A Comprehensive Guide To Spark SQL With Examples

Shubham Sinha Follow
Jan 2, 2017 · 15 min read

Spark SQL Tutorial — Edureka

Apache Spark is a lightning-fast cluster computing framework designed for fast computation. It is one of the most successful projects in the Apache Software Foundation. With the advent of real-time processing framework in Big Data Ecosystem, companies are using Apache Spark rigorously in their solutions, and hence this has increased the demand of professionals with ***Apache Spark training***. Spark SQL is a new module in Spark which integrates relational processing with Spark's functional programming API. It supports querying data either via SQL or via the Hive Query Language.

For those of you familiar with RDBMS, Spark SQL will be an easy transition from your earlier tools where you can extend the boundaries of traditional relational data processing. Through this article, I will introduce you to this new exciting domain of Spark SQL and together we will equip ourselves to lead our organization to leverage the benefits of relational processing and call complex analytics libraries in Spark.

The following provides the storyline for the blog:

1. Why Spark SQL came into the picture?
2. Spark SQL Overview
3. Spark SQL Libraries
4. Features of Spark SQL
5. Querying using Spark SQL
6. Adding Schema to RDDs
7. RDDs as Relations
8. Caching Tables In-Memory

Why Spark SQL Came Into Picture?

Spark SQL originated as Apache Hive to run on top of Spark and is now integrated with the Spark stack. Apache Hive had certain limitations as mentioned below. Spark SQL was built to overcome these drawbacks and replace Apache Hive.

Limitations With Hive:

- Hive launches MapReduce jobs internally for executing the ad-hoc queries. MapReduce lags in the performance when it comes to the analysis of medium-sized datasets (10 to 200 GB).
- Hive has no resume capability. This means that if the processing dies in the middle of a workflow, you cannot resume from where it got stuck.
- Hive cannot drop encrypted databases in cascade when trash is enabled and leads to an execution error. To overcome this, users have to use Purge option to skip trash instead of drop.

These drawbacks gave way to the birth of Spark SQL.

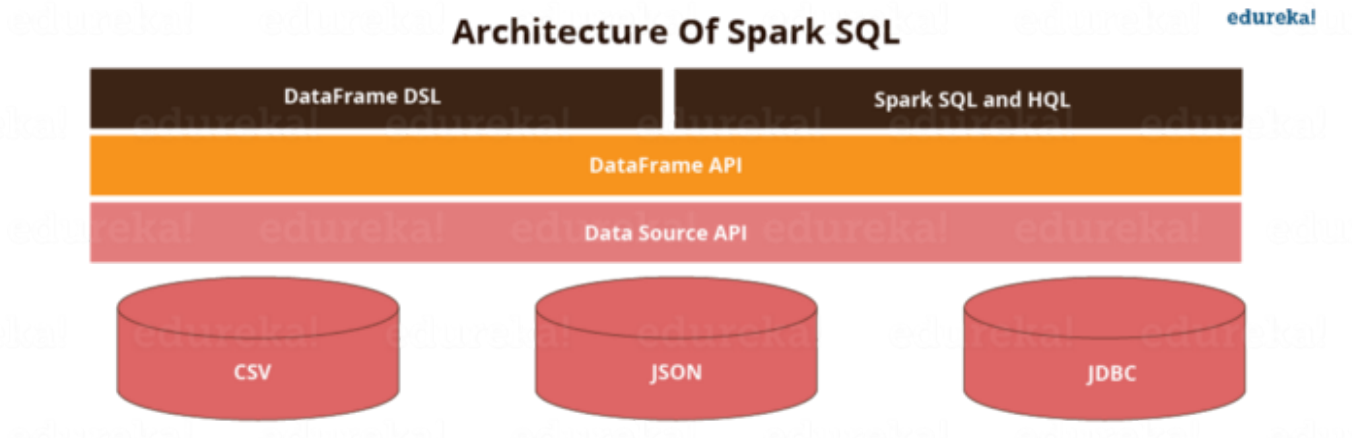
Spark SQL Overview

Spark SQL integrates relational processing with Spark's functional programming. It provides support for various data sources and makes it possible to weave SQL queries with code transformations thus resulting in a very powerful tool.

Let us explore, what Spark SQL has to offer. Spark SQL blurs the line between RDD and relational table. It offers much tighter integration between relational and procedural processing, through declarative DataFrame APIs which integrates with Spark code. It also provides higher optimization. DataFrame API and Datasets API are the ways to interact with Spark SQL.

With Spark SQL, Apache Spark is accessible to more users and improves optimization for the current ones. Spark SQL provides DataFrame APIs which perform relational operations on both external data sources and Spark's built-in distributed collections. It introduces an extensible optimizer called Catalyst as it helps in supporting a wide range of data sources and algorithms in Big-data.

Spark runs on both Windows and UNIX-like systems (e.g. Linux, Microsoft, Mac OS). It is easy to run locally on one machine — all you need is to have java installed on your system PATH, or the JAVA_HOME environment variable pointing to a Java installation.



Spark SQL Libraries

Spark SQL has the following four libraries which are used to interact with relational and procedural processing:

Data Source API (Application Programming Interface):

This is a universal API for loading and storing structured data.

- It has built in support for Hive, Avro, JSON, JDBC, Parquet, etc.
- Supports third-party integration through Spark packages.
- Support for smart sources.
- It is a Data Abstraction and Domain Specific Language (DSL) applicable on the structure and semi-structured data.

DataFrame API:

- DataFrame API is a distributed collection of data in the form of named column and row.
- It is lazily evaluated like Apache Spark Transformations and can be accessed through SQL Context and Hive Context.
- It processes the data in the size of Kilobytes to Petabytes on a single-node cluster to multi-node clusters.
- Supports different data formats (Avro, CSV, Elastic Search, and Cassandra) and storage systems (HDFS, HIVE Tables, MySQL, etc.).
- Can be easily integrated with all Big Data tools and frameworks via Spark-Core.
- Provides API for Python, Java, Scala, and R Programming.
- A DataFrame is a distributed collection of data organized into a named column. It is equivalent to a relational table in SQL used for storing data into tables.

SQL Interpreter And Optimizer:

SQL Interpreter and Optimizer is based on functional programming constructed in Scala.

- It is the newest and most technically evolved component of SparkSQL.
- It provides a general framework for transforming trees, which is used to perform analysis/evaluation, optimization, planning, and run time code spawning.
- This supports cost-based optimization (run time and resource utilization is termed as cost) and rule-based optimization, making queries run much faster than their

RDD (Resilient Distributed Dataset) counterparts.

e.g. Catalyst is a modular library which is made as a rule-based system. Each rule in framework focuses on the distinct optimization.

SQL Service:

SQL Service is the entry point for working along structured data in Spark. It allows the creation of DataFrame objects as well as the execution of SQL queries.

Features Of Spark SQL

The following are the features of Spark SQL:

Integration With Spark

Spark SQL queries are integrated with Spark programs. Spark SQL allows us to query structured data inside Spark programs, using SQL or a DataFrame API which can be used in Java, Scala, Python and R. To run streaming computation, developers simply write a batch computation against the DataFrame / Dataset API, and Spark automatically increments the computation to run it in a streaming fashion. This powerful design means that developers don't have to manually manage state, failures, or keeping the application in sync with batch jobs. Instead, the streaming job always gives the same answer as a batch job on the same data.

Uniform Data Access

DataFrames and SQL support a common way to access a variety of data sources, like Hive, Avro, Parquet, ORC, JSON, and JDBC. This joins the data across these sources. This is very helpful to accommodate all the existing users into Spark SQL.

Hive Compatibility

Spark SQL runs unmodified Hive queries on current data. It rewrites the Hive front-end and meta store, allowing full compatibility with current Hive data, queries, and UDFs.

Standard Connectivity

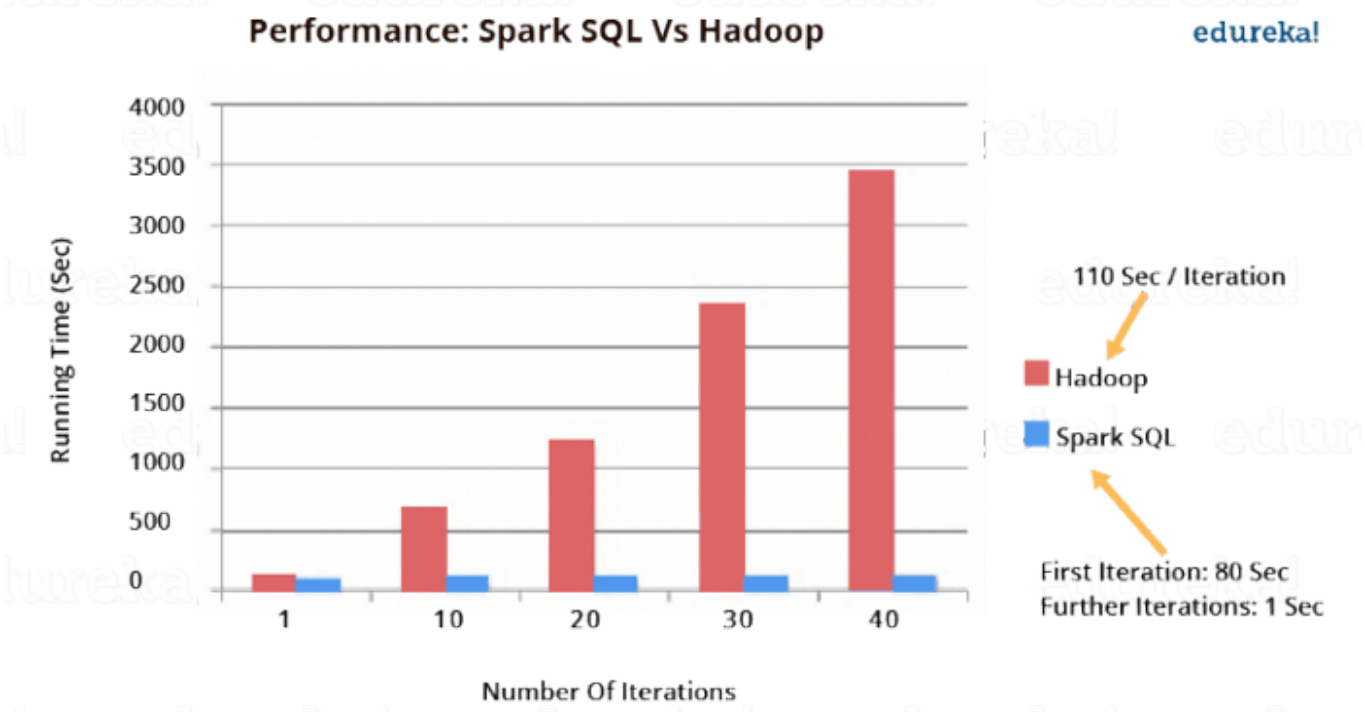
The connection is through JDBC or ODBC. JDBC and ODBC are the industry norms for connectivity for business intelligence tools.

