

Spark can be done with or without hadoop in our environment and we are having an environment where we are having an environment where hadoop is just installed and we will only be starting the services as typically that is the use case.

Start **Hadoop** in the following way

- a) start-dfs.sh
- b) start-yarn.sh

The following will be the daemons which will be started

```
notroot@ubuntu:~$ jps
2520 SecondaryNameNode
2731 Jps
1933 DataNode
1821 NameNode
2158 NodeManager
2014 ResourceManager
```

**Then we will start with the History Server like this :-**

```
notroot@ubuntu:~$ mr-jobhistory-daemon.sh start historyserver
starting historyserver, logging to /home/notroot/lab/software/hadoop-2.7.2/logs/mapred-
notroot-historyserver-ubuntu.out
```

```
notroot@ubuntu:~$ jps
2776 JobHistoryServer
2520 SecondaryNameNode
2812 Jps
1933 DataNode
1821 NameNode
2158 NodeManager
2014 ResourceManager
```

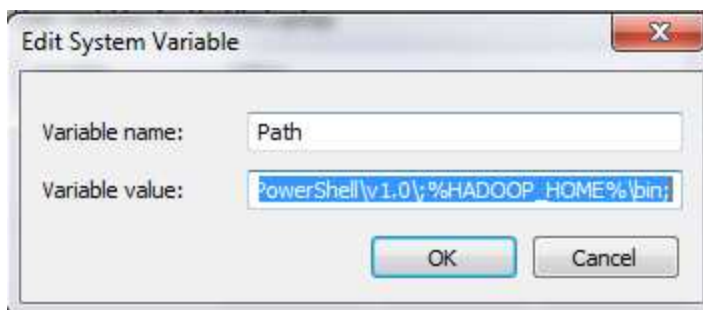
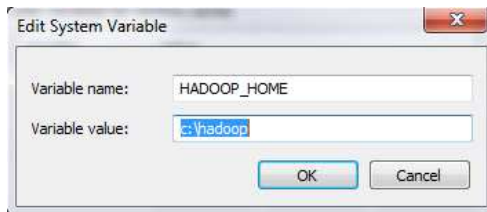
<https://spark.apache.org/docs/2.3.0/>

Check the Version of java which you have in the image:-

```
notroot@ubuntu:~$ java -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
notroot@ubuntu:~$ _
```

To compile the code in windows

- a) Create a folder called hadoop in c:\ and a bin folder inside hadoop and copy winutil.exe inside the bin folder



### Example 1:

```
package com.evenkat;
```

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.SparkConf;
import scala.Tuple2;
import java.util.Arrays;
```

```
public class WordCounter {
```

```
    public static void main(String[] args){
        //Create a SparkContext to initialize
        SparkConf conf = new
        SparkConf().setMaster("local").setAppName("Word Count");
```

```
// Create a Java version of the Spark Context
```

```
JavaSparkContext sc = new JavaSparkContext(conf);
```

```
// Load the text into a Spark RDD, which is a  
distributed representation of each line of text
```

```
JavaRDD<String> textFile = sc.textFile("file:///home/notroot/lab/data/words");
```

```
JavaPairRDD<String, Integer> counts = textFile  
.flatMap(s -> Arrays.asList(s.split("[.]")).iterator())  
.mapToPair(word -> new Tuple2<>(word, 1))  
.reduceByKey((a, b) -> a + b);
```

```
counts.foreach(p -> System.out.println(p));
```

```
System.out.println("Total words: " + counts.count());
```

```
counts.saveAsTextFile("/output/wcount_java");  
sc.close();  
}  
}
```

```
notroot@ubuntu:~/lab/programs$
```

```
spark-submit --class com.evenkat.WordCounter SparkJava.jar
```

[The SparkJava is the final jar file created and we move it to the programs directory]

## Example 2:

```
package com.evenkat;
```

```
import org.apache.log4j.Level;
```

```
import org.apache.log4j.Logger;
```

```
import org.apache.spark.SparkConf;
```

```
import org.apache.spark.api.java.JavaRDD;
```

```
import org.apache.spark.api.java.JavaSparkContext;
```

```
import java.util.Arrays;
```

```
import java.util.Map;
```

```

public class WordCount {
    public static void main(String[] args) throws Exception {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkConf conf = new
SparkConf().setAppName("wordCounts").setMaster("local[2]");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaRDD<String> lines = sc.textFile("/input/words");

        JavaRDD<String> words = lines.flatMap(line -> Arrays.asList(line.split("
")).iterator());
        Map<String, Long> wordCounts = words.countByKey();
        for (Map.Entry<String, Long> entry : wordCounts.entrySet()) {
            System.out.println(entry.getKey() + " : " + entry.getValue());
        }
        sc.close();
    }
}

```

### Example 3

Source File: word\_count.text

```

package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

import java.util.Arrays;
import java.util.Map;

public class WordCount {

    public static void main(String[] args) throws Exception {

```

```

Logger.getLogger("org").setLevel(Level.ERROR);
SparkConf conf = new
SparkConf().setAppName("wordCounts").setMaster("local[2]");
JavaSparkContext sc = new JavaSparkContext(conf);

JavaRDD<String> lines = sc.textFile("in/word_count.text");
JavaRDD<String> words = lines.flatMap(line -> Arrays.asList(line.split("
))).iterator());

Map<String, Long> wordCounts = words.countByKey();

for (Map.Entry<String, Long> entry : wordCounts.entrySet()) {
    System.out.println(entry.getKey() + " : " + entry.getValue());
}
sc.close();
}
}

```

#### Example 4. Problem Statement

Source File: airports.text

| name of the airport   | name of the airport  | ICAO code   | longitude   | timezone DST  |
|---|--|---|---|---|
| 3998, "Son Sant Joan", "Palma de Mallorca", "Spain", "PMI", "LEPA", 39.55361, 2.727778, 24, 1, "E", "Europe/Madrid" | 3999, "Darwin Intl", "Darwin", "Australia", "DRW", "YPDN", -12.408333, 130.87266, 103, 9, 5, "N", "Australia/Darwin" | 4000, "Surat Thani", "Surat Thani", "Thailand", "URT", "VTSB", 9.1325, 99.135556, 286, 7, "U", "Asia/Bangkok" | 4001, "Bacolod Intl", "Nyang U", "Burma", "VYU", "VYBR", 21.173333, 94.924666, 298, 6, 5, "U", "Asia/Rangoon" | 4002, "Goswami", "Caticlan", "Philippines", "MPH", "RPM", 9.214999, 121.95, 8, "N", "Asia/Manila" |

package com.evenkat;

public class AirportsInUsaProblem {

public static void main(String[] args) throws Exception {

/\* Create a Spark program to read the airport data from in/airports.text, find all the airports which are located in United States and output the airport's name and the city's name to out/airports\_in\_usa.text.

Each row of the input file contains the following columns:

Airport ID, Name of airport, Main city served by airport, Country where airport is located, IATA/FAA code,  
ICAO Code, Latitude, Longitude, Altitude, Timezone, DST, Timezone in Olson format

Sample output:

"Putnam County Airport", "Greencastle"

"Dowagiac Municipal Airport", "Dowagiac"

```
...
*/
}
}
5675,"Sao Filipe Airport","Sao Filipe, Fogo Island","Cape Verde","SFL",
5674,"Praia International Airport","Praia, Santiago Island","Cape Verde
```

#### Example 4: Solution of example of Filter and Map

```
package com.evenkat;
```

```
public class Utils {
```

```
    private Utils(){
    };

```

```
    // a regular expression which matches commas but not commas within double
    quotations

```

```
    public static final String COMMA_DELIMITER = ",(?=([^\"]*\"[^\"]*\")*[^\"]*$)";
}

```

```
package com.evenkat;
```

```
import com.evenkat.Utils;
```

```
import org.apache.commons.lang.StringUtils;
```

```
import org.apache.spark.SparkConf;
```

```
import org.apache.spark.api.java.JavaRDD;
```

```
import org.apache.spark.api.java.JavaSparkContext;
```

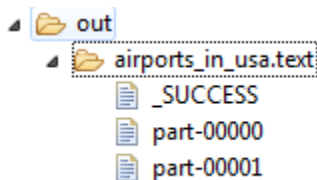
```
//Example of filter and map transformation
```

```
public class AirportsInUsaSolution {
```

```

public static void main(String[] args) throws Exception {
    SparkConf conf = new SparkConf().setAppName("airports").setMaster("local[2]");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> airports = sc.textFile("in/airports.text");
    JavaRDD<String> airportsInUSA = airports.filter(line ->
line.split(Utils.COMMA_DELIMITER)[3].equals("\"United States\""));
    JavaRDD<String> airportsNameAndCityNames = airportsInUSA.map(line -> {
        String[] splits = line.split(Utils.COMMA_DELIMITER);
        return StringUtils.join(new String[]{splits[1], splits[2]}, ",");
        //Name and City and join with a ,
    }
);
    airportsNameAndCityNames.saveAsTextFile("out/airports_in_usa.text");
    sc.close();
}
}

```



### Sample Solution

"Putnam County Airport", "Greencastle"  
 "Dowagiac Municipal Airport", "Dowagiac"  
 "Cambridge Municipal Airport", "Cambridge"  
 "Door County Cherryland Airport", "Sturgeon Bay"  
 "Shoestring Aviation Airfield", "Stewartstown"

### **Example 5: Problem Statement**

```

package com.evenkat;

public class AirportsByLatitudeProblem {

```

```
public static void main(String[] args) throws Exception {
```

```
    /* Create a Spark program to read the airport data from in/airports.text, find all the
    airports whose latitude are bigger than 40.
```

```
    Then output the airport's name and the airport's latitude to
    out/airports_by_latitude.text.
```

```
    Each row of the input file contains the following columns:
```

```
    Airport ID, Name of airport, Main city served by airport, Country where airport is
    located, IATA/FAA code,
```

```
    ICAO Code, Latitude, Longitude, Altitude, Timezone, DST, Timezone in Olson
    format
```

```
    Sample output:
```

```
    "St Anthony", 51.391944
```

```
    "Tofino", 49.082222
```

```
    ...
```

```
    */
```

```
    }
```

```
}
```

## Solution

```
package com.evenkat;
```

```
import evenkat.Utils;
```

```
import org.apache.commons.lang.StringUtils;
```

```
import org.apache.spark.SparkConf;
```

```
import org.apache.spark.api.java.JavaRDD;
```

```
import org.apache.spark.api.java.JavaSparkContext;
```

```
public class AirportsByLatitudeSolution {
```

```
    public static void main(String[] args) throws Exception {
```

```
        SparkConf conf = new SparkConf().setAppName("airports").setMaster("local[2]");
```

```
        JavaSparkContext sc = new JavaSparkContext(conf);
```

```
        JavaRDD<String> airports = sc.textFile("in/airports.text");
```

```
        JavaRDD<String> airportsInUSA = airports.filter(line ->
        Float.valueOf(line.split(Utils.COMMA_DELIMITER)[6]) > 40);
```

```
        JavaRDD<String> airportsNameAndCityNames = airportsInUSA.map(line -> {
```

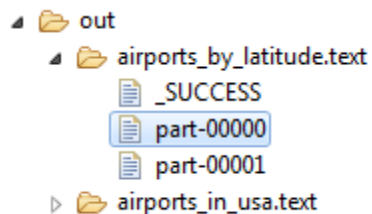


```

        String[] splits = line.split(Utils.COMMA_DELIMITER);
        return StringUtils.join(new String[]{splits[1], splits[6]}, ",");
    }
);
airportsNameAndCityNames.saveAsTextFile("out/airports_by_latitude.text");
sc.close();
}
}

```

---



### Sample Output

```

Narsarsuaq",61.160517
"Nuuq",64.190922
"Sondre Stromfjord",67.016969
"Thule Air Base",76.531203
"Akureyri",65.659994

```

---

### Example 6: Now trying with a Maven Project

```

package com.evenkat;

import java.util.ArrayList;
import java.util.List;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
public class Main {

```

```

public static void main(String[] args)
{
    List<Integer> inputData = new ArrayList<>();
    inputData.add(35);
    inputData.add(12);
    inputData.add(90);
    inputData.add(20);

    Logger.getLogger("org").setLevel(Level.WARN);

    SparkConf conf = new
SparkConf().setAppName("startingSpark").setMaster("local[*]");
    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<Integer> myRdd = sc.parallelize(inputData);

    // 1] This is to call the reduce function
    Integer result = myRdd.reduce((value1, value2 ) -> value1 + value2 );

    // 2] This is find the sqrt of the values in the RDD
    JavaRDD<Double> sqrtRdd = myRdd.map( value -> Math.sqrt(value) );

    sqrtRdd.foreach( value -> System.out.println(value));
    //
    // sqrtRdd.foreach( System.out::println );
    // This is Java 8 way of doing it
    // Incase if you have a multiple core there you might an exception called
    // NotSerializableException
    sqrtRdd.collect().forEach( System.out::println );

    // 3] How many elements in sqrtRdd
    System.out.println(sqrtRdd.count());
    // using just map and reduce
    JavaRDD<Long> singleIntegerRdd = sqrtRdd.map( value -> 1L);
    Long count = singleIntegerRdd.reduce((value1, value2) -> value1 + value2);
    System.out.println(count);

    System.out.println(result);
    sc.close();
}
}

```

pom.xml

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>Spark</groupId>
  <artifactId>Third1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>2.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.10</artifactId>
      <version>2.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-hdfs</artifactId>
      <version>2.2.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.5.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>

```

```

        <configuration>
            <source>1.8</source>
            <target>1.8</target>
            <archive>
                <manifest>
                    <mainClass>Main</mainClass>
                </manifest>
            </archive>
            <descriptorRefs>
                <descriptorRef>jar-with-
dependencies</descriptorRef>
            </descriptorRefs>
        </configuration>
        <executions>
            <execution>
                <id>make-assembly</id> <!-- this is used for inheritance merges -->
                <phase>package</phase> <!-- bind to the packaging phase -->
                <goals>
                    <goal>single</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>
</project>

```

## Example 7

```

package com.evenkat;

import java.util.ArrayList;
import java.util.List;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

import com.google.common.collect.Iterables;
import scala.Tuple2;

public class Main1 {

