***Spark SQL***

Spark SQL is the module provided for processing the structured data processing

There are several ways to interact with sparksql

1. Sql / HiveQL
2. Data Frame
3. Dataset

Spark SQL can be used to read data from the existing hive installation

When running SQL from within another programming language the results are returns in the form of data frames

Data Frame is a distributed collection of data organized into named columns. It is conceptually equivalent as of a table in RDBMS.

Data Frames can be constructed from a wide array of [sources](https://spark.apache.org/docs/1.6.0/sql-programming-guide.html#data-sources) such as: structured data files, tables in Hive, external databases, or existing RDDs

Entry Point into all functionality in Spark SQL is the SQLContext class we can create the basic SQLContext as below

Val sqlcontext = new org.apache.spark.sql.SQLContext (sc)

To implicitly convert the RDD to Data Frame

Import sqlcontext.implicits.\_

Creating the Data Frame

Val input = sqlcontext.read.json (“/home/esakkipillai/datalake/cars.json”)

The above statement will create a data frame. Data Frame has set of functions which were listed as below: -

**Running SQL Queries**

The sql function on a SQLContext enables applications to run SQL queries programmatically and returns the result as a Data Frame.

Val df = sqlcontext.sql (“select \* from table”) will return the results as data frame

*Creating the Data Sets*

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

*// Data Frames can be converted to a Dataset by providing a class. Mapping will be done by name.*

val path = “/home/esakkipillai/datalake/peopledetails.json”

val input = sqlcontext.read.json (path).as [Person]

SPARK SQL Supports two different methods for converting RDD into Data Frames

1. Inferring the Schema Using Reflection
2. Programmatically Specifying the Schema

Inferring the Schema Using Reflection

Case class defines the schema of the table. The names of the arguments to the case class are read using reflection and used as column names. case class can also be nested or contain complex structures such as sequences or Array.

This RDD can be convert into Data Frame and then be registered as the temp Table. These tables can be used in subsequent SQL statements.

Val sqlcontext = new org.apache.spark.SQLContext(sc)

Import sqlcontext.implicits.\_

Case Person (name :String , age : Long)

Val input = sqlcontext.read.json(““/home/esakkipillai/datalake/peopledetails.json”).map(\_.split(“ ,”)).map( p => Person( p(0) , p(1).trim.toInt() ) .toDF()

Input.registerTempTable(“people”)

Val teen = sqlcontext.sql(“select \* from people where age >=13 AND age <=19”)

Programmatically Specifying the Schema

Data Frame can be created programmatically with three steps.

1. Create an RDD of Rows from the original RDD;
2. Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
3. Apply the schema to the RDD of Rows via create Data Frame method provided by SQLContext.

Reading the Json Datasets

Spark SQL can automatically infer the schema of a JSON dataset and load it as a DataFrame.

Val input = sqlcontext.read.json(filepath)

Reading Hive Tables

SPARKSQL supports reading and writing data to apache hive.

Hive has a large number of dependencies, it is not included in the default Spark assembly. Hive support is enabled by adding the -Phive and -Phive-thriftserver flags to Spark’s build  This command builds a new assembly jar that includes Hive. Note that this Hive assembly jar must also be present on all of the worker nodes, as they will need access to the Hive serialization and deserialization libraries (SerDes) in order to access data stored in Hive.

When we run the spark under yarn-client mode

1. datanucles jar should be present in the lib\_managed/jars directory
2. hive-site.xml should be present in the conf/ directory

these two should be available on the driver and all the executor launched by the yarn client

The convenient way to do this is adding them through the --jars option and --file option of the spark-submit command.

1. When working with hive we should create the HIVECONTEXT which inherits from the sqlcontext
2. Hivecontext adds support for finding the tables in metastore and writing the queries using hiveql
3. When not cobfigured by the hive-site.xml the context will automatically create the metastore\_db in the current directory and create the warehouse directory indicated by the hiveconf which defaults to the /user/hive/warehouse.

Note that you may need to grant write privilege on /user/hive/warehouse to the user who starts the spark application.