Performance And Scalability

Spark SQL incorporates a cost-based optimizer, code generation, and columnar storage to make queries agile alongside computing thousands of nodes using the Spark engine,

which provides full mid-query fault tolerance. The interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimization. Spark SQL can directly read from multiple sources (files, HDFS, JSON/Parquet files, existing RDDs, Hive, etc.). It ensures fast execution of existing Hive queries.

The image below depicts the performance of Spark SQL when compared to Hadoop. Spark SQL executes up to 100x times faster than Hadoop.



User-Defined Functions

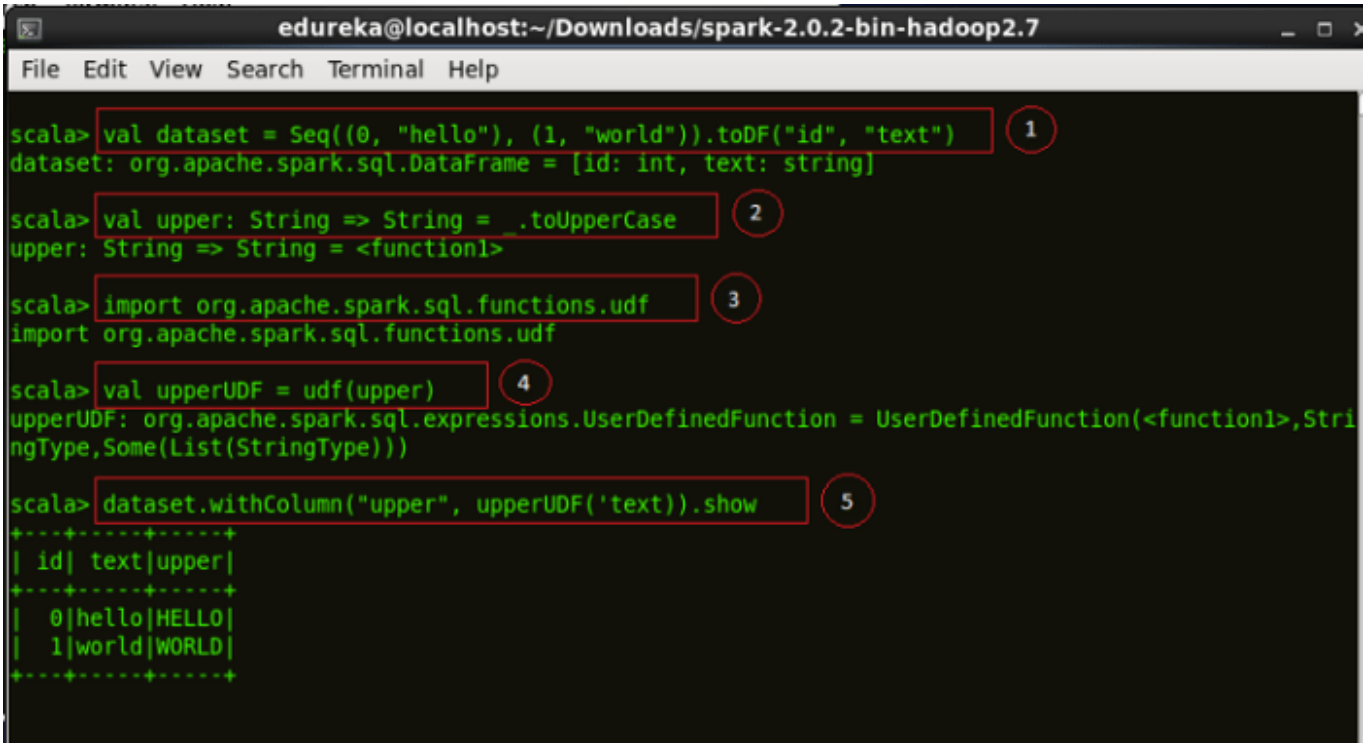
Spark SQL has language integrated User-Defined Functions (UDFs). UDF is a feature of Spark SQL to define new Column-based functions that extend the vocabulary of Spark SQL's DSL for transforming Datasets. UDFs are black boxes in their execution.

The example below defines a UDF to convert a given text to upper case.

Code explanation:

1. Creating a dataset "hello world"
2. Defining a function 'upper' which converts a string into upper case.
3. We now import the 'udf' package into Spark.
4. Defining our UDF, 'upperUDF' and importing our function 'upper'.
5. Displaying the results of our User Defined Function in a new column 'upper'.

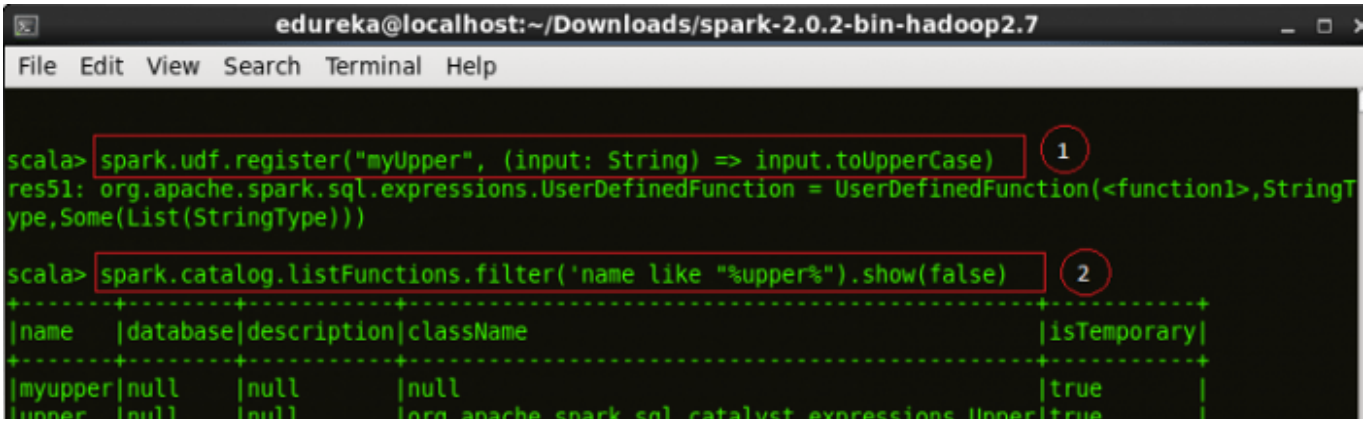
```
val dataset = Seq((0, "hello"),(1, "world")).toDF("id","text")
val upper: String =&amp;amp;amp;amp;gt; String =_.toUpperCase
import org.apache.spark.sql.functions.udf
val upperUDF = udf(upper)
dataset.withColumn("upper", upperUDF('text')).show
```



Code explanation:

1. We now register our function as ‘myUpper’
2. Cataloging our UDF among the other functions.

```
spark.udf.register("myUpper", (input:String) =&amp;amp;amp;amp;gt;
input.toUpperCase)
spark.catalog.listFunctions.filter('name like "%upper%").show(false)
```

