CHAPTER 4

MACHINE LEARNING, STREAMS AND DATABASE ON SPARK

4.1 SPARK: REAL TIME CLUSTER COMPUTING FRAMEWORK [36]

Apache Spark is an open-source cluster computing framework for real-time processing. It is of the most successful projects in the Apache Software Foundation. Spark has clearly evolved as the market leader for Big Data processing. Today, Spark is being adopted by major players like Amazon, eBay, and Yahoo! Many organizations run Spark on clusters with thousands of nodes. We are excited to begin this exciting journey through this Spark Tutorial blog. This blog is the first blog in the upcoming Apache Spark blog series which will include Spark Streaming, Spark Interview Questions, Spark MLlib and others.

When it comes to Real Time Data Analytics, Spark stands as the go-to tool across all other solutions. Through this blog, I will introduce you to this new exciting domain of Apache Spark and we will go through a complete use case, Earthquake Detection using Spark.

4.1.1 Real Time Analytics [36]

Before we begin, let us have a look at the amount of data generated every minute by social media leaders.

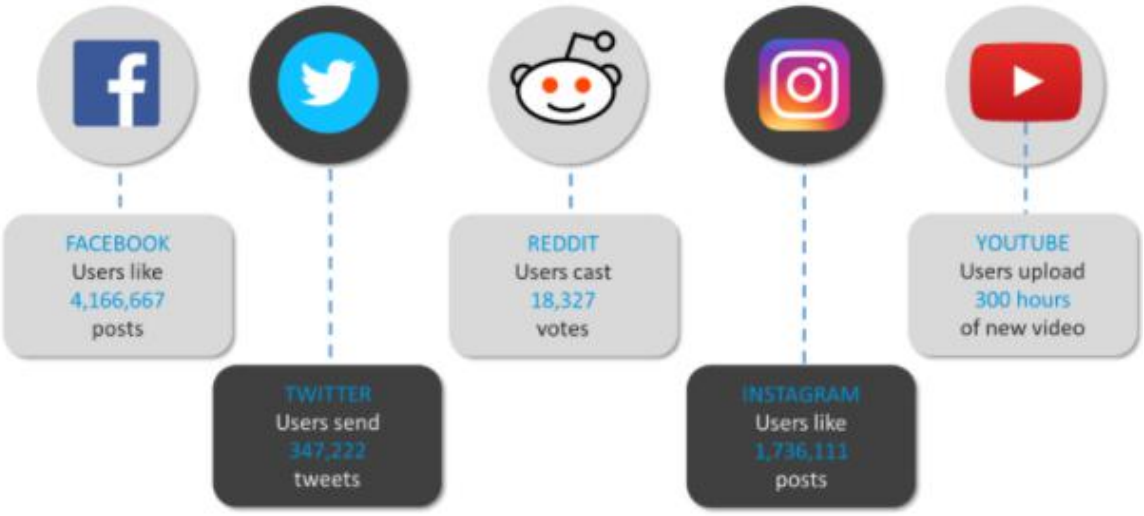


Figure 4.1 Amount of data generated every minute. [36]

As we can see, there is a colossal amount of data that the internet world necessitates to process in seconds. We will go through all the stages of handling big data in enterprises and discover the need for a *Real Time Processing Framework* called **Apache Spark**.

To begin with, let me introduce you to few domains using real-time analytics big time in today's world.

Figure 4.2 Examples of Real Time Analytics. [36]

We can see that Real Time Processing of Big Data is ingrained in every aspect of our lives. From fraud detection in banking to live surveillance systems in government, automated machines in healthcare to live prediction systems in the stock market, everything around us revolves around processing big data in near real time.

Let us look at some of these use cases of Real Time Analytics:

1. **Healthcare**: Healthcare domain uses Real Time analysis to continuously check the medical status of critical patients. Hospitals on the look out for blood and organ transplants need to stay in a real-time contact with each other during emergencies. Getting medical attention on time is a matter of life and death for patients.
2. **Government**: Government agencies perform Real Time Analysis mostly in the field of national security. Countries need to continuously keep a track of all the military and police agencies for updates regarding threats to security.
3. **Telecommunications**: Companies revolving around services in the form of calls, video chats and streaming use real-time analysis to reduce customer churn and stay

ahead of the competition. They also extract measurements of jitter and delay in mobile networks to improve customer experiences.

4. **Banking**: Banking transacts with almost all of the world's money. It becomes very important to ensure fault tolerant transactions across the whole system. Fraud detection is made possible through real-time analytics in banking.
5. **Stock Market**: Stockbrokers use real-time analytics to predict the movement of stock portfolios. Companies re-think their business model after using real-time analytics to analyze the market demand for their brand.

4.1.2 Why Spark when Hadoop is already there? [36]

The first of the many questions everyone asks when it comes to Spark is, "Why Spark when we have Hadoop already?".

To answer this, we have to look at the concept of batch and real-time processing. Hadoop is based on the concept of batch processing where the processing happens of blocks of data that have already been stored over a period of time. At the time, Hadoop broke all the expectations with the revolutionary MapReduce framework in 2005. Hadoop MapReduce is the best framework for processing data in batches.

This went on until 2014, till Spark overtook Hadoop. The USP for Spark was that it could process data in real time and was about 100 times faster than Hadoop MapReduce in batch processing large data sets.

The following figure gives a detailed explanation of the differences between processing in Spark and Hadoop.

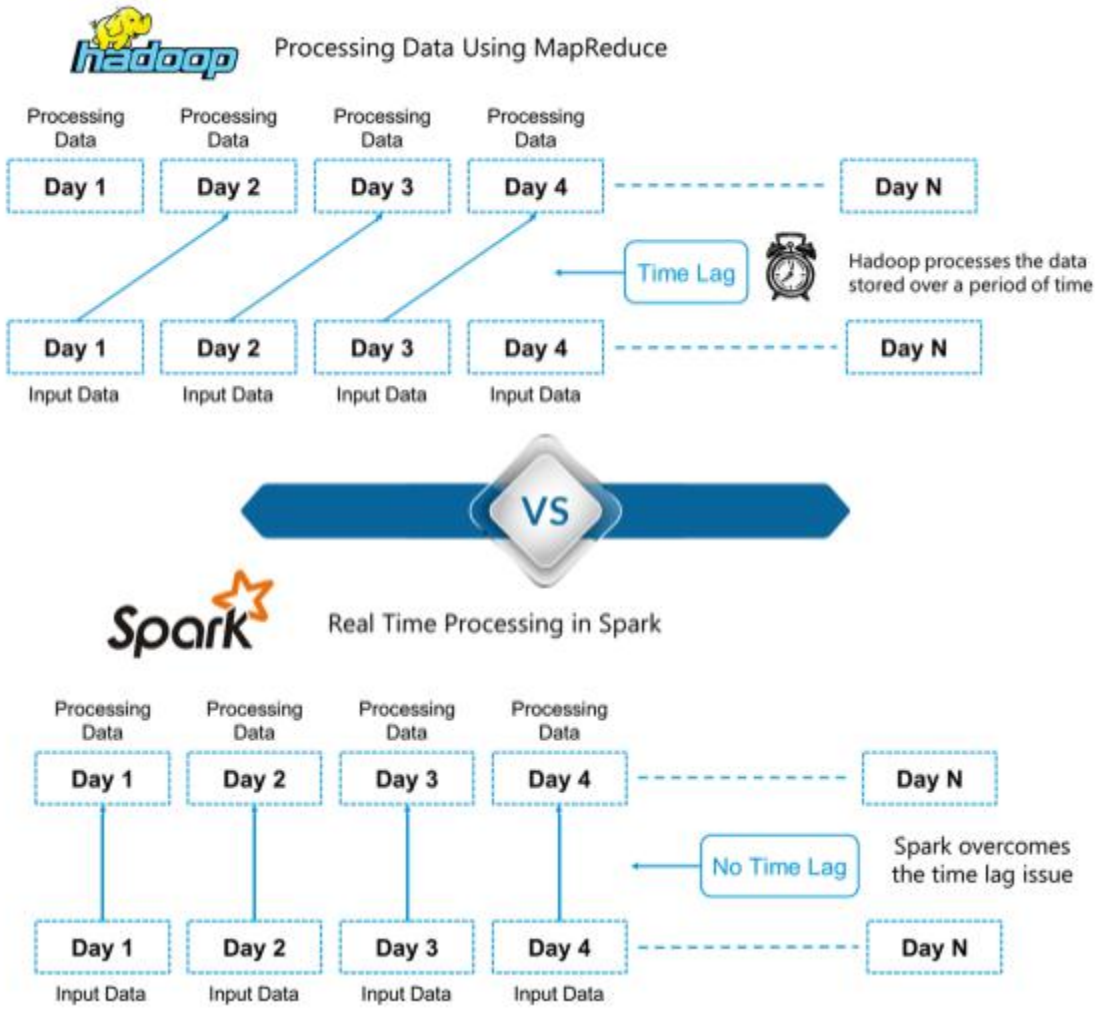


Figure 4.3 Differences between Hadoop and Spark. [36]

Here, we can draw out one of the key differentiators between Hadoop and Spark. Hadoop is based on batch processing of big data. This means that the data is stored over a period of time and is then processed using Hadoop. Whereas in Spark, processing can take place in real-time. This real-time processing power in Spark helps us to solve the use cases of Real Time Analytics we saw in the previous section. Alongside this, Spark is also able to do batch processing 100 times faster than that of Hadoop MapReduce (Processing framework in Apache Hadoop). Therefore, Apache Spark is the go-to tool for big data processing in the industry.

4.1.3 What is Apache Spark? [36]

Apache Spark is an open-source cluster computing framework for real-time processing. It has a thriving open-source community and is the most active Apache project at the

moment. Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.
It was built on top of Hadoop MapReduce and it extends the MapReduce model to efficiently use more types of computations.

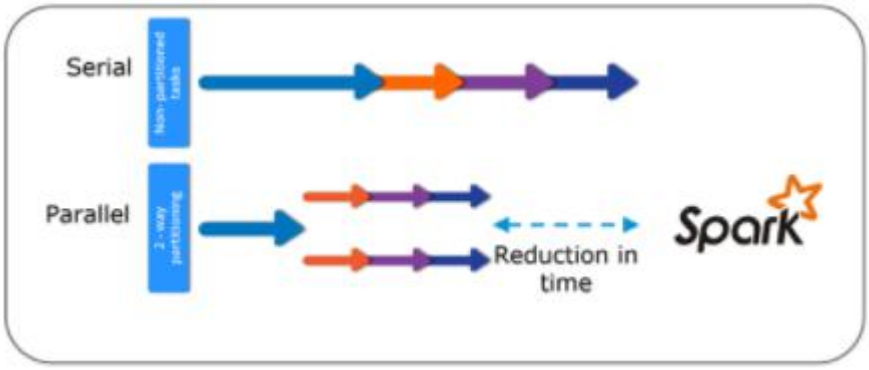


Figure 4.4 Real Time Processing in Apache Spark. [36]

4.1.4 Features of Apache Spark [36]

Spark has the following features:

Figure 4.5 Spark Features. [36]

Let us look at the features in detail:

Polyglot:
Spark provides high-level APIs in Java, Scala, Python and R. Spark code can be written in any of these four languages. It provides a shell in Scala and Python. The Scala shell can be accessed through ./bin/spark-shell and Python shell through ./bin/pyspark from the installed directory.

Speed:
Spark runs up to 100 times faster than Hadoop MapReduce for large-scale data processing. Spark is able to achieve this speed through controlled partitioning. It manages data using partitions that help parallelize distributed data processing with minimal network traffic.

Multiple Formats:
Spark supports multiple data sources such as Parquet, JSON, Hive and Cassandra apart from the usual formats such as text files, CSV and RDBMS tables. The Data Source API provides a pluggable mechanism for accessing structured data though Spark SQL. Data sources can be more than just simple pipes that convert data and pull it into Spark.

Lazy Evaluation:
Apache Spark delays its evaluation till it is absolutely necessary. This is one of the key factors contributing to its speed. For transformations, Spark adds them to a DAG (Directed Acyclic Graph) of computation and only when the driver requests some data, does this DAG actually gets executed.

Real Time Computation:
Spark's computation is real-time and has low latency because of its in-memory computation. Spark is designed for massive scalability and the Spark team has documented users of the system running production clusters with thousands of nodes and supports several computational models.

Hadoop Integration:
Apache Spark provides smooth compatibility with Hadoop. This is a boon for all the Big Data engineers who started their careers with Hadoop. Spark is a potential replacement for the MapReduce functions of Hadoop, while Spark has the ability to run on top of an existing Hadoop cluster using YARN for resource scheduling.

Machine Learning:
Spark's MLlib is the machine learning component which is handy when it comes to big data processing. It eradicates the need to use multiple tools, one for processing and one for machine learning. Spark provides data engineers and data scientists with a powerful, unified engine that is both fast and easy to use.

4.1.5 Getting Started With Spark [36]

The first step in getting started with Spark is installation. Let us install Apache Spark 2.1.0 on Linux systems:

Installation:

1. The prerequisites for installing Spark is having Java and Scala installed.
2. Download Java in case it is not installed using below commands.

```
sudo apt-get install python-software-properties
sudo apt-add-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

3. Download the latest Scala version from Scala Lang Official page[37]. Once installed, set the scala path in ~/.bashrc file as shown below.

```
export SCALA_HOME=Path_Where_Scala_File_Is_Located
export PATH=$SCALA_HOME/bin:PATH
```

4. Download Spark 2.1.0 from the Apache Spark Downloads page[38]. You can also choose to download a previous version.
5. Extract Spark tar using below command.

```
tar -xvf spark-2.1.0-bin-hadoop2.7.tgz
```

6. Set the Spark_Path in ~/.bashrc file.