```

```

@SuppressWarnings("resource")
public static void main(String[] args)
{
    List<String> inputData = new ArrayList<>();
    inputData.add("WARN: Tuesday 4 September 0405");
    inputData.add("ERROR: Tuesday 4 September 0408");
    inputData.add("FATAL: Wednesday 5 September 1632");
    inputData.add("ERROR: Friday 7 September 1854");
    inputData.add("WARN: Saturday 8 September 1942");

    Logger.getLogger("org").setLevel(Level.WARN);

    SparkConf conf = new
SparkConf().setAppName("startingSpark").setMaster("local[*]");
    JavaSparkContext sc = new JavaSparkContext(conf);

//      JavaRDD<String> originalLogMessages = sc.parallelize(inputData);
/*      JavaPairRDD<String, String> pair = originalLogMessages.mapToPair(
rawValue -> {
        String [] columns = rawValue.split(":");
        String level = columns[0];
        String date = columns[1];

        return new Tuple2<>(level,date);
// the Tuple2 take 2 string arguments and they can be removed only
// when we parameterize the JavaPairRDD object
    });
    We just created the second element in JavaPairRDD as String.
    */

    /*JavaPairRDD<String, Long> pair = originalLogMessages.mapToPair(
rawValue -> {
        String [] columns = rawValue.split(":");
        String level = columns[0];
        return new Tuple2<>(level,1L);
    });

    JavaPairRDD<String, Long> sumsRdd = pair.reduceByKey( (value1,
value2) -> value1+value2);
    sumsRdd.foreach(tuple -> System.out.println(tuple._1 + " has " + tuple._2
+ " instances"));*/

/*      sc.parallelize(inputData)
        .mapToPair(rawValue -> new Tuple2<>(rawValue.split(":")[0] , 1L ))
        .reduceByKey((value1, value2) -> value1 + value2)

```

```

        .foreach(tuple -> System.out.println(tuple._1 + " has " + tuple._2 + "
instances")); */

        //groupbykey version - not recommended due to performance (see later)
AND the iterable
        //is awkward to work with.
        sc.parallelize(inputData)
        .mapToPair(rawValue -> new Tuple2<>(rawValue.split(":")[0] , 1L ))
        .groupByKey()
        .foreach( tuple -> System.out.println(tuple._1 + " has " +
        Iterables.size(tuple._2) + " instances" ) );
    }
    sc.close();
}

```

### Example 8:

```

package com.evenkat;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

import com.google.common.collect.Iterables;

import scala.Tuple2;

public class Main2 {
    @SuppressWarnings("resource")
    public static void main(String[] args)
    {
        List<String> inputData = new ArrayList<>();
        inputData.add("WARN: Tuesday 4 September 0405");
        inputData.add("ERROR: Tuesday 4 September 0408");
        inputData.add("FATAL: Wednesday 5 September 1632");
        inputData.add("ERROR: Friday 7 September 1854");
        inputData.add("WARN: Saturday 8 September 1942");

        Logger.getLogger("org").setLevel(Level.WARN);
    }
}

```

```

        SparkConf conf = new
SparkConf().setAppName("startingSpark").setMaster("local[*]");
        JavaSparkContext sc = new JavaSparkContext(conf);

//          JavaRDD<String> sentences = sc.parallelize(inputData);
//          JavaRDD<String> words = sentences.flatMap(value ->
Arrays.asList(value.split(" ")).iterator());
//          words.foreach(System.out::println);

//          JavaRDD<String> filteredWords = words.filter(word -> word.length() > 1);
//          filteredWords.foreach(System.out::println);

        sc.parallelize(inputData)
          .flatMap(value -> Arrays.asList(value.split(" ")).iterator())
          .filter(word -> word.length() > 1)
          .foreach(System.out::println);

        sc.close();
    }
}

```

### Example 9:

#### InputFile: Sample1.txt

```

package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

public class FilterEx {
    @SuppressWarnings("resource")
    public static void main(String[] args) {

        Logger.getLogger("org").setLevel(Level.WARN);
        SparkConf sparkConf = new
SparkConf().setAppName("test").setMaster("local");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

        JavaRDD<String> radiusData =
jsc.textFile("D:\\SparkWithJava\\data\\Sample1.txt");
        JavaRDD<String> filtered = radiusData.filter(line -> {

```

```

        String[] split = line.split(",");

        if (split[2].contains("port=27")) {
            return true;
        } else {
            return false;
        }
    });
    filtered.foreach(x -> System.out.println(x));

    //Example of Count and Take
    System.out.println("Count is :" + radiusData.count());
    System.out.println("Example of Take \n");
    for (String string : radiusData.take(10)) {
        System.out.println(string);
    }

    //Example of Collect
    System.out.println("Example of Collect\n");

    for (String string : radiusData.collect()) {
        System.out.println(string);
    }
}
jsc.close();
}

```

## Example 10

### InputFile: Sample2.txt and Sample 3.txt

```

package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

import scala.Tuple2;

public class Distinct_Union_Intersection_Subtract_Cartisian {
    public static void main(String[] args) {
        Logger.getLogger("org").setLevel(Level.WARN);
    }
}

```



```

        SparkConf sparkConf = new
SparkConf().setAppName("test").setMaster("local");

        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        JavaRDD<String> rdd =
jsc.textFile("D:\\SparkWithJava\\data\\Sample2.txt");
        JavaRDD<String> rdd2 =
jsc.textFile("D:\\SparkWithJava\\data\\Sample3.txt");

        JavaRDD<String> distinct_rdd =rdd.distinct();
        System.out.println("Distinct \n " );
        for (String string : distinct_rdd.collect()) {
            System.out.println(string);
        }

        JavaRDD<String> union_rdd=rdd.union(rdd2);
        System.out.println("Union \n");
        for (String string : union_rdd.collect()) {
            System.out.println(string);
        }
        JavaRDD<String> intersection_rdd=rdd.intersection(rdd2);
        System.out.println("Intersection \n");
        for (String string :intersection_rdd.collect()) {
            System.out.println(string);
        }

        JavaRDD<String> subtract_rdd=rdd.subtract(rdd2);
        System.out.println("Subtract \n");
        for (String string : subtract_rdd.collect()) {
            System.out.println(string);
        }

        JavaPairRDD<String,String> cartesian_rdd=rdd.cartesian(rdd2);
        System.out.println("Cartesian \n");
        for (Tuple2<String,String> string : cartesian_rdd.collect()) {
            System.out.println(string._1+"-----"+string._2);
        }
        jsc.close();
    }
}

```

**Example 11:**

**DataFile: Sample4.txt**

```

package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;

import scala.Tuple2;

public class FilterKeyValue {

    public static void main(String[] args) {

        Logger.getLogger("org").setLevel(Level.WARN);
        SparkConf sparkConf = new
SparkConf().setAppName("test").setMaster("local");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

        JavaRDD<String> rdd =
jsc.textFile("D:\\SparkWithJava\\data\\Sample4.txt");

        JavaPairRDD<String, String> pairRdd = rdd.mapToPair(new
PairFunction<String, String, String>() {

            private static final long serialVersionUID = 1L;

            @Override
            public Tuple2<String, String> call(String x) throws Exception {
                // TODO Auto-generated method
                return new Tuple2<String, String>(x.split(",")[0], x);
            }
        });

        JavaPairRDD<String, String> filteredRdd=pairRdd.filter(x -> {
            if (Integer.parseInt(x._2.split(",")[4]) > 200) {
                return true;
            } else {
                return false;
            }
        });

        System.out.println("Filtered key values being \n");
    }
}

```

```

        for (Tuple2 string : filteredRdd.collect()) {
            System.out.println("Key "+string._1+" Value "+string._2);
        }
//Sometimes working with pairs can be awkward if we want to access only the value
//part of our pair RDD. Since this is a common pattern, Spark provides the
//mapValues(func) function

JavaPairRDD<String, String> d=pairRdd.mapValues(new Function<String, String>() {
    private static final long serialVersionUID = 1L;

    public String call(String arg0) throws Exception {
        // TODO Auto-generated method stub
        return arg0;
    }
});

System.out.println("Only values being shown \n");
for (Tuple2<String, String> string : d.collect()) {
    System.out.println(" Value "+string._2);
}
}

```

After seeing the examples of Transformations & Actions, we will move on with the Advanced Concepts of Spark

## Accumulators

<https://spark.apache.org/docs/2.3.0/rdd-programming-guide.html#accumulators>

Let open the 2016-stack-overflow-survey-responses.csv document in excel and we interested in the 3<sup>rd</sup> column and the 15 column in this example.

| O1 |       | salary_midpoint |             |          |              |                       |          |            |   |           |           |             |           |               |                 |         |            |            |            |
|----|-------|-----------------|-------------|----------|--------------|-----------------------|----------|------------|---|-----------|-----------|-------------|-----------|---------------|-----------------|---------|------------|------------|------------|
|    | A     | B               | C           | D        | E            | F                     | G        | H          | I   | J         | K         | L           | M         | N             | O               | P       | Q          | R          | S          |
| 1  |       | collector       | country     | un_subre | so_region    | age_range             | age_midp | gender     | self_ident  | occupatio | occupatio | experie     | experie   | salary_ran    | salary_midpoint | big_mac | tech_do    | tech_wan   | aliens     |
| 2  | 1888  | Facebook        | Afghanistan | Southern | Central As   | 20-24                 | 22       | Male       | Programmer  |           |           |             |           |               |                 |         |            |            |            |
| 3  | 4637  | Facebook        | Afghanistan | Southern | Central As   | 30-34                 | 32       | Male       | Develope  | Mobile de | Mobile De | 6 - 10 year | 8         | \$40,000 - \$ | 45000           |         | iOS; Obj   | Swift      | Yes        |
| 4  | 11164 | Facebook        | Afghanistan | Southern | Central Asia |                       |          |            |   |           |           |             |           |               |                 |         |            |            |            |
| 5  | 21378 | Facebook        | Afghanistan | Southern | Central Asia |                       |          | Female     | Engineer  | DevOps    | DevOps    | 11+ years   | 13        | Less than     | 5000            |         |            |            | Other (ple |
| 6  | 30280 | Facebook        | Afghanistan | Southern | Central As   | > 60                  | 65       | Prefer not | Developer; Engineer; Programmer; Sr. Developer; Manager; Rockstar; Ninja; Guru; Expert; Full-stack Developer; Full Stack Ov |           |           |             |           |               |                 |         |            |            |            |
| 7  | 31355 | Facebook        | Afghanistan | Southern | Central As   | 20-24                 | 22       | Prefer not | Ninja; Guru; Hacker   |           |           |             |           |               |                 |         |            |            |            |
| 8  | 31743 | Facebook        | Afghanistan | Southern | Central As   | Prefer not to disclos | Other    | Hacker     | Growth he   | Growth he | 11+ years | 13          | More than | 210000        |                 |         | Android; J | Android    | Yes        |
| 9  | 51301 | Facebook        | Afghanistan | Southern | Central As   | 25-29                 | 27       | Male       | Engineer  | Back-end  | Back-end  | 1 - 2 years | 1.5       | Less than     | 5000            |         | JavaScript | Android; i | Yes        |
| 10 | 13017 | Facebook        | Albania     | Southern | Eastern Eu   | 25-29                 | 27       | Other      | Developer; Engineer; Programmer; Sr. Developer; Rockstar; Ninja; Guru; Hacker   |           |           |             |           |               |                 |         |            |            |            |

We have the following questions to answer in this example

- How many records do we have in this survey result
- How many records are missing the salary middle point
- How many records are from Canada

## Example 12

**DataFile:** 2016-stack-overflow-survey-responses.csv

```
package com.evenkat;

import com.evenkat.Utils;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.util.LongAccumulator;
import scala.Option;

public class StackOverFlowSurvey {
    public static void main(String[] args) throws Exception {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkConf conf = new SparkConf().setAppName("StackOverFlowSurvey").setMaster("local[1]");

        SparkContext sparkContext = new SparkContext(conf);
        JavaSparkContext javaSparkContext = new JavaSparkContext(sparkContext);

        final LongAccumulator total = new LongAccumulator();
        final LongAccumulator missingSalaryMidPoint = new LongAccumulator();

        total.register(sparkContext, Option.apply("total"), false);
        missingSalaryMidPoint.register(sparkContext, Option.apply("missing salary middle point"), false);

        JavaRDD<String> responseRDD = javaSparkContext.textFile("in/2016-stack-overflow-survey-
responses.csv");
        JavaRDD<String> responseFromCanada = responseRDD.filter(response -> {
            String[] splits = response.split(Utils.COMMA_DELIMITER, -1);
            total.add(1);
            if (splits[14].isEmpty()) {
                missingSalaryMidPoint.add(1);
            }
            return splits[2].equals("Canada");
        });
        System.out.println("Count of responses from Canada: " + responseFromCanada.count());
        System.out.println("Total count of responses: " + total.value());
        System.out.println("Count of responses missing salary middle point: " +
missingSalaryMidPoint.value());
    }
}
```

Out of the box, spark supports several accumulators of types such as double and long. Spark also allows users to define custom accumulator types and custom aggregation operations such as finding the maximum of the accumulated values instead of adding them.

We will have to extend the AccumulatorV2 abstract class to define our own custom accumulators

<https://spark.apache.org/docs/preview/api/java/org/apache/spark/util/AccumulatorV2.html>

<https://medium.com/@shrechak/leveraging-custom-accumulators-in-apache-spark-2-0-f4fef23f19f1>

To check the accumulators we need to go to the HistoryPage to check the details and hence we will also be trying this example in our cluster environment by creating a final jar called Example12.jar and moving it via winscp to the lab/programs directory.

Our file for testing should also be present in the /input directory of hdfs and we will put it in this fashion

- 1) Copy 2016-stack-overflow-survey-responses.csv file in to lab/data
- 2) `notroot@ubuntu:~/lab/data$ hdfs dfs -copyFromLocal 2016-stack-overflow-survey-responses.csv /input`
- 3) Confirm that the file is present by checking the 50070 page

| /input     |         |            |           |                        |             |            |  |
|------------|---------|------------|-----------|------------------------|-------------|------------|--|
| Permission | Owner   | Group      | Size      | Last Modified          | Replication | Block Size | Name   |
| -rw-r--r-- | notroot | supergroup | 2.25 MB   | 10/9/2019, 3:49:12 PM  | 1           | 128 MB     | <a href="#">2016-stack-overflow-survey-responses.csv</a> |
| -rw-r--r-- | notroot | supergroup | 168.47 MB | 10/8/2019, 12:40:18 AM | 1           | 128 MB     | <a href="#">txns</a>                                     |
| -rw-r--r-- | notroot | supergroup | 37 B      | 10/9/2019, 2:14:23 AM  | 1           | 128 MB     | <a href="#">words</a>                                    |

Before executing the code, ensure that you change the java file to load the textFile in this fashion:-

```
JavaRDD<String> responseRDD = javaSparkContext.textFile("/input/2016-stack-overflow-survey-responses.csv");
```

Now ensure that you have the HistoryServer up and running and to start it again.

```
notroot@ubuntu:~/lab/software/spark-2.3.0-bin-hadoop2.7/sbin$ ./start-history-server.sh
```

```
starting org.apache.spark.deploy.history.HistoryServer, logging to /home/notroot/lab/software/spark-2.3.0-bin-hadoop2.7/logs/spark-notroot-org.apache.spark.deploy.history.HistoryServer-1-ubuntu.out
```

Then create the jar and move it to the programs directory. Then we will execute it in this fashion:-

```
notroot@ubuntu:~/lab/programs$ spark-submit --class com.evenkat.StackOverFlowSurvey Example12.jar
```

After that we can look at the details of the job and see the Accumulator that we had created:-

#### Accumulators

| Accumulable                 | Value |
|-----------------------------|-------|
| missing salary middle point | 566   |
| total                       | 2000  |

**Example 13:** Now we will add another accumulator to calculate the bytes processed

```
package com.evenkat;
```

```
import com.evenkat.Utils;  
import org.apache.log4j.Level;
```

```

import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.util.LongAccumulator;
import scala.Option;
public class StackOverFlowSurveyFollowUp {
    public static void main(String[] args) throws Exception {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkConf conf = new SparkConf().setAppName("StackOverFlowSurvey").setMaster("local[1]");
        SparkContext sparkContext = new SparkContext(conf);
        JavaSparkContext javaSparkContext = new JavaSparkContext(sparkContext);

        final LongAccumulator total = new LongAccumulator();
        final LongAccumulator missingSalaryMidPoint = new LongAccumulator();
        final LongAccumulator processedBytes = new LongAccumulator();

        total.register(sparkContext, Option.apply("total"), false);
        missingSalaryMidPoint.register(sparkContext, Option.apply("missing salary middle point"), false);
        processedBytes.register(sparkContext, Option.apply("Processed bytes"), true);

        JavaRDD<String> responseRDD = javaSparkContext.textFile("in/2016-stack-overflow-survey-
responses.csv");
        JavaRDD<String> responseFromCanada = responseRDD.filter(response -> {
            // update processedBytes accumulator with the size of the current response
            processedBytes.add(response.getBytes().length);

            // split the reponse using commas.
            String[] splits = response.split(Utils.COMMA_DELIMITER, -1);
            // increase the total accumulator by 1
            total.add(1);

            // increase the missingSalaryMidPoint accumulator by 1 if the salary middle point is not present in
the reponse
            if (splits[14].isEmpty()) {
                missingSalaryMidPoint.add(1);
            }
            // return true if the reponse is from Canada
            return splits[2].equals("Canada");
        });

        System.out.println("Count of responses from Canada: " + responseFromCanada.count());
        System.out.println("Number of bytes processed: " + processedBytes.value());
        System.out.println("Total count of responses: " + total.value());
        System.out.println("Count of responses missing salary middle point: " +
missingSalaryMidPoint.value());
    }
}

```

#### Output:

Count of responses from Canada: 77  
Number of bytes processed: 2360248

Total count of responses: 2000

Count of responses missing salary middle point: 566

## Broadcast Variables

They allow developers to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used for example to give every node, a copy of a large input dataset in an efficient manner.