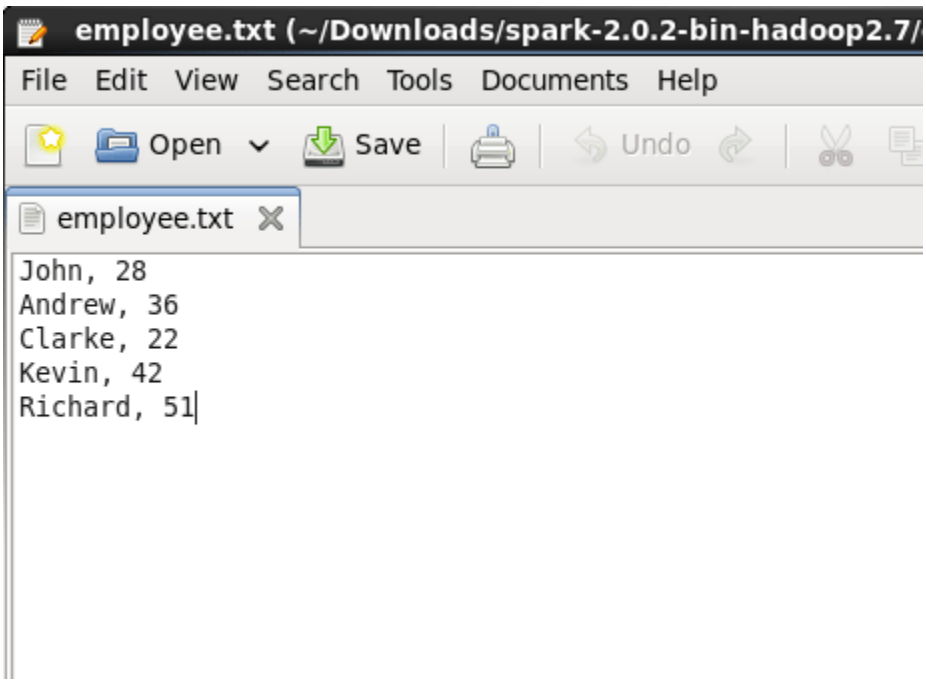


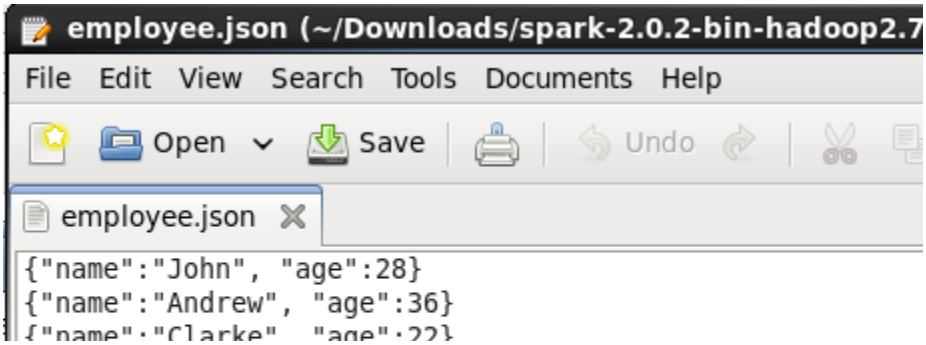
Querying Using Spark SQL

We will now start querying using Spark SQL. Note that the actual SQL queries are similar to the ones used in popular SQL clients.

Starting the Spark Shell. Go to the Spark directory and execute ./bin/spark-shell in the terminal to being the Spark Shell.

For the querying examples shown in the blog, we will be using two files, 'employee.txt' and 'employee.json'. The images below show the content of both the files. Both these files are stored at 'examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala' inside the folder containing the Spark installation (~/Downloads/spark-2.0.2-bin-hadoop2.7). So, all of you who are executing the queries, place them in this directory or set the path to your files in the lines of code below.



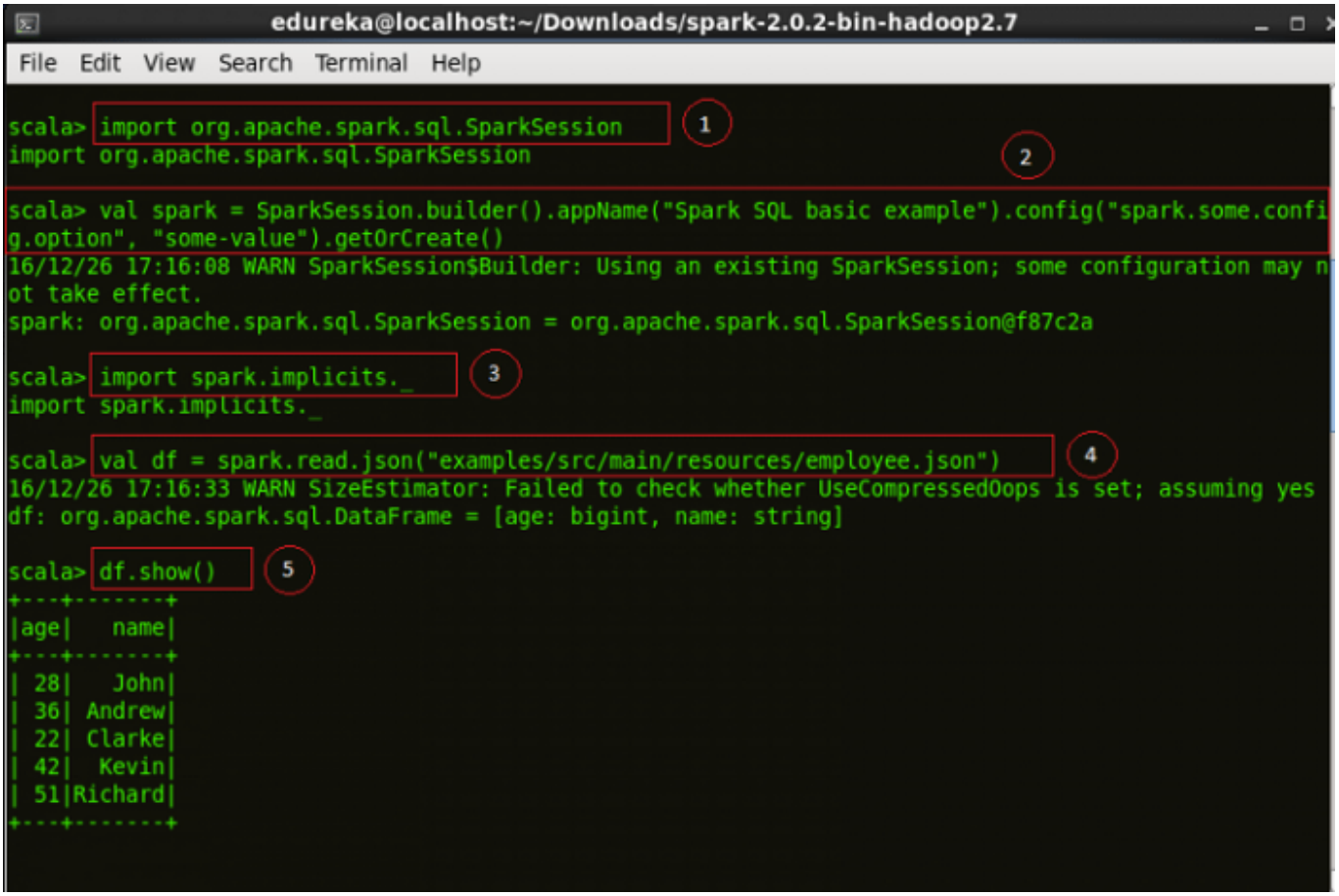


```
{"name":"Kevin", "age":42}
{"name":"Richard", "age":51}
```

Code explanation:

1. We first import a Spark Session into Apache Spark.
2. Creating a Spark Session 'spark' using the 'builder()' function.
3. Importing the Implicts class into our 'spark' Session.
4. We now create a DataFrame 'df' and import data from the 'employee.json' file.
5. Displaying the DataFrame 'df'. The result is a table of 5 rows of ages and names from our 'employee.json' file.

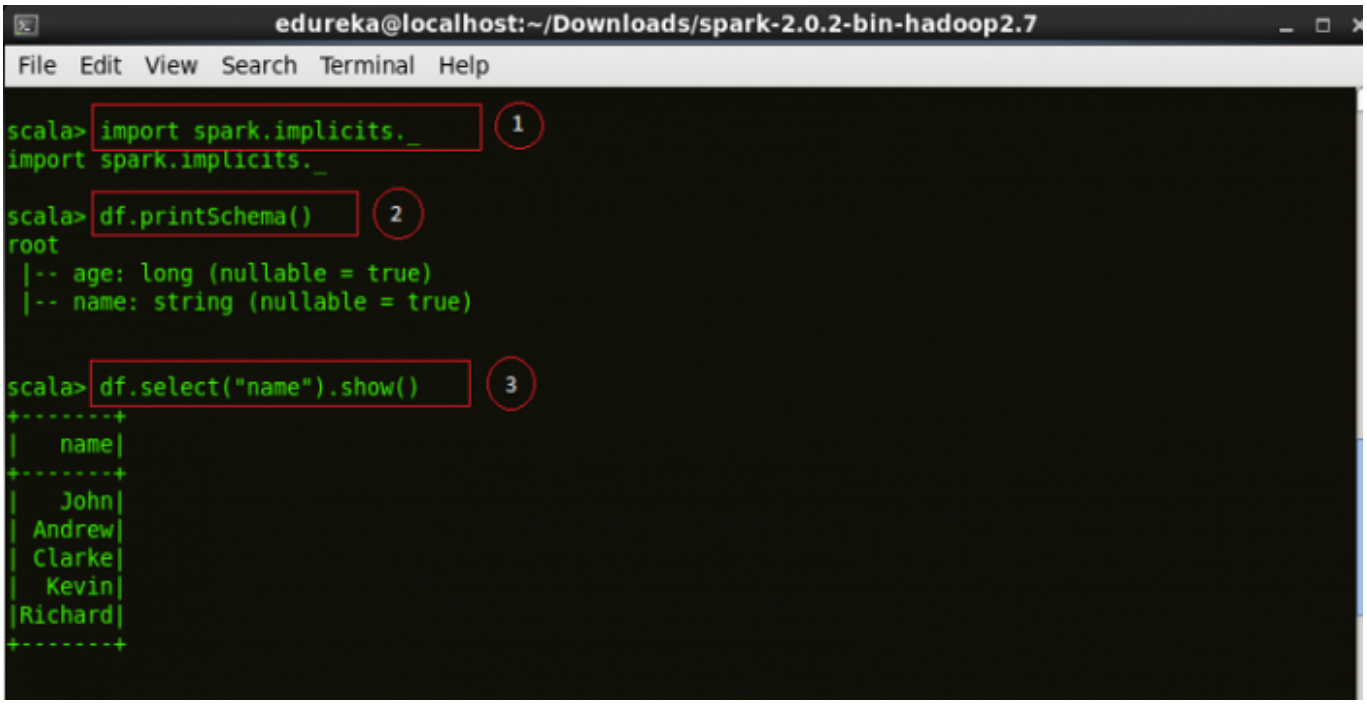
```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("Spark SQL basic
example").config("spark.some.config.option", "some-
value").getOrCreate()
import spark.implicits._
val df = spark.read.json("examples/src/main/resources/employee.json")
df.show()
```