```
export SPARK_HOME=Path_Where_Spark_Is_Installed
export PATH=$PATH:$SPARK_HOME/bin
```

Before we move further, let us start up Apache Spark on our systems and get used to the main concepts of Spark like Spark Session, Data Sources, RDDs, DataFrames and other libraries.

Spark Shell:
Spark's shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively.

Spark Session:
In earlier versions of Spark, Spark Context was the entry point for Spark. For every other API, we needed to use different contexts. For streaming, we needed StreamingContext, for SQL sqlContext and for hive HiveContext. To solve this issue, SparkSession came into the picture. It is essentially a combination of SQLContext, HiveContext and future StreamingContext.

Data Sources:
The Data Source API provides a pluggable mechanism for accessing structured data though Spark SQL. Data Source API is used to read and store structured and semi-structured data into Spark SQL. Data sources can be more than just simple pipes that convert data and pull it into Spark.

RDD:
Resilient Distributed Dataset (RDD) 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. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Dataset:
A Dataset is a distributed collection of data. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). The Dataset API is available in Scala and Java.

DataFrames:
A DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases or existing RDDs.

4.1.6 Using Spark with Hadoop [36]

The best part of Spark is its compatibility with Hadoop. As a result, this makes for a very powerful combination of technologies. Here, we will be looking at how Spark can benefit from the best of Hadoop.

Hadoop components can be used alongside Spark in the following ways:

- **HDFS**: Spark can run on top of HDFS to leverage the distributed replicated storage.

- **MapReduce**: Spark can be used along with MapReduce in the same Hadoop cluster or separately as a processing framework.
- **YARN**: Spark applications can be made to run on YARN (Hadoop NextGen).
- **Batch & Real Time Processing**: MapReduce and Spark are used together where MapReduce is used for batch processing and Spark for real-time processing.

4.1.7 Spark Components [36]

Spark components are what make Apache Spark fast and reliable. A lot of these Spark components were built to resolve the issues that cropped up while using Hadoop MapReduce. Apache Spark has the following components:

1. Spark Core
2. Spark Streaming
3. Spark SQL
4. GraphX
5. MLlib (Machine Learning)

Spark Core

Spark Core is the base engine for large-scale parallel and distributed data processing. The core is the distributed execution engine and the Java, Scala, and Python APIs offer a platform for distributed ETL application development. Further, additional libraries which are built atop the core allow diverse workloads for streaming, SQL, and machine learning. It is responsible for:

1. Memory management and fault recovery
2. Scheduling, distributing and monitoring jobs on a cluster
3. Interacting with storage systems

Spark Streaming

Spark Streaming is the component of Spark which is used to process real-time streaming data. Thus, it is a useful addition to the core Spark API. It enables high-throughput and fault-tolerant stream processing of live data streams. The fundamental stream unit is DStream which is basically a series of RDDs (Resilient Distributed Datasets) to process the real-time data.

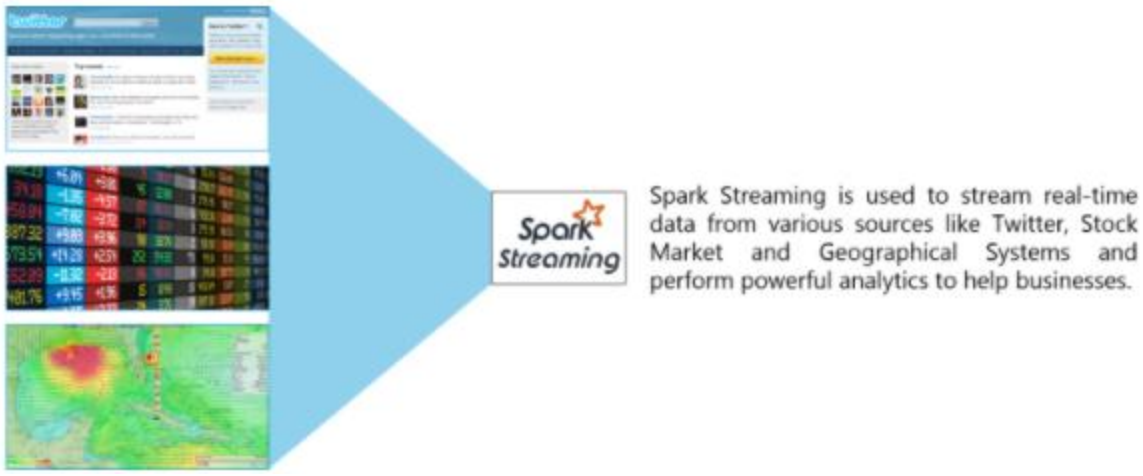


Figure 4.6 Spark Streaming. [36]

Spark SQL

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.

Spark SQL integrates relational processing with Spark's functional programming. Further, 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.

The following are the four libraries of Spark SQL.

1. Data Source API
2. DataFrame API
3. Interpreter & Optimizer
4. SQL Service

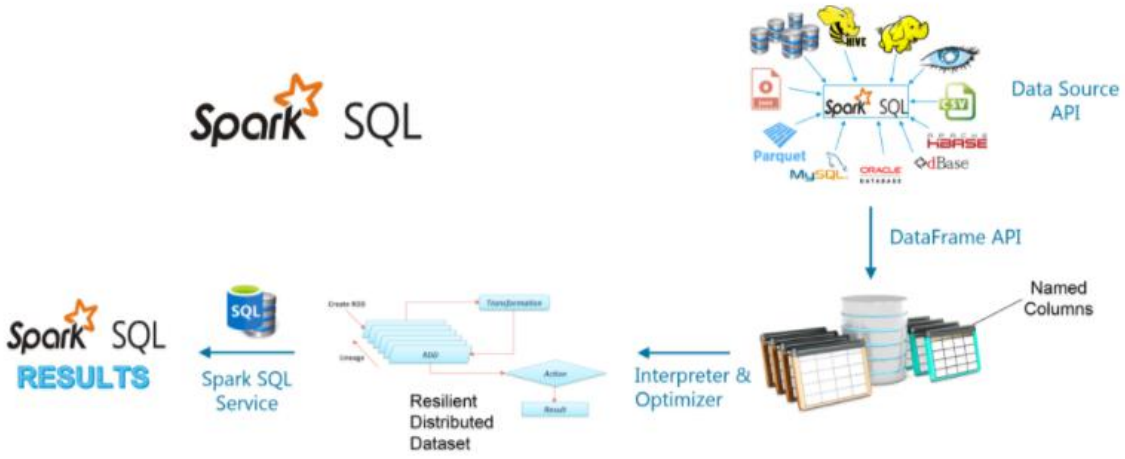


Figure 4.7 Spark SQL process using all the four libraries in sequence. [36]

GraphX

GraphX is the Spark API for graphs and graph-parallel computation. Thus, it extends the Spark RDD with a Resilient Distributed Property Graph.

The property graph is a directed multigraph which can have multiple edges in parallel. Every edge and vertex have user defined properties associated with it. Here, the parallel edges allow multiple relationships between the same vertices. At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge.

To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and mapReduceTriplets) as well as an optimized variant of the Pregel API. In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

MlLib (Machine Learning)

MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.

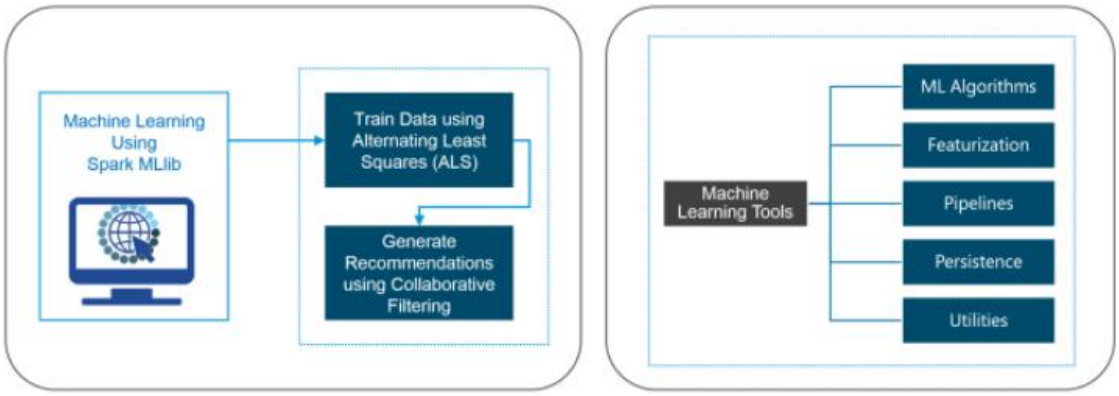


Figure 4.8 Machine Learning Flow Diagram / Machine Learning Tools. [36]

4.1.8 Earthquake Detection using Spark [36]

Now that we have understood the core concepts of Spark, let us solve a real-life problem using Apache Spark. This will help give us the confidence to work on any Spark projects in the future.

Problem Statement: *To design a Real Time Earthquake Detection Model to send lifesaving alerts, which should improve its machine learning to provide near real-time computation results.*

Use Case – Requirements:

1. Process data in real-time
2. Handle input from multiple sources
3. Easy to use system
4. Bulk transmission of alerts

We will use Apache Spark which is the perfect tool for our requirements.

Use Case – Dataset:

EARTHQUAKE ROC DATASET																		
	S Wave							P Wave										
Classification Index	First Activity	Time Taken	Acceleration	Building Strength	Velocity	Sa	Sd	First Activity	Time Taken	Acceleration	Building Strength	Velocity	Sa	Sd	Total Weight	Sum * ROC	ROC	AVG * ROC
0	3	11	14	19	39	42	55	64	67	73	75	76	80	83	701	618.168	0.881837	623.2843
0	3	6	17	27	35	40	57	63	69	73	74	76	81	103	724	638.4503	0.881837	623.2843
0	4	6	15	21	35	40	57	63	67	73	74	77	80	83	695	612.877	0.881837	623.2843
0	5	6	15	22	36	41	47	66	67	72	74	76	80	83	690	608.4678	0.881837	623.2843
0	2	6	16	22	36	40	54	63	67	73	75	76	80	83	693	611.1133	0.881837	623.2843
0	2	6	14	20	37	41	47	64	67	73	74	76	82	83	686	604.9405	0.881837	623.2843
0	1	6	14	22	36	42	49	64	67	72	74	77	80	83	687	605.8223	0.881837	623.2843
0	1	6	17	19	39	42	53	64	67	73	74	76	80	83	694	611.9952	0.881837	623.2843
0	2	6	18	20	37	42	48	64	71	73	74	76	81	83	695	612.877	0.881837	623.2843
1	5	11	15	32	39	40	52	63	67	73	74	76	78	83	708	624.3409	0.881837	623.2843
0	5	16	30	35	41	64	67	73	74	76	80	83			644	567.9033	0.881837	623.2843
0	5	6	15	20	37	40	50	63	67	73	75	76	80	83	690	608.4678	0.881837	623.2843
0	5	7	16	29	39	40	48	63	67	73	74	76	78	83	698	615.5225	0.881837	623.2843
0	1	11	18	20	37	42	59	62	71	72	74	76	80	83	706	622.5772	0.881837	623.2843
1	5	18	19	39	40	63	67	73	74	76	80	83			637	561.7304	0.881837	623.2843

Figure 4.9 Use Case – Earthquake Dataset. [36]

You can download the complete dataset from [39]

Before moving ahead, there is one concept we have to learn that we will be using in our Earthquake Detection System and it is called Receiver Operating Characteristic (ROC). An ROC curve is a graphical plot that illustrates the performance of a binary classifier system as its discrimination threshold is varied. We will use the dataset to obtain an ROC value using Machine Learning in Apache Spark.

Use Case – Flow Diagram:

The following illustration clearly explains all the steps involved in our Earthquake Detection System.

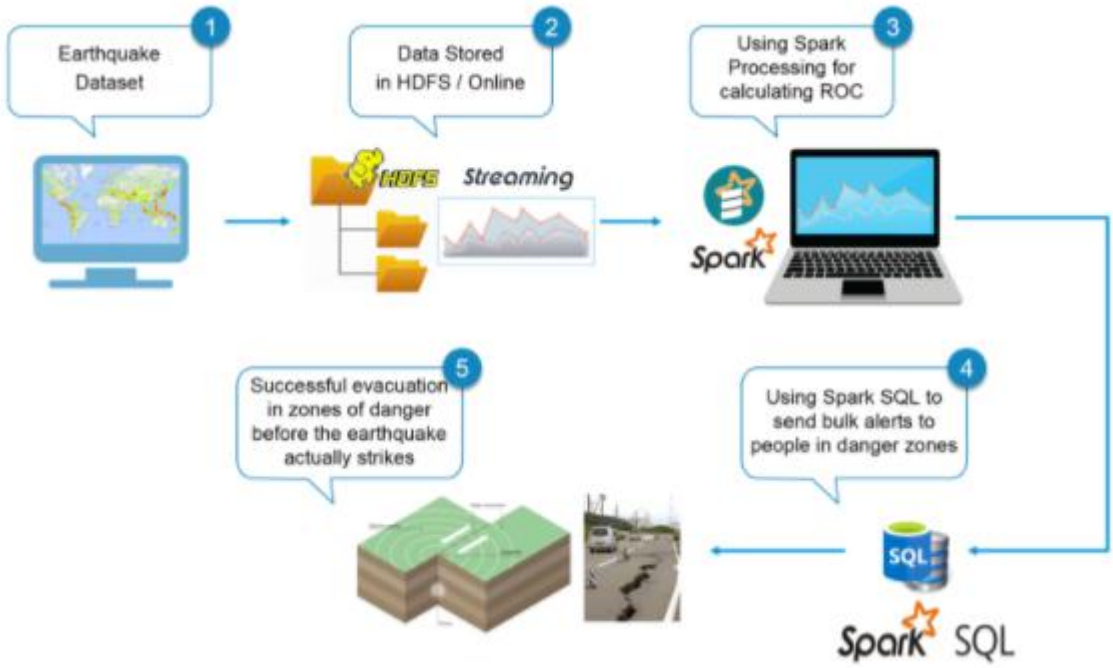


Figure 4.10 Flow diagram of Earthquake Detection using Apache Spark. [36]

Use Case – Spark Implementation:

Moving ahead, now let us implement our project using Eclipse IDE for Spark.

Find the Pseudo Code below:

```
//Importing the necessary classes
import org.apache.spark._
...
//Creating an Object earthquake
object earthquake {
 def main(args: Array[String]) {

//Creating a Spark Configuration and Spark Context
val sparkConf = new SparkConf().setAppName("earthquake").setMaster("local[2]")
val sc = new SparkContext(sparkConf)

//Loading the Earthquake ROC Dataset file as a LibSVM file
val data = MLUtils.loadLibSVMFile(sc, *Path to the Earthquake File* )

//Training the data for Machine Learning
val splits = data.randomSplit( *Splitting 60% to 40%* , seed = 11L)
val training = splits(0).cache()
val test = splits(1)

//Creating a model of the trained data
val numIterations = 100
val model = *Creating SVM Model with SGD* (  *Training Data* , *Number of Iterations* )

//Using map transformation of model RDD
val scoreAndLabels = *Map the model to predict features*

//Using Binary Classification Metrics on scoreAndLabels
val metrics = * Use Binary Classification Metrics on scoreAndLabels *(scoreAndLabels)
val auROC = metrics. *Get the area under the ROC Curve*()

//Displaying the area under Receiver Operating Characteristic
println("Area under ROC = " + auROC)
 }
}
```

The full source code of Earthquake Detection using Apache Spark is available in [40]

From our Spark program, we obtain the ROC value to be 0.088137. We will be transforming this value to get the area under the ROC curve.

Use Case – Visualizing Results:

We will plot the ROC curve and compare it with the specific earthquake points. Where ever the earthquake points exceed the ROC curve, such points are treated as major earthquakes. As per our algorithm to calculate the Area under the ROC curve, we can assume that these major earthquakes are above 6.0 magnitude on the Richter scale.

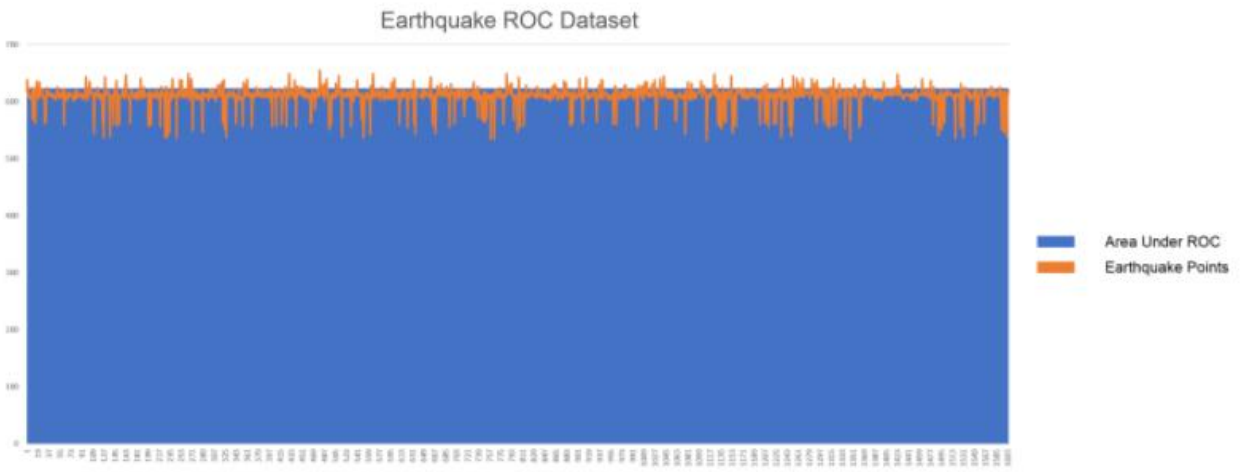


Figure 4.11 Earthquake ROC Curve. [36]

The above image shows the Earthquake line in orange. The area in blue is the ROC curve that we have obtained from our Spark program. Let us zoom into the curve to get a better picture.

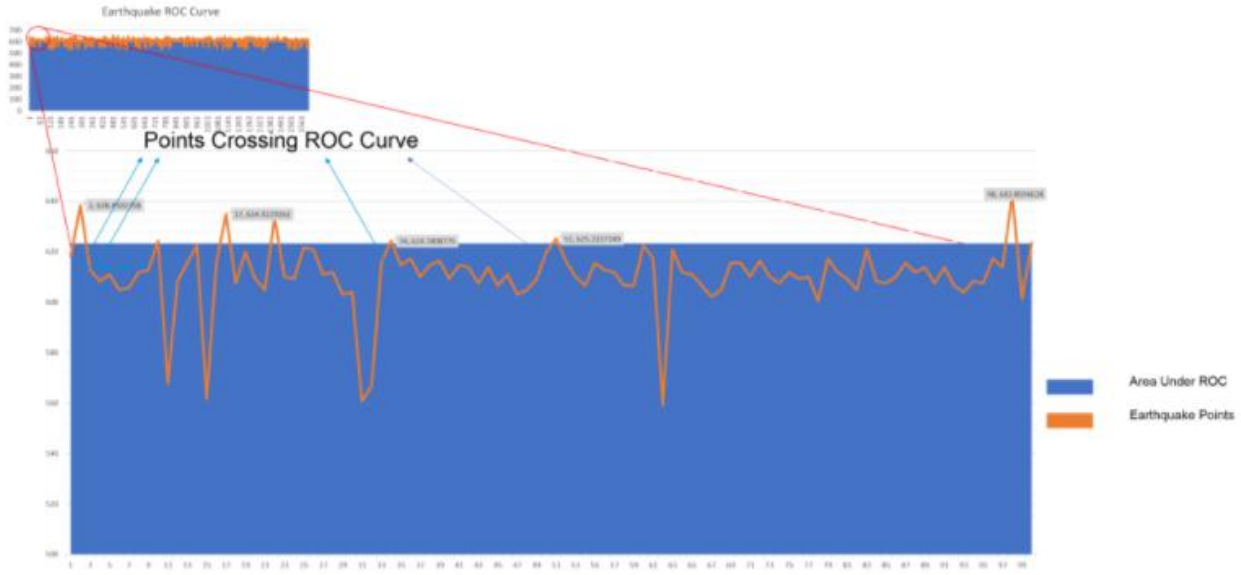


Figure 4.12 Visualizing Earthquake Points. [36]

We have plotted the earthquake curve against the ROC curve. At points where the orange curve is above the blue region, we have predicted the earthquakes to be major, i.e., with magnitude greater than 6.0. Thus armed with this knowledge, we could use Spark SQL and query an existing Hive table to retrieve email addresses and send people personalized warning emails. Thus we have used technology once more to save human life from trouble and make everyone's life better.

4.2 SENTIMENT ANALYSIS USING APACHE SPARK [36]

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Spark Streaming can be used to stream live data and processing can happen in real time. Spark Streaming's ever-growing user base consists of household names like Uber, Netflix and Pinterest.

When it comes to Real Time Data Analytics, Spark Streaming provides a single platform to ingest data for fast and live processing in Apache Spark. Through this blog, I will introduce you to this new exciting domain of Spark Streaming and we will go through a complete use case, Twitter Sentiment Analysis using Spark Streaming.

4.2.1 What is Streaming? [36]

Data Streaming is a technique for transferring data so that it can be processed as a steady and continuous stream. Streaming technologies are becoming increasingly important with the growth of the Internet.

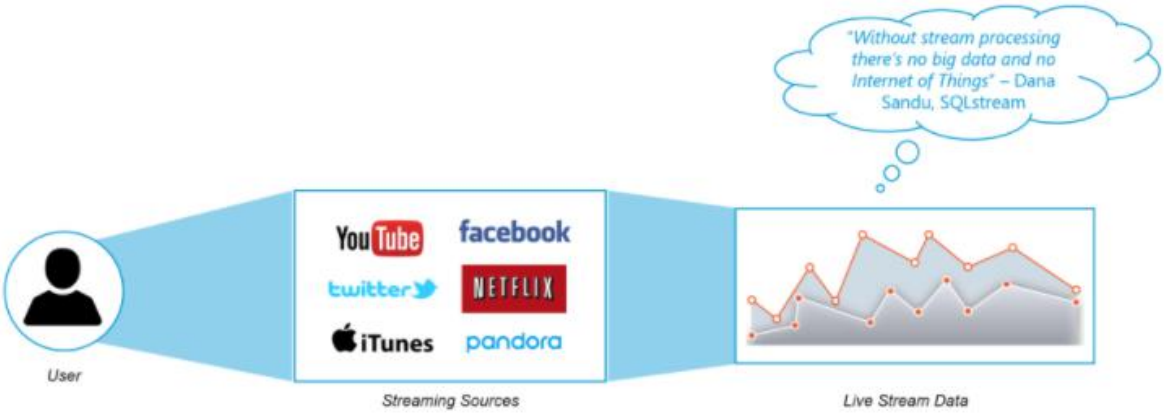


Figure 4.13 What is Streaming? [36]

4.2.2 Why Spark Streaming? [36]

We can use Spark Streaming to stream real-time data from various sources like Twitter, Stock Market and Geographical Systems and perform powerful analytics to help businesses.

4.2.3 Spark Streaming Overview [36]

Spark Streaming is used for processing real-time streaming data. It is a useful addition to the core Spark API. Spark Streaming enables high-throughput and fault-tolerant stream processing of live data streams.

Figure 4.14 Streams in Spark Streaming [36]

The fundamental stream unit is DStream which is basically a series of RDDs to process the real-time data.

4.2.4 Spark Streaming Features [36]

1. **Scaling**: Spark Streaming can easily scale to hundreds of nodes.
2. **Speed**: It achieves low latency.
3. **Fault Tolerance**: Spark has the ability to efficiently recover from failures.
4. **Integration**: Spark integrates with batch and real-time processing.
5. **Business Analysis**: Spark Streaming is used to track the behavior of customers which can be used in business analysis.

4.2.5 Spark Streaming Workflow [36]

Spark Streaming workflow has four high-level stages. The first is to stream data from various sources. These sources can be streaming data sources like Akka, Kafka, Flume, AWS or Parquet for real-time streaming. The second type of sources includes HBase, MySQL, PostgreSQL, Elastic Search, Mongo DB and Cassandra for static/batch streaming. Once this happens, Spark can be used to perform Machine Learning on the data through its MLlib API. Further, Spark SQL is used to perform further operations on this data. Finally, the streaming output can be stored into various data storage systems like HBase, Cassandra, MemSQL, Kafka, Elastic Search, HDFS and local file system.

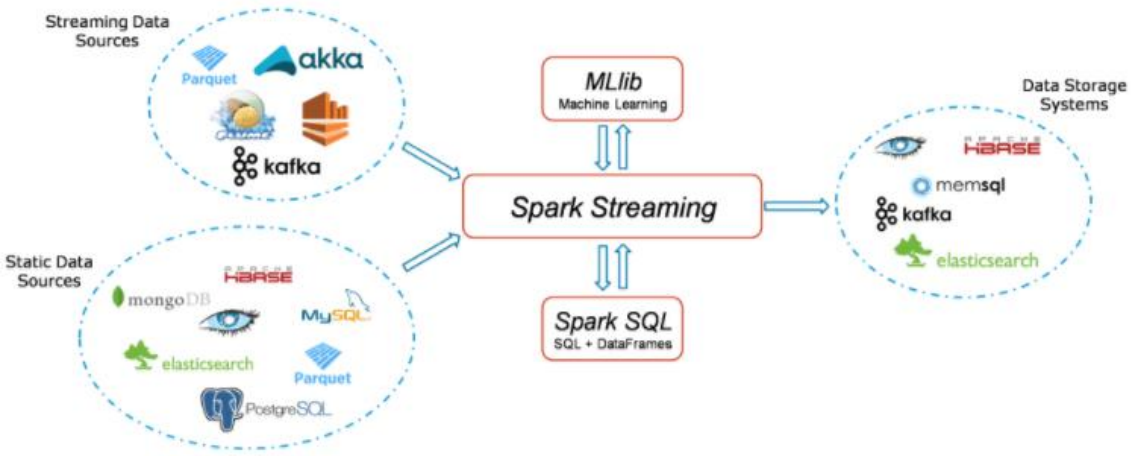


Figure 4.15 Overview Of Spark Streaming [36]

4.2.6 Spark Streaming Fundamentals [36]

1. Streaming Context
2. DStream
3. Caching
4. Accumulators, Broadcast Variables and Checkpoints

***Streaming Context* [36]**

Streaming Context consumes a stream of data in Spark. It registers an Input DStream to produce a Receiver object. It is the main entry point for Spark functionality. Spark provides a number of default implementations of sources like Twitter, Akka Actor and ZeroMQ that are accessible from the context.

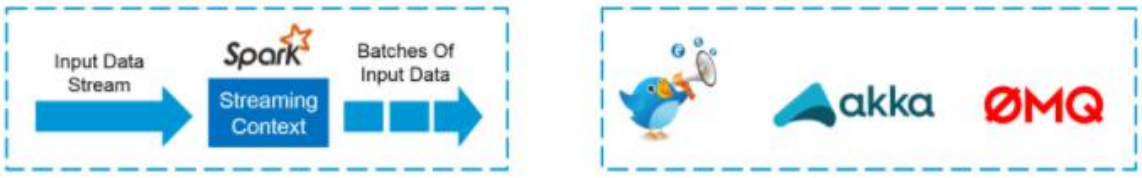


Figure 4.16 Spark Streaming Context / Default Implementation Sources [36]

A StreamingContext object can be created from a SparkContext object. A SparkContext represents the connection to a Spark cluster and can be used to create RDDs, accumulators and broadcast variables on that cluster.