Val hivecontext = new org.apache.spark.sql.hive.HiveContext(sc)

Hivecontext.sql(“CREATE TABLE IF NOT EXISTS src( Key INT , Value STRING ) “ )

Hivecontext.sql(“LOAD DATA LOCAL INPATH ‘/USER/ESAKKIPILLAI/DATLAKE/SAMPLE.TXT’ INTO TABLE SRC ”)

Hivecontext.sql(“SELECT \* FROM SRC”).collect().foreach(println)

Data Frame Operations in Text File

As an example, the following creates a DataFrame based on the content of a text file. Read a text document named fruits.txt with the following content and generate a table based on the schema in the text document.

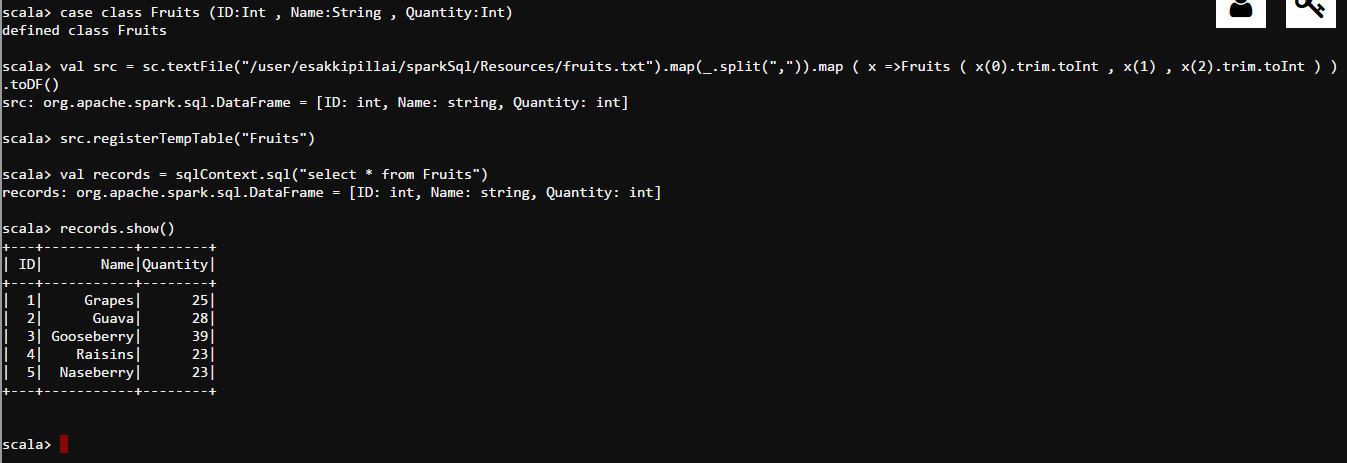
1, Grapes, 25

2, Guava, 28

3, Gooseberry, 39

4, Raisins, 23

5, Naseberry, 23



Data Frame Operations in CSV File

As an example, the following creates a Data Frame based on the content of a CSV file. Read a csv document named cars.csv with the following content and generate a table based on the schema in the csv document.