All broadcast variables will be kept at all the worker nodes for use in one or more spark operations

## Example 14

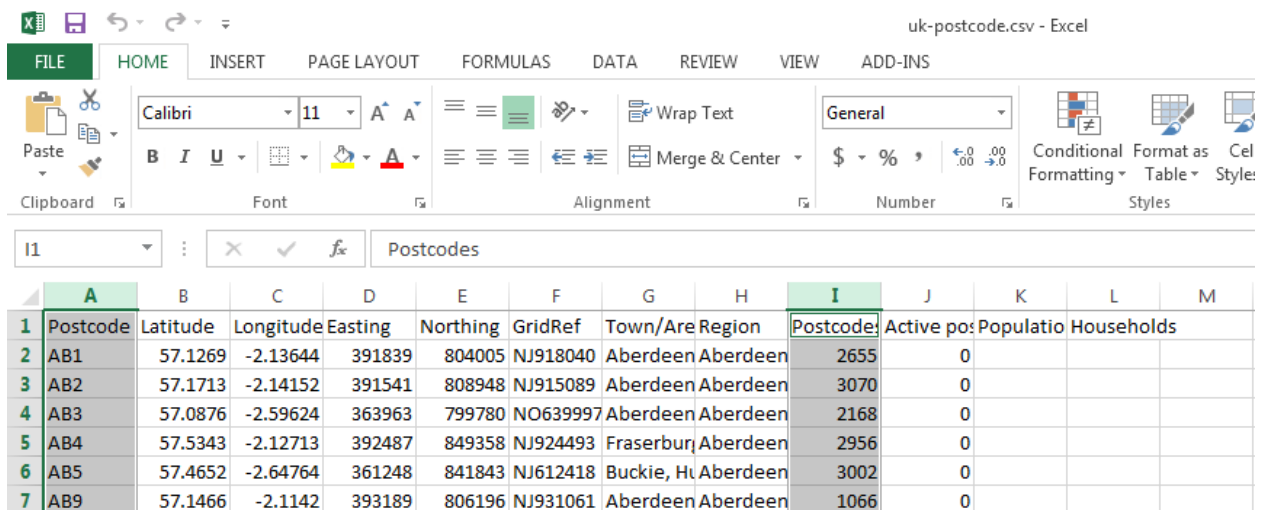
**Data File:** uk-makerspaces-identifiable-data.csv & uk-postcode.csv

Note: The csv file has a header column in the first line.

The main question to be answered here is: How are the maker spaces distributed across different regions in UK. However only know the postcode for each region and do not know the region. To answer that we have another dataset called uk-postcode.csv.

So by using the above, for each postcode prefix we can find the region it belongs to and by combining the datasets we can answer the above question. The solution would be

- Load the postcode dataset and broadcast it across the cluster.
- Load the maker space dataset and call map operation on the maker space RDD to look up the region using the postcode of the maker space.



|   | A        | B        | C         | D       | E        | F        | G          | H        | I        | J           | K          | L          | M |
|---|----------|----------|-----------|---------|----------|----------|------------|----------|----------|-------------|------------|------------|---|
|   | Postcode | Latitude | Longitude | Easting | Northing | GridRef  | Town/Area  | Region   | Postcode | Active post | Population | Households |   |
| 1 | AB1      | 57.1269  | -2.13644  | 391839  | 804005   | NJ918040 | Aberdeen   | Aberdeen | 2655     | 0           |            |            |   |
| 2 | AB2      | 57.1713  | -2.14152  | 391541  | 808948   | NJ915089 | Aberdeen   | Aberdeen | 3070     | 0           |            |            |   |
| 3 | AB3      | 57.0876  | -2.59624  | 363963  | 799780   | NO639997 | Aberdeen   | Aberdeen | 2168     | 0           |            |            |   |
| 4 | AB4      | 57.5343  | -2.12713  | 392487  | 849358   | NJ924493 | Fraserburg | Aberdeen | 2956     | 0           |            |            |   |
| 5 | AB5      | 57.4652  | -2.64764  | 361248  | 841843   | NJ612418 | Buckie, H  | Aberdeen | 3002     | 0           |            |            |   |
| 6 | AB9      | 57.1466  | -2.1142   | 393189  | 806196   | NJ931061 | Aberdeen   | Aberdeen | 1066     | 0           |            |            |   |

When we talk about the post code in the maker space RDD it is the full postcode that is W1T 3AC, E14 9TC and SW12 7YC. Whereas the post code in the postcode dataset is only the prefix of the postcode that is W1T, E14 and SW12.

package com.evenkat;

```
import com.evenkat.Utils;  
import org.apache.log4j.Level;  
import org.apache.log4j.Logger;
```

```

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.broadcast.Broadcast;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

public class UkMakerSpaces {

    public static void main(String[] args) throws Exception {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkConf conf = new SparkConf().setAppName("UkMakerSpaces").setMaster("local[1]");
        JavaSparkContext jsc = new JavaSparkContext(conf);

        final Broadcast<Map<String, String>> postCodeMap = jsc.broadcast(loadPostCodeMap());

        JavaRDD<String> makerSpaceRdd = jsc.textFile("in/uk-makerspaces-identifiable-
data.csv");

        JavaRDD<String> regions = makerSpaceRdd
            .filter(line -> !line.split(Utils.COMMA_DELIMITER, -1)[0].equals("Timestamp"))
            .map(line -> {
                Optional<String> postPrefix = getPostPrefix(line);
                if (postPrefix.isPresent() && postCodeMap.value().containsKey(postPrefix.get())) {
                    return postCodeMap.value().get(postPrefix.get());
                }
                return "Unknown";
            });
        for (Map.Entry<String, Long> regionCounts : regions.countByValue().entrySet()) {
            System.out.println(regionCounts.getKey() + " : " + regionCounts.getValue());
        }
        jsc.close();
    }

    private static Optional<String> getPostPrefix(String line) {
        String[] splits = line.split(Utils.COMMA_DELIMITER, -1);
        String postcode = splits[4];
        if (postcode.isEmpty()) {
            return Optional.empty();
        }
        return Optional.of(postcode.split(" ")[0]);
    }

    private static Map<String, String> loadPostCodeMap() throws FileNotFoundException {
        @SuppressWarnings("resource")
        Scanner postCode = new Scanner(new File("in/uk-postcode.csv"));
        Map<String, String> postCodeMap = new HashMap<>();
        while (postCode.hasNextLine()) {
            String line = postCode.nextLine();
            String[] splits = line.split(Utils.COMMA_DELIMITER, -1);
            postCodeMap.put(splits[0], splits[7]);
        }
    }
}

```



```

        return postCodeMap;
    }
}

```

So the procedure of using the Broadcast variable is like this:

- a) Create a Broadcast variable T by calling `SparkContext.broadcast()` on an object of type T.
- b) It is important that the broadcast variable is serializable because it needs to be passed from the driver program to all the worker nodes in the cluster across the wire.
- c) We can also broadcast a custom java object, but we need to make sure that the object implements the serializable interface.
- d) The variable will be send to each node only once and should be considered as read only as no new updates will be propagated to other nodes.
- e) The value of the broadcast can be accessed by calling the value method in each node.

## Caching and Persistence

Sometimes we would like to call actions on the same RDD multiple times and if we do this in a normal fashion the RDDs and all of its dependencies are recomputed each time an action is called on the RDD.

This can be very expensive, especially for some iterative algorithms, which would call actions on the same dataset many times and if we want to reuse an RDD in multiple actions, you can ask Spark to persist by calling the `persist` method on the RDD. When we persist an RDD, the first time it is computed in the action and it will be kept in memory across the nodes.

### Example 15

```

package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.storage.StorageLevel;

import java.util.Arrays;
import java.util.List;

public class PersistExample {

    public static void main(String[] args) throws Exception {

        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkConf conf = new SparkConf().setAppName("reduce").setMaster("local[*]");
        JavaSparkContext sc = new JavaSparkContext(conf);

        List<Integer> inputIntegers = Arrays.asList(1, 2, 3, 4, 5);
        JavaRDD<Integer> integerRdd = sc.parallelize(inputIntegers);
    }
}

```

```

integerRdd.persist(StorageLevel.MEMORY_ONLY());

integerRdd.reduce((x, y) -> x * y);

System.out.println (" The count is " +integerRdd.count());

sc.close();
}
}

```

The different storage level available to us mentioned in detailed in this link.  
<https://spark.apache.org/docs/2.3.0/rdd-programming-guide.html#rdd-persistence>

In the above page also look at the link of which Storage Level to use.

### Joins - Example 16

```

package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.Optional;
import scala.Tuple2;

import java.util.Arrays;

public class JoinOperations {

    public static void main(String[] args) throws Exception {

        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkConf conf = new SparkConf().setAppName("JoinOperations").setMaster("local[1]");

        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaPairRDD<String, Integer> ages = sc.parallelizePairs(Arrays.asList(new
        Tuple2<>("Tom", 29),
                                new Tuple2<>("John", 22)));

        JavaPairRDD<String, String> addresses = sc.parallelizePairs(Arrays.asList(new
        Tuple2<>("James", "USA"),
                                new Tuple2<>("John", "UK")));

        JavaPairRDD<String, Tuple2<Integer, String>> join = ages.join(addresses);

        join.saveAsTextFile("out/age_address_join.text");

        JavaPairRDD<String, Tuple2<Integer, Optional<String>>> leftOuterJoin =
        ages.leftOuterJoin(addresses);

        leftOuterJoin.saveAsTextFile("out/age_address_left_out_join.text");
    }
}

```

```

    JavaPairRDD<String, Tuple2<Optional<Integer>, String>> rightOuterJoin =
ages.rightOuterJoin(addresses);

    rightOuterJoin.saveAsTextFile("out/age_address_right_out_join.text");

    JavaPairRDD<String, Tuple2<Optional<Integer>, Optional<String>>> fullOuterJoin =
ages.fullOuterJoin(addresses);

    fullOuterJoin.saveAsTextFile("out/age_address_full_out_join.text");
}
}

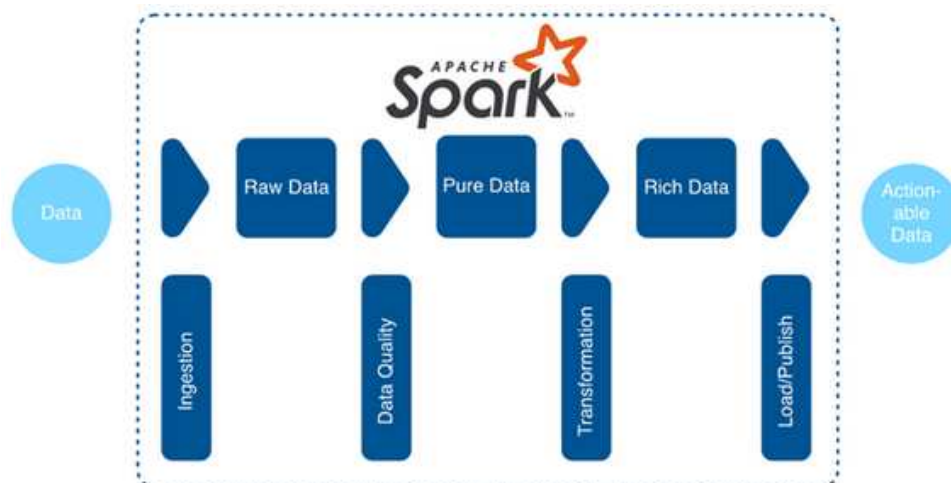
```

### Best Practices in Join

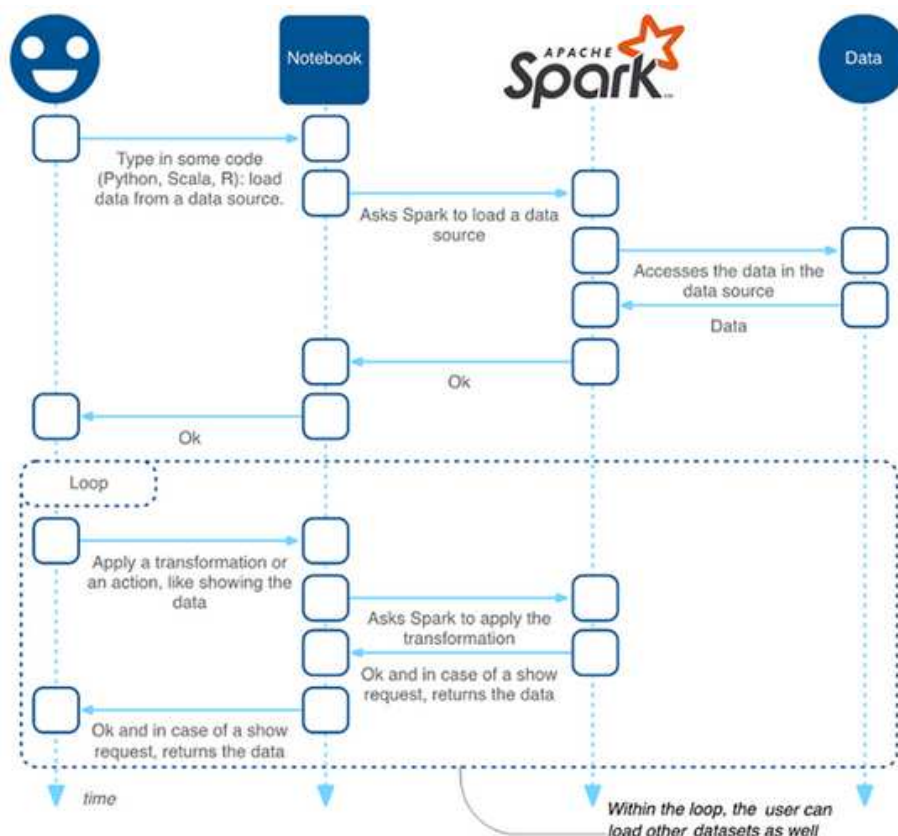
- 1 If both RDDs have duplicate keys, join operation can dramatically expand the size of the data and hence it is recommended to perform a distinct operation to reduce the key space if possible.
- 2 Join operations may require large network transfers or even create data sets beyond our capability to handle
- 3 Joins in general are expensive since they require that corresponding keys from each RDDs are located at the same partition so that they can be combined locally.