```
import org.apache.spark._
import org.apache.spark.streaming._
var ssc = new StreamingContext(sc,Seconds(1))
```

DStream [36]

Discretized Stream (DStream) is the basic abstraction provided by Spark Streaming. It is a continuous stream of data. It is received from a data source or a processed data stream generated by transforming the input stream.

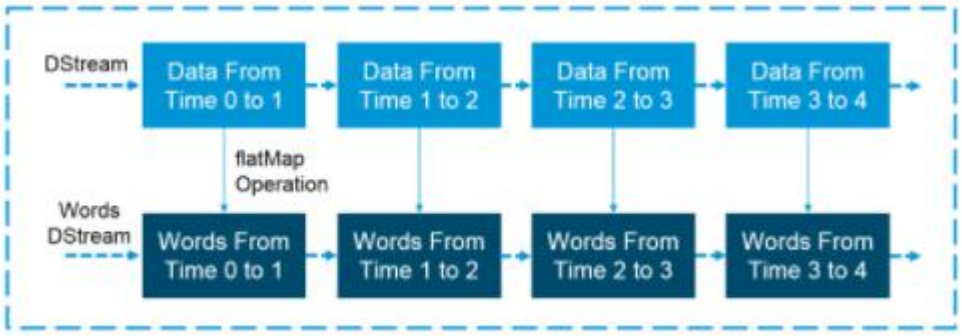


Figure 4.17 Extracting words from an Input DStream [36]

Internally, a DStream is represented by a continuous series of RDDs and each RDD contains data from a certain interval.

Input DStreams: Input DStreams are DStreams representing the stream of input data received from streaming sources.

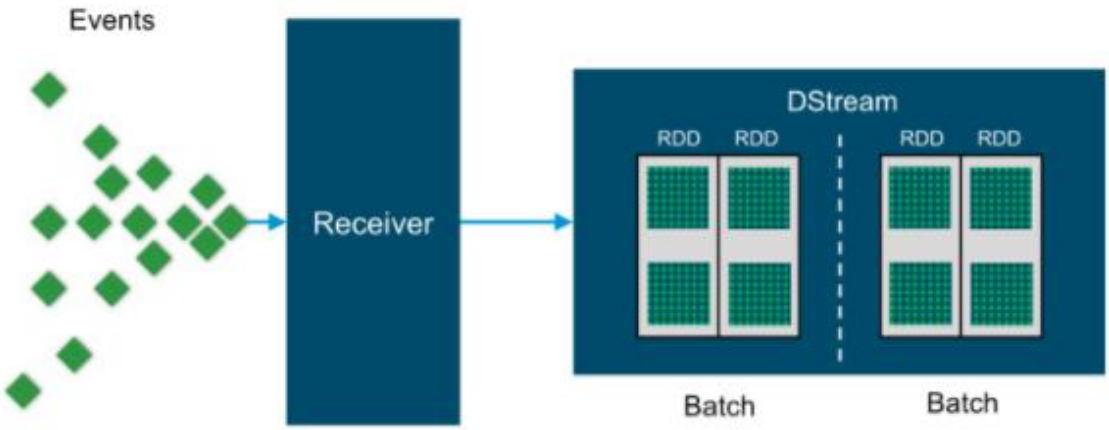


Figure 4.18 The Receiver sends data onto the Input DStream where each Batch contains RDDs [36]

Every input DStream is associated with a Receiver object which receives the data from a source and stores it in Spark's memory for processing.

Transformations on DStreams:

Any operation applied on a DStream translates to operations on the underlying RDDs. Transformations allow the data from the input DStream to be modified similar to RDDs. DStreams support many of the transformations available on normal Spark RDDs.

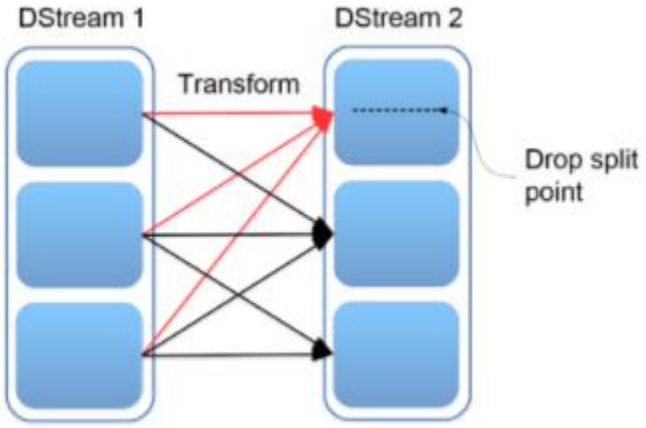


Figure 4.19 DStream Transformations [36]

The following are some of the popular transformations on DStreams:

map(*func*)	map(*func*) returns a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	flatMap(*func*) is similar to map(*func*) but each input item can be mapped to 0 or more output items and returns a new DStream by passing each source element through a function *func*.
filter(*func*)	filter(*func*) returns a new DStream by selecting only the records of the source DStream on which *func* returns true.
reduce(*func*)	reduce(*func*) returns a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func*.
groupBy(*func*)	groupBy(*func*) returns the new RDD which basically is made up with a key and corresponding list of items of that group.

Output DStreams:

Output operations allow DStream's data to be pushed out to external systems like databases or file systems. Output operations trigger the actual execution of all the DStream transformations.

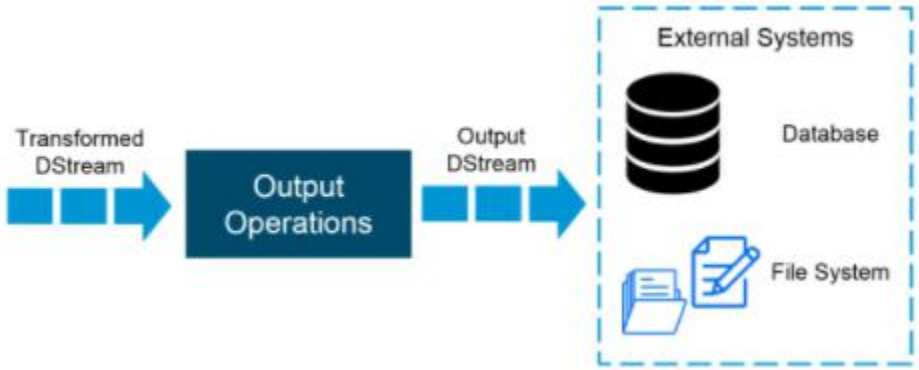


Figure 4.20 Output Operations on DStreams [36]

Caching [36]

DStreams allow developers to cache/ persist the stream's data in memory. This is useful if the data in the DStream will be computed multiple times. This can be done using the persist() method on a DStream.

Figure 4.21 Caching into 2 Nodes [36]

For input streams that receive data over the network (such as Kafka, Flume, Sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault-tolerance.

Accumulators, Broadcast Variables and Checkpoints

Accumulators: Accumulators are variables that are only added through an associative and commutative operation. They are used to implement counters or sums. Tracking accumulators in the UI can be useful for understanding the progress of running stages.

Spark natively supports numeric accumulators. We can create named or unnamed accumulators.

Broadcast Variables: Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Checkpoints: Checkpoints are similar to checkpoints in gaming. They make it run 24/7 and make it resilient to failures unrelated to the application logic.

Figure 4.22 Features of Checkpoints [36]

4.2.7 Use Case – Twitter Sentiment Analysis [36]

Now that we have understood the core concepts of Spark Streaming, let us solve a real-life problem using Spark Streaming.

Problem Statement: *To design a Twitter Sentiment Analysis System where we populate real-time sentiments for crisis management, service adjusting and target marketing.*

Applications of Sentiment Analysis:

1. Predict the success of a movie
2. Predict political campaign success
3. Decide whether to invest in a certain company
4. Targeted advertising
5. Review products and services

Spark Streaming Implementation:

Find the Pseudo Code below:

```
//Import the necessary packages into the Spark Program
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkContext._
...
import java.io.File

object twitterSentiment {

def main(args: Array[String]) {
if (args.length < 4) {
System.err.println("Usage: TwitterPopularTags <consumer key> <consumer secret> "
    + "<access token> <access token secret> [<filters>]")
System.exit(1)
}

StreamingExamples.setStreamingLogLevels()
//Passing our Twitter keys and tokens as arguments for authorization
val Array(consumerKey, consumerSecret, accessToken, accessTokenSecret) =
    args.take(4)
val filters = args.takeRight(args.length - 4)

// Set the system properties so that Twitter4j library used by twitter stream
// Use them to generate OAuth credentials
System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
...
System.setProperty("twitter4j.oauth.accessTokenSecret", accessTokenSecret)

val sparkConf = new SparkConf().setAppName("twitterSentiment").setMaster("local[2]")
val ssc = new Streaming Context
val stream = TwitterUtils.createStream(ssc, None, filters)

//Input DStream transformation using flatMap
val tags = stream.flatMap { status => Get Text From The Hashtags }

//RDD transformation using sortBy and then map function
tags.countByValue()
.foreachRDD { rdd =>
val now = Get current time of each Tweet
rdd
.sortBy(_._2)
.map(x => (x, now))
//Saving our output at ~/twitter/ directory
.saveAsTextFile(s"~/twitter/$now")
```

```
}

//DStream transformation using filter and map functions
val tweets = stream.filter {t =>
val tags = t. Split On Spaces .filter(_.startsWith("#")). Convert To Lower Case
tags.exists { x => true }
}

val data = tweets.map { status =>
val sentiment = SentimentAnalysisUtils.detectSentiment(status.getText)
val tagss = status.getHashtagEntities.map(_.getText.toLowerCase)
(status.getText, sentiment.toString, tagss.toString())
}

data.print()
//Saving our output at ~/ with filenames starting like twitters
data.saveAsTextFiles("~/twitters","20000")