Code explanation:

1. Importing the Implicts class into our ‘spark’ Session.
2. Printing the schema of our ‘df’ DataFrame.
3. Displaying the names of all our records from ‘df’ DataFrame.

```
import spark.implicits._
df.printSchema()
df.select("name").show()
```



Code explanation:

1. Displaying the DataFrame after incrementing everyone’s age by two years.
2. We filter all the employees above age 30 and display the result.

```
df.select($"name", $"age" + 2).show()
df.filter($"age" &amp;amp;amp;amp;gt; 30).show()
```

```
| Clarke|       24|
|  Kevin|       44|
|Richard|       53|
+-------+---------+


scala> df.filter($"age" > 30).show()      (2)
+---+-------+
|age|   name|
+---+-------+
| 36| Andrew|
| 42|  Kevin|
| 51|Richard|
+---+-------+
```

Code explanation:

1. Counting the number of people with the same ages. We use the 'groupBy' function for the same.
2. Creating a temporary view 'employee' of our 'df' DataFrame.
3. Perform a 'select' operation on our 'employee' view to display the table into 'sqlDF'.
4. Displaying the results of 'sqlDF'.

```
df.groupBy("age").count().show()
df.createOrReplaceTempView("employee")
val sqlDF = spark.sql("SELECT * FROM employee")
sqlDF.show()
```

```
edureka@localhost:~/Downloads/spark-2.0.2-bin-hadoop2.7
File  Edit  View  Search  Terminal  Help

scala> df.groupBy("age").count().show()      (1)
+---+-----+
|age|count|
+---+-----+
| 22|    1|
| 51|    1|
| 28|    1|
| 36|    1|
| 42|    1|
+---+-----+


scala> df.createOrReplaceTempView("employee")      (2)

scala> val sqlDF = spark.sql("SELECT * FROM employee")      (3)
sqlDF: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

scala> sqlDF.show()      (4)
+---+-------+
|age|   name|
+---+-------+
| 28|   John|
| 36| Andrew|
| 22| Clarke|
| 42|  Kevin|
| 51|Richard|
```

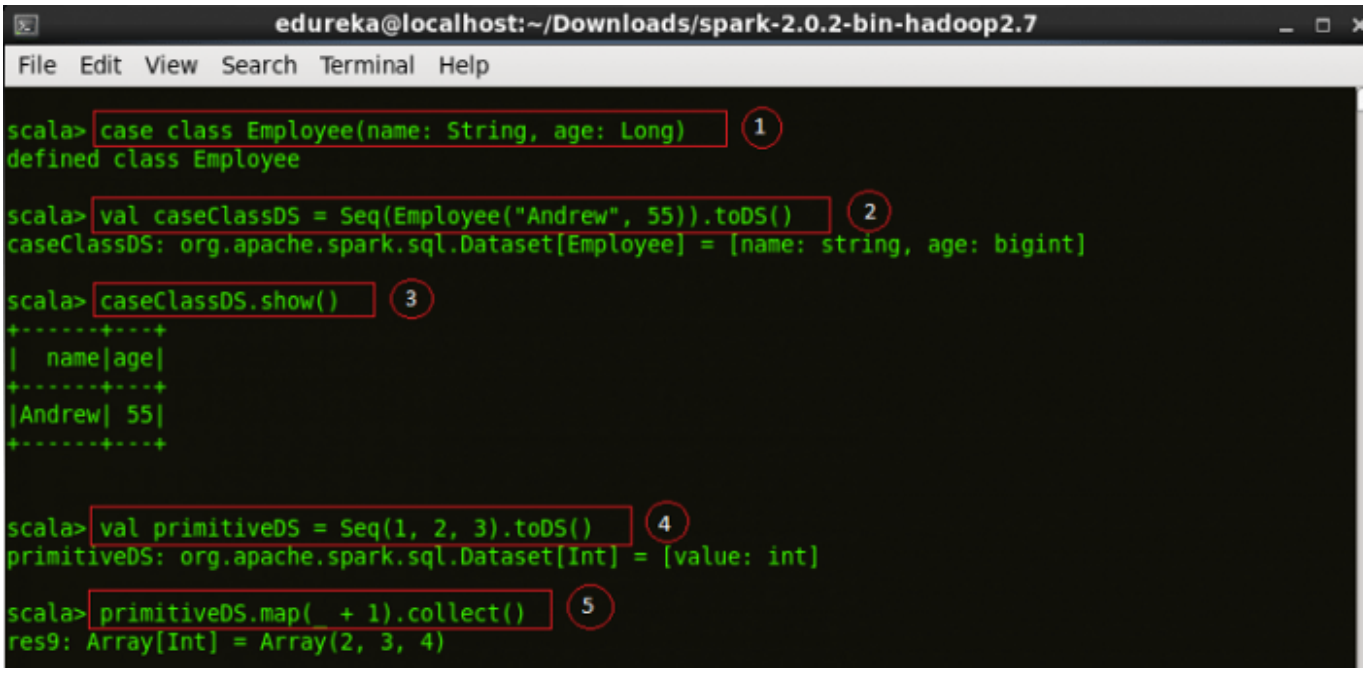
Creating Datasets

After understanding DataFrames, let us now move on to Dataset API. The below code creates a Dataset class in SparkSQL.

Code explanation:

1. Creating a class 'Employee' to store the name and age of an employee.
2. Assigning a Dataset 'caseClassDS' to store the record of Andrew.
3. Displaying the Dataset 'caseClassDS'.
4. Creating a primitive Dataset to demonstrate the mapping of DataFrames into Datasets.
5. Assigning the above sequence into an array.

```
case class Employee(name: String, age: Long)
val caseClassDS = Seq(Employee("Andrew", 55)).toDS()
caseClassDS.show()
val primitiveDS = Seq(1, 2, 3).toDS
()primitiveDS.map(_ + 1).collect()
```

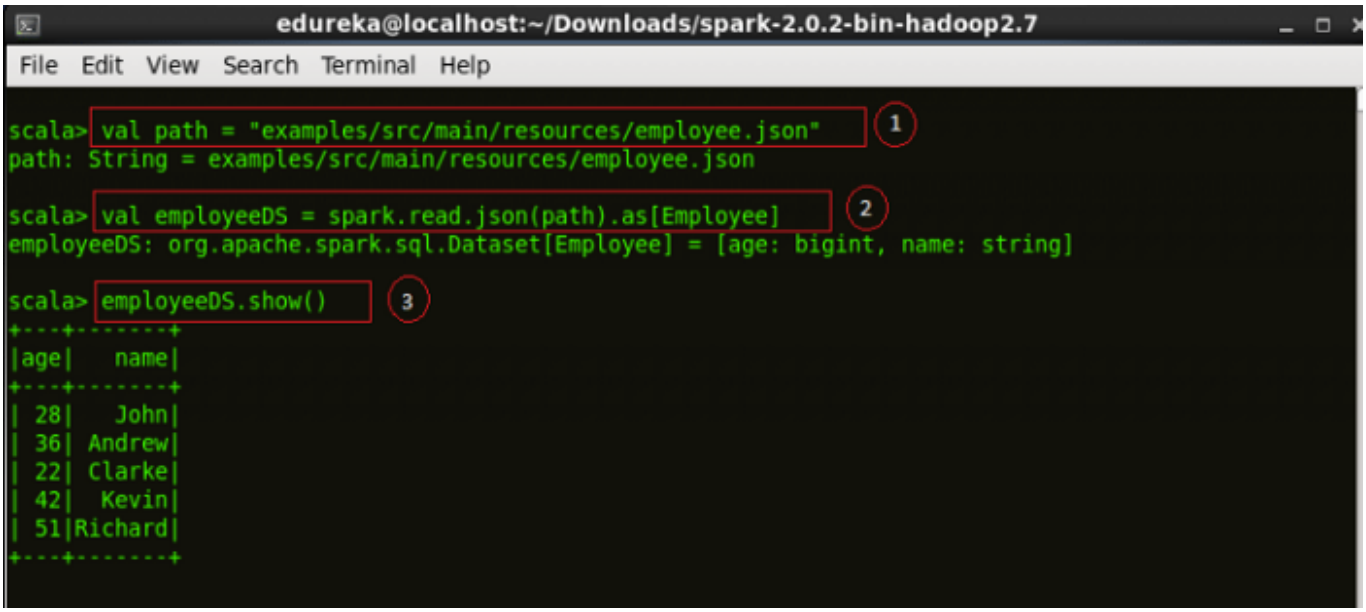


Code explanation:

1. Setting the path to our JSON file 'employee.json'.

2. Creating a Dataset and from the file.
3. Displaying the contents of 'employeeDS' Dataset.

```
val path = "examples/src/main/resources/employee.json"
val employeeDS = spark.read.json(path).as[Employee]
employeeDS.show()
```



