_



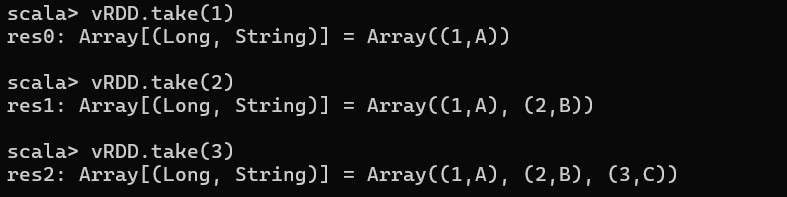
val vertices = Array((1L,("A")),(2L,("B")),(3L,("C")))



val vRDD = sc.parallelize(vertices)



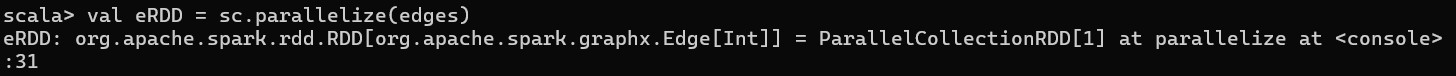
vRDD.take(1) vRDD.take(2) vRDD.take(3)



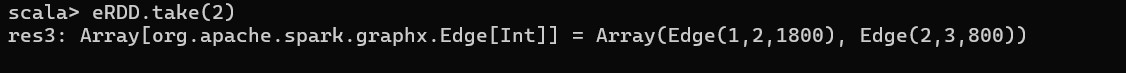
val edges = Array(Edge(1L,2L,1800),Edge(2L,3L,800),Edge(3L,1L,1400))



val eRDD = sc.parallelize(edges)



eRDD.take(2)



val nowhere = "nowhere"



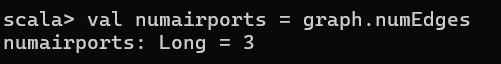
val graph = Graph(vRDD,eRDD,nowhere)



#To check number of Airports val numairports = graph.numVertices

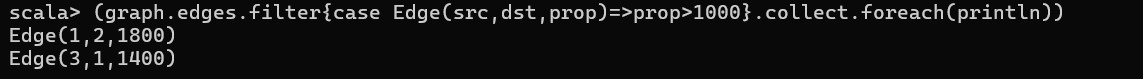


#To check routes val numairports = graph.numEdges

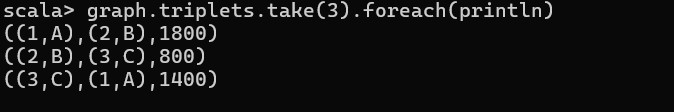


#Route having distance > 1000

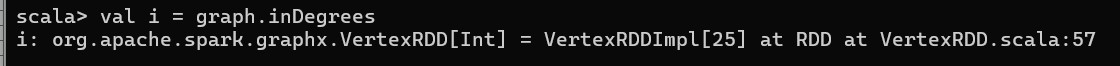
(graph.edges.filter{case Edge(src,dst,prop)=>prop>1000}.collect.foreach(println))



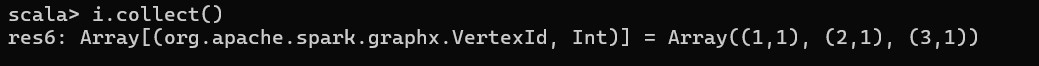
#Triplet Information graph.triplets.take(3).foreach(println)



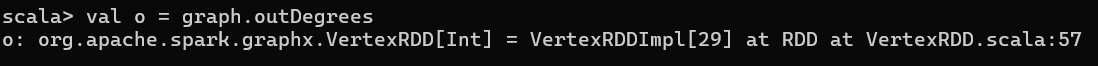
#Indegree val i = graph.inDegrees



i.collect()



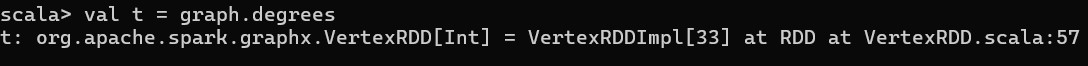
#Outdegrees val o = graph.outDegrees



o.collect()



#Total Degree val t = graph.degrees



t.collect()



**Practical 04**

**Aim :** PySpark

**Steps :**

1. Make a CSV file with data related to Name Age Salary and Experience.



1. Code

from pyspark.sql import SparkSession

# Create a Spark session spark = SparkSession.builder.appName("Read CSV").getOrCreate()

# Path to your CSV file csv\_file\_path = "Student.csv"

# Read the CSV file into a DataFrame. inferSchema tries to determine the datatype of values in the fields.

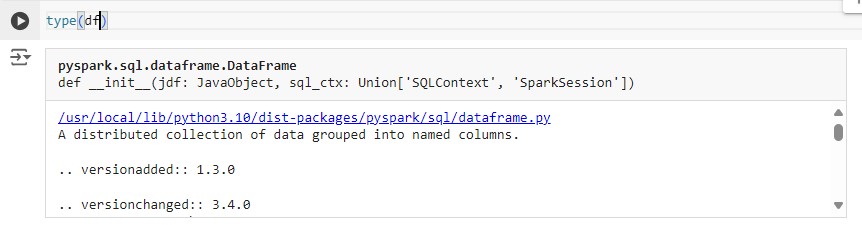
df = spark.read.csv(csv\_file\_path, header=True, inferSchema=True)