ssc.start()
ssc.awaitTermination()
 }
}
```

The full source code of Twitter Sentiment Analysis using Spark Streaming is available in [41]

Results:

The following are the results that are displayed in the Eclipse IDE while running the Twitter Sentiment Streaming program.

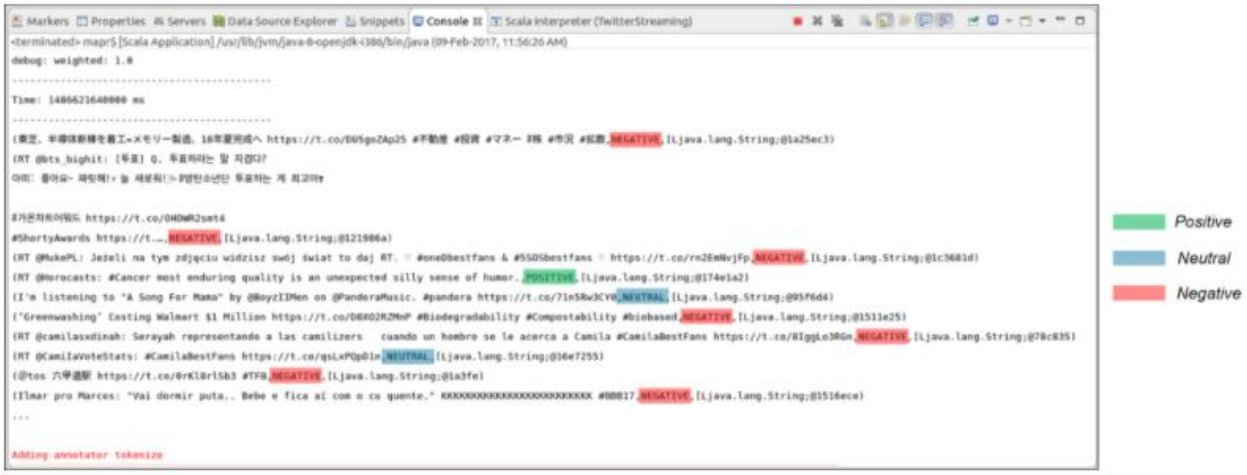


Figure 4.23 Sentiment Analysis Output in Eclipse IDE [36]

As we can see in the screenshot, all the tweets are categorized into Positive, Neutral and Negative according to the sentiment of the contents of the tweets.

The output of the Sentiments of the Tweets is stored into folders and files according to the time they were created. This output can be stored on the local file system or HDFS as necessary. The output directory looks like this:

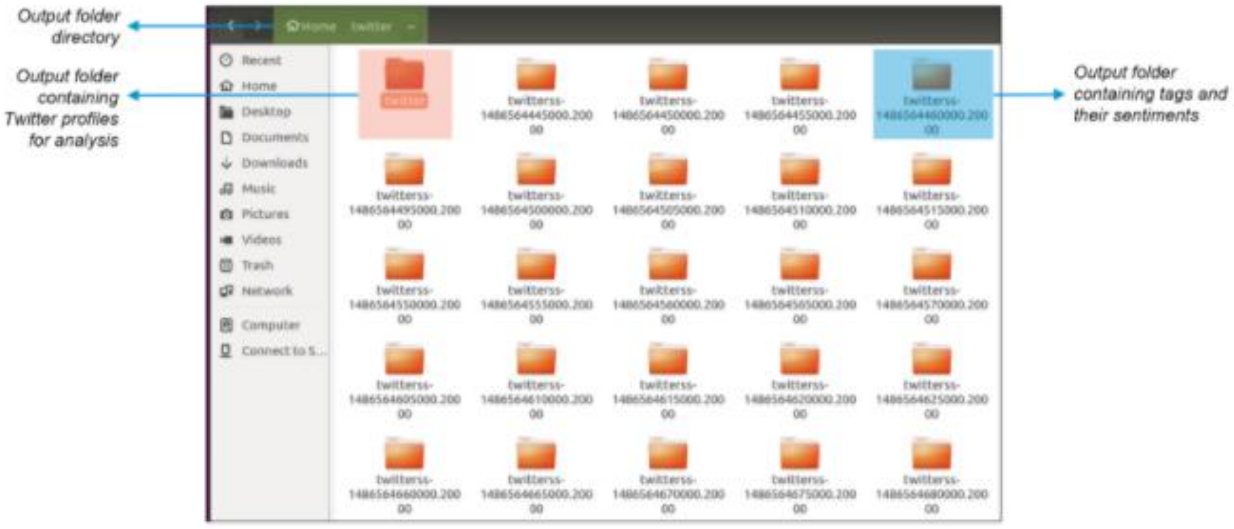


Figure 4.24 Output folders inside our 'twitter' project folder [36]

Here, inside the twitter directory, we can find the usernames of the Twitter users along with the timestamp for every tweet as shown below:

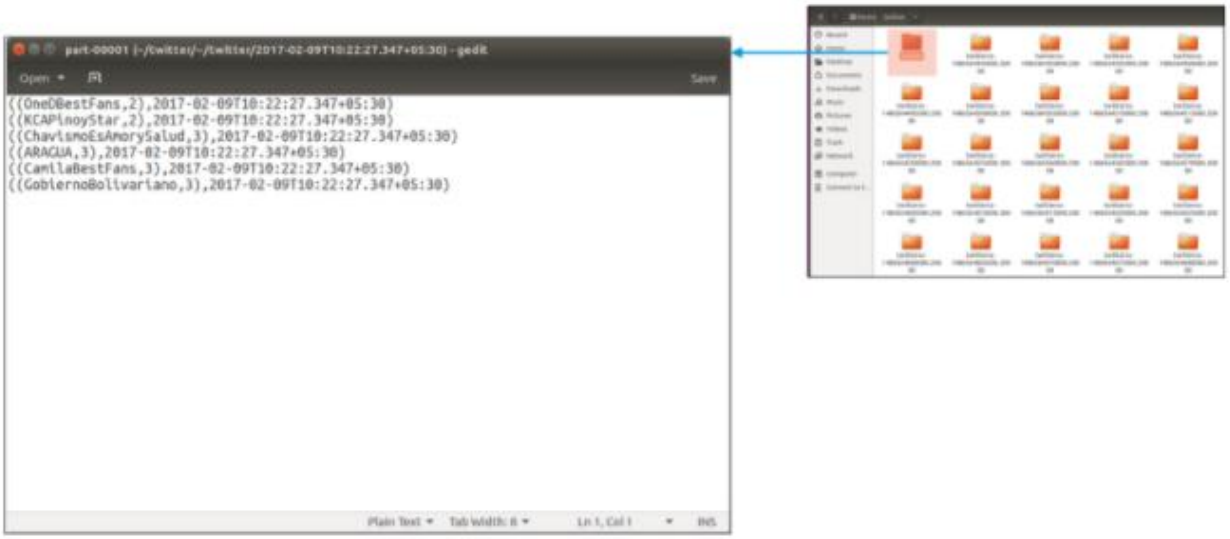


Figure 4.25 Output file containing Twitter usernames with timestamp [36]

Now that we have got the Twitter usernames and timestamp, let us look at the Sentiments and tweets stored in the main directory. Here, every tweet is followed by the sentiment emotion. This Sentiment that is stored is further used for analyzing a vast multitude of insights by companies.

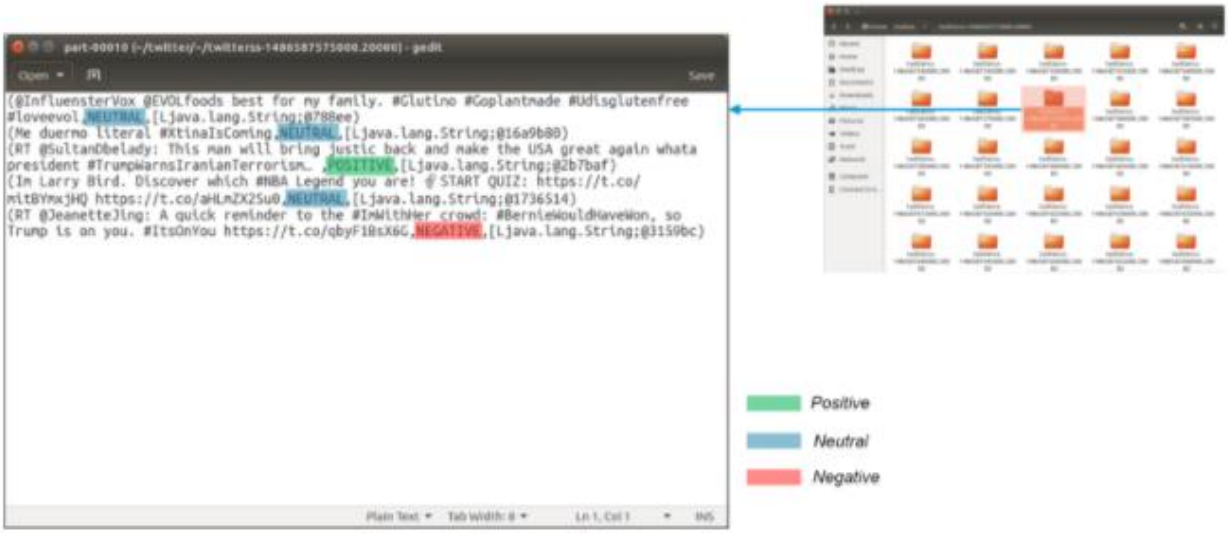


Figure 4.26 Output file containing tweets with sentiments [36]

Tweaking Code:

Now, let us modify our code a little to get sentiments for specific hashtags (topics). Currently, Donald Trump, the President of the United States is trending across news channels and online social media. Let us look at the sentiments associated with the keyword ‘Trump‘.

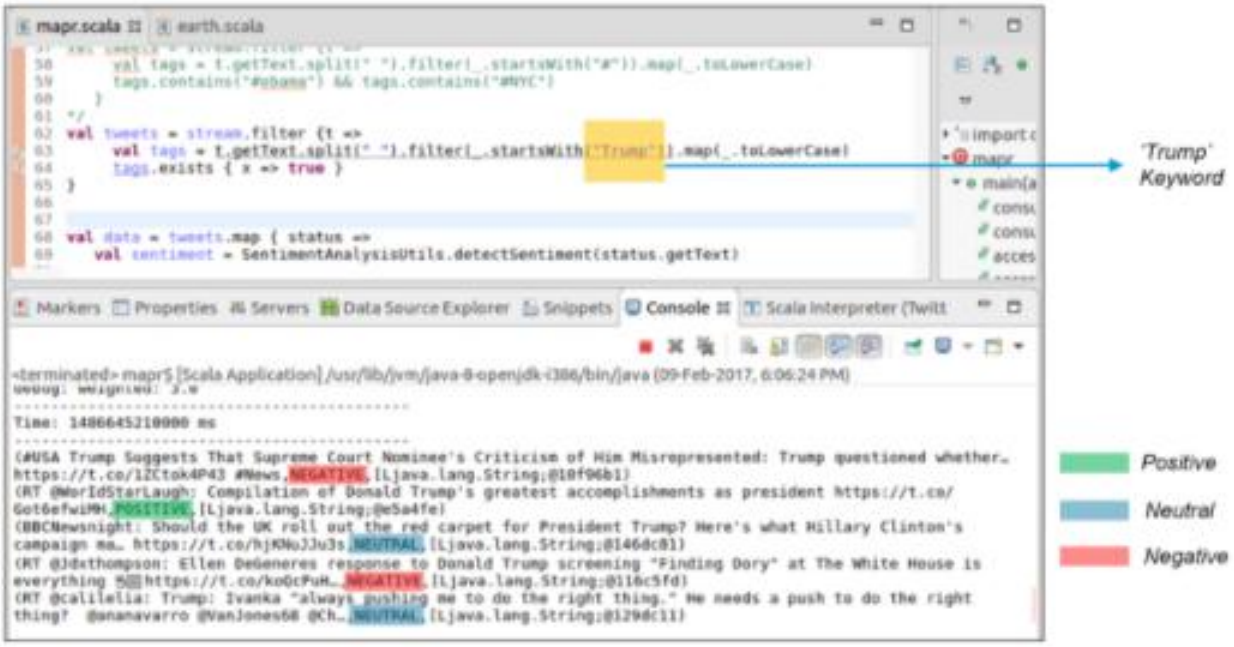


Figure 4.27 Performing Sentiment Analysis on Tweets with 'Trump' Keyword [36]

Moving Ahead:

As we have seen from our Sentiment Analysis demonstration, we can extract sentiments of particular topics just like we did for 'Trump'. Similarly, Sentiment Analytics can be used in crisis management, service adjusting and target marketing by companies around the world.

Companies using Spark Streaming for Sentiment Analysis have applied the same approach to achieve the following:

1. Enhancing the customer experience
2. Gaining competitive advantage
3. Gaining Business Intelligence
4. Revitalizing a losing brand

4.3 SPARK MLLIB – MACHINE LEARNING LIBRARY OF APACHE SPARK [36]

Spark MLlib is Apache Spark's Machine Learning component. One of the major attractions of Spark is the ability to scale computation massively, and that is exactly what you need for machine learning algorithms. But the limitation is that all machine learning

algorithms cannot be effectively parallelized. Each algorithm has its own challenges for parallelization, whether it is task parallelism or data parallelism.

Having said that, Spark is becoming the de-facto platform for building machine learning algorithms and applications. The developers working on the Spark MLlib are implementing more and more machine algorithms in a scalable and concise manner in the Spark framework. Through this blog, we will learn the concepts of Machine Learning, Spark MLlib, its utilities, algorithms and a complete use case of Movie Recommendation System.

4.3.1 What is Machine Learning? [36]

Evolved from the study of pattern recognition and computational learning theory in artificial intelligence, machine learning explores the study and construction of algorithms that can learn from and make predictions on data – such algorithms overcome following strictly static program instructions by making data-driven predictions or decisions, through building a model from sample inputs.

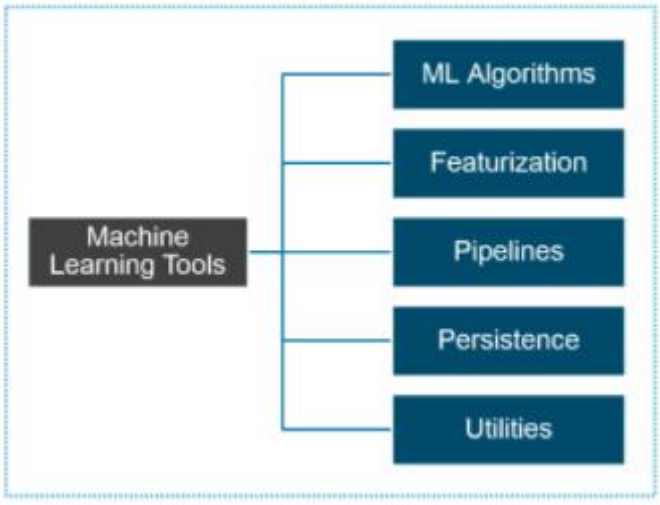


Figure 4.28 Machine Learning tools [36]

Machine learning is closely related to computational statistics, which also focuses on prediction-making through the use of computers. It has strong ties to mathematical optimization, which delivers methods, theory and application domains to the field. Within the field of data analytics, machine learning is a method used to devise complex models and algorithms that lend themselves to a prediction which in commercial use is known as predictive analytics.

There are three categories of Machine learning tasks:

1. **Supervised Learning**: Supervised learning is where you have input variables (x) and an output variable (Y) and you use an algorithm to learn the mapping function from the input to the output.
2. **Unsupervised Learning**: Unsupervised learning is a type of machine learning algorithm used to draw inferences from datasets consisting of input data without labeled responses.
3. **Reinforcement Learning**: A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). The program is provided feedback in terms of rewards and punishments as it navigates its problem space. This concept is called reinforcement learning.

4.3.2 Spark MLlib Overview [36]

Spark MLlib is used to perform machine learning in Apache Spark. MLlib consists popular algorithms and utilities.

MLlib Overview:

- spark.mllib contains the original API built on top of RDDs. It is currently in maintenance mode.
- spark.ml provides higher level API built on top of DataFrames for constructing ML pipelines. spark.ml is the primary Machine Learning API for Spark at the moment.

4.3.3 Spark MLlib Tools [36]

Spark MLlib provides the following tools:

- **ML Algorithms**: ML Algorithms form the core of MLlib. These include common learning algorithms such as classification, regression, clustering and collaborative filtering.
- **Featurization**: Featurization includes feature extraction, transformation, dimensionality reduction and selection.
- **Pipelines**: Pipelines provide tools for constructing, evaluating and tuning ML Pipelines.

- **Persistence**: Persistence helps in saving and loading algorithms, models and Pipelines.
- **Utilities**: Utilities for linear algebra, statistics and data handling.

4.3.4 MLlib Algorithms [36]

The popular algorithms and utilities in Spark MLlib are:

1. Basic Statistics
2. Regression
3. Classification
4. Recommendation System
5. Clustering
6. Dimensionality Reduction
7. Feature Extraction
8. Optimization

Let us look at some of these in detail.

Basic Statistics
Basic Statistics includes the most basic of machine learning techniques. These include:

1. **Summary Statistics**: Examples include mean, variance, count, max, min and numNonZeros.
2. **Correlations**: Spearman and Pearson are some ways to find correlation.
3. **Stratified Sampling**: These include sampleBykey and sampleByKeyExact.
4. **Hypothesis Testing**: Pearson's chi-squared test is an example of hypothesis testing.
5. **Random Data Generation**: RandomRDDs, Normal and Poisson are used to generate random data.

Regression
Regression analysis is a statistical process for estimating the relationships among variables. It includes many techniques for modeling and analyzing several variables when the focus is on the relationship between a dependent variable and one or more independent variables. More specifically, regression analysis helps one understand how the typical value of the dependent variable changes when any one of the independent variables is varied, while the other independent variables are held fixed.

Regression analysis is widely used for prediction and forecasting, where its use has substantial overlap with the field of machine learning. Regression analysis is also used to

understand which among the independent variables are related to the dependent variable, and to explore the forms of these relationships. In restricted circumstances, regression analysis can be used to infer causal relationships between the independent and dependent variables.

Classification

Classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known. It is an example of pattern recognition.

Here, an example would be assigning a given email into "spam" or "non-spam" classes or assigning a diagnosis to a given patient as described by observed characteristics of the patient (gender, blood pressure, presence or absence of certain symptoms, etc.).

Recommendation System

A recommendation system is a subclass of information filtering system that seeks to predict the "rating" or "preference" that a user would give to an item. Recommender systems have become increasingly popular in recent years, and are utilized in a variety of areas including movies, music, news, books, research articles, search queries, social tags, and products in general.

Recommender systems typically produce a list of recommendations in one of two ways – through collaborative and content-based filtering or the personality-based approach.

- **Collaborative Filtering** approaches building a model from a user's past behavior (items previously purchased or selected and/or numerical ratings given to those items) as well as similar decisions made by other users. This model is then used to predict items (or ratings for items) that the user may have an interest in.
- **Content-Based Filtering** approaches utilize a series of discrete characteristics of an item in order to recommend additional items with similar properties.

Further, these approaches are often combined as Hybrid Recommender Systems.

Clustering

Clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). So, it is the main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine

learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression and computer graphics.

Dimensionality Reduction
Dimensionality Reduction is the process of reducing the number of random variables under consideration, via obtaining a set of principal variables. It can be divided into feature selection and feature extraction.

- **Feature Selection**: Feature selection finds a subset of the original variables (also called features or attributes).
- **Feature Extraction**: This transforms the data in the high-dimensional space to a space of fewer dimensions. The data transformation may be linear, as in Principal Component Analysis(PCA), but many nonlinear dimensionality reduction techniques also exist.

Feature Extraction
Feature Extraction starts from an initial set of measured data and builds derived values (features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations. This is related to dimensionality reduction.

Optimization
Optimization is the selection of the best element (with regard to some criterion) from some set of available alternatives.

In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function. The generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics. More generally, optimization includes finding "best available" values of some objective function given a defined domain (or input), including a variety of different types of objective functions and different types of domains.

4.3.5 Use Case – Movie Recommendation System [36]

Problem Statement: To build a Movie Recommendation System which recommends movies based on a user's preferences using Apache Spark.

Our Requirements:
So, let us assess the requirements to build our movie recommendation system:

1. Process huge amount of data
2. Input from multiple sources
3. Easy to use
4. Fast processing

As we can assess our requirements, we need the best Big Data tool to process large data in short time. Therefore, Apache Spark is the perfect tool to implement our Movie Recommendation System.

Let us now look at the Flow Diagram for our system.

Figure 4.29 Flow Diagram for the system [36]

As we can see, the following uses Streaming from Spark Streaming. We can stream in real time or read data from Hadoop HDFS.

Getting Dataset:

For our Movie Recommendation System, we can get user ratings from many popular websites like IMDB, Rotten Tomatoes and Times Movie Ratings. This dataset is available in many formats such as CSV files, text files and databases. We can either stream the data live from the websites or download and store them in our local file system or HDFS.

Dataset:

The below figure shows how we can collect dataset from popular websites.

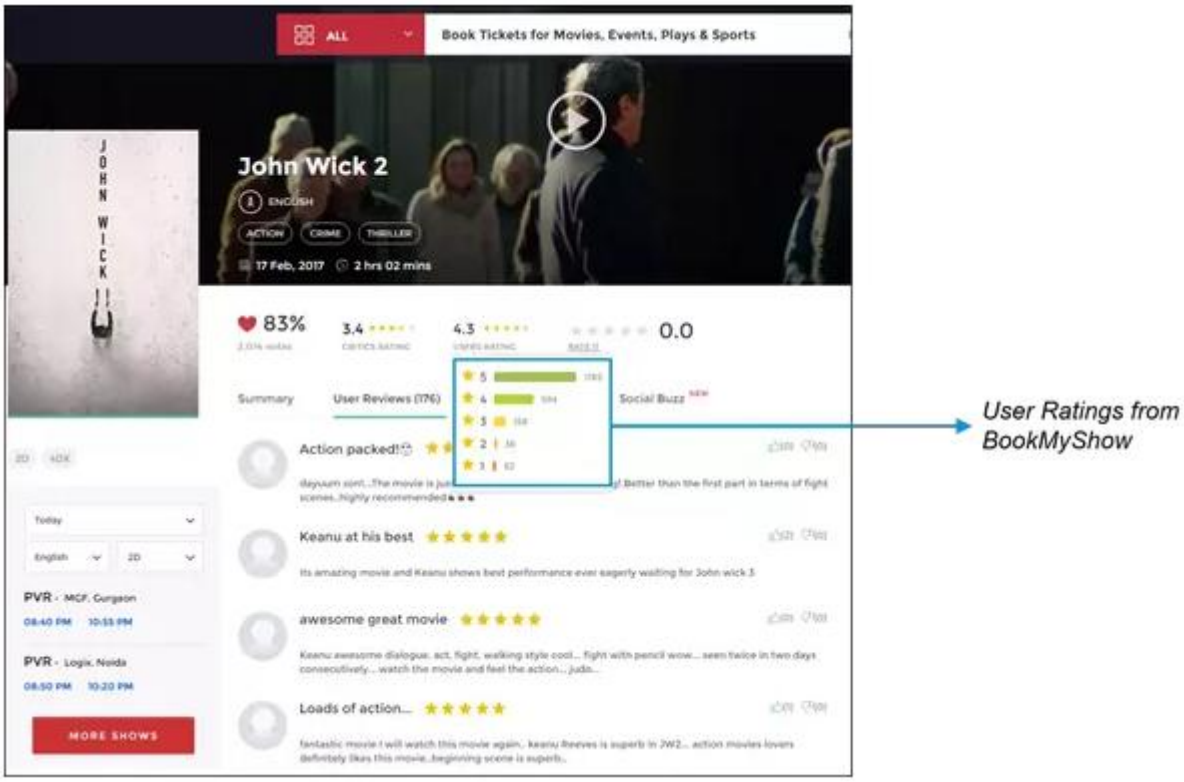
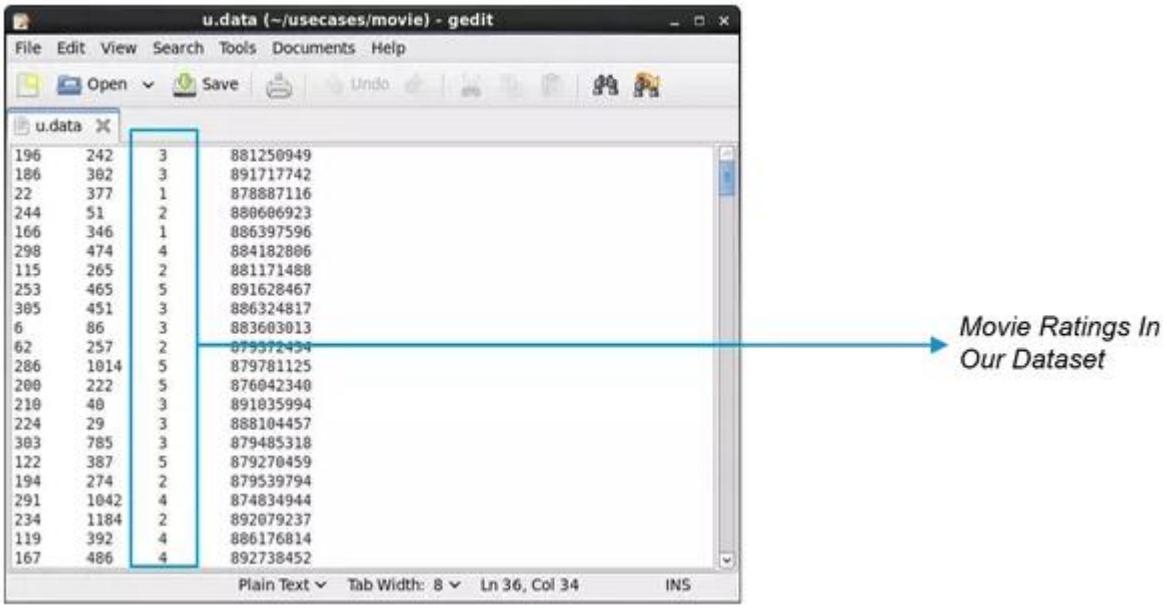


Figure 4.30 How to collect dataset from popular websites [36]

Once we stream the data into Spark, it looks somewhat like this.



Machine Learning:

The whole recommendation system is based on Machine Learning algorithm Alternating Least Squares. Here, ALS is a type of regression analysis where regression is used to draw a line amidst the data points in such a way so that the sum of the squares of the distance from each data point is minimized. Thus, this line is then used to predict the values of the function where it meets the value of the independent variable.

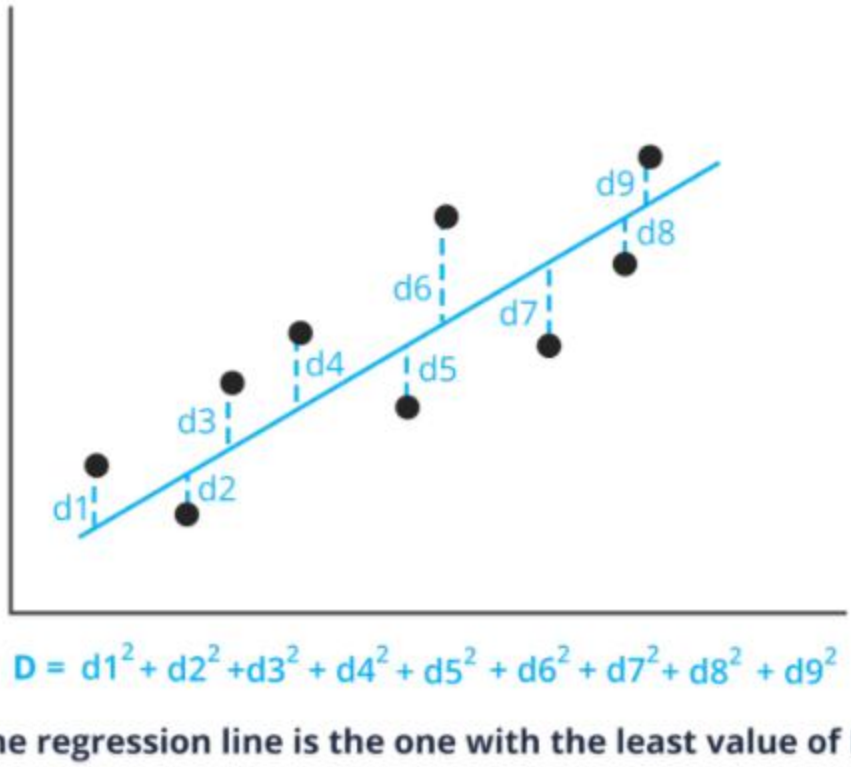


Figure 4.31 Machine Learning Algorithm – Regression Alternating Least Squares [36]

The blue line in the diagram is the best-fit regression line. For this line, the value of the dimension D is minimum. All other red lines will always be farther from the dataset as a whole.

Spark MLlib Implementation:

1. We will use Collaborative Filtering(CF) to predict the ratings for users for particular movies based on their ratings for other movies.
2. We then collaborate this with other users' rating for that particular movie.
3. To get the following results from our Machine Learning, we need to use Spark SQL's DataFrame, Dataset and SQL Service.

Here is the pseudo code for our program:

```
import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.Rating
import org.apache.spark.SparkConf
//Import other necessary packages

object Movie {
def main(args: Array[String]) {

val conf = new SparkConf().setAppName("Movie").setMaster("local[2]")
val sc = new SparkContext(conf)
val rawData = sc.textFile(" *Read Data from Movie CSV file* ")

//rawData.first()
val rawRatings = rawData.map( *Split rawData on tab delimiter* )
val ratings = rawRatings.map { *Map case array of User, Movie and Rating* }

//Training the data
val model = ALS.train(ratings, 50, 5, 0.01)
model.userFeatures
model.userFeatures.count
model.productFeatures.count
val predictedRating = *Predict for User 789 for movie 123*
val userId = *User 789*
val K = 10
val topKRecs = model.recommendProducts( *Recommend for User for the particular
value of K* )
println(topKRecs.mkString("\n"))
val movies = sc.textFile(" *Read Movie List Data* ")
val titles = movies.map(line => line.split("\\|").take(2)).map(array =>
(array(0).toInt,array(1))).collectAsMap()
val titlesRDD = movies.map(line => line.split("\\|").take(2)).map(array =>
(array(0).toInt,array(1))).cache()
titles(123)
val moviesForUser = ratings.*Search for User 789*
val sqlContext= *Create SQL Context*
val moviesRecommended = sqlContext.*Make a DataFrame of recommended movies*
moviesRecommended.registerTempTable("moviesRecommendedTable")
sqlContext.sql("Select count(*) from moviesRecommendedTable").foreach(println)
moviesForUser. *Sort the ratings for User 789* .map( *Map the rating to movie title* ).
*Print the rating*
val results = moviesForUser.sortBy(-_.rating).take(30).map(rating =>
(titles(rating.product), rating.rating))
 }
}
```

The full source code of Movie Recommendation System using Spark MLlib is available in [42].

Once we generate predictions, we can use Spark SQL to store the results into an RDBMS system. Further, this can be displayed on a web application.

Results:

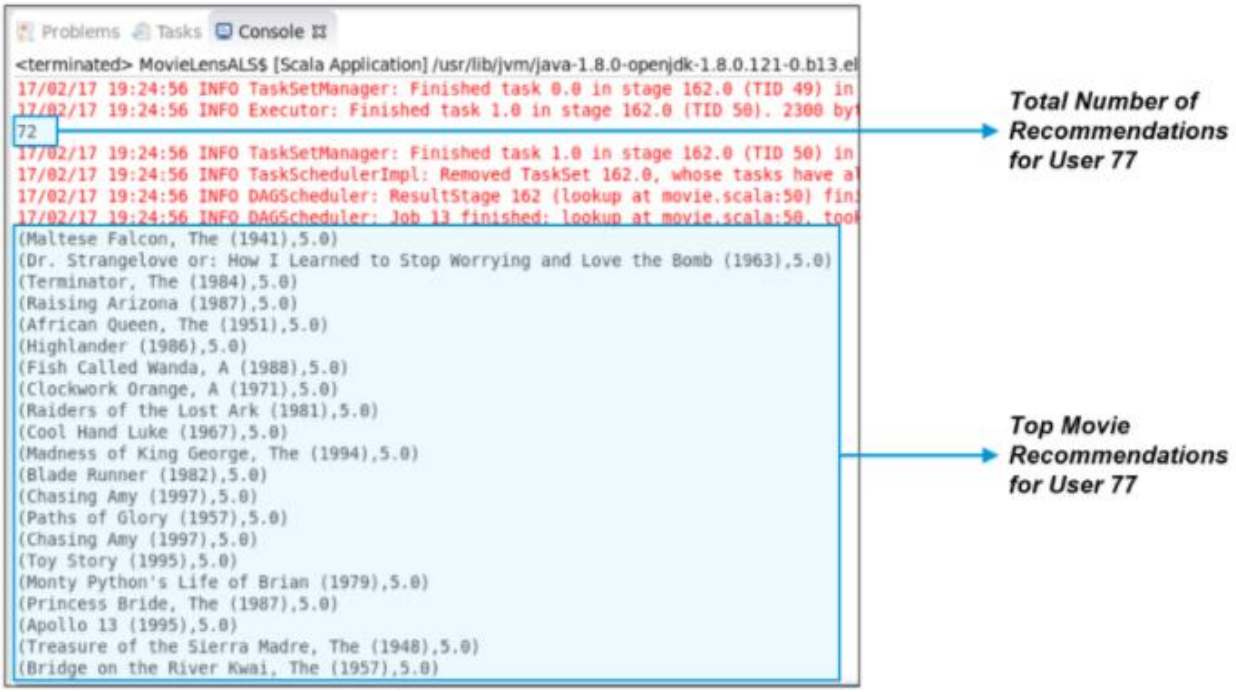


Figure 4.32 Movies recommended for User 77 [36]

4.4 SPARK SQL TUTORIAL – UNDERSTANDING SPARK SQL WITH EXAMPLES [36]

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 blog, 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.

4.4.1 Why Spark SQL Came Into Picture? [36]

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.

4.4.2 Limitations with Hive: [36]

- 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.

4.4.3 Spark SQL Overview [36]

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 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.

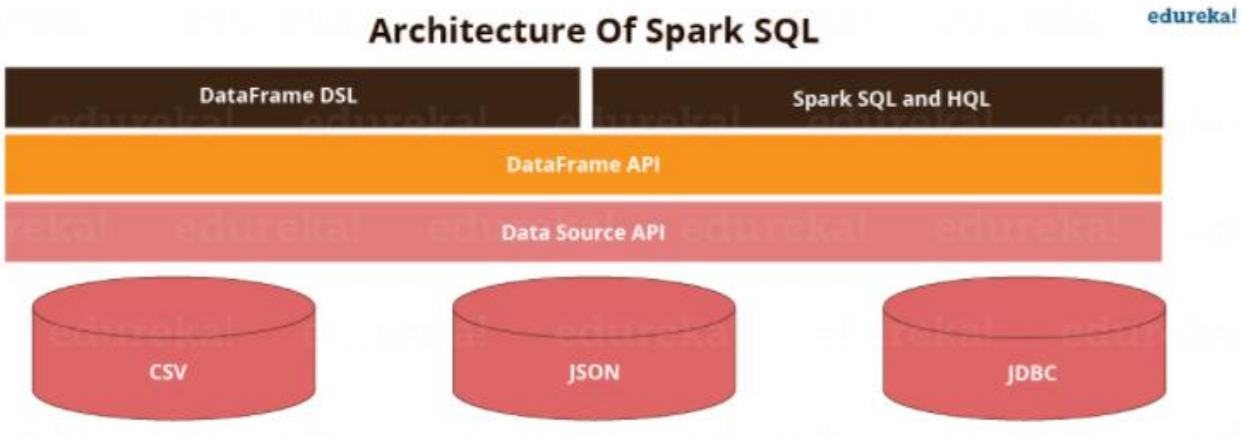


Figure 4.33 Architecture of Spark SQL. [36]

4.4.4 Spark SQL Libraries [36]

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

1. **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.

2. **DataFrame API:**

A DataFrame is a distributed collection of data organized into named column. It is equivalent to a relational table in SQL used for storing data into tables.

- It is a Data Abstraction and Domain Specific Language (DSL) applicable on structure and semi structured data.
- DataFrame API is 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.

3. 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.

4. 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.

4.4.5 Features Of Spark SQL [36]

The following are the features of Spark SQL:

1. 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.

2. 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.

3. 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.

4. Standard Connectivity

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

5. 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 upto 100x times faster than Hadoop.

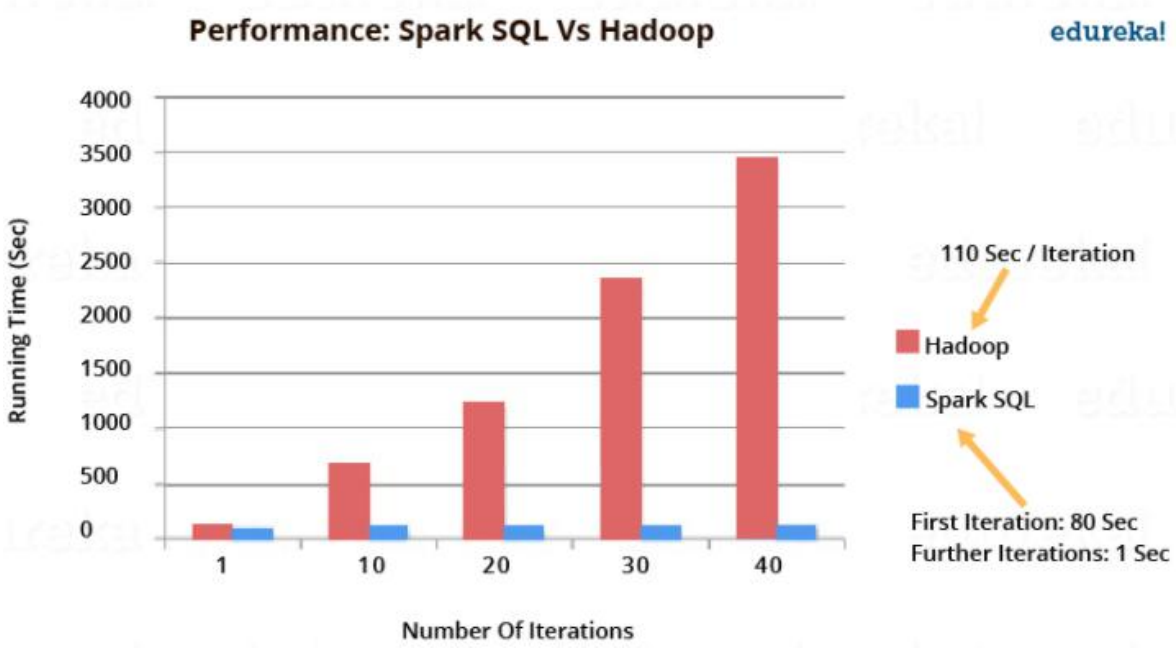


Figure 4.34 Runtime of Spark SQL vs Hadoop. Spark SQL is faster
Source: Cloudera Apache Spark Blog. [36]

6. 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'.

```
1  val dataset = Seq((0, "hello"),(1, "world")).toDF("id","text")
2  val upper: String => String =_.toUpperCase
3  import org.apache.spark.sql.functions.udf
4  val upperUDF = udf(upper)
5  dataset.withColumn("upper", upperUDF('text')).show
```

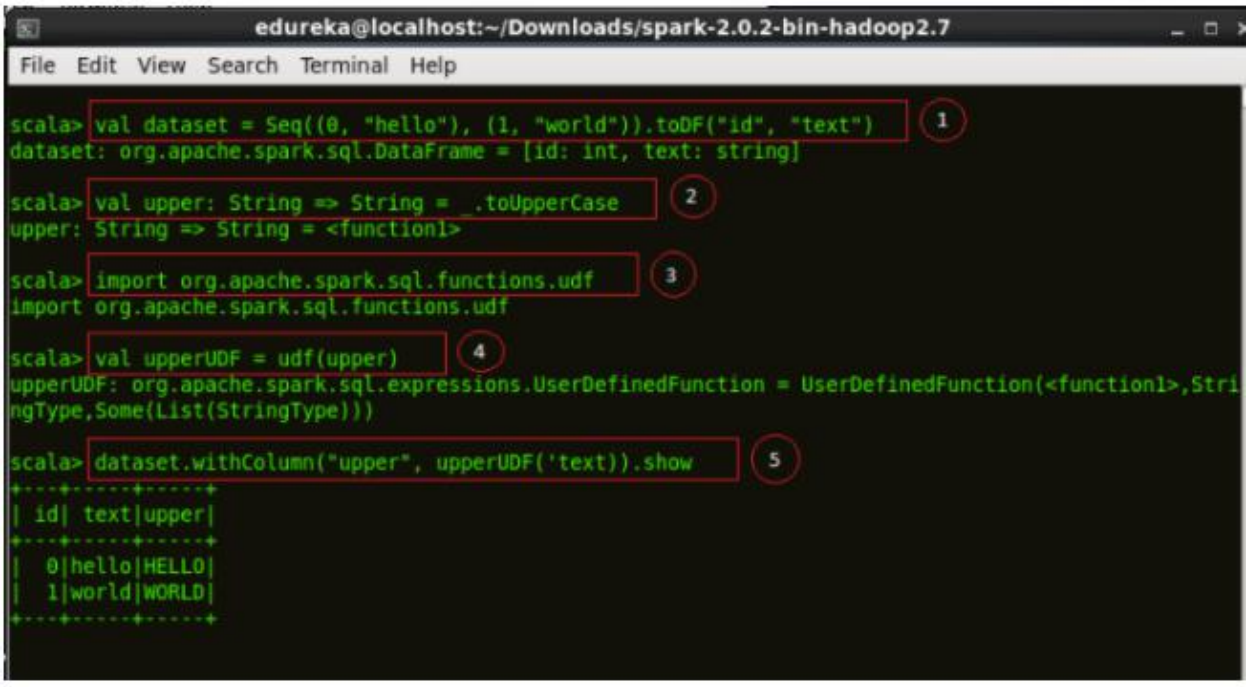


Figure 4.35 Demonstration of a User Defined Function, upperUDF. [36]

Code explanation:

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

```
1  spark.udf.register("myUpper", (input:String) => input.toUpperCase)
2  spark.catalog.listFunctions.filter('name like "%upper%").show(false)
```

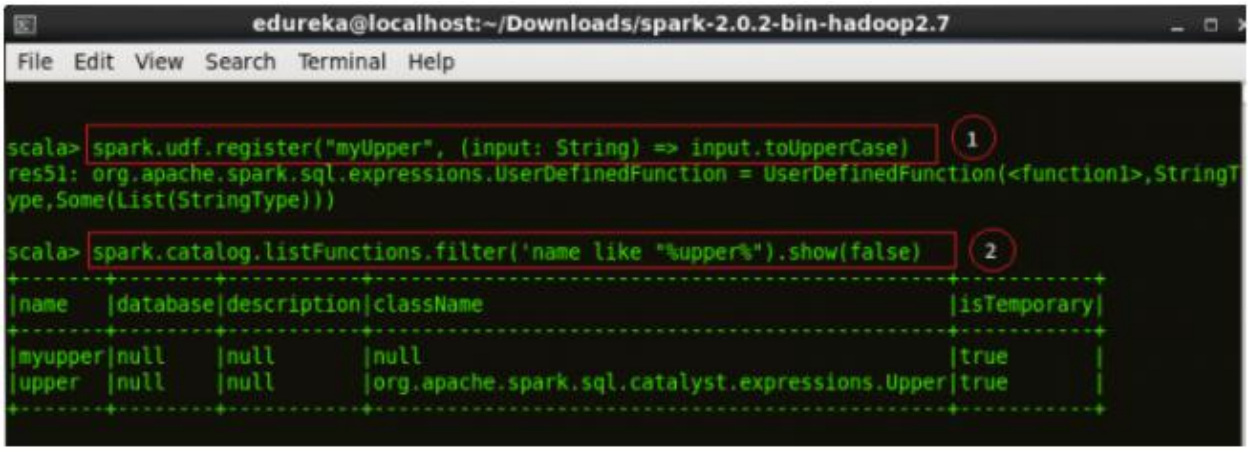


Figure 4.36 Results of the User Defined Function, upperUDF. [36]

4.4.6 Querying Using Spark SQL [36]

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 here, 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.

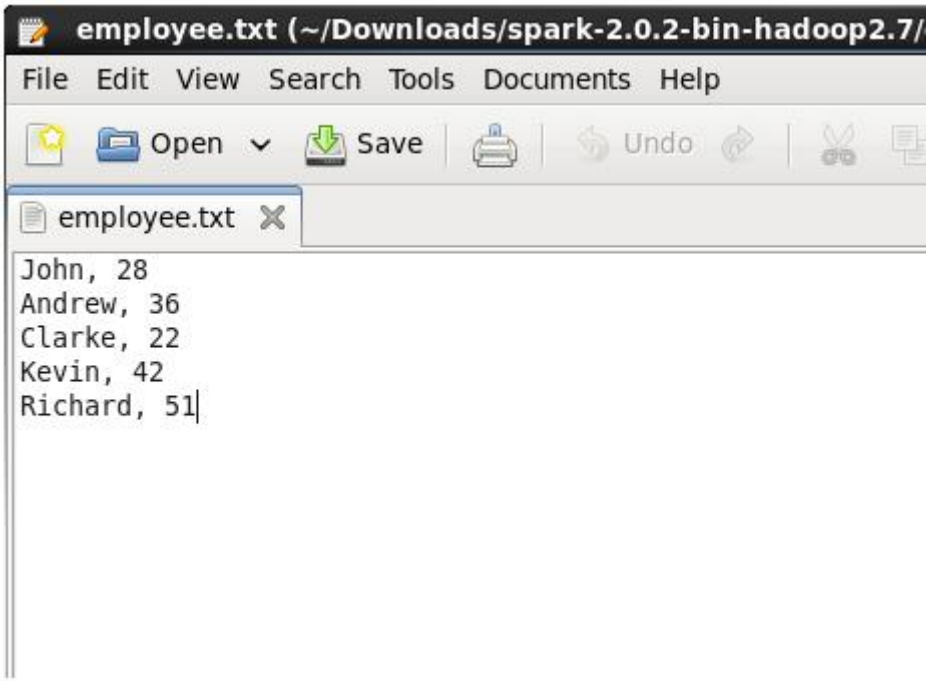


Figure 4.37 Contents of employee.txt. [36]

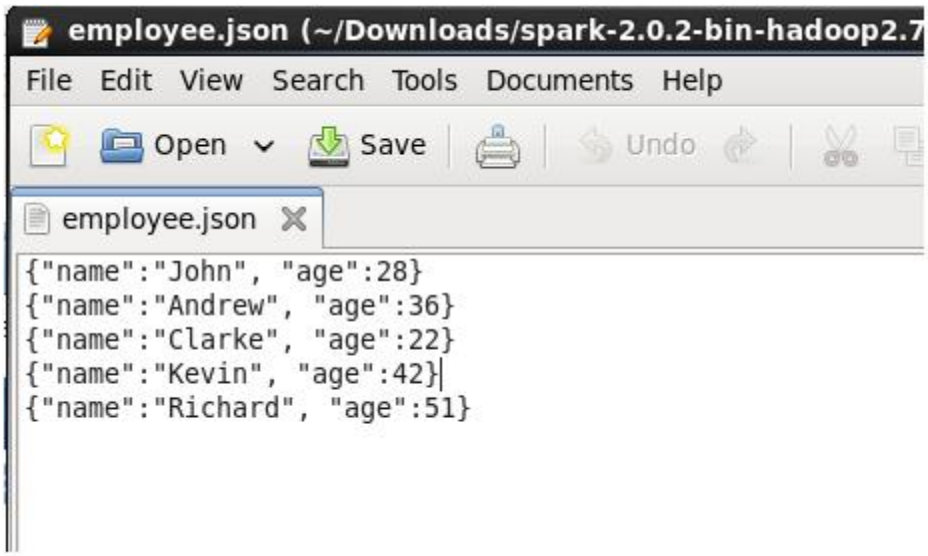


Figure 4.38 Contents of employee.json. [36]

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()
```