Adding Schema To RDDs

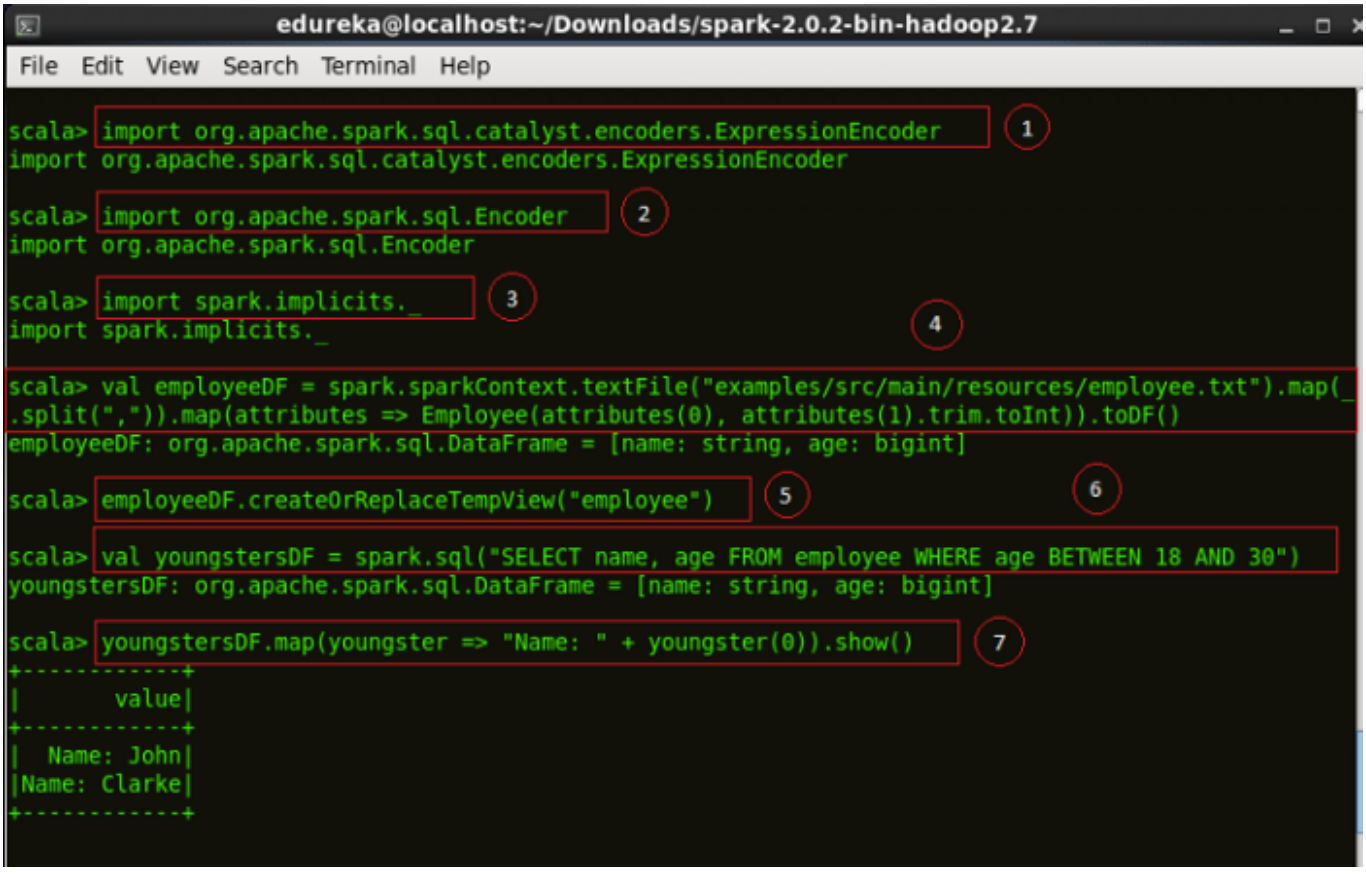
Spark introduces the concept of an RDD (Resilient Distributed Dataset), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel. An RDD can contain any type of object and is created by loading an external dataset or distributing a collection from the driver program.

Schema RDD is a RDD where you can run SQL on. It is more than SQL. It is a unified interface for structured data.

Code explanation:
1. Importing Expression Encoder for RDDs. RDDs are similar to Datasets but use encoders for serialization.
2. Importing Encoder library into the shell.
3. Importing the Implicts class into our 'spark' Session.
4. Creating an 'employeeDF' DataFrame from 'employee.txt' and mapping the columns based on the delimiter comma ',' into a temporary view 'employee'.
5. Creating the temporary view 'employee'.

6. Defining a DataFrame ‘youngstersDF’ which will contain all the employees between the ages of 18 and 30.
7. Mapping the names from the RDD into ‘youngstersDF’ to display the names of youngsters.

```
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.Encoder
import spark.implicits._
val employeeDF =
spark.sparkContext.textFile("examples/src/main/resources/employee.txt
").map(_.split(",")).map(attributes =&amp;amp;amp;amp;gt;
Employee(attributes(0), attributes(1).trim.toInt)).toDF()
employeeDF.createOrReplaceTempView("employee")
val youngstersDF = spark.sql("SELECT name, age FROM employee WHERE
age BETWEEN 18 AND 30")
youngstersDF.map(youngster =&amp;amp;amp;amp;gt; "Name: " +
youngster(0)).show()
```

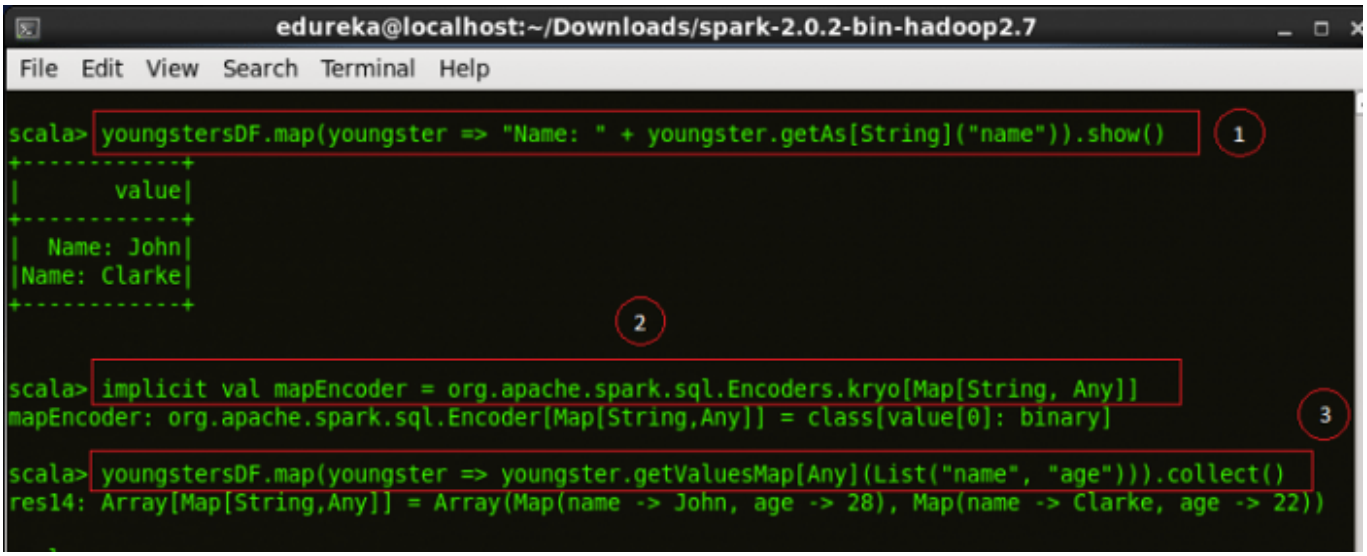


Code explanation:

1. Converting the mapped names into string for transformations.
2. Using the mapEncoder from Implicits class to map the names to the ages.

3. Mapping the names to the ages of our 'youngstersDF' DataFrame. The result is an array with names mapped to their respective ages.

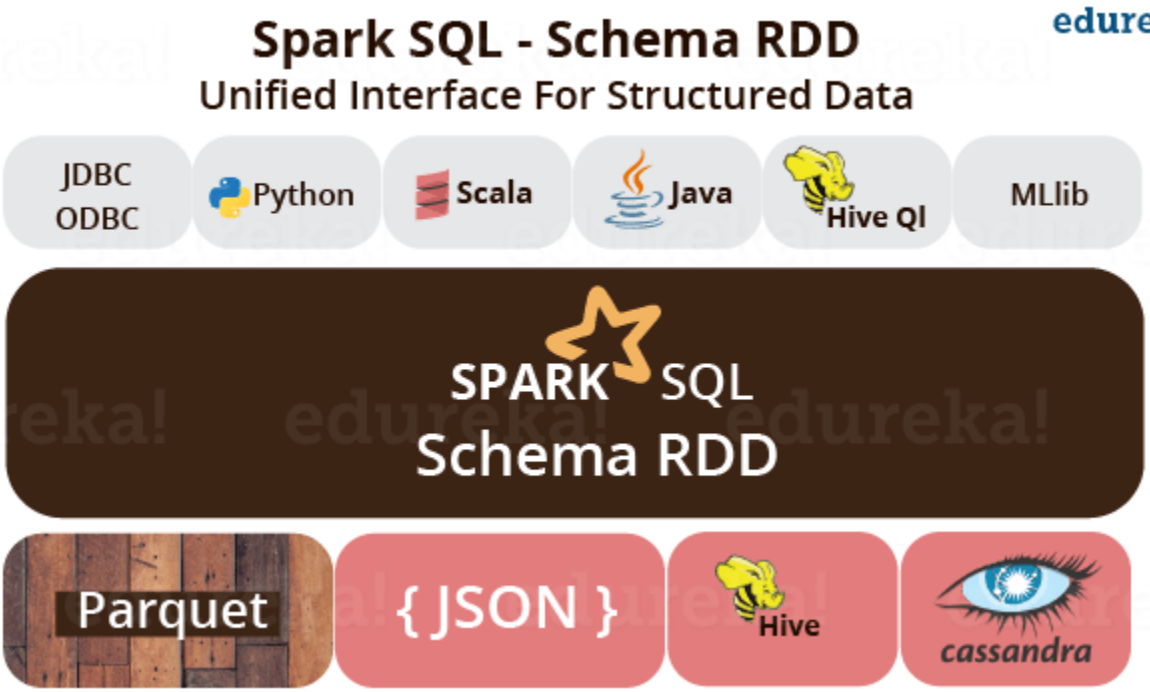
```
youngstersDF.map(youngster =&amp;amp;amp;amp;gt; "Name: " +
youngster.getAs[String]("name")).show()
implicit val mapEncoder =
org.apache.spark.sql.Encoders.kryo[Map[String, Any]]
youngstersDF.map(youngster =&amp;amp;amp;gt;
youngster.getValuesMap[Any](List("name", "age"))).collect()
```



RDDs support two types of operations:

- Transformations: These are the operations (such as map, filter, join, union, and so on) performed on an RDD which yield a new RDD containing the result.
- Actions: These are operations (such as reduce, count, first, and so on) that return a value after running a computation on an RDD.