# Display the DataFrame df.show()

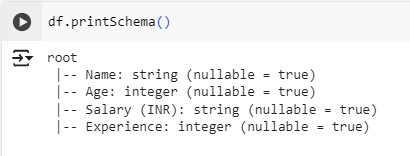
Output



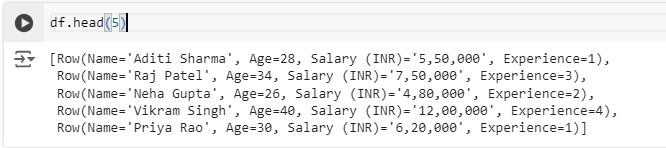
type(df)



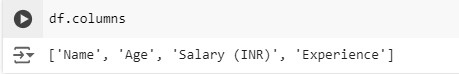
df.printSchema()



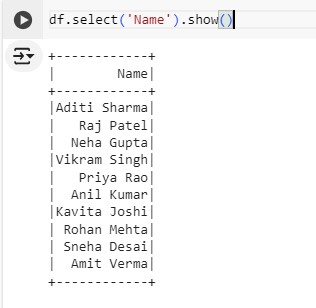
df.head(5) or df.show(5)



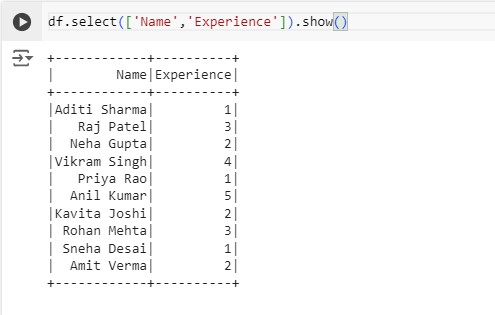
df.columns



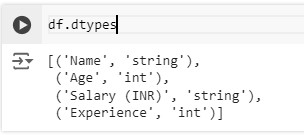
df.select('Name').show()



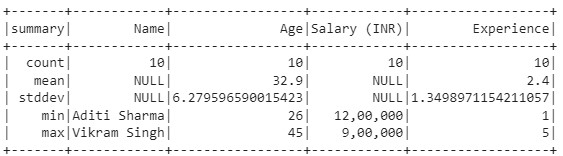
df.select(['Name','Experience']).show()



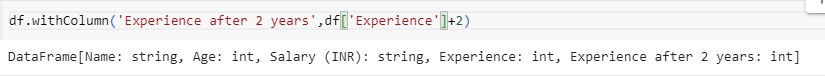
df.dtypes



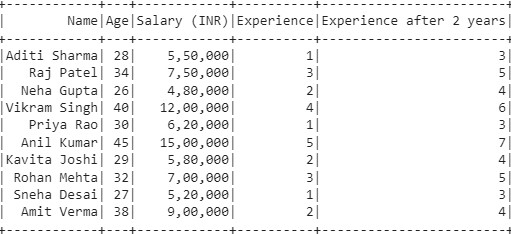
df.describe().show()



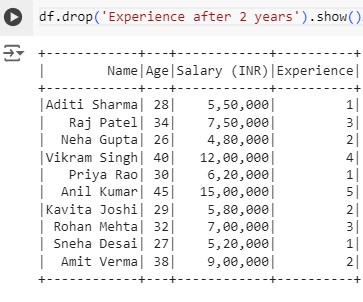
Adding columns to the dataframe df = df.withColumn('Experience after 2 years',df['Experience']+2)



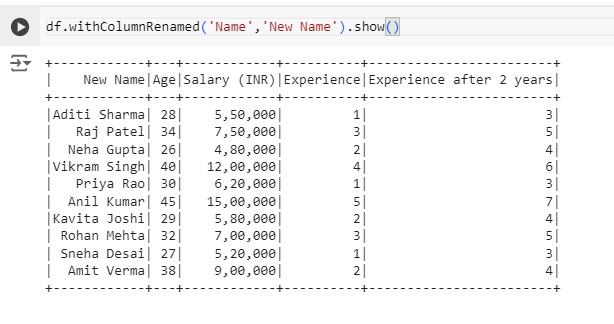
df.show()