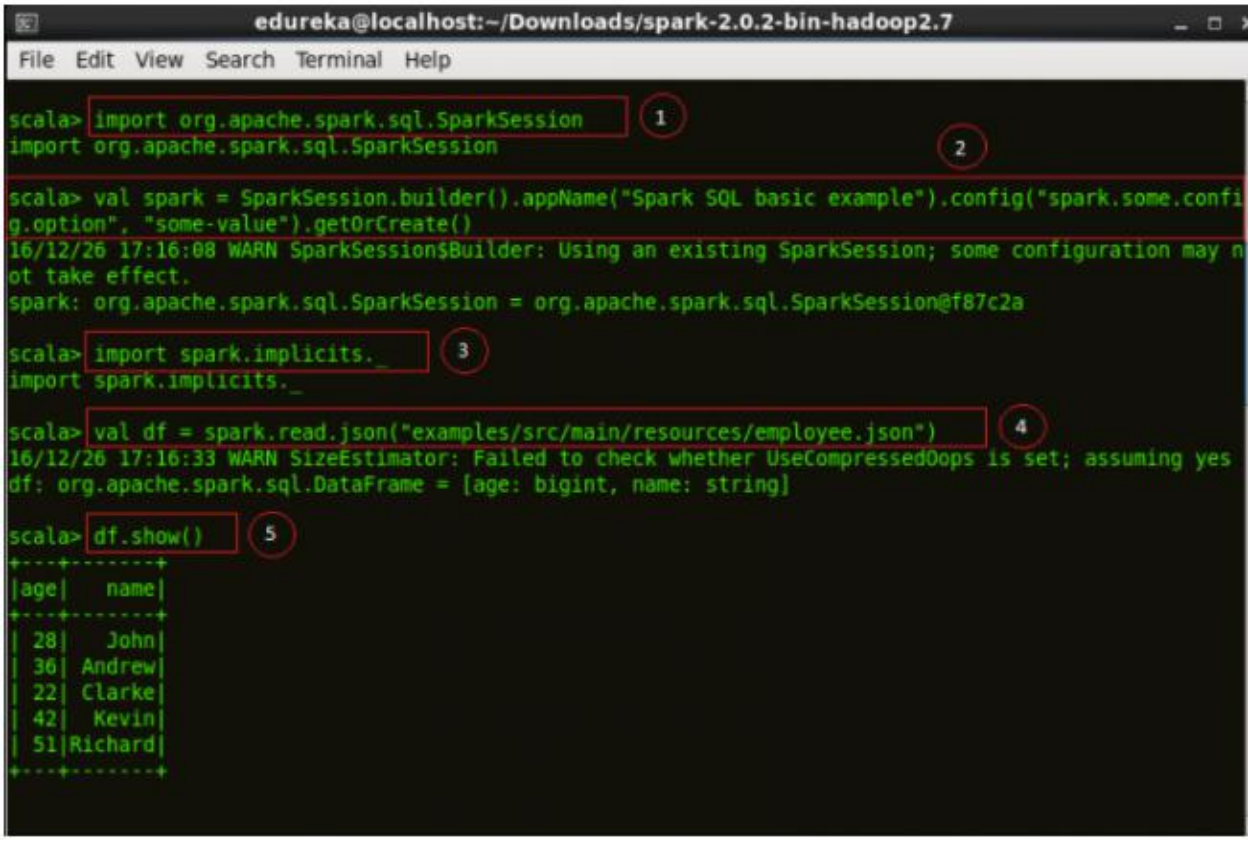


Figure 4.39 Starting a Spark Session and displaying DataFrame of employee.json. [36]

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.

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

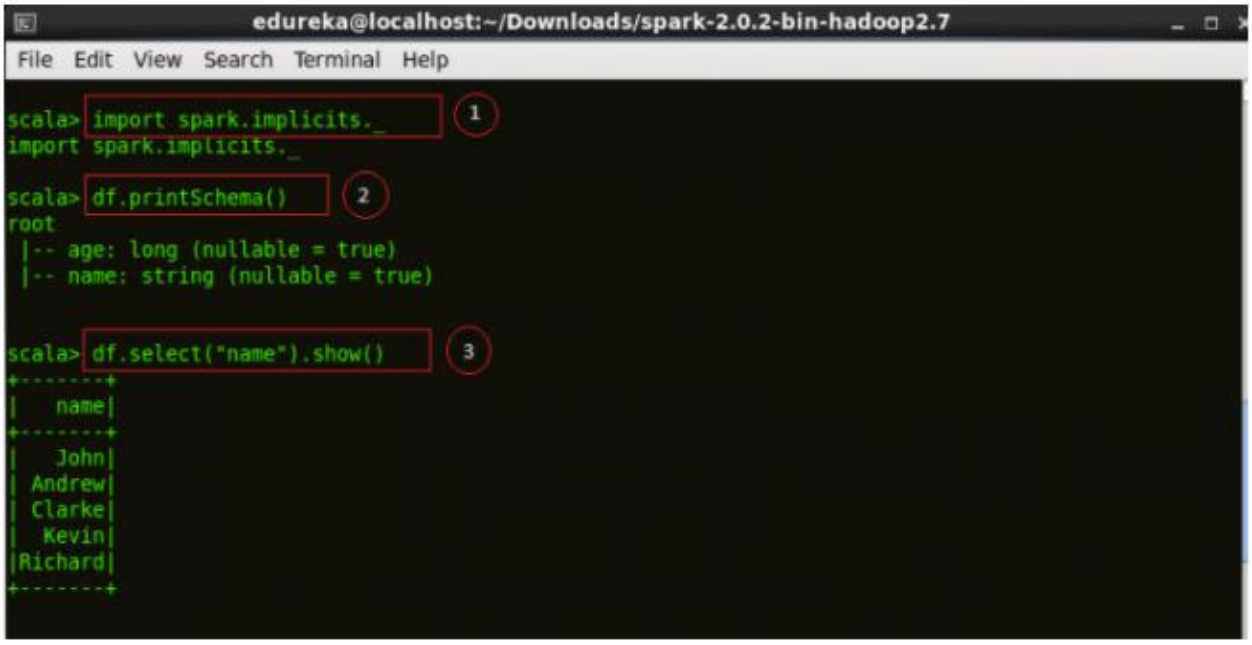


Figure 4.40 Schema of a DataFrame. [36]

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.

```
1  df.select($"name", $"age" + 2).show()
2  df.filter($"age" > 30).show()
```

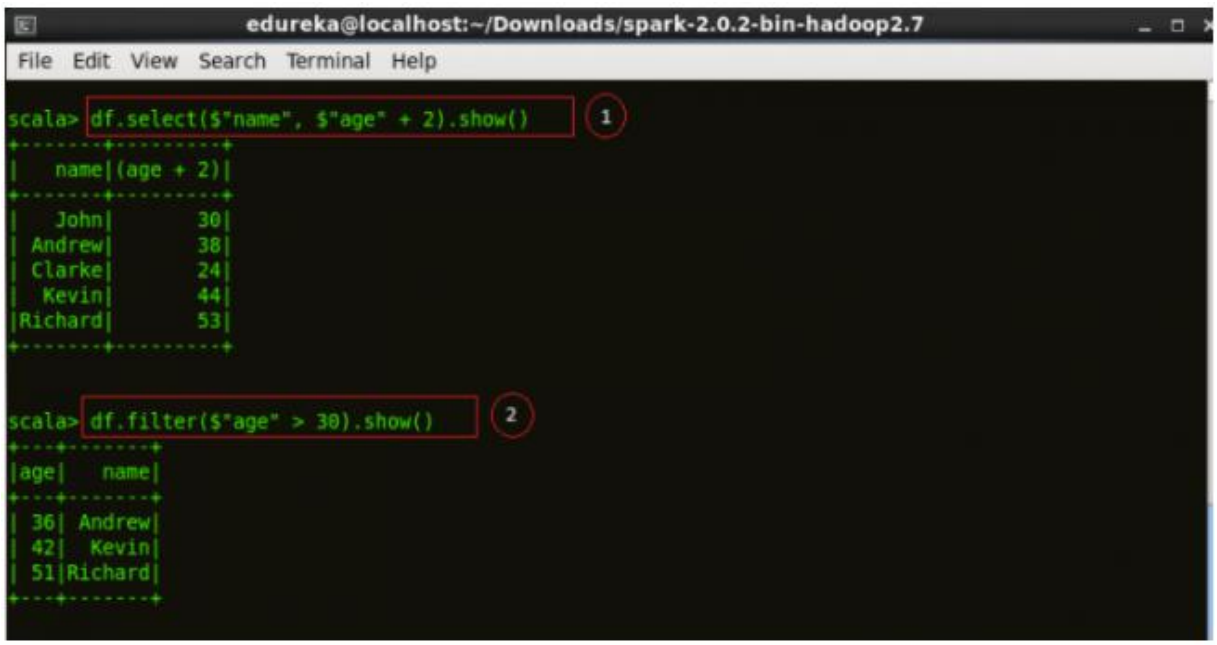


Figure 4.41 Basic SQL operations on employee.json. [36]

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()
```

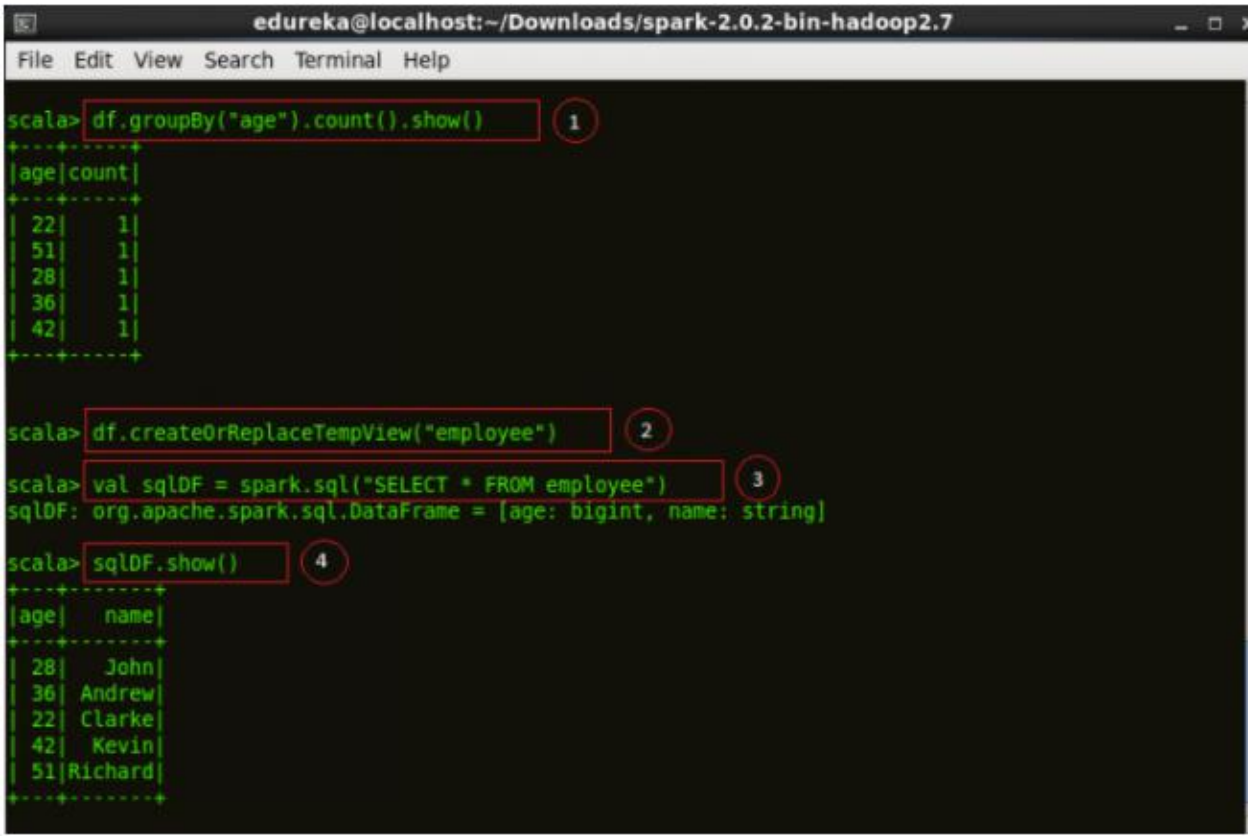


Figure 4.42 SQL operations on employee.json. [36]

4.4.7 Creating Datasets [36]

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 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 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()
```

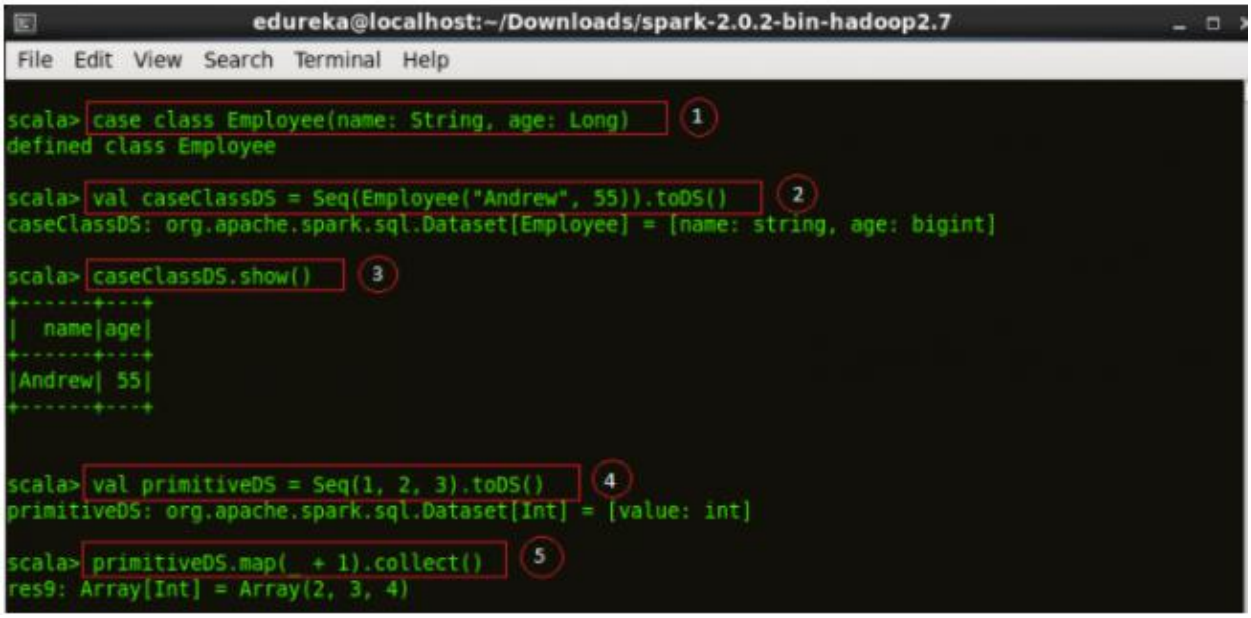


Figure 4.43 Creating a Dataset. [36]

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()
```

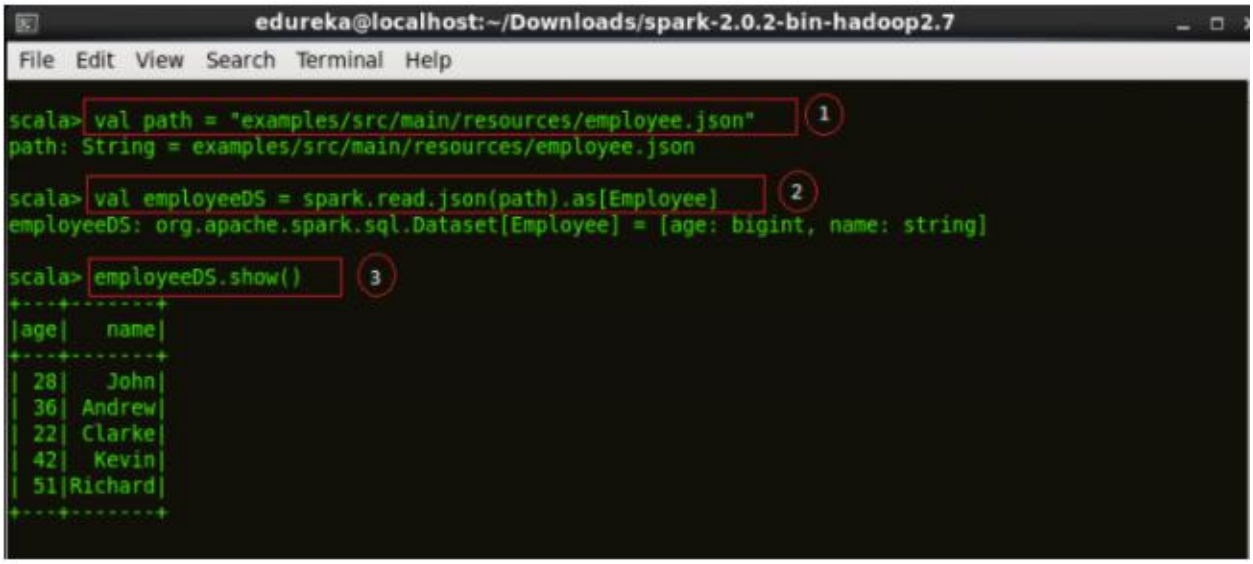


Figure 4.44 Creating a Dataset from a JSON file. [36]

4.4.8 Adding Schema To RDDs [36]

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 => 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 => "Name: " + youngster(0)).show()
```