<https://databricks.com/session/optimizing-apache-spark-sql-joins>

Spark in a typical data processing scenario. The first step is ingesting the data. At this stage, the data is raw: you may want to apply some data quality (DQ). You are now ready to transform your data. Once you have transformed your data, it is richer. It is time to publish or share them, so that people in your organization can perform actions and make decisions based on it.



Sequence diagram for a data scientist using Spark: the user “talks” to the notebook, which calls Spark when needed. Spark directly handles ingestion. Each square represents a step, each arrow represents a sequence. The diagram should be read chronologically, starting from the top.



## DataFrame & DataSet

Spark SQL introduces a tabular data abstraction called DataFrame since Spark 1.3. It is a data abstraction for working with structured data.

It uses capabilities of RDD and applies a structure called schema to the data, allowing spark to manage the schema and only pass data between nodes, in a much more efficient way than using Java Serialization.

Inline an RDD, data is organized in to named Columns like a table in a RDBMS.

A Dataset is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each Dataset also has an untyped view called a DataFrame, which is a Dataset of Row.

Dataset in Spark provides Optimized query using Catalyst Query Optimizer and Tungsten. Catalyst Query Optimizer is an execution-agnostic framework. It represents and manipulates a data-flow graph. Data flow graph is a tree of expressions and relational operators

Using Dataset we can check syntax and analysis at compile time. It is not possible using DataFrame, RDDs or regular SQL queries.

SparkSession is the entry point to the SparkSQL. It is a very first object that we create while developing Spark SQL applications using fully typed Dataset data abstractions. Using SparkSession.Builder, we can create an instance of SparkSession. And can stop SparkSession using the stop method (spark.stop).

## **Analysis of RDD vs DataFrame vs DataSet in Apache Spark**

### **1] Spark Release**

- RDD – The RDD APIs have been on Spark since the 1.0 release.
- DataFrames – Spark introduced DataFrames in Spark 1.3 release.
- DataSet – Spark introduced Dataset in Spark 1.6 release.

### **2] Data Representation**

- RDD – RDD is a distributed collection of data elements spread across many machines in the cluster. RDDs are a set of Java or Scala objects representing data.
- DataFrame – A DataFrame is a distributed collection of data organized into named columns. It is conceptually equal to a table in a relational database.
- DataSet – It is an extension of DataFrame API that provides the functionality of – type-safe, object-oriented programming interface of the RDD API and performance benefits of the Catalyst query optimizer of a DataFrame API.

### **3] Data Formats**

- RDD – It can easily and efficiently process data which is structured as well as unstructured. But like DataFrame and DataSet, RDD does not infer the schema of the ingested data and requires the user to specify it.
- DataFrame – It works only on structured data. It organizes the data in the named column. DataFrames allow the Spark to manage schema.
- DataSet – It also efficiently processes structured data. It represents data in the form of JVM objects of row or a collection of row object. Which is represented in tabular forms through encoders.

### **4] Data Sources API**

- RDD – Data source API allows that an RDD could come from any data source e.g. text file, a database via JDBC etc. and easily handle data with no predefined structure.
- DataFrame – Data source API allows Data processing in different formats (AVRO, CSV, JSON, and storage system HDFS, HIVE tables, MySQL). It can read and write from various data sources that are mentioned above.
- DataSet – Dataset API of spark also support data from different sources.

## 5] **Immutability and Interoperability**

- RDD – RDDs contains the collection of records which are partitioned. The basic unit of parallelism in an RDD is called partition. Each partition is one logical division of data which is immutable and created through some transformation on existing partitions. Immutability helps to achieve consistency in computations. We can move from RDD to DataFrame (If RDD is in tabular format) by toDF() method or we can do the reverse by the .rdd method.
- DataFrame – After transforming into DataFrame one cannot regenerate a domain object. For example, if you generate testDF from testRDD, then you won't be able to recover the original RDD of the test class.
- DataSet – It overcomes the limitation of DataFrame to regenerate the RDD from DataFrame.

## 6] **Compile-time type safety**

- RDD – RDD provides a familiar object-oriented programming style with compile-time type safety.
- DataFrame – If you are trying to access the column which does not exist in the table in such case DataFrame APIs does not support compile-time error. It detects attribute error only at runtime.
- DataSet – It provides compile-time type safety.

## 7] **Optimization**

- RDD – No inbuilt optimization engine is available in RDD. When working with structured data, RDDs cannot take advantages of sparks advance optimizers. For example, catalyst optimizer and Tungsten execution engine. Developers optimize each RDD on the basis of its attributes.
- DataFrame – Optimization takes place using catalyst optimizer. DataFrames use catalyst tree transformation framework in four phases: a) Analyzing a logical plan to resolve references. b) Logical plan optimization. c) Physical planning. d) Code generation to compile parts of the query to Java bytecode.
- Dataset – It includes the concept of DataFrame Catalyst optimizer for optimizing query plan.

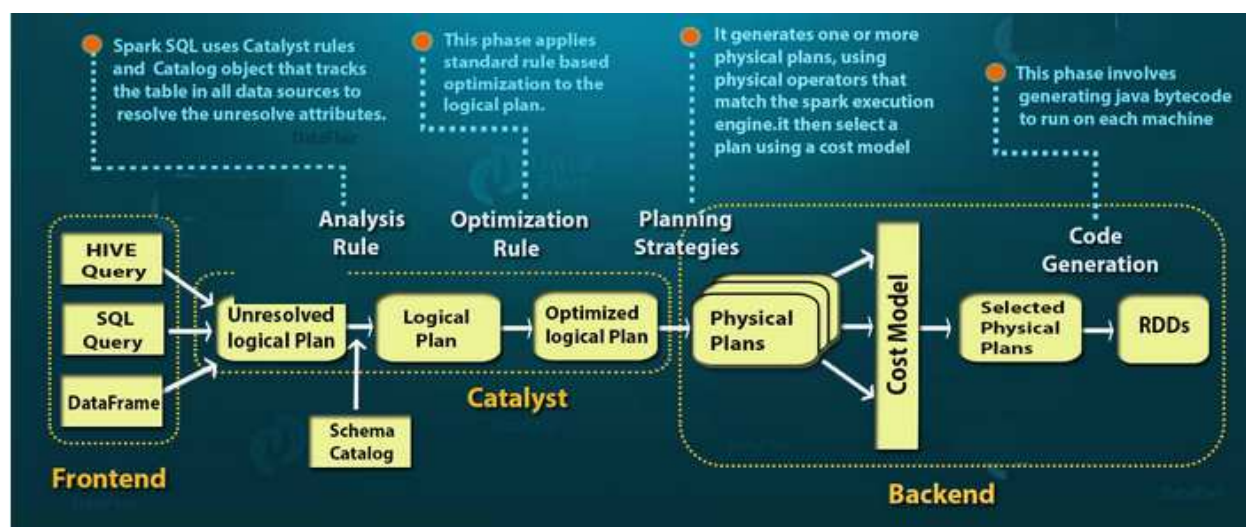
## 8] **Serialization**

- RDD – Whenever Spark needs to distribute the data within the cluster or write the data to disk, it does so use Java serialization. The overhead of serializing individual Java and Scala objects is expensive and requires sending both data and structure between nodes.
- DataFrame – Spark DataFrame Can serialize the data into off-heap storage (in memory) in binary format and then perform many transformations directly on this off heap memory because spark understands the schema. There is no need to use java serialization to encode the data. It provides a Tungsten physical execution backend which explicitly manages memory and dynamically generates bytecode for expression evaluation.
- DataSet – When it comes to serializing data, the Dataset API in Spark has the concept of an encoder which handles conversion between JVM objects to tabular representation. It stores tabular representation using spark internal Tungsten binary format. Dataset allows

performing the operation on serialized data and improving memory use. It allows on-demand access to individual attribute without de-serializing the entire object.

<https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>

<https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html>



## The DataFrame from a Java perspective

If your background is Java and you have some JDBC experience, then the DataFrame will look like a ResultSet. It contains data, it has an API.

Similarities:

- Data is accessible through a simple API.
- You can access the schema.

Differences:

- You do not browse through it with a next() method.
- Its API is extensible through UDF (user-defined function): you can write or wrap existing code and add it to Spark. This code will then be accessible in a distributed mode. You will study UDF in chapter 16.
- If you want to access the data, you first get the Row then goes through the columns of the row with getters (similar to a ResultSet).
- Metadata is fairly basic as there is no primary or foreign keys or index in Spark.

In Java, a DataFrame is implemented as a Dataset<Row> (pronounced “a dataset of rows”).

### The DataFrame from an RDBMS perspective

If you come more from an RDBMS background, you may find that a DataFrame is like a table.

Similarities:

- Data is described in columns and rows.
- Columns are strongly typed.

Differences:

- Data can be nested, like in a JSON or XML document.
- You don't update or delete entire rows: you create new DataFrames.
- You can easily add or remove columns.
- There are no constraints, indices, primary, or foreign keys, or triggers on the DataFrame.

### Example 17

```
package com.evenkat;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;

public class CsvToDataframeApp {

    public static void main(String[] args) {
        CsvToDataframeApp app = new CsvToDataframeApp();
        app.start();
    }

    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        Session spark = Session.builder()
            .appName("CSV to Dataset")
            .master("local")
            .getOrCreate();

        Dataset<Row> df = spark.read().format("csv")
            .option("header", "true")
            .load("in/books.csv");

        df.show(5);
    }
}
```

Ensure that you create an in folder and have books.csv file over there.



Ensure that when you install postgresdb I have used the default username and the password used is hadoop123. You will have to remember your password.

**Also when accessing the database we will be using a Maven Project so that the dependencies will be downloaded.**

```
Properties prop = new Properties();
    prop.setProperty("driver", "org.postgresql.Driver");
    prop.setProperty("user", "postgres");
    prop.setProperty("password", "hadoop123");
```

### Example 18

```
import static org.apache.spark.sql.functions.concat;
import static org.apache.spark.sql.functions.lit;

import java.util.Properties;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SaveMode;
import org.apache.spark.sql.Session;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;

public class CsvToRelationalDatabaseApp {

    public static void main(String[] args) {
        CsvToRelationalDatabaseApp app = new CsvToRelationalDatabaseApp();
        app.start();
    }

    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        Session spark = Session.builder() //
            .appName("CSV to DB")
            .master("local")
            .getOrCreate();
        Dataset<Row> df = spark.read() //
            .format("csv")
            .option("header", "true")
            .load("in/authors.csv");
        df = df.withColumn( //
            "name",
            concat(df.col("lname"), //
                lit(", "), df.col("fname"))); //
        String dbConnectionUrl = "jdbc:postgresql://localhost/course_data";
        // We should ensure that the database is created.
        Properties prop = new Properties(); //
        prop.setProperty("driver", "org.postgresql.Driver");
        prop.setProperty("user", "postgres");
        prop.setProperty("password", "hadoop123");
```

```

df.write() //
  .mode(SaveMode.Overwrite)
  .jdbc(dbConnectionUrl, "project2", prop);

System.out.println("Process complete");
}
}

```

**The standard pom.xml file will look like this:**

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.spark</groupId>
  <artifactId>project1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
    <scala.version>2.11</scala.version>
    <spark.version>2.3.1</spark.version>
    <postgresql.version>42.1.4</postgresql.version>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <!-- Spark -->
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_${scala.version}</artifactId>
      <version>${spark.version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_${scala.version}</artifactId>
      <version>${spark.version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-mllib_${scala.version}</artifactId>
      <version>${spark.version}</version>
    </dependency>
  </dependencies>

```

```

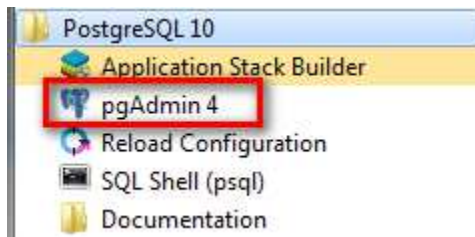
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>${postgresql.version}</version>
    </dependency>
  </dependencies>