Transformations in Spark are "lazy", meaning that they do not compute their results right away. Instead, they just "remember" the operation to be performed and the dataset (e.g., file) to which the operation is to be performed. The transformations are computed only when an action is called and the result is returned to the driver program and stored as Directed Acyclic Graphs (DAG). This design enables Spark to run more efficiently. For example, if a big file was transformed in various ways and passed to the first action, Spark would only process and return the result for the first line, rather than do the work for the entire file.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory using the persist or cache method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it.

RDDs As Relations

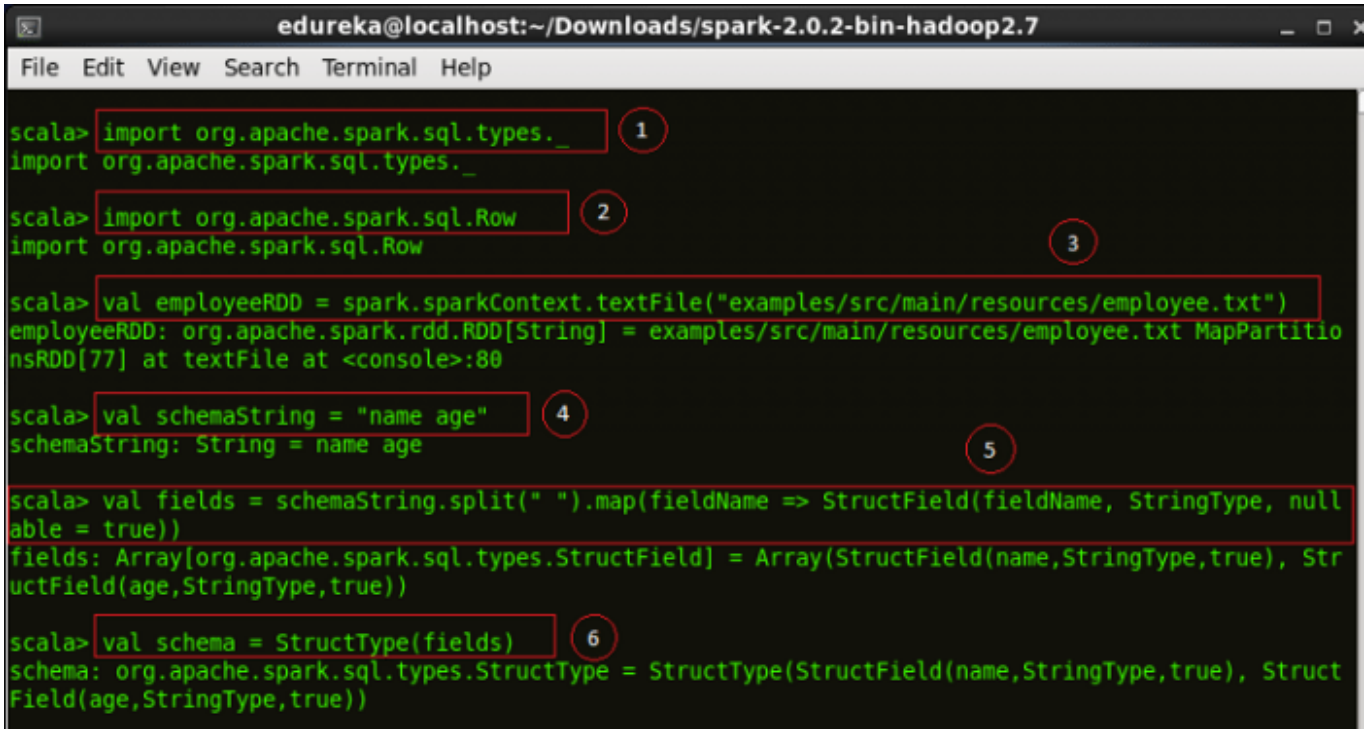
Resilient Distributed Datasets (RDDs) are distributed memory abstraction which lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs can be created from any data source. Eg: Scala collection, local file system, Hadoop, Amazon S3, HBase Table, etc.

Specifying Schema

Code explanation:

1. Importing the ‘types’ class into the Spark Shell.
2. Importing ‘Row’ class into the Spark Shell. Row is used in mapping RDD Schema.
3. Creating an RDD ’employeeRDD’ from the text file ’employee.txt’.
4. Defining the schema as “name age”. This is used to map the columns of the RDD.
5. Defining ‘fields’ RDD which will be the output after mapping the ’employeeRDD’ to the schema ‘schemaString’.
6. Obtaining the type of ‘fields’ RDD into ‘schema’.

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
val employeeRDD =
spark.sparkContext.textFile(“examples/src/main/resources/employee.txt
”)
val schemaString = “name age”
val fields = schemaString.split(“ ”).map(fieldName
=&amp;amp;amp;amp;gt; StructField(fieldName, StringType, nullable =
true))
val schema = StructType(fields)
```

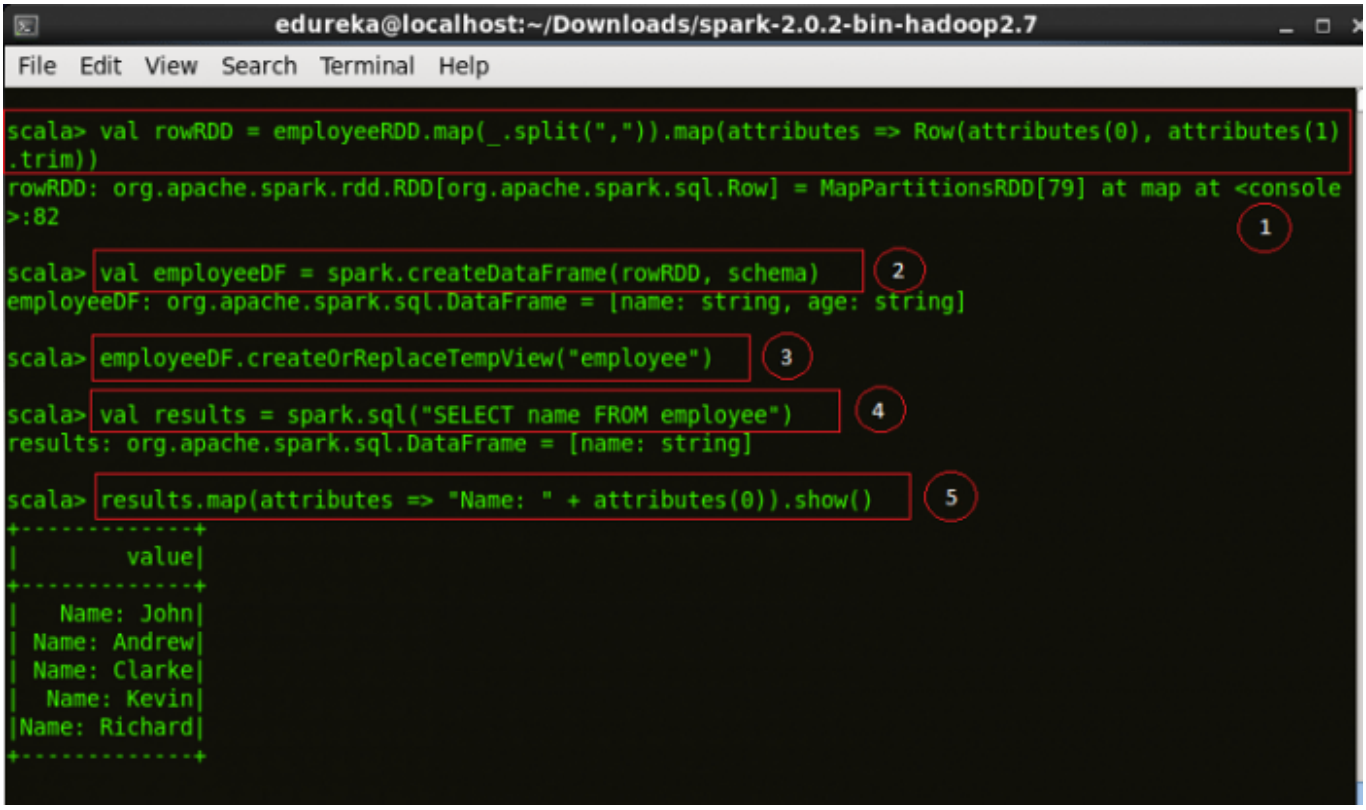


Code explanation:

1. We now create a RDD called ‘rowRDD’ and transform the ’employeeRDD’ using the ‘map’ function into ‘rowRDD’.
2. We define a DataFrame ’employeeDF’ and store the RDD schema into it.
3. Creating a temporary view of ’employeeDF’ into ’employee’.
4. Performing the SQL operation on ’employee’ to display the contents of employee.
5. Displaying the names of the previous operation from the ’employee’ view.