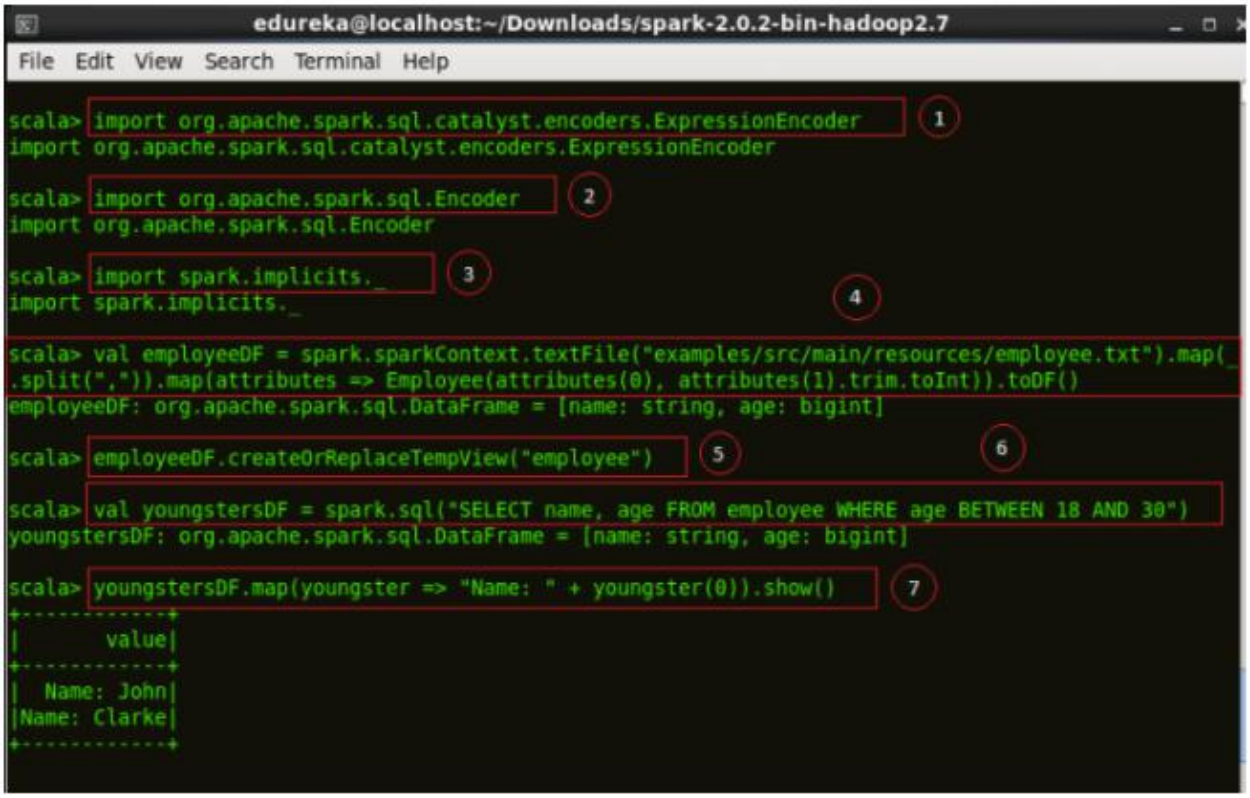


Figure 4.45 Creating a DataFrame for transformations. [36]

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

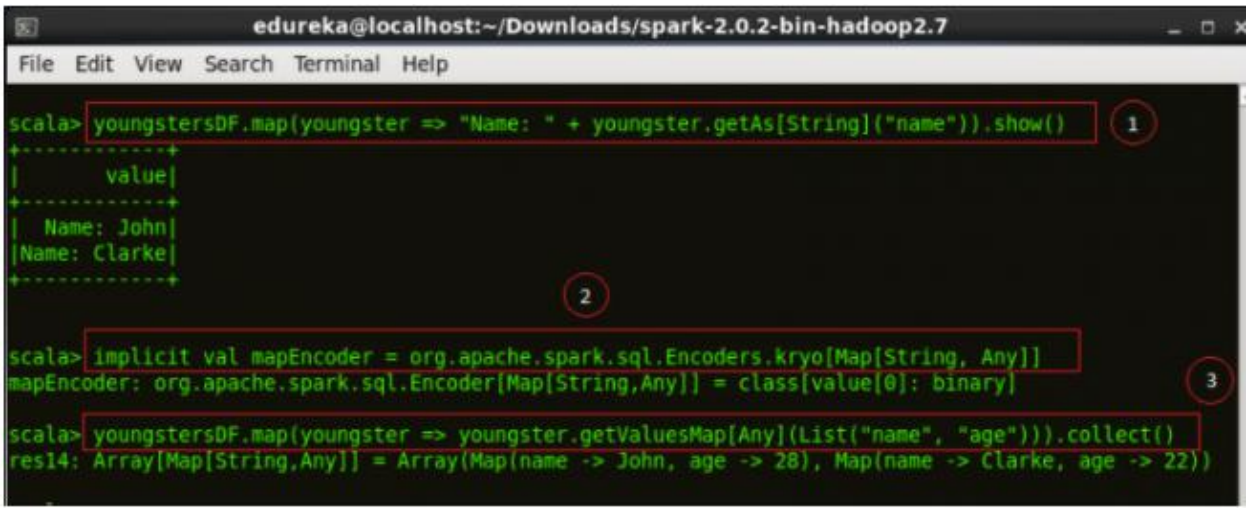


Figure 4.46 Mapping using DataFrames. [36]

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 first action, Spark would only process and return the result for the first line, rather than do the work for the entire file.

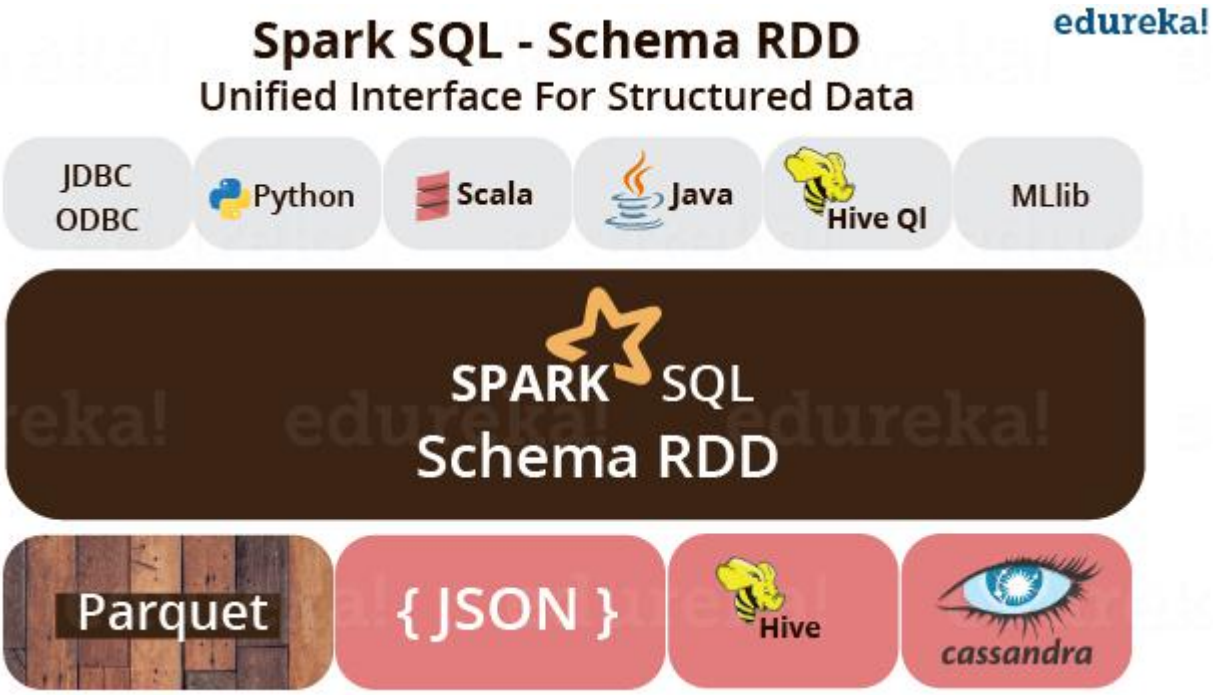


Figure 4.47 Ecosystem of Schema RDD in Spark SQL. [36]

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.

4.4.9 RDDs As Relations [36]

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 a 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 => StructField(fieldName,
StringType, nullable = true))
val schema = StructType(fields)
```

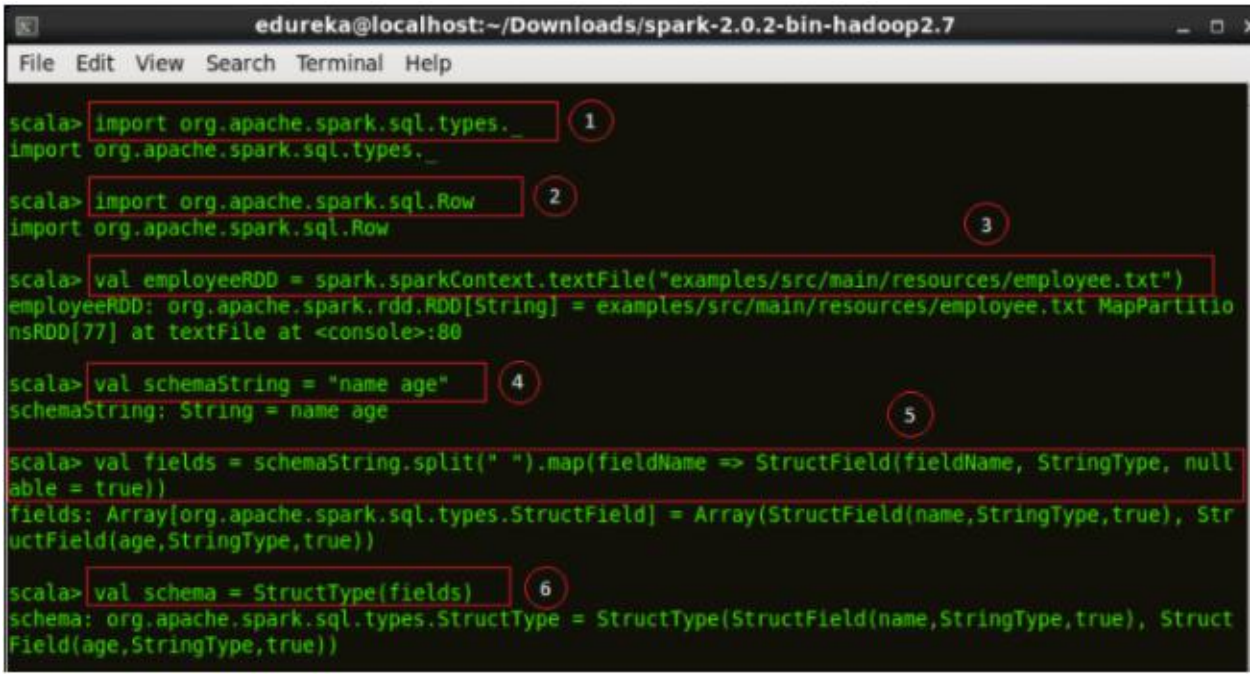


Figure 4.48 Specifying Schema for RDD transformation. [36]

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.

```
1 val rowRDD = employeeRDD.map(_.split(",")).map(attributes => Row(attributes(0), attributes(1).trim))
2 val employeeDF = spark.createDataFrame(rowRDD, schema)
3 employeeDF.createOrReplaceTempView("employee")
4 val results = spark.sql("SELECT name FROM employee")
5 results.map(attributes => "Name: " + attributes(0)).show()
```

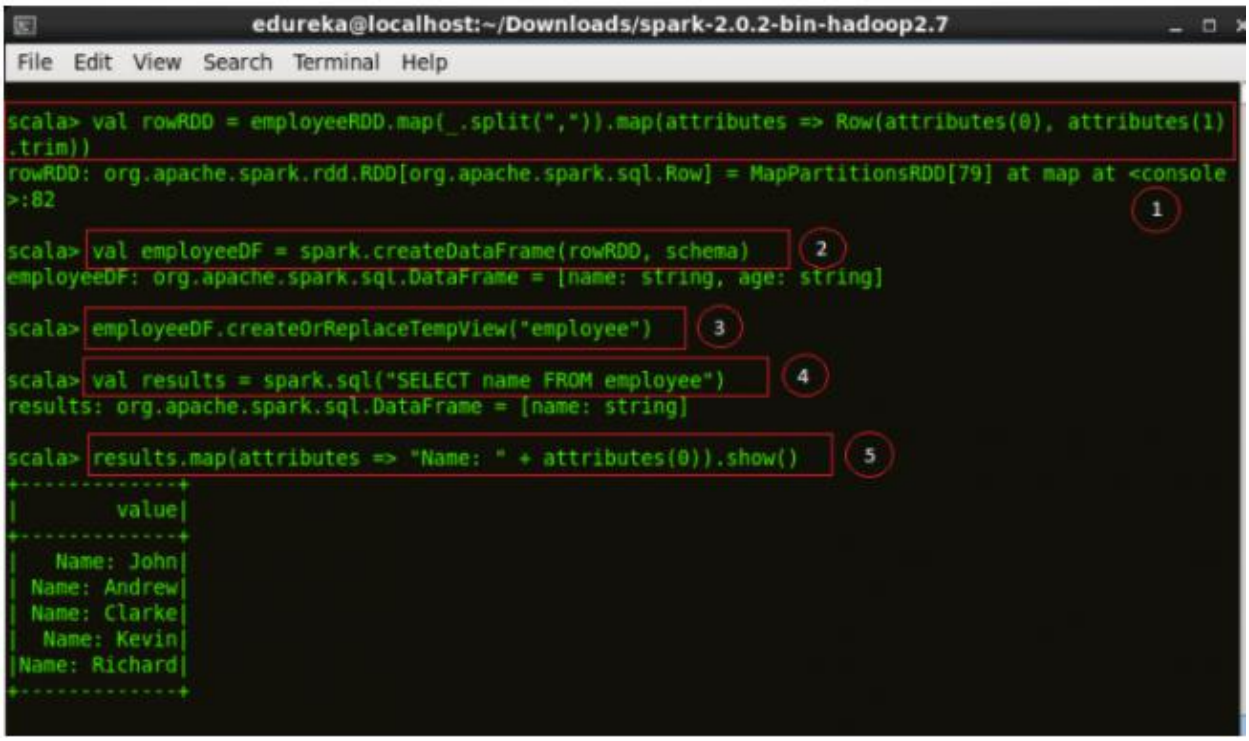


Figure 4.49 Result of RDD transformation. [36]

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

4.4.10 Caching Tables In-Memory [36]

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

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

4.4.11 Loading Data Programmatically [36]

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")
```

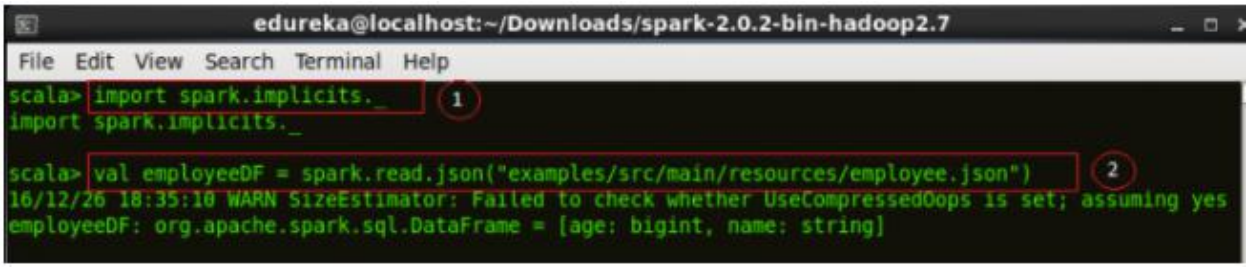


Figure 4.50 Loading a JSON file into DataFrame. [36]

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 => "Name: " + attributes(0)).show()
```

Figure 4.51 Displaying results from a Parquet DataFrame. [36]

4.4.12 JSON Datasets [36]

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.

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

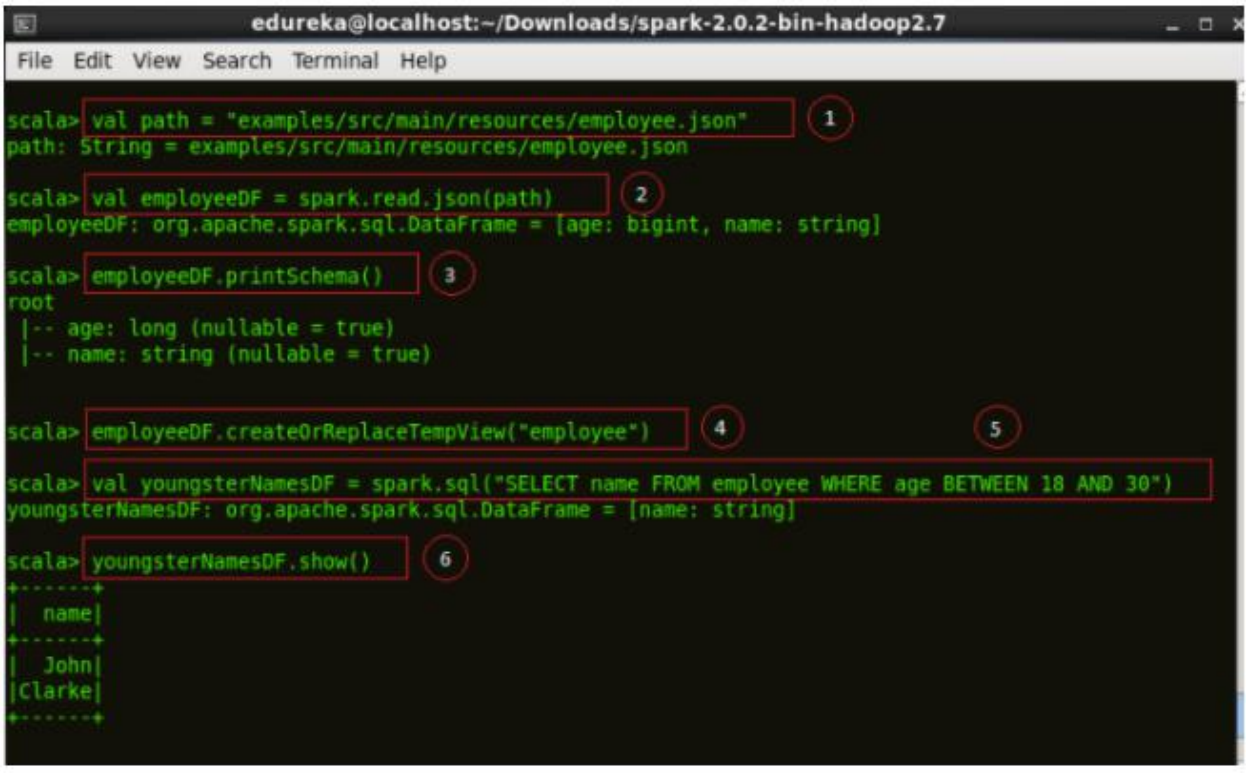


Figure 4.52 Operations on JSON Datasets. [36]

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()
```