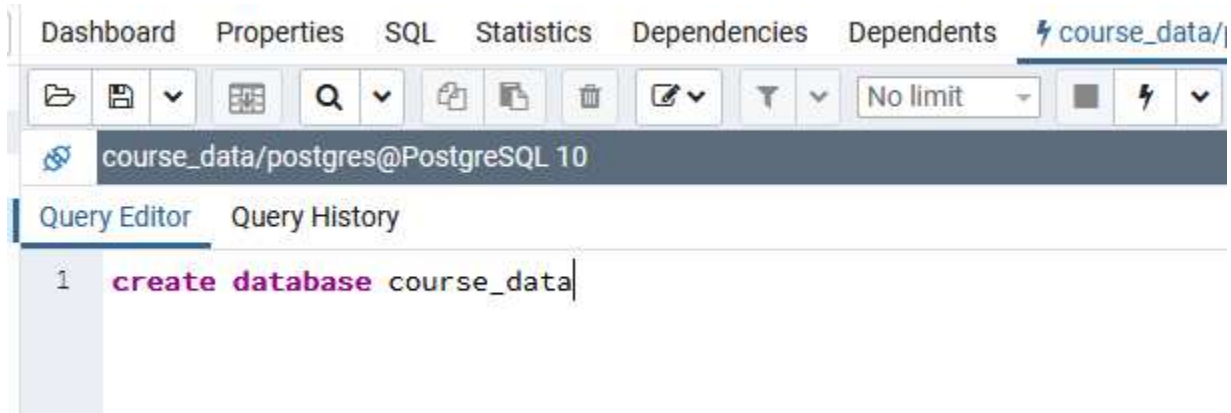
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <executions>
        <execution>
          <id>copy-dependencies</id>
          <phase>prepare-package</phase>
          <goals>
            <goal>copy-dependencies</goal>
          </goals>
          <configuration>
            <outputDirectory>
              ${project.build.directory}/libs
            </outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
          <configuration>
            <mainClass>com.spark.Application</mainClass>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
</project>

```

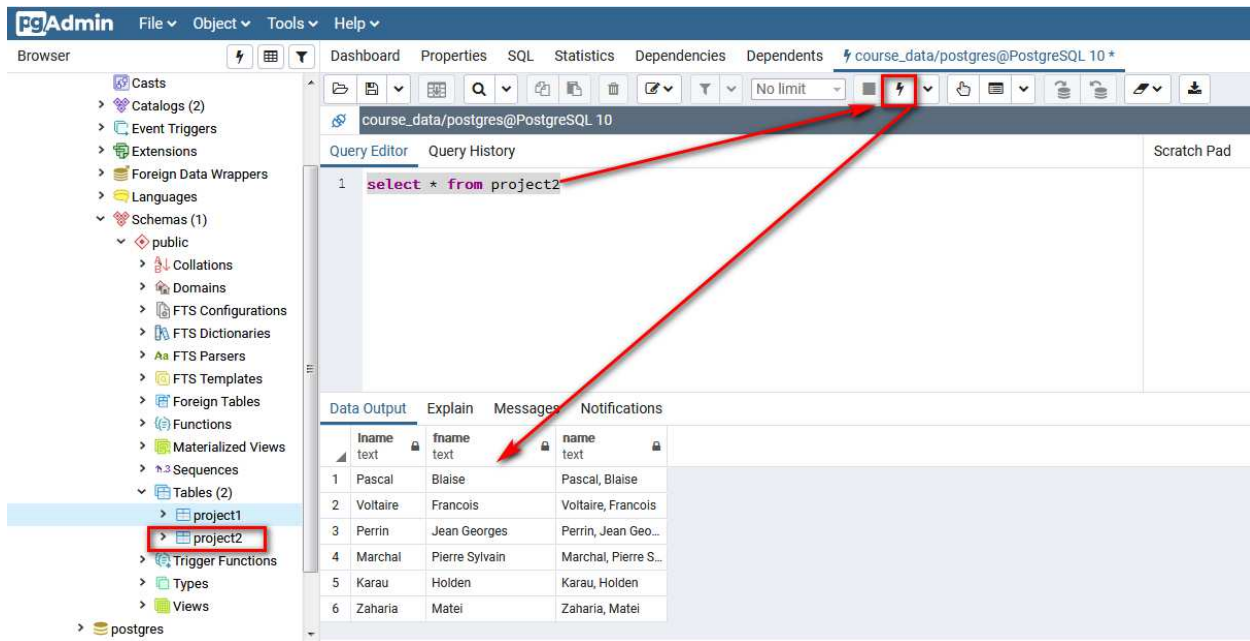
Start pg Admin 4 tool from here:



Before running the sample, ensure that you have created the database like this :-



It will take you to a browser view from where can execute the queries :-



### Example 19. Another example to reinforce the concepts.

```
package com.evenkat;
```

```
import static org.apache.spark.sql.functions.concat;
import static org.apache.spark.sql.functions.lit;
```

```
import java.util.Properties;
```

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SaveMode;
import org.apache.spark.sql.SparkSession;
```

```
public class Application {
```

```
    public static void main(String args[]) throws InterruptedException {
        Logger.getLogger("org").setLevel(Level.ERROR);
        // Create a session
        SparkSession spark = new SparkSession.Builder()
            .appName("CSV to DB")
            .master("local[*]")
            .getOrCreate();
```

```
        // get data
        Dataset<Row> df = spark.read().format("csv")
            .option("header", true)
            .load("in/name_and_comments.txt");
```

```
//        df.show();
```

```
// Transformation
df = df.withColumn("full_name",
                  concat(df.col("last_name"), lit(", "), df.col("first_name")))
                  .filter(df.col("comment").rlike("\\d+"))
                  .orderBy(df.col("last_name").asc());

//we are first doing a concat function for full name and adding a literal
//with , so that it will come as a fourth column

// Then we are doing a filter with a rlike function.

// Then we do a orderBy on the last_name col with ascending option.

//DataSets are immutable

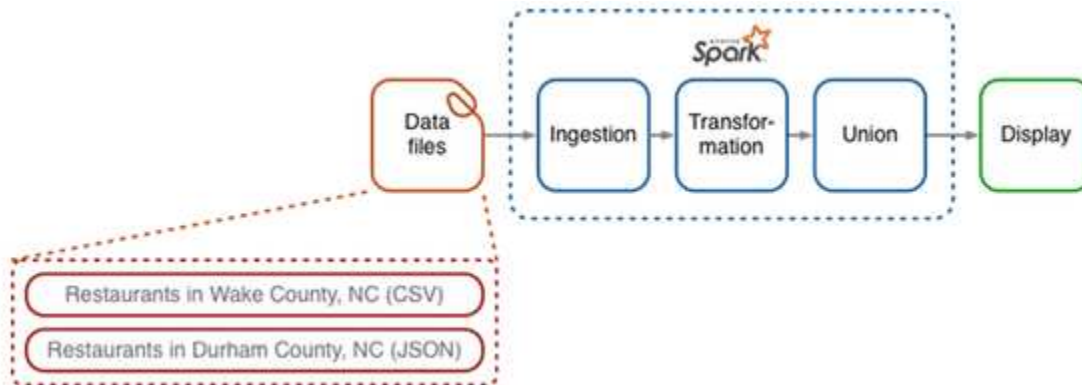
// Write to destination
//      df.show();
String dbConnectionUrl = "jdbc:postgresql://localhost/course_data"; // <<- You
need to create this database as shown earlier.
Properties prop = new Properties();
prop.setProperty("driver", "org.postgresql.Driver");
prop.setProperty("user", "postgres");
prop.setProperty("password", "hadoop123"); // <- The password you used while
installing Postgres

df.write()
  .mode(SaveMode.Overwrite)
  .jdbc(dbConnectionUrl, "project1", prop);
}
```

|   | A     | B         | C         | D        | E            | F         | G            | H      | I          | J          | K   | L         | M      | N     | O               | P       |
|---|-------|-----------|-----------|----------|--------------|-----------|--------------|--------|------------|------------|---|-----------|--------|-------|-----------------|---------|
| 1 |       | collector | country   | un_subre | so_region    | age_range | age_midpoint | gender | self_id    | occupation | occupatio   | exper     | exper  | salar | salary_midpoint | big_mac |
| 2 | 1888  | Facebook  | Afghanist | Southern | Central As   | 20-24     |              | 22     | Male       | Programmer |   |           |        |       |                 |         |
| 3 | 4637  | Facebook  | Afghanist | Southern | Central As   | 30-34     |              | 32     | Male       | Develope   | Mobile developer - iOS  | Mobile De | 6 - 10 | 8     | \$40,0          | 45000   |
| 4 | 11164 | Facebook  | Afghanist | Southern | Central Asia |           |              |        |            |            |   |           |        |       |                 |         |
| 5 | 21378 | Facebook  | Afghanist | Southern | Central Asia |           |              |        | Female     | Engineer   | DevOps  | DevOps    | 11+ ye | 13    | Less t          | 5000    |
| 6 | 30280 | Facebook  | Afghanist | Southern | Central As   | > 60      |              | 65     | Prefer not | Developer  | Engineer; Programmer; Sr. Developer; Manager; Rockstar; Ninja; Guru; Expert; Full-stack Developer; Full Stack O |           |        |       |                 |         |

In this you will perform two simple ingestions, then, you will study the schemas and storage part, in order to understand the behavior of DataFrames as it is used in an application. The first ingestion is a list of restaurants in Wake County, North Carolina, the second data set consists of restaurants in Durham County, North Carolina. You will then transform the dataset so you will be able to combine them in a union.

Restaurants\_in\_Durham\_County\_NC.csv  
Restaurants\_in\_Wake\_County\_NC.csv



The destination DataFrame, after the transformation, needs to have the same schema after both transformations.

### Example 20

```
package com.evenkat;
```

```
import static org.apache.spark.sql.functions.concat;
import static org.apache.spark.sql.functions.lit;
```

```
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.Partition;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;
```

```
public class WakeCountyApp {
```

```
    public static void main(String[] args) {
        WakeCountyApp app =
            new WakeCountyApp();
        app.start();
    }
```

```
    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkSession spark = SparkSession.builder()
            .appName("Restaurants in Wake County, NC")
            .master("local")
            .getOrCreate();
```

```
        Dataset<Row> df = spark.read().format("csv")
            .option("header", "true")
            .load("in/Restaurants_in_Wake_County_NC.csv");
```

```

Dataset<Row> df1 = spark.read().format("csv")
    .option("header", "true")
    .load("in/Restaurants_in_Durham_County_NC.csv");

System.out.println("*** Right after ingestion Printing Restaurants_in_Wake_County_NC");
df.show(5);
df.printSchema();
System.out.println("We have " + df.count() + " records.");

System.out.println("*** Right after ingestion Printing Restaurants_in_Durham_County_NC");
df1.show(5);
df1.printSchema();
System.out.println("We have " + df1.count() + " records.");
}
}

```

\*\*\* Right after ingestion

| OBJECTID | HSISID      | NAME                 | ADDRESS1             | ADDRESS2 | CITY  | STATE | POSTALCODE |
|----------|-------------|----------------------|----------------------|----------|-------|-------|------------|
| 58442    | 04092017687 | Pho Sure             | 7451 Six Forks RD    | null     | 27615 | NC    | 27615      |
| 58443    | 04092016126 | PANERA BREAD #1650   | 1001 BEAVER CREEK... | null     | APEX  | NC    | 27502      |
| 58444    | 04092025146 | Harris Teeter #58... | 750 W Williams ST    | null     | Apex  | NC    | 27502      |
| 58445    | 04092012835 | Mcdonald's #17721    | 900 US 64 HWY        | null     | APEX  | NC    | 27502      |
| 58446    | 04092021939 | DONOVAN'S DISH       | 800 W WILLIAMS ST    | STE 112  | APEX  | NC    | 27502      |

| PHONENUMBER    | RESTAURANTOPENDATE   | FACILITYTYPE | PERMITID | X            | Y           | GEOCODESTATUS |
|----------------|----------------------|--------------|----------|--------------|-------------|---------------|
| (919) 745-1168 | 2017-09-27T00:00:... | Restaurant   | 16446    | -78.64620405 | 35.88177389 | M             |
| (919) 589-0026 | 2012-03-27T00:00:... | Restaurant   | 9256     | -78.87238568 | 35.74374477 | M             |
| (919) 362-3782 | 2017-10-11T00:00:... | Food Stand   | 17466    | -78.86276064 | 35.73803218 | M             |
| (919) 380-0906 | 1998-11-16T00:00:... | Restaurant   | 3251     | 0            | 0           | U             |
| (919) 651-8309 | 2016-02-08T00:00:... | Food Stand   | 20625    | -78.86480181 | 35.73964723 | M             |

root

```

|-- OBJECTID: string (nullable = true)
|-- HSISID: string (nullable = true)
|-- NAME: string (nullable = true)
|-- ADDRESS1: string (nullable = true)
|-- ADDRESS2: string (nullable = true)
|-- CITY: string (nullable = true)
|-- STATE: string (nullable = true)
|-- POSTALCODE: string (nullable = true)
|-- PHONENUMBER: string (nullable = true)
|-- RESTAURANTOPENDATE: string (nullable = true)
|-- FACILITYTYPE: string (nullable = true)
|-- PERMITID: string (nullable = true)
|-- X: string (nullable = true)
|-- Y: string (nullable = true)
|-- GEOCODESTATUS: string (nullable = true)

```



We have 3623 records.

```
ID;Premise_Name;Premise_Address1;Premise_Address2;Premise_City;Premise_State;Premise_Zip;
Premise_Phone;
Hours_Of_Operation;Opening_Date;Closing_Date;Seats;Water;Sewage;Insp_Freq;Est_Group_Desc;Risk;
Smoking_Allowed;Type_Description;Rpt_Area_Desc;Status;Transitional_Type_Desc;geolocation
```

```
root
```

```
|--
ID;Premise_Name;Premise_Address1;Premise_Address2;Premise_City;Premise_State;Premise_Zip;Prem
ise_Phone;Hours_Of_Operation;Opening_Date;Closing_Date;Seats;Water;Sewage;Insp_Freq;Est_Group
_Desc;Risk;Smoking_Allowed;Type_Description;Rpt_Area_Desc;Status;Transitional_Type_Desc;geoloca
tion: string (nullable = true)
```

We have 2472 records.

The goal is to merge two DataFrames, like an SQL union of two tables. To make the union effective, you will need to have similarly named columns in both DataFrames. **To get there, you can easily imagine that the schema of our first dataset was modified too, this is how it looks like:**

```
root
```

```
|-- datasetId: string (nullable = true)
|-- name: string (nullable = true)
|-- address1: string (nullable = true)
|-- address2: string (nullable = true)
|-- city: string (nullable = true)
|-- state: string (nullable = true)
|-- zip: string (nullable = true)
|-- tel: string (nullable = true)
|-- dateStart: string (nullable = true)
|-- type: string (nullable = true)
|-- geoX: string (nullable = true)
|-- geoY: string (nullable = true)
|-- county: string (nullable = false)
|-- id: string (nullable = true)
```

You will use four methods of the DataFrame and two static functions.

```
df = df.withColumn("county", lit("Wake"))

    .withColumnRenamed("HSISID", "datasetId")

    .withColumnRenamed("NAME", "name")

    .withColumnRenamed("ADDRESS1", "address1")

    .withColumnRenamed("ADDRESS2", "address2")
```

```

.withColumnRenamed("CITY", "city")
.withColumnRenamed("STATE", "state")
.withColumnRenamed("POSTALCODE", "zip")
.withColumnRenamed("PHONENUMBER", "tel")
.withColumnRenamed("RESTAURANTOPENDATE", "dateStart")
.withColumnRenamed("FACILITYTYPE", "type")
.withColumnRenamed("X", "geoX")
.withColumnRenamed("Y", "geoY")
.drop("OBJECTID")
.drop("PERMITID")
.drop("GEOCODESTATUS");

```

A Creating a new column called “county” containing the value “Wake” in every record

B Simply renaming column names to what you need in your new data set

C Columns to be dropped

Let’s look at the methods and functions you need now:

- withColumn() (method) creates a new column from an expression or a column.
- withColumnRenamed() (method) renames a column.
- col() (method) gets a column from its name. Some methods will take the column name as an argument, some requires a Column object.
- drop() (method) drops a column from the dataframe. This method accepts both an instance of a Column object or a column name.
- lit() (functions) creates a column with a value, literally, a literal value.
- concat() (functions) concatenates the values in a set of columns.