year,make,model,comment,blank

"2012","Tesla","S","No comment",

1997,Ford,E350,"Go get one now they are going fast",

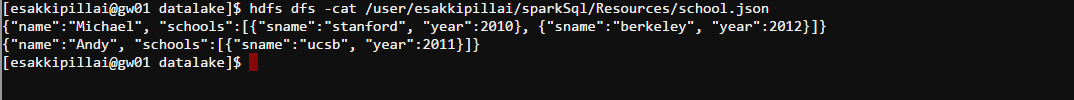
2015,Chevy,Volt

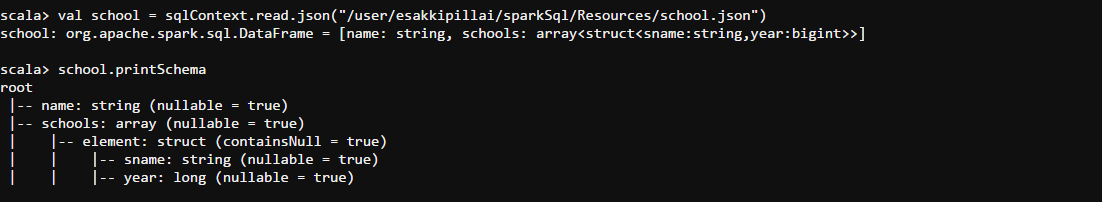
Getting Class not Found Exception

Data Frame Operations in JSON File

A sample Json Data

hdfs dfs -cat /user/esakkipillai/sparkSql/Resources/school.json





Show



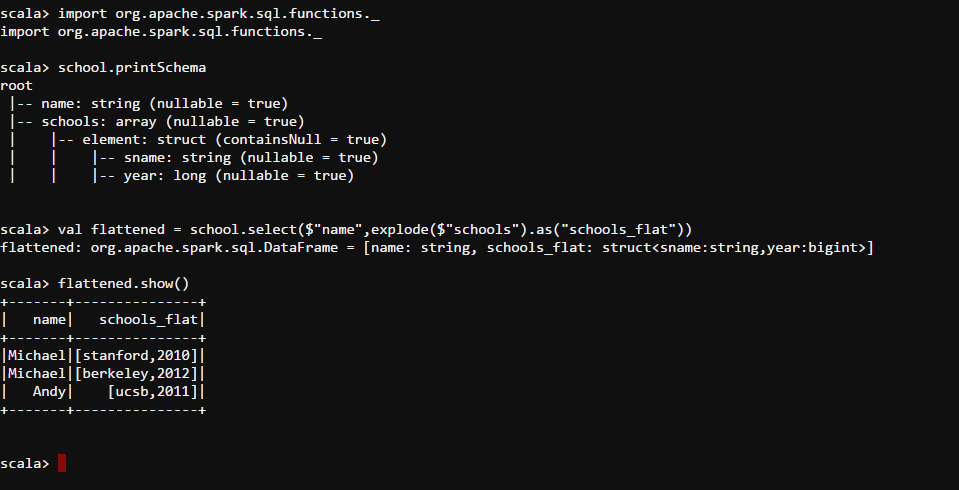
**Flatten / Explode an Array**

If your JSON object contains nested arrays of structs, how will you access the elements of an array? One way is by flattening it. For instance, in the example above, each JSON object contains a "schools" array. We can simply flatten "schools" with the *explode()*function.

import org.apache.spark.sql.functions.\_

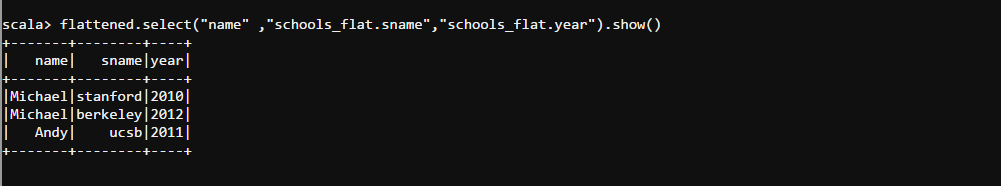
val flattened = school.select($"name",explode($"schools").as("schools\_flat"))

flattened.show()

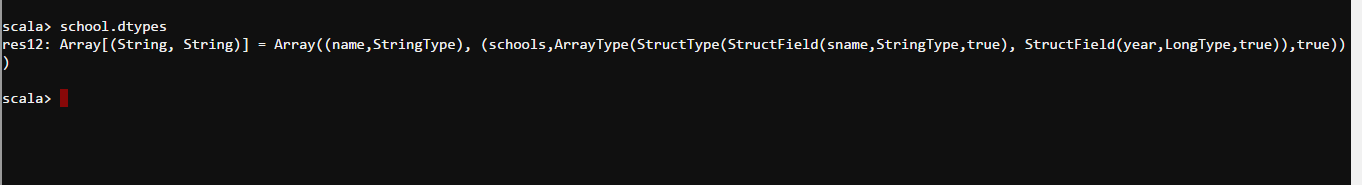


Accessing each element

flattened.select("name" ,"schools\_flat.sname","schools\_flat.year").show()



Dtypes will return the entire column name with the data types as an Array



We can create the Json data as below