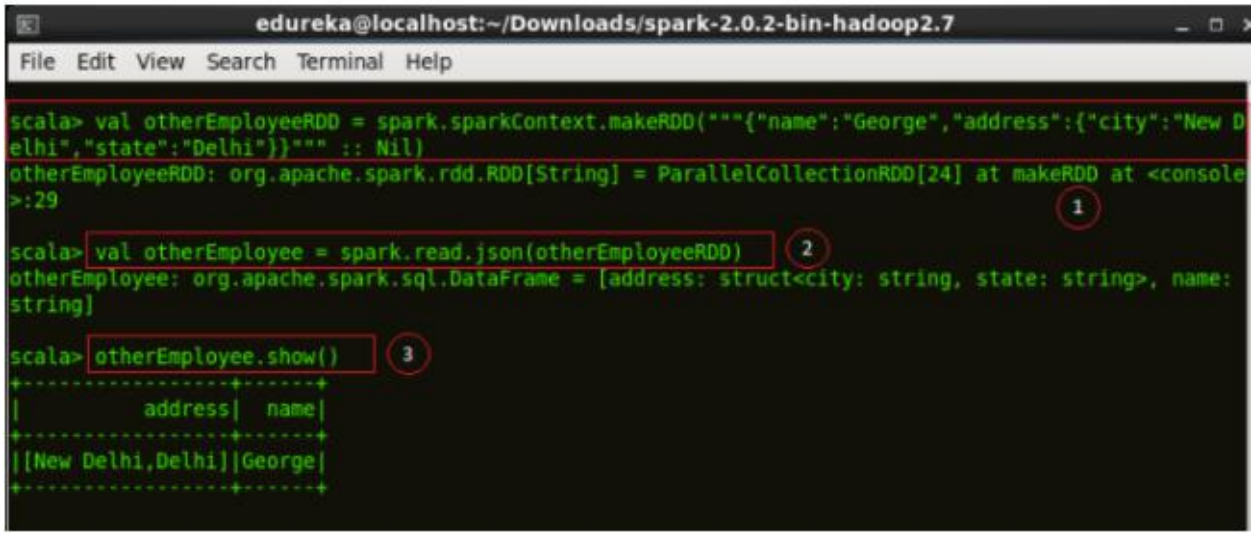


Figure 4.53 RDD transformations on JSON Dataset. [36]

4.4.13 Hive Tables [36]

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)")
```

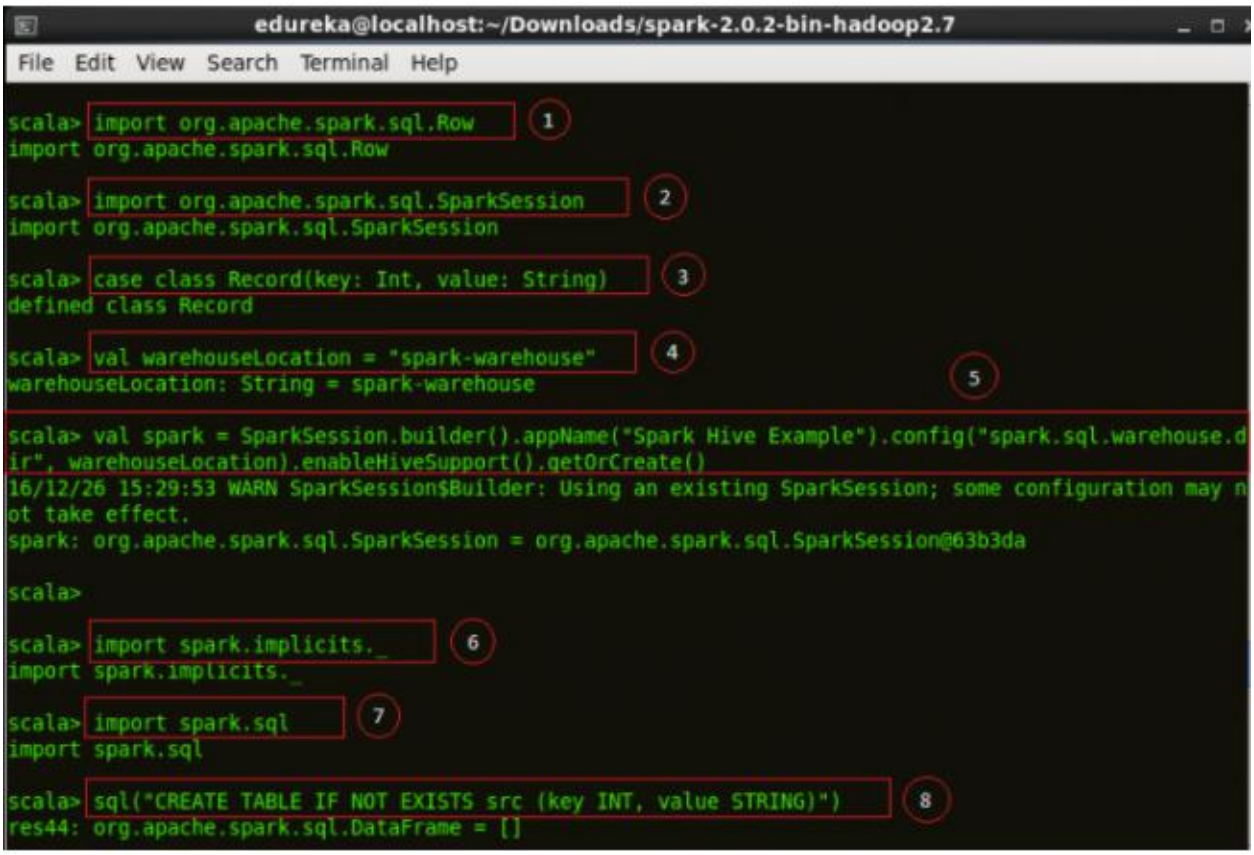


Figure 4.54 Building a Session for Hive. [36]

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.

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

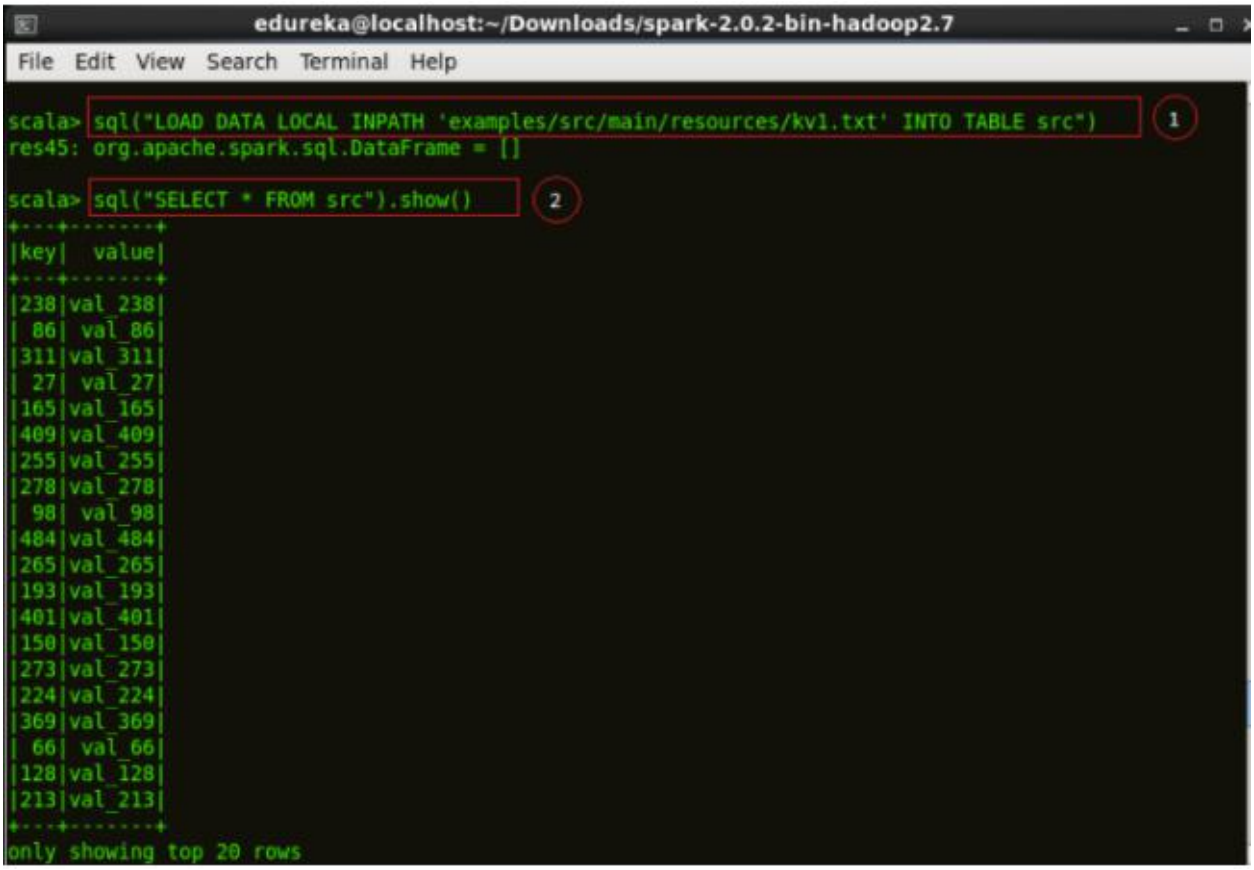


Figure 4.55 Selection using Hive tables. [36]

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 < 10 ORDER BY key")
val stringsDS = sqlDF.map {case Row(key: Int, value: String) => s"Key: $key, Value:
$value"}
stringsDS.show()
```

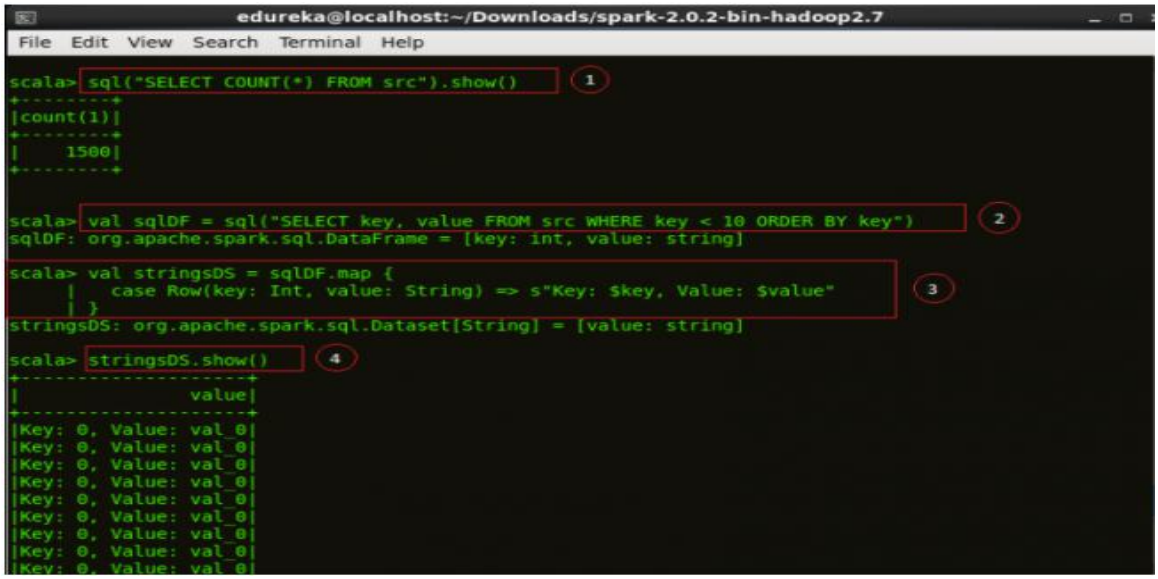


Figure 4.56 Creating DataFrames from Hive tables. [36]

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.

```
1  val recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, s"val_$i")))
2  recordsDF.createOrReplaceTempView("records")
3  sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
```

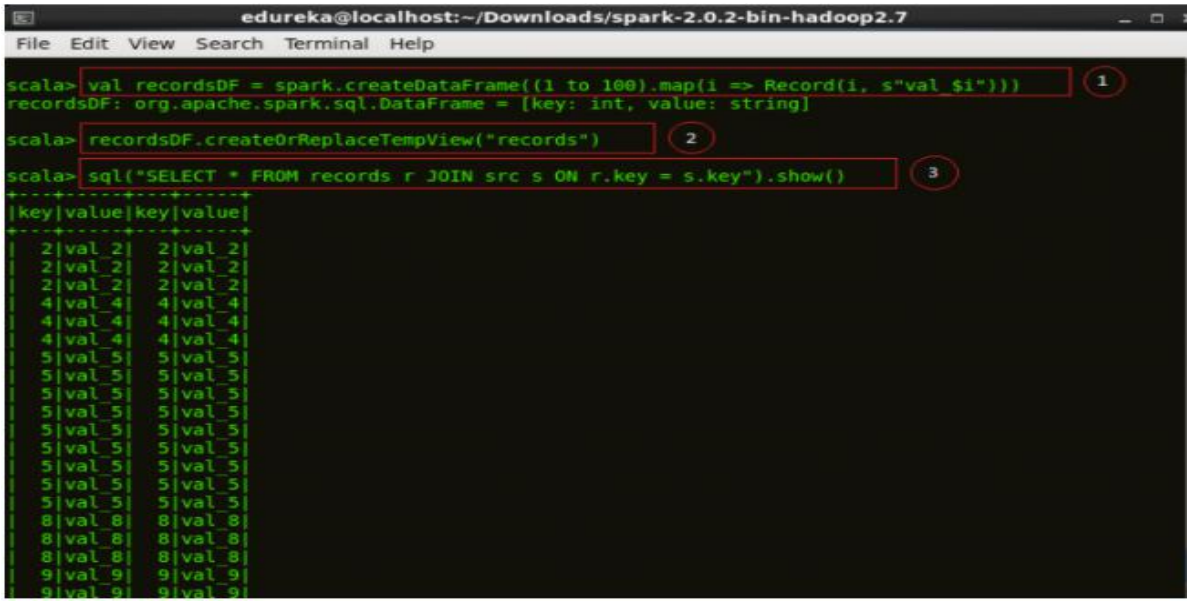


Figure 4.57 Recording the results of Hive operations. [36]

REFERENCES:

1. Agira corporate blog [Online]. Available:< http://www.agiratech.com/introduction-to-big-data-analytics/> [Accessed: 10-Feb-2018].
2. Tutorialspoints [Online]. Available:< https://www.tutorialspoint.com/big_data_analytics/index.htm> [Accessed: 10-Feb-2018]
3. Bart Baesens, Analytics in a Big Data World: The Essential Guide to Data Science and its Applications (Wiley and SAS Business Series), Wiley; 1 edition, May 19, 2014
4. J. Han and M. Kamber, Data Mining: Concepts and Techniques, 2nd ed. (Morgan Kaufmann, Waltham, MA, US, 2006); D. J. Hand, H. Mannila, and P. Smyth, Principles of Data Mining (MIT Press, Cambridge, Massachusetts, London, England, 2001); P. N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining (Pearson, Upper Saddle River, New Jersey, US, 2006).
5. D. Martens, J. Vanthienen, W. Verbeke, and B. Baesens, "Performance of Classification Models from a User Perspective." Special issue, Decision Support Systems 51, no. 4 (2011): 782–793.
6. J. Banasik, J. N. Crook, and L. C. Thomas, "Sample Selection Bias in Credit Scoring Models" in Proceedings of the Seventh Conference on Credit Scoring and Credit Control (Edinburgh University, 2001).
7. R. J. A. Little and D. B. Rubin, Statistical Analysis with Missing Data (Wiley-Interscience, Hoboken, New Jersey, 2002).
8. Bernard Marr, "Big Data Using Smart Big Data, Analytics and Metrics to Make Better Decisions and Improve Performance", Wiley, 2015
9. Mayer-Schonberger, V. and Cukier, K., "Big Data: A Revolution That Will Transform How We Live, Work and Think", London: John Murray Publishers, 2013
10. Neilson, J.P. and Mistry, R.T., "Fetal electrocardiogram plus heart rate recording for fetal monitoring during labour", Cochrane Database of Systematic Reviews (2), 2000
11. Dekker, J.M., Schouten, E.G., Klootwijk, P., Pool, J., Swenne, C.A. and Kromhout, D., "Heart rate variability from short electrocardiographic recordings predicts mortality from all causes in Middle-aged and elderly men", The Zutphen Study, American Journal of Epidemiology 145 (10), 1997
12. IACP Centre for Social Media Fun Facts http://www.iacpsocialmedia.org/ Resources/FunFacts.aspx
13. BBC Two (2014) Bang goes the Theory, May 2014, Series 8: Big Data.
14. SAS Whitepaper, "Big Data meets Big Data Analytics: Three key technologies for extracting real-time business value from the Big Data that threatens to overwhelm traditional computing architectures", 2012
15. IDC (2014) The Digital Universe Study, April 2014. Sponsored by EMC2
16. Chui M, Loffler M, and Roberts R, "The Internet of Things". ¨ McKinsey Quarterly, March 2010.
17. Kosinski, M., Stillwell, D. and Graepel, T. (2013) Private traits and attributes are predictable from digital records of human behavior. Published online: http://www.pnas.org/content/early
18. David Loshin, "Big Data Analytics: From Strategic Planning to Enterprise Integration with Tools, Techniques, NoSQL, and Graph", Morgan Kaufmann, 2013

19. Hanson J. An introduction to the Hadoop distributed file system, accessed via http://www.ibm.com/developerworks/library/wa-introhdfs.
20. Apache's HDFS Architecture Guide, accessed via http://hadoop.apache.org/docs/stable/hdfs_design.html.
21. Murthy A. Introducing Apache Hadoop YARN, accessed via http://hortonworks.com/blog/introducing-apache-hadoop-yarn/.
22. Zookeeper, accessed via http://zookeeper.apache.org/.
23. Apache, accessed via http://hive.apache.org/.
24. Pig, accessed via http://pig.apache.org/.
25. Mahout, accesed via http://mahout.apache.org/.
26. Thomas Erl, Wajid Khattak, and Paul Buhler, "Big Data Fundamentals, Concepts, Drivers & Techniques", Prentice Hall, 2016
27. Sean Owen,‵ Robin Anil,‵ Ted Dunning,‵ Ellen Friedman, "Mahout in Action", Manning Publications, 2011
28. Practical eCommerce, "10 Questions on Product Recommendations," http://mng.bz/b6A5
29. Google Blogoscoped, "Overall Number of Picasa Photos" (March 12, 2007), http://blogoscoped.com/archive/2007-03-12-n67.html
30. TutorialsPoint [Online], Available: https://www.tutorialspoint.com/mahout/mahout_introduction.htm, [Accessed: 24-Feb-2018]
31. Wisdomjobs [Online], Available: https://www.wisdomjobs.com/e-university/hadoop-tutorial-484/a-brief-history-of-hadoop-14745.html [Accessed: 26-Feb-2018]
32. Edureka [Online], Available: https://www.edureka.co/blog/hadoop-tutorial/ [Accessed: 26-Feb-2018]
33. bmc [Online], Available: http://www.bmc.com/guides/hadoop-examples.html [Accessed: 26-Feb-2018]
34. bmc [Online], Available: http://www.bmc.com/guides/hadoop-benefits-business-case.html [Accessed: 26-Feb-2018]
35. JavaTPoint [Online], Available: https://www.javatpoint.com/hadoop-tutorial [Accessed: 27-Feb-2018]
36. Edureka [Online], Available: https://www.edureka.co/blog/spark-tutorial/ [Accessed: 04-March-2018]
37. http://www.scala-lang.org/ [Accessed: 04-March-2018]
38. http://spark.apache.org/downloads.html [Accessed: 04-March-2018]
39. https://drive.google.com/file/d/0B7Yoht-ttAeuWGd5SC1LcVZUbkk/view [Accessed: 04-March-2018]
40. https://docs.google.com/forms/d/e/1FAIpQLScJyoUgebjUlRyG9JrXY-aF9M4P2ZuMhFW7pzXzWaZ2IMEcxw/viewform [Accessed: 04-March-2018]
41. https://docs.google.com/forms/d/e/1FAIpQLSfMzE3sUoIASRbVanVwdlUX2h-1vXiksSxusHjSqhj_CR6RhQ/viewform [Accessed: 04-March-2018]
42. https://docs.google.com/forms/d/e/1FAIpQLSdPiCNiObU145181i0IyBEmHQtk5V09Wyq7F0ZYVyx1H-faYA/viewform [Accessed: 04-March-2018]