You also need a unique identifier for each record. You can call this column “id” and build it by concatenating:

1. The state.
2. An underscore (\_).
3. The county.
4. An underscore (\_).
5. The identifier within the dataset.

```

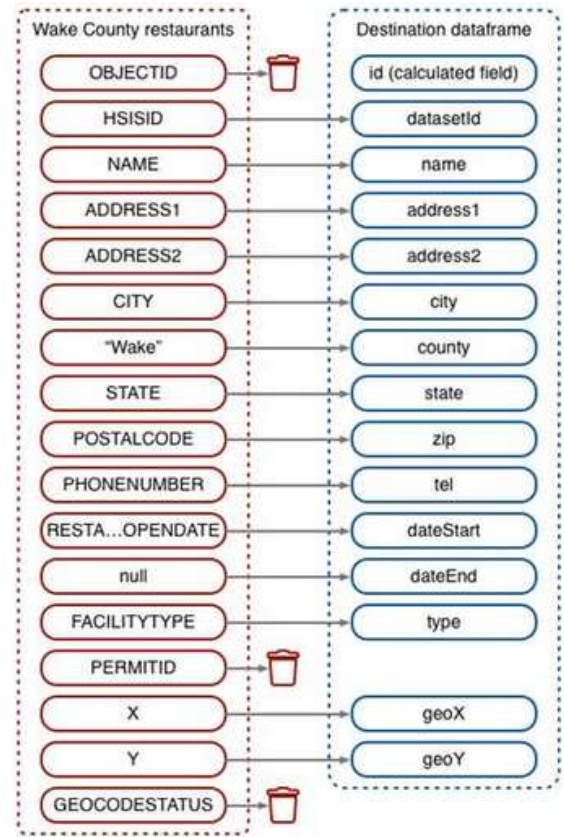
df = df.withColumn("id", concat(
    df.col("state"), lit("_"),
    df.col("county"), lit("_"),
    df.col("datasetId")))

```

```

System.out.println("*** Dataframe transformed");
df.show(5);
df.printSchema();

```



Partitions are created and data is assigned to each partition automatically based on your infrastructure (number of nodes and size of the dataset). You can repartition the DataFrame to use four partitions using the `repartition()` method. Repartition can increase performance and that would be the main reason for it

```

System.out.println("*** Looking at partitions");
Partition[] partitions = df.rdd().partitions();
int partitionCount = partitions.length;
System.out.println("Partition count before repartition: " +
partitionCount);

```

```

df = df.repartition(4);
System.out.println("Partition count after repartition: " +
df.rdd().partitions().length);

```

**The complete code with the above changes:-**

```

package com.evenkat;

```

```

import static org.apache.spark.sql.functions.concat;
import static org.apache.spark.sql.functions.lit;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.Partition;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;
import org.apache.spark.sql.types.StructType;

public class WakeCountyApp2 {

    public static void main(String[] args) {
        WakeCountyApp2 app =
            new WakeCountyApp2();
        app.start();
    }

    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkSession spark = SparkSession.builder()
            .appName("Restaurants in Wake County, NC")
            .master("local")
            .getOrCreate();

        Dataset<Row> df = spark.read().format("csv")
            .option("header", "true")
            .load("in/Restaurants_in_Wake_County_NC.csv");

        Dataset<Row> df1 = spark.read().format("csv")
            .option("header", "true")
            .load("in/Restaurants_in_Durham_County_NC.csv");

        //      System.out.println("**** Right after ingestion Printing Restaurants_in_Wake_County_NC");
        //      df.show(5);
        //      df.printSchema();
        //      System.out.println("We have " + df.count() + " records.");

        //      System.out.println("**** Right after ingestion Printing
        Restaurants_in_Durham_County_NC");
        //      df1.show(5);
        //      df1.printSchema();
        //      System.out.println("We have " + df1.count() + " records.");

        df = df.withColumn("county", lit("Wake"))
            .withColumnRenamed("HSISID", "datasetId")

```

```

        .withColumnRenamed("NAME", "name")
        .withColumnRenamed("ADDRESS1", "address1")
        .withColumnRenamed("ADDRESS2", "address2")
        .withColumnRenamed("CITY", "city")
        .withColumnRenamed("STATE", "state")
        .withColumnRenamed("POSTALCODE", "zip")
        .withColumnRenamed("PHONENUMBER", "tel")
        .withColumnRenamed("RESTAURANTOPENDATE", "dateStart")
        .withColumnRenamed("FACILITYTYPE", "type")
        .withColumnRenamed("X", "geoX")
        .withColumnRenamed("Y", "geoY")
        .drop("OBJECTID")
        .drop("PERMITID")
        .drop("GEOCODESTATUS");

df = df.withColumn("id", concat(
    df.col("state"), lit("_"),
    df.col("county"), lit("_"),
    df.col("datasetId")));

System.out.println("*** Dataframe transformed");
df.show(5);
df.printSchema();

System.out.println("*** Looking at partitions");
Partition[] partitions = df.rdd().partitions();
int partitionCount = partitions.length;
System.out.println("Partition count before repartition: " +
    partitionCount);

df = df.repartition(4);
System.out.println("Partition count after repartition: " +
    df.rdd().partitions().length);
}
}

```

The output of the earlier program.

\*\*\* Dataframe transformed

| datasetId   | name                 | address1             | address2 | city  | state | zip   | tel            |
|-------------|----------------------|----------------------|----------|-------|-------|-------|----------------|
| 04092017687 | Pho Sure             | 7451 Six Forks RD    | null     | 27615 | NC    | 27615 | (919) 745-1168 |
| 04092016126 | PANERA BREAD #1650   | 1001 BEAVER CREEK... | null     | APEX  | NC    | 27502 | (919) 589-0026 |
| 04092025146 | Harris Teeter #58... | 750 W Williams ST    | null     | Apex  | NC    | 27502 | (919) 362-3782 |
| 04092012835 | Mcdonald`s #17721    | 900 US 64 HWY        | null     | APEX  | NC    | 27502 | (919) 380-0906 |
| 04092021939 | DONOVAN'S DISH       | 800 W WILLIAMS ST    | STE 112  | APEX  | NC    | 27502 | (919) 651-8309 |

| dateStart            | type       | geoX         | geoY        | county | id                  |
|----------------------|------------|--------------|-------------|--------|---------------------|
| 2017-09-27T00:00:... | Restaurant | -78.64620405 | 35.88177389 | Wake   | NC_Wake_04092017687 |
| 2012-03-27T00:00:... | Restaurant | -78.87238568 | 35.74374477 | Wake   | NC_Wake_04092016126 |
| 2017-10-11T00:00:... | Food Stand | -78.86276064 | 35.73803218 | Wake   | NC_Wake_04092025146 |
| 1998-11-16T00:00:... | Restaurant | 0            | 0           | Wake   | NC_Wake_04092012835 |
| 2016-02-08T00:00:... | Food Stand | -78.86480181 | 35.73964723 | Wake   | NC_Wake_04092021939 |

only showing top 5 rows

root

```
-- datasetId: string (nullable = true)
-- name: string (nullable = true)
-- address1: string (nullable = true)
-- address2: string (nullable = true)
-- city: string (nullable = true)
-- state: string (nullable = true)
-- zip: string (nullable = true)
-- tel: string (nullable = true)
-- dateStart: string (nullable = true)
-- type: string (nullable = true)
-- geoX: string (nullable = true)
-- geoY: string (nullable = true)
-- county: string (nullable = false)
-- id: string (nullable = true)
```

\*\*\* Looking at partitions

Partition count before repartition: 1

Partition count after repartition: 4

### Now digging in to the schema

We learnt about having access to the schema using `printSchema()`. It is important to know the structure of the data, and specifically how Spark sees them. There is a way to know more details about the schema by calling the `schema()` method

```
StructType schema = df.schema();
System.out.println("*** Schema as a tree:");
schema.printTreeString();
String schemaAsString = schema.mkString();
System.out.println("*** Schema as string: " + schemaAsString);
String schemaAsJson = schema.prettyJson();
System.out.println("*** Schema as JSON: " + schemaAsJson);
```

**Add this at the end and test it out and it will look like this:**

\*\*\* Schema as a tree:

root

```
-- datasetId: string (nullable = true)
```

```

|-- name: string (nullable = true)
|-- address1: string (nullable = true)
|-- address2: string (nullable = true)
|-- city: string (nullable = true)
|-- state: string (nullable = true)
|-- zip: string (nullable = true)
|-- tel: string (nullable = true)
|-- dateStart: string (nullable = true)
|-- type: string (nullable = true)
|-- geoX: string (nullable = true)
|-- geoY: string (nullable = true)
|-- county: string (nullable = false)
|-- id: string (nullable = true)

```

\*\*\* Schema as string:

```

StructField(datasetId,StringType,true)StructField(name,StringType,true)StructField(address1,StringType,
true)StructField(address2,StringType,true)StructField(city,StringType,true)StructField(state,StringType,tr
ue)StructField(zip,StringType,true)StructField(tel,StringType,true)StructField(dateStart,StringType,true)S
tructField(type,StringType,true)StructField(geoX,StringType,true)StructField(geoY,StringType,true)Struct
Field(county,StringType,false)StructField(id,StringType,true)

```

\*\*\* Schema as JSON: {

```

  "type" : "struct",
  "fields" : [ {
    "name" : "datasetId",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "name",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "address1",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "address2",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "city",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {

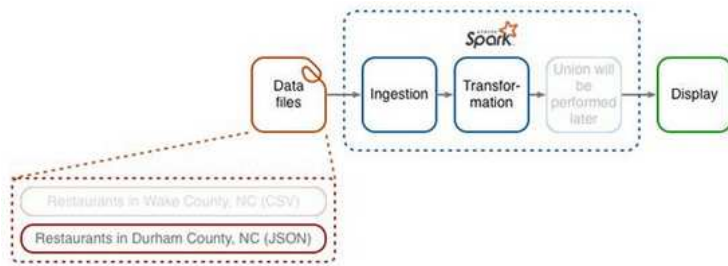
```

```

    "name" : "state",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "zip",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "tel",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "dateStart",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "type",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "geoX",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "geoY",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "county",
    "type" : "string",
    "nullable" : false,
    "metadata" : { }
  }, {
    "name" : "id",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }
}

```





```

SparkSession spark = SparkSession.builder()
    .appName("Restaurants in Durham County, NC")
    .master("local[*]")
    .getOrCreate();

// Reads a JSON file called Restaurants_in_Durham_County_NC.json, stores
// it in a dataframe
Dataset<Row> df = spark.read().format("json")
    .load("in/Restaurants_in_Durham_County_NC.json");
System.out.println("*** Right after ingestion");
df.show(5);
df.printSchema();

```

Once the data is in the DataFrame, APIs to manipulate the data are the same and hence we can start transforming the DataFrame.

```

*** Dataframe transformed
+-----+-----+-----+-----+
|datasetId|          fields|          geometry|    record_timestamp|...
+-----+-----+-----+-----+
|   56060|[, Full-Service R...|[-78.9573299, 35...|2017-07-13T09:15:...|...
|   58123|[, Nursing Home, ...|[-78.8895483, 36...|2017-07-13T09:15:...|...
|   70266|[, Fast Food Rest...|[-78.9593263, 35...|2017-07-13T09:15:...|...
|   97837|[, Full-Service R...|[-78.9060312, 36...|2017-07-13T09:15:...|...
|   60690|[, , [36.0556347, ...|[-78.9135175, 36...|2017-07-13T09:15:...|...
+-----+-----+-----+-----+

```

```

root
 |-- datasetId: string (nullable = true)      B
 |-- fields: struct (nullable = true)        A
 ...
 |    |-- premise_name: string (nullable = true)
 ...
 |-- geometry: struct (nullable = true)
 ...
 |-- record_timestamp: string (nullable = true)

```

```

|-- recordid: string (nullable = true)
|-- county: string (nullable = false)
|-- name: string (nullable = true)
|-- address1: string (nullable = true)
|-- address2: string (nullable = true)
|-- city: string (nullable = true)
|-- state: string (nullable = true)
|-- zip: string (nullable = true)
|-- tel: string (nullable = true)
|-- dateStart: string (nullable = true)
|-- dateEnd: string (nullable = true)
|-- type: string (nullable = true)
|-- geoX: double (nullable = true)
|-- geoY: double (nullable = true)
|-- id: string (nullable = true)

```

- A The fields coming from the original dataset, before the transformation  
 B The new fields you will create

To access the fields in a structure, you can:

- 1) Use the dot (.) symbol in the path.
- 2) To access an element in an array, use the getItem() method.

```

df = df.withColumn("county", lit("Durham"))
      .withColumn("datasetId", df.col("fields.id"))
      .withColumn("name", df.col("fields.premise_name"))
      .withColumn("address1", df.col("fields.premise_address1"))
      .withColumn("address2", df.col("fields.premise_address2"))
      .withColumn("city", df.col("fields.premise_city"))
      .withColumn("state", df.col("fields.premise_state"))
      .withColumn("zip", df.col("fields.premise_zip"))
      .withColumn("tel", df.col("fields.premise_phone"))
      .withColumn("dateStart", df.col("fields.opening_date"))
      .withColumn("dateEnd", df.col("fields.closing_date"))
      .withColumn("type",
        split(df.col("fields.type_description"), " - ").getItem(1))
      .withColumn("geoX", df.col("fields.geolocation").getItem(0))
      .withColumn("geoY", df.col("fields.geolocation").getItem(1));

```

- A As with the CSV, you can add a column with the county name  
 B Accessing the nested fields with . (dot)  
 C The description has the <id> - <label> notation, you can split the field on " - " and get the second element  
 D Extract the first element of the array as the latitude (geoX)  
 E Extract the second element of the array as the longitude (geoY)  
 As you created all the fields and columns, creating the id field is the same operation you did for the CSV file

```

df = df.withColumn("id",
  concat(df.col("state"), lit("_"),
    df.col("county"), lit("_"),
    df.col("datasetId")));

```

```

System.out.println("*** Dataframe transformed");
df.show(5);
df.printSchema();

```

Finally, looking at partitions is also equivalent as for your CSV file.