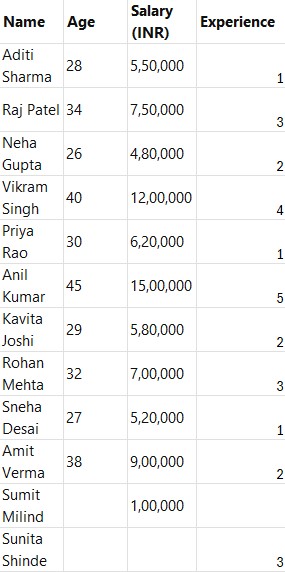
Dropping columns from dataframe df.drop('Experience after 2 years').show()

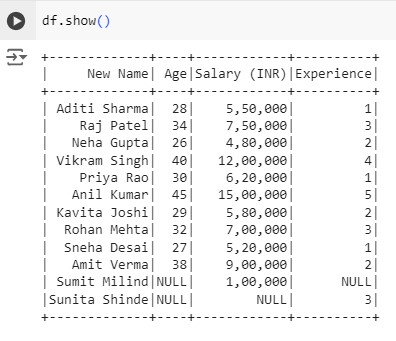


df.withColumnRenamed('Name','New Name').show()

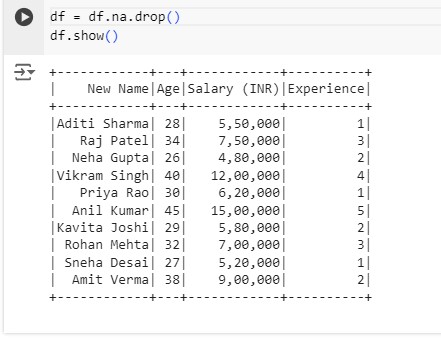


With new data that has null values



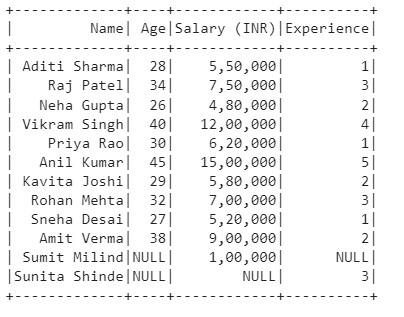


df = df.na.drop() df.show()

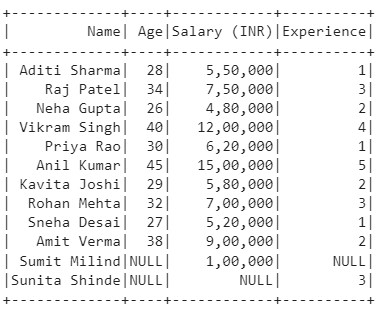


#Drops entries that have all columns as null df = df.na.drop(how="all") df.show()

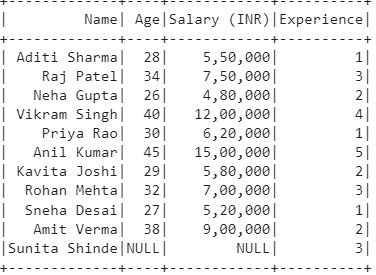
Here none will be dropped

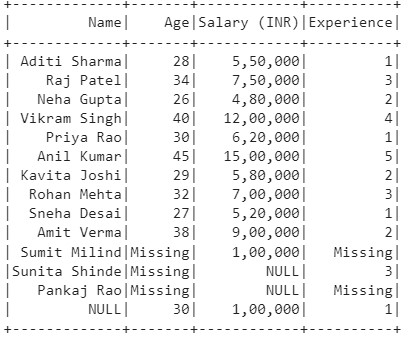


#Only drop the rows that have given threshold number of NULL COLUMNS df = df.na.drop(how="any", thresh = 2) df.show()

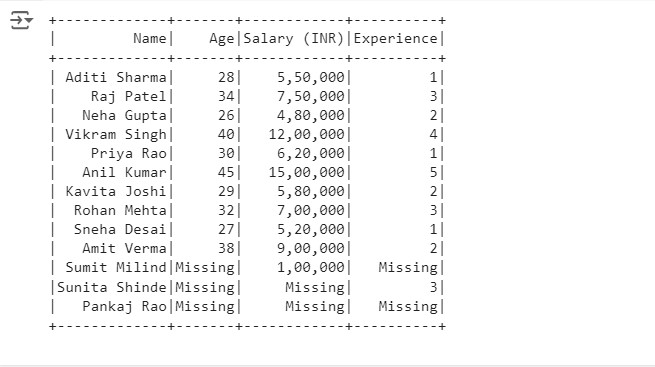


# Drops rows that has Experience as NULL df = df.na.drop(how = "any",subset = ['Experience']) df.show()

df = df.na.fill('Missing',['Age','Experience']).show()



#Fills in the said text in the string columns that are NULL df = df.na.fill("Missing") df.show()



from pyspark.ml.feature import Imputer imputer = Imputer(

inputCols = ['Age','Experience'], outputCols = ["{}\_imputed".format(c) for c in

['Age','Experience']]).setStrategy("mean") imputer.fit(df).transform(df).show()