```
val rowRDD = employeeRDD.map(_.split(",")).map(attributes
=&amp;amp;amp;amp;gt; Row(attributes(0), attributes(1).trim))
val employeeDF = spark.createDataFrame(rowRDD, schema)
employeeDF.createOrReplaceTempView("employee")
val results = spark.sql("SELECT name FROM employee")
```

```
results.map(attributes =&amp;amp;amp;amp;gt; "Name: " +
attributes(0)).show()
```



Even though RDDs are defined, they don't contain any data. The computation to create the data in an RDD is only done when the data is referenced. e.g. Caching results or writing out the RDD.

Caching Tables In-Memory

Spark SQL caches tables using an in-memory columnar format:

1. Scan only required columns
2. Fewer allocated objects
3. Automatically selects the best comparison

Loading Data Programmatically

The below code will read employee.json file and create a DataFrame. We will then use it to create a Parquet file.

Code explanation:

1. Importing Implicits class into the shell.

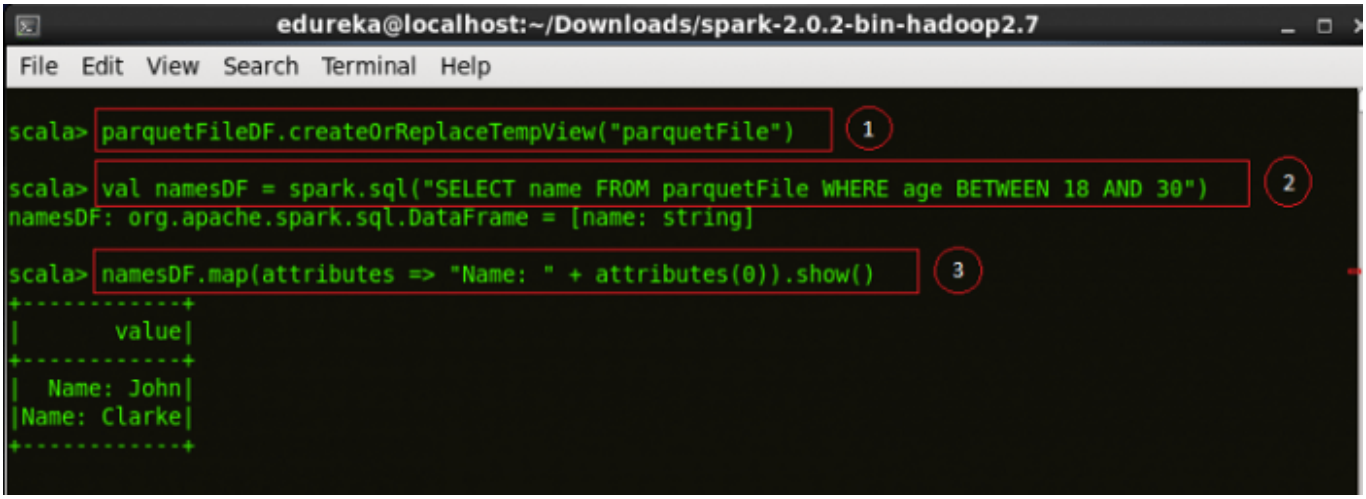
2. Creating an ’employeeDF’ DataFrame from our ’employee.json’ file.

```
import spark.implicits._
val employeeDF =
spark.read.json(“examples/src/main/resources/employee.json”)
```

Code explanation:

1. Creating a ‘parquetFile’ temporary view of our DataFrame.
2. Selecting the names of people between the ages of 18 and 30 from our Parquet file.
3. Displaying the result of the Spark SQL operation.

```
employeeDF.write.parquet("employee.parquet")
val parquetFileDF = spark.read.parquet("employee.parquet")
parquetFileDF.createOrReplaceTempView("parquetFile")
val namesDF = spark.sql("SELECT name FROM parquetFile WHERE age
BETWEEN 18 AND 30")
namesDF.map(attributes =&amp;amp;amp;amp;gt; "Name: " +
attributes(0)).show()
```



JSON Datasets

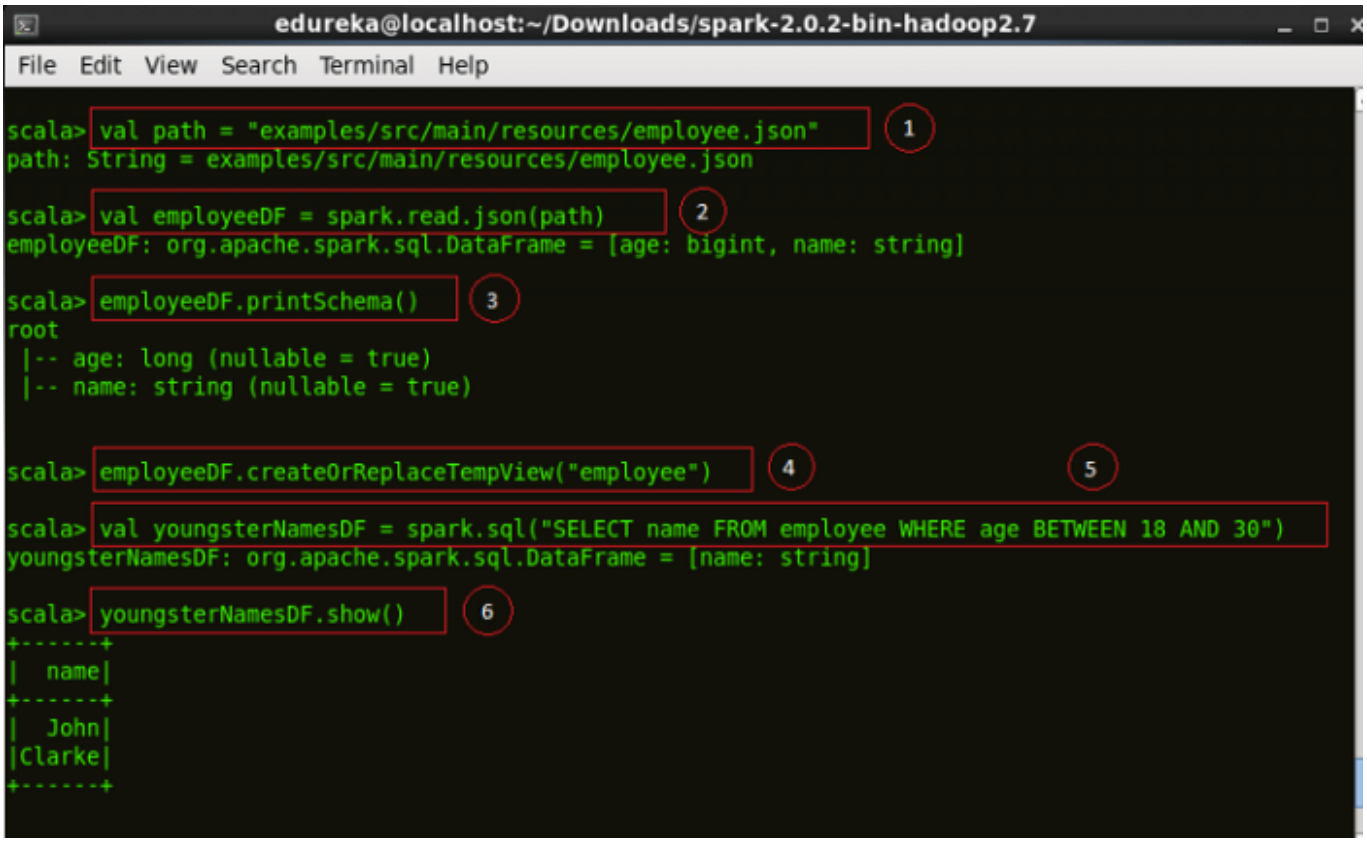
We will now work on JSON data. As Spark SQL supports JSON dataset, we create a DataFrame of employee.json. The schema of this DataFrame can be seen below. We then define a Youngster DataFrame and add all the employees between the ages of 18 and 30.

Code explanation:

1. Setting to path to our ’employee.json’ file.

2. Creating a DataFrame 'employeeDF' from our JSON file.
3. Printing the schema of 'employeeDF'.
4. Creating a temporary view of the DataFrame into 'employee'.
5. Defining a DataFrame 'youngsterNamesDF' which stores the names of all the employees between the ages of 18 and 30 present in 'employee'.
6. Displaying the contents of our DataFrame.

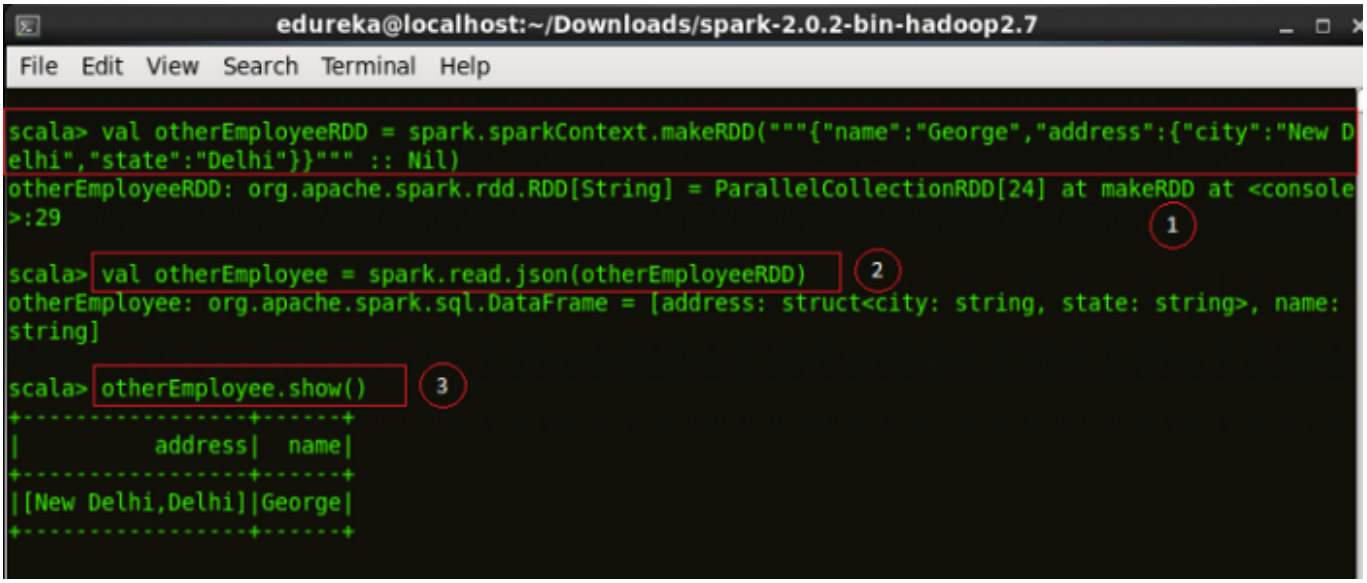
```
val path = "examples/src/main/resources/employee.json"
val employeeDF = spark.read.json(path)
employeeDF.printSchema()
employeeDF.createOrReplaceTempView("employee")
val youngsterNamesDF = spark.sql("SELECT name FROM employee WHERE age
BETWEEN 18 AND 30")
youngsterNamesDF.show()
```



Code explanation:

1. Creating a RDD 'otherEmployeeRDD' which will store the content of employee George from New Delhi, Delhi.
2. Assigning the contents of 'otherEmployeeRDD' into 'otherEmployee'.
3. Displaying the contents of 'otherEmployee'.