The **code** for doing all of this is below:

```

package com.evenkat;

import static org.apache.spark.sql.functions.concat;
import static org.apache.spark.sql.functions.lit;
import static org.apache.spark.sql.functions.split;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.Partition;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;

public class DurhamCountyApp {

    public static void main(String[] args) {
        DurhamCountyApp app =
            new DurhamCountyApp();
        app.start();
    }

    private void start() {
        // Creates a session on a local master
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkSession spark = SparkSession.builder()
            .appName("Restaurants in Durham County, NC")
            .master("local[*]")
            .getOrCreate();

        // Reads a JSON file called Restaurants_in_Durham_County_NC.json, stores
        // it in a dataframe
        Dataset<Row> df = spark.read().format("json")
            .load("in/Restaurants_in_Durham_County_NC.json");
        System.out.println("*** Right after ingestion");
        df.show(5);
        df.printSchema();
        System.out.println("We have " + df.count() + " records.");

        df = df.withColumn("county", lit("Durham"))
            .withColumn("datasetId", df.col("fields.id"))
            .withColumn("name", df.col("fields.premise_name"))
    }
}

```

```

        .withColumn("address1", df.col("fields.premise_address1"))
        .withColumn("address2", df.col("fields.premise_address2"))
        .withColumn("city", df.col("fields.premise_city"))
        .withColumn("state", df.col("fields.premise_state"))
        .withColumn("zip", df.col("fields.premise_zip"))
        .withColumn("tel", df.col("fields.premise_phone"))
        .withColumn("dateStart", df.col("fields.opening_date"))
        .withColumn("dateEnd", df.col("fields.closing_date"))
        .withColumn("type",
            split(df.col("fields.type_description"), " - ").getItem(1))
        .withColumn("geoX", df.col("fields.geolocation").getItem(0))
        .withColumn("geoY", df.col("fields.geolocation").getItem(1));
df = df.withColumn("id",
    concat(df.col("state"), lit("_"),
        df.col("county"), lit("_"),
        df.col("datasetId")));

System.out.println("*** Dataframe transformed");
df.show(5);
df.printSchema();

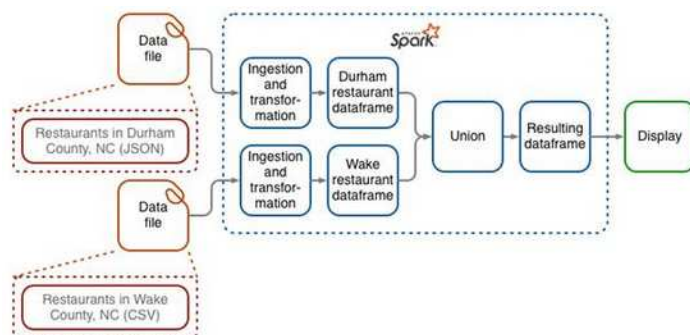
System.out.println("*** Looking at partitions");
Partition[] partitions = df.rdd().partitions();
int partitionCount = partitions.length;
System.out.println("Partition count before repartition: " +
    partitionCount);

df = df.repartition(4);
System.out.println("Partition count after repartition: " +
    df.rdd().partitions().length);
    }
}

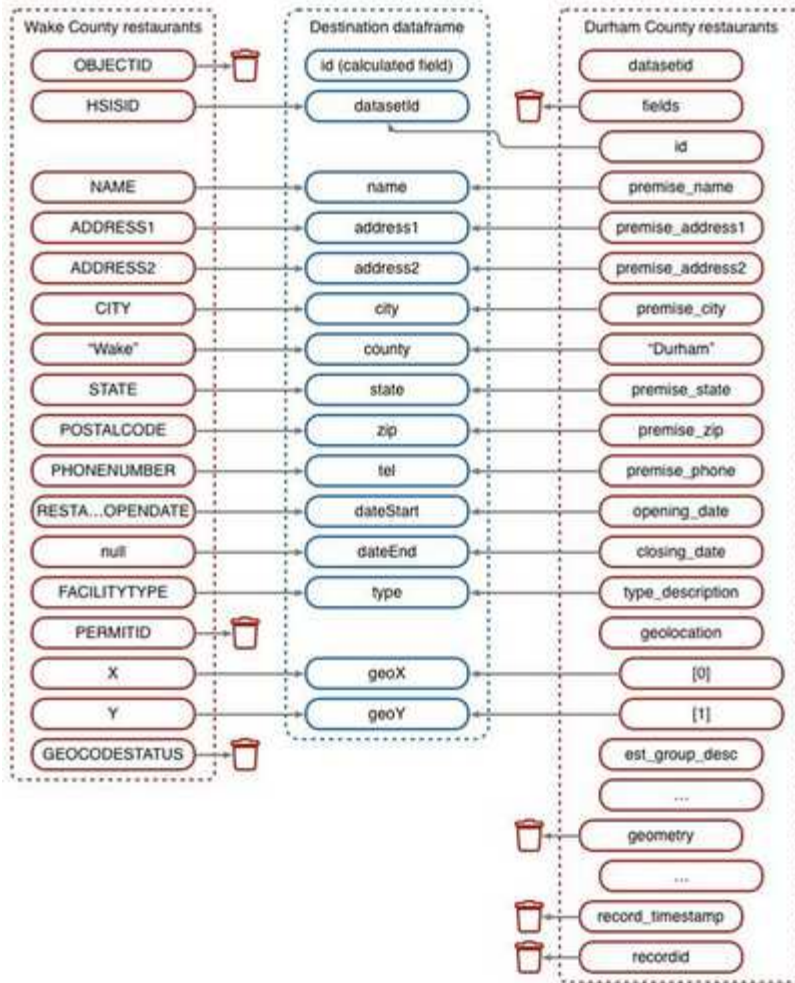
```

## Combining two DataFrames

We will combine two datasets, in a SQL union-like way to build a bigger dataset. This will allow you to perform analytics on more data points.



We can reuse most of the code you wrote for the ingestion and transformation. However, to perform a union, you will have to make sure that the schemas are strictly identical, otherwise Spark will not be able to perform the union.



Imports are the same, to make it a little easier, the SparkSession's instance is a private member, initialized in the start() method. The rest is isolated in three methods:

1. buildWakeRestaurantsDataframe () builds the DataFrame containing the restaurants in Wake County.
2. buildDurhamRestaurantsDataframe() builds the DataFrame containing the restaurants in Durham County.
3. combineDataframes() combines the two DataFrame using a SQL-like union.

**Note that** when you drop a parent column in buildDurhamRestaurantsDataframe, all the nested columns are being dropped as well. The nested columns under the fields and geometry fields are dropped as you dropped the parent. So, when you dropped the field's column, all subfields like risk, seats, sewage, and so on are being dropped at the same time.

You now have two DataFrames, with the same number of columns, you can now union them in the `combineDataframes()` method. There are two ways of combining two DataFrames using a SQL-like union: you can use the `union()` and `unionByName()` methods.

The `union()` method does not care about the name of the column, just the order of them: it will always union column 1 from the first DataFrame, with column 1 from the second dataset, then move to column 2, 3... regardless of their name. After a few transformative operations, where you create new columns, rename them, dumps them, or combine them, it is very difficult to remember if they are in the right order. If fields don't match, it may result in inconsistent data at worse, a program stop at best. On the other hand, `unionByName()` matches column by names, which is safer.

**Note that** both methods require to have the same number of columns on both sides

When you load a small (typically under 128MB) dataset in a DataFrame, Spark will only create one partition. However, in this scenario Spark created a partition for the CSV-based dataset and one for the JSON-based dataset: two datasets in two distinct DataFrames result in at least two partitions (at least one for each dataset). Joining them will create a unique DataFrame, but it will rely on the two original partitions (or more). You can try to modify the example by playing with `repartition()` and see how Spark will create the datasets and the partitions.

**The DataFrame is a Dataset<Row>:** You can have datasets of almost any POJO (plain old java object), but only the dataset of rows (`Dataset<Row>`) is called a DataFrame.

A Row uses a very efficient storage called Tungsten. This is not the case with your POJOs.

**The complete code:**

```
package com.evenkat;

import static org.apache.spark.sql.functions.concat;
import static org.apache.spark.sql.functions.lit;
import static org.apache.spark.sql.functions.split;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.Partition;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;

public class DataframeUnionApp {

    private Session spark;

    public static void main(String[] args) {
        DataframeUnionApp app =
```

```

        new DataframeUnionApp();
    app.start();
}
private void start() {
    Logger.getLogger("org").setLevel(Level.ERROR);
    // Creates a session on a local master
    this.spark = SparkSession.builder()
        .appName("Union of two dataframes")
        .master("local")
        .getOrCreate();

    Dataset<Row> wakeRestaurantsDf = buildWakeRestaurantsDataframe();
    Dataset<Row> durhamRestaurantsDf = buildDurhamRestaurantsDataframe();
    combineDataframes(wakeRestaurantsDf, durhamRestaurantsDf);
}

private void combineDataframes(Dataset<Row> df1, Dataset<Row> df2) {
    Dataset<Row> df = df1.unionByName(df2);
    df.show(5);
    df.printSchema();
    System.out.println("We have " + df.count() + " records.");

    Partition[] partitions = df.rdd().partitions();
    int partitionCount = partitions.length;
    System.out.println("Partition count: " + partitionCount);
}

private Dataset<Row> buildWakeRestaurantsDataframe() {
    Dataset<Row> df = this.spark.read().format("csv")
        .option("header", "true")
        .load("in/Restaurants_in_Wake_County_NC.csv");
    df = df.withColumn("county", lit("Wake"))
        .withColumnRenamed("HSISID", "datasetId")
        .withColumnRenamed("NAME", "name")
        .withColumnRenamed("ADDRESS1", "address1")
        .withColumnRenamed("ADDRESS2", "address2")
        .withColumnRenamed("CITY", "city")
        .withColumnRenamed("STATE", "state")
        .withColumnRenamed("POSTALCODE", "zip")
        .withColumnRenamed("PHONENUMBER", "tel")
        .withColumnRenamed("RESTAURANTOPENDATE", "dateStart")
        .withColumn("dateEnd", lit(null))
        .withColumnRenamed("FACILITYTYPE", "type")
        .withColumnRenamed("X", "geoX")
        .withColumnRenamed("Y", "geoY")
        .drop(df.col("OBJECTID"))
        .drop(df.col("GEOCODESTATUS"))
        .drop(df.col("PERMITID"));
}

```

```

df = df.withColumn("id", concat(
    df.col("state"),
    lit("_"),
    df.col("county"), lit("_"),
    df.col("datasetId")));

// if you want to play with repartitioning
// df = df.repartition(4);

return df;
}

private Dataset<Row> buildDurhamRestaurantsDataframe() {
    Dataset<Row> df = this.spark.read().format("json")
        .load("in/Restaurants_in_Durham_County_NC.json");
    df = df.withColumn("county", lit("Durham"))
        .withColumn("datasetId", df.col("fields.id"))
        .withColumn("name", df.col("fields.premise_name"))
        .withColumn("address1", df.col("fields.premise_address1"))
        .withColumn("address2", df.col("fields.premise_address2"))
        .withColumn("city", df.col("fields.premise_city"))
        .withColumn("state", df.col("fields.premise_state"))
        .withColumn("zip", df.col("fields.premise_zip"))
        .withColumn("tel", df.col("fields.premise_phone"))
        .withColumn("dateStart", df.col("fields.opening_date"))
        .withColumn("dateEnd", df.col("fields.closing_date"))
        .withColumn("type",
            split(df.col("fields.type_description"), " - ").getItem(1))
        .withColumn("geoX", df.col("fields.geolocation").getItem(0))
        .withColumn("geoY", df.col("fields.geolocation").getItem(1))
        .drop(df.col("fields"))
        .drop(df.col("geometry"))
        .drop(df.col("record_timestamp"))
        .drop(df.col("recordid"));
    df = df.withColumn("id",
        concat(df.col("state"), lit("_"),
            df.col("county"), lit("_"),
            df.col("datasetId")));

    // if you want to play with repartitioning
    // df = df.repartition(4);

    return df;
}
}

```

=====➔



Let us look at creating a simple dataset of String. This will illustrate the usage of datasets using a simple object we are all familiar with, the string. You will then be able to create datasets of more complex objects.

### Example 21

```
package com.evenkat;

import java.util.Arrays;
import java.util.List;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Encoders;
import org.apache.spark.sql.Session;

public class ArrayToDatasetApp {

    public static void main(String[] args) {
        ArrayToDatasetApp app =
            new ArrayToDatasetApp();
        app.start();
    }

    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        Session spark = Session.builder()
            .appName("Array to Dataset<String>")
            .master("local")
            .getOrCreate();

        String[] stringList =
            new String[] { "Jean", "Liz", "Pierre", "Lauric" };
        List<String> data = Arrays.asList(stringList);
        Dataset<String> ds = spark.createDataset(data, Encoders.STRING());
        ds.show();
        ds.printSchema();
    }
}

//Encoders are helping build the dataset for conversion
```

```

+-----+
| value |
+-----+
|  Jean |
|   Liz |
| Pierre |
| Lauric |
+-----+

```

```

root
|-- value: string (nullable = true)

```

To use the extended methods of the DataFrame vs. the dataset, you can easily convert a dataset into a DataFrame by calling the `toDF()` method.

```

Dataset<Row> df = ds.toDF();
df.show();
df.printSchema();

```

Note: You will have to import the Row object → `import org.apache.spark.sql.Row;`

We will learn how to convert a dataset to a DataFrame and back: it is useful if you want to manipulate your existing POJOs and the extended API that only apply to the DataFrame.

We will read a CSV file containing books, in a DataFrame. You will convert the DataFrame to a dataset of books, and back to a DataFrame. Although it sounds like an obnoxious flow, you could be involved in part of those operations at various stages.

Imagine the following use case: you have an existing `bookProcessor()` method in your arsenal of libraries. This method takes a Book POJO and publishes it, via APIs, on a merchant website like Amazon, FNAC, or Flipkart. You definitely do not want to rewrite this method just to work with Spark, you want to continue sending it a Book POJO: you can load thousands of books, store them in a dataset of books, and when you are going to iterate over them using distributed processing to call your existing `bookProcessor()` method without modification.

```

package com.evenkat;

```

```

import static org.apache.spark.sql.functions.concat;
import static org.apache.spark.sql.functions.expr;
import static org.apache.spark.sql.functions.lit;
import static org.apache.spark.sql.functions.to_date;

```

```

import java.io.Serializable;
import java.text.SimpleDateFormat;

```

```

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Encoders;

```

```

import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;

import com.evenkat1.Book;

public class CsvToDatasetBookToDataframeApp implements Serializable {
    private static final long serialVersionUID = -1L;

    class BookMapper implements MapFunction<Row, Book> {
        private static final long serialVersionUID = -2L;

        public Book call(Row value) throws Exception {
            Book b = new Book();
            b.setId(value.getAs("id"));
            b.setAuthorId(value.getAs("authorId"));
            b.setLink(value.getAs("link"));
            b.setTitle(value.getAs("title"));

            // date case
            String dateAsString = value.getAs("releaseDate");
            if (dateAsString != null) {
                SimpleDateFormat parser = new SimpleDateFormat("M/d/yy");
                b.setReleaseDate(parser.parse(dateAsString));
            }
            return b;
        }
    }

    public static void main(String[] args) {
        CsvToDatasetBookToDataframeApp app =
            new CsvToDatasetBookToDataframeApp();
        app.start();
    }

    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkSession spark = SparkSession.builder()
            .appName("CSV to dataframe to Dataset<Book> and back")
            .master("local")
            .getOrCreate();

        String filename = "in/books.csv";
        Dataset<Row> df = spark.read().format("csv")
            .option("inferSchema", "true")
            .option("header", "true")
            .load(filename);

        System.out.println("*** Books ingested in a dataframe");
    }
}