```
val otherEmployeeRDD =
spark.sparkContext.makeRDD("""{"name":"George","address":{"city":"New
Delhi","state":"Delhi"}}""" :: Nil)
val otherEmployee = spark.read.json(otherEmployeeRDD)
otherEmployee.show()
```



Hive Tables

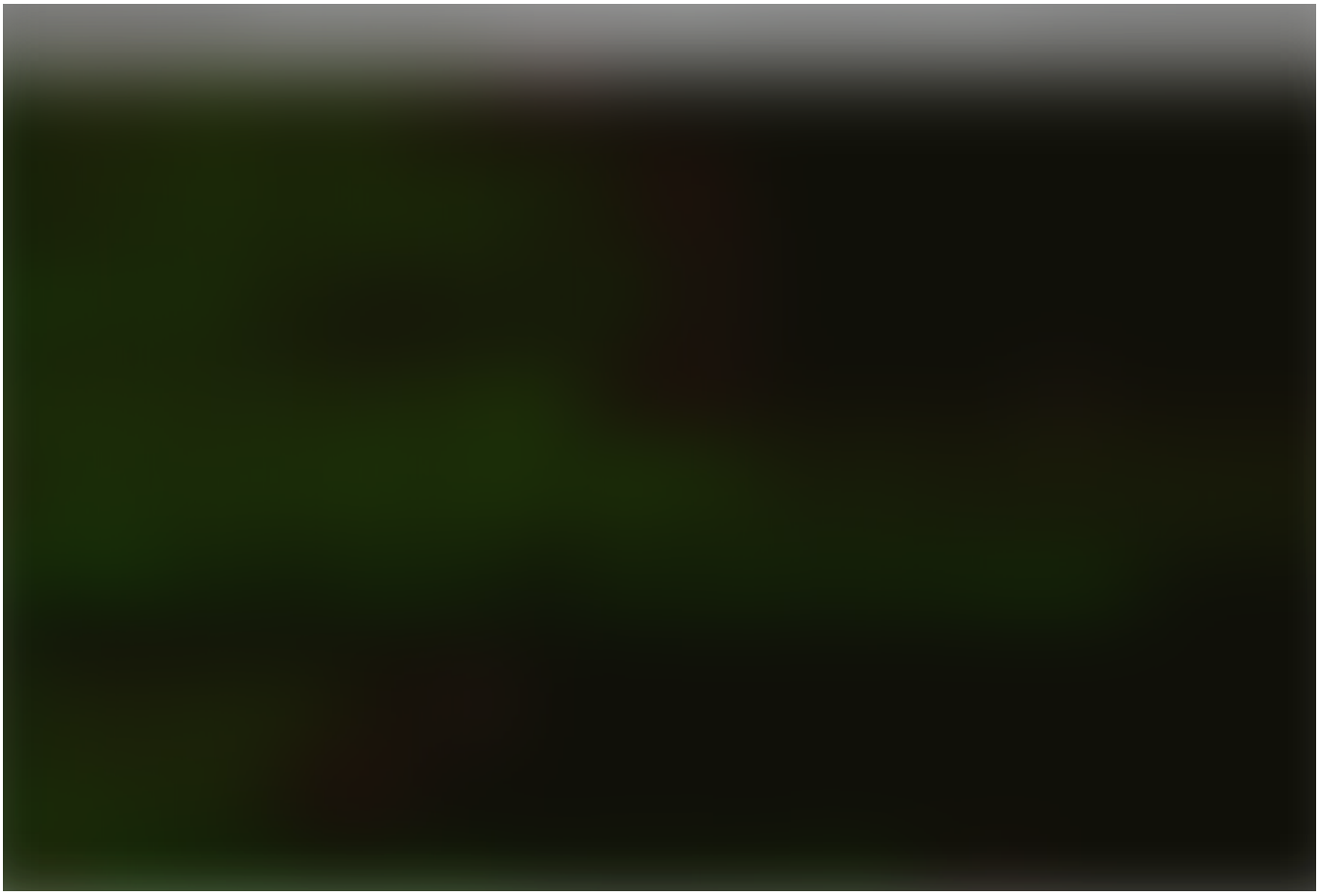
We perform a Spark example using Hive tables.

Code explanation:

1. Importing 'Row' class into the Spark Shell. Row is used in mapping RDD Schema.
2. Importing Spark Session into the shell.
3. Creating a class 'Record' with attributes Int and String.
4. Setting the location of 'warehouseLocation' to Spark warehouse.
5. We now build a Spark Session 'spark' to demonstrate Hive example in Spark SQL.
6. Importing Implicits class into the shell.
7. Importing SQL library into the Spark Shell.
8. Creating a table 'src' with columns to store key and value.

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.SparkSession
case class Record(key: Int, value: String)
val warehouseLocation = "spark-warehouse"
val spark = SparkSession.builder().appName("Spark Hive
Example").config("spark.sql.warehouse.dir",
warehouseLocation).enableHiveSupport().getOrCreate()
import spark.implicits._
```

```
import spark.sql
sql(“CREATE TABLE IF NOT EXISTS src (key INT, value STRING)”)
```



Code explanation:

1. We now load the data from the examples present in Spark directory into our table ‘src’.
2. The contents of ‘src’ is displayed below.

```
sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt'
INTO TABLE src")
sql("SELECT * FROM src").show()
```

Code explanation:

1. We perform the ‘count’ operation to select the number of keys in ‘src’ table.
2. We now select all the records with ‘key’ value less than 10 and store it in the ‘sqlDF’ DataFrame.
3. Creating a Dataset ‘stringDS’ from ‘sqlDF’.
4. Displaying the contents of ‘stringDS’ Dataset.

```
sql(“SELECT COUNT(*) FROM src”).show()
val sqlDF = sql(“SELECT key, value FROM src WHERE key
&amp;amp;amp;amp;lt; 10 ORDER BY key”) val stringsDS = sqlDF.map
{case Row(key: Int, value: String) =&amp;amp;amp;amp;gt; s”Key: $key,
Value: $value”}
stringsDS.show()
```

Code explanation:

1. We create a DataFrame ‘recordsDF’ and store all the records with key values 1 to 100.
2. Create a temporary view ‘records’ of ‘recordsDF’ DataFrame.
3. Displaying the contents of the join of tables ‘records’ and ‘src’ with ‘key’ as the primary key.

```
val recordsDF = spark.createDataFrame((1 to 100).map(i
=&amp;amp;amp;amp;gt; Record(i, s”val_$i”)))
recordsDF.createOrReplaceTempView(“records”)
sql(“SELECT * FROM records r JOIN src s ON r.key = s.key”).show()
```

So folks, that's an end to this article on Spark SQL Tutorial.

So this is it! I hope this blog was informative and added value to your knowledge. If you wish to check out more articles on the market's most trending technologies like Artificial Intelligence, DevOps, Ethical Hacking, then you can refer to Edureka's official site.

Do look out for other articles in this series which will explain the various other aspects of Spark.

1. Apache Spark Architecture
2. Spark Streaming Tutorial
3. Spark MLlib
4. Apache Spark Tutorial
5. Spark GraphX Tutorial
6. Spark Java Tutorial

Big Data Spark Spark Sql Apache Spark Rdd