```

```

df.show(5);
df.printSchema();

Dataset<Book> bookDs = df.map(
    new BookMapper(),
    Encoders.bean(Book.class));
System.out.println("*** Books are now in a dataset of books");
bookDs.show(5, 17);
bookDs.printSchema();

Dataset<Row> df2 = bookDs.toDF();

df2 = df2.withColumn(
    "releaseDateAsString",
    concat(
        expr("releaseDate.year + 1900"), lit("-"),
        expr("releaseDate.month + 1"), lit("-"),
        df2.col("releaseDate.date")));

df2 = df2
    .withColumn(
        "releaseDateAsDate",
        to_date(df2.col("releaseDateAsString"), "yyyy-MM-dd"))
    .drop("releaseDateAsString");

System.out.println("*** Books are back in a dataframe");
df2.show(5, 13);
df2.printSchema();
}
}

package com.evenkat1;

import java.util.Date;

public class Book {
    int id;
    int authorId;
    String title;
    Date releaseDate;
    String link;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}

```

```

    }

    public int getAuthorId() {
        return authorId;
    }

    public void setAuthorId(int authorId) {
        this.authorId = authorId;
    }

    public void setAuthorId(Integer authorId) {
        if (authorId == null) {
            this.authorId = 0;
        } else {
            this.authorId = authorId;
        }
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Date getReleaseDate() {
        return releaseDate;
    }

    public void setReleaseDate(Date releaseDate) {
        this.releaseDate = releaseDate;
    }

    public String getLink() {
        return link;
    }

    public void setLink(String link) {
        this.link = link;
    }
}

```

The output would be as under:-

\*\*\* Books ingested in a dataframe

| id | authorId | title                | releaseDate | link                 |
|----|----------|----------------------|-------------|----------------------|
| 1  | 1        | Fantastic Beasts ... | 11/18/16    | http://amzn.to/2k... |
| 2  | 1        | Harry Potter and ... | 10/6/15     | http://amzn.to/2l... |
| 3  | 1        | The Tales of Beed... | 12/4/08     | http://amzn.to/2k... |
| 4  | 1        | Harry Potter and ... | 10/4/16     | http://amzn.to/2k... |
| 5  | 2        | Informix 12.10 on... | 4/23/17     | http://amzn.to/2i... |

only showing top 5 rows

root

A

-- id: integer (nullable = true)  
-- authorId: integer (nullable = true)  
-- title: string (nullable = true)  
-- releaseDate: string (nullable = true)  
-- link: string (nullable = true)

B

\*\*\* Books are now in a dataset of books

| authorId | id | link              | releaseDate         | title             |
|----------|----|-------------------|---------------------|-------------------|
| 1        | 1  | http://amzn.to... | [18, 0, 0, 10, ...] | Fantastic Beas... |
| 1        | 2  | http://amzn.to... | [6, 0, 0, 9, 0...]  | Harry Potter a... |
| 1        | 3  | http://amzn.to... | [4, 0, 0, 11, ...]  | The Tales of B... |
| 1        | 4  | http://amzn.to... | [4, 0, 0, 9, 0...]  | Harry Potter a... |
| 2        | 5  | http://amzn.to... | [23, 0, 0, 3, ...]  | Informix 12.10... |

only showing top 5 rows

root

C

-- authorId: integer (nullable = true)  
-- id: integer (nullable = true)  
-- link: string (nullable = true)  
-- releaseDate: struct (nullable = true)  
| -- date: integer (nullable = true)  
| -- hours: integer (nullable = true)  
| -- minutes: integer (nullable = true)  
| -- month: integer (nullable = true)  
| -- seconds: integer (nullable = true)  
| -- time: long (nullable = true)  
| -- year: integer (nullable = true)  
-- title: string (nullable = true)

D

E

D

D

D

D

D

D

```

*** Books are back in a dataframe
+-----+-----+-----+-----+-----+-----+
|authorId| id|      link|  releaseDate|      title|releaseDateAsDate|
+-----+-----+-----+-----+-----+-----+
|      1|  1|http://amz...|[18, 0, 0,...|Fantastic ...|      2016-11-18|
|      1|  2|http://amz...|[6, 0, 0, ...|Harry Pott...|      2015-10-06|
|      1|  3|http://amz...|[4, 0, 0, ...|The Tales ...|      2008-12-04|
|      1|  4|http://amz...|[4, 0, 0, ...|Harry Pott...|      2016-10-04|
|      2|  5|http://amz...|[23, 0, 0,...|Informix 1...|      2017-04-23|
+-----+-----+-----+-----+-----+-----+
only showing top 5 rows

```

root

```

|-- authorId: integer (nullable = true)
|-- id: integer (nullable = true)
|-- link: string (nullable = true)
|-- releaseDate: struct (nullable = true)
|   |-- date: integer (nullable = true)
|   |-- hours: integer (nullable = true)
|   |-- minutes: integer (nullable = true)
|   |-- month: integer (nullable = true)
|   |-- seconds: integer (nullable = true)
|   |-- time: long (nullable = true)
|   |-- year: integer (nullable = true)
|-- title: string (nullable = true)
|-- releaseDateAsDate: date (nullable = true)

```

A The order of fields is the order in the file

B Seen as a string when parsed

C The fields are now alphabetically sorted (same for nested fields), this is Spark behavior

D "Exploded" as the components of a date when converted to a Dataset<Books>

E Day of the date

The fields are sorted after you converted your DataFrame to a dataset, this is not something you asked the application to do, and it is a bonus. However, remember to use `unionByName()` (instead of `union()`) if you plan on combining datasets after that as fields may have shifted.

```

Dataset<Book> bookDs = df.map(
    new BookMapper(),
    Encoders.bean(Book.class));

```

When called, the `map()` method will:

- Go through every record of the DataFrame.
- Call an instance of a class implementing `MapFunction<Row, Book>`, in your case `BookMapper`. Note that it is instantiated only once, whatever the number of records you will have to process.
- Return a `Dataset<Book>` (your goal).

When you implement yours, make sure you have the right signature and implementation, as this could be tricky. The skeleton, including signature and required method, is:

```
class AnyMapper implements MapFunction<T, U> {  
    @Override  
    public U call(T value) throws Exception {  
        ...  
    }  
}
```

This is what we are doing in our BookMapper mapper class we build.

```
class BookMapper implements MapFunction<Row, Book> {  
    private static final long serialVersionUID = -2L;  
  
    public Book call(Row value) throws Exception {  
        Book b = new Book();  
        b.setId(value.getAs("id"));  
        b.setAuthorId(value.getAs("authorId"));  
        b.setLink(value.getAs("link"));  
        b.setTitle(value.getAs("title"));  
  
        // date case  
        String dateAsString = value.getAs("releaseDate");  
        if (dateAsString != null) {  
            SimpleDateFormat parser = new SimpleDateFormat("M/d/yy");  
            b.setReleaseDate(parser.parse(dateAsString));  
        }  
        return b;  
    }  
}
```

We will also need a simple POJO representing a book (the Book POJO) which we had shown earlier and we created this in a new package.

So, let's convert the dataset back to a DataFrame to study this part of the mechanism. You will study an interesting case with the date, as the date is split in a nested structure. We are now ready to convert the dataset to a DataFrame, and perform a few transformations, like changing the date from this abominable structure to a date column in your DataFrame.

```
Dataset<Row> df2 = bookDs.toDF();
```

However, we still have this strange date format, so let's correct that. The first step is to transform the date to a string with a representation of date, in this situation, we will use the ANSI/ISO format: YYYY-MM-DD, like 1971-10-05.

Remember that years in Java start in 1900, so 1971 is 71, while 2004 is 104. Similarly, months start at 0, making October, the tenth month of the year, month #9. Using the Java methods to build the date



would require using a mapping function. This is the way to build a dataset or a DataFrame through iteration over the data. You could also use UDF (user defined functions).

```
df2 = df2.withColumn(                                     A
    "releaseDateAsString",                               B
    concat(                                               C
        expr("releaseDate.year + 1900"), lit("-"),      D
        expr("releaseDate.month + 1"), lit("-"),        E
        df2.col("releaseDate.date")));                  F
```

- A Create a column
- B Called releaseDateAsString
- C Which value is the concatenation of
- D The expression is the sum of the release year and 1900
- E The expression is the month plus one
- F And the day of the month

The `expr()` static function will compute an expression and return a column. It can use field names. The expression `releaseDate.year + 1900` will be evaluated and turned into a column containing the value. Once you have a date as a string, you can convert it to a date as a date, using the `to_date()` static function

Despite major efforts around the DataFrame, RDDs are not disappearing. And nobody wants them to disappear: they remain the low-level storage tier used by Spark.

<https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html>

### Ingesting a multiline JSON file

The data file to be used in this process is: `countrytravelinfo.json`

#### Example 21

```
package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;

public class MultilineJsonToDataframeApp {

    public static void main(String[] args) {
        MultilineJsonToDataframeApp app =
            new MultilineJsonToDataframeApp();
        app.start();
    }
}
```

```

private void start() {
    Logger.getLogger("org").setLevel(Level.ERROR);
    SparkSession spark = SparkSession.builder()
        .appName("Multiline JSON to Dataframe")
        .master("local")
        .getOrCreate();

    Dataset<Row> df = spark.read()
        .format("json")
        .option("multiline", true) // Most Imp function
        .load("in/countrytravelinfo.json");

    df.show(3);
    df.printSchema();
}
}

```

If you forget the multiline option, your DataFrame will be composed of the single column, called `_corrupt_record`

```

+-----+-----+-----+-----+-----+
|destination_description|entry_exit_requirements|geopoliticalarea|health|iso_code|
+-----+-----+-----+-----+-----+
|<div style="margi...|<div style="margi...|Afghanistan|<div style="margi...|AF|
|<div style="margi...|<div style="margi...|Albania|<div style="margi...|AL|
|<div style="margi...|<div style="margi...|Algeria|<div style="margi...|DZ|
+-----+-----+-----+-----+-----+

+-----+-----+-----+-----+-----+
|last_update_date|local_laws_and_special_circumstances|safety_and_security|tag|
+-----+-----+-----+-----+-----+
|Last Updated: May...|<div style="margi...|<div style="margi...|AF|
|Last Updated: Jul...|<div style="margi...|<div style="margi...|AL|
|Last Updated: Jan...|<div style="margi...|<div style="margi...|DZ|
+-----+-----+-----+-----+-----+

+-----+-----+
|travel_embassyAndConsulate|travel_transportation|
+-----+-----+
|<div style="margi...|<div style="margi...|
|<div style="margi...|<div style="margi...|
|<div style="margi...|<div style="margi...|
+-----+-----+

```

Ingestion via XML file

**DataFile: NASA\_Patents.xml**

Look at this following link:

<https://data.nasa.gov/>  
<https://data.gov.in/>  
<https://www.data.gov/>  
<https://data.ca.gov/>

```

package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;

public class XmlToDataframeApp {
    public static void main(String[] args) {
        XmlToDataframeApp app = new XmlToDataframeApp();
        app.start();
    }

    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkSession spark = SparkSession.builder()
            .appName("XML to Dataframe")
            .master("local")
            .getOrCreate();

        Dataset<Row> df = spark.read().format("xml") //
            .option("rowTag", "row")
            // Element or tag that indicates a record in the XML file
            .load("in/NASA_Patents.xml");

        df.show(5);
        df.printSchema();
    }
}

```

Add the following to pom.xml file to include the spark dependency for xml.

In the Properties tag.

```

<scala.version>2.11</scala.version>
<spark-xml.version>0.4.1</spark-xml.version>

```

In the Dependency tag.

```

<dependency>
<groupId>com.databricks</groupId>
<artifactId>spark-xml_${scala.version}</artifactId>
<version>${spark-xml.version}</version>
<exclusions>
<exclusion>
<groupId>org.slf4j</groupId>

```

```
<artifactId>slf4j-simple</artifactId>
</exclusion>
</exclusions>
</dependency>
```

### Ingesting a text file

**DataFile:** Romeo\_Juliet.txt

```
package com.evenkat;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;

public class TextToDataframeApp {
    public static void main(String[] args) {
        TextToDataframeApp app = new TextToDataframeApp();
        app.start();
    }

    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        Session spark = Session.builder()
            .appName("Text to Dataframe")
            .master("local")
            .getOrCreate();

        Dataset<Row> df = spark.read().format("text") //
            .load("in/Romeo_Juliet.txt");

        df.show(10);
        df.printSchema();
    }
}
```

Other File Format

<https://www.datanami.com/2018/05/16/big-data-file-formats-demystified/>

<https://community.hitachivantara.com/s/article/hadoop-file-formats-its-not-just-csv-anymore>

## Testing with Avro

### DataFile: weather.avro

In the Properties tag.

```
<spark-avro.version>4.0.0</spark-avro.version>
```

In the Dependency tag.

```
<dependency>
  <groupId>com.databricks</groupId>
  <artifactId>spark-avro_${scala.version}</artifactId>
  <version>${spark-avro.version}</version>
</dependency>
```

```
package com.evenkat;
```

```
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;
```

```
public class AvroToDataframeApp {

    public static void main(String[] args) {
        AvroToDataframeApp app = new AvroToDataframeApp();
        app.start();
    }

    private void start() {
        Logger.getLogger("org").setLevel(Level.ERROR);
        SparkSession spark = SparkSession.builder()
            .appName("Avro to Dataframe")
            .master("local")
            .getOrCreate();

        Dataset<Row> df = spark.read()
            .format("com.databricks.spark.avro")
            .load("in/weather.avro");

        df.show(10);
        df.printSchema();
        System.out.println("The dataframe has " + df.count()
            + " rows.");
    }
}
```

```
}
```

```
+-----+-----+-----+
| station|      time|temp|
+-----+-----+-----+
|011990-99999|-619524000000|  0|
|011990-99999|-619506000000| 22|
|011990-99999|-619484400000| -11|
|012650-99999|-655531200000| 111|
|012650-99999|-655509600000|  78|
+-----+-----+-----+
```

```
root|
|-- station: string (nullable = true)
|-- time: long (nullable = true)
|-- temp: integer (nullable = true)
```

The dataframe has 5 rows.

## Testing with ORC

### Data File: ORCExample.orc

The implementation parameter can have either the native value or hive value. The native implementation means that it uses the implementation coming with Spark. It is the default value starting with Spark v2.4.

```
package com.evenkat;
```

```
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;
```

```
public class OrcToDataframeApp {
    public static void main(String[] args) {
        OrcToDataframeApp app = new OrcToDataframeApp();
        app.start();
    }
}
```

```
private void start() {
    Logger.getLogger("org").setLevel(Level.ERROR);
    SparkSession spark = SparkSession.builder()
        .appName("ORC to Dataframe")
        .config("spark.sql.orc.impl", "native")
        .master("local")
        .getOrCreate();
}
```

```
Dataset<Row> df = spark.read()
```

```

        .format("orc")
        .load("in/ORCExample.orc");

    df.show(10);
    df.printSchema();
    System.out.println("The dataframe has " + df.count() + " rows.");
}
}

```

| _col0 | _col1 | _col2 | _col3   | _col4 | _col5 | _col6 | _col7 | _col8 |
|-------|-------|-------|---------|-------|-------|-------|-------|-------|
| 1     | M     | M     | Primary | 500   | Good  | 0     | 0     | 0     |
| 2     | F     | M     | Primary | 500   | Good  | 0     | 0     | 0     |
| 3     | M     | S     | Primary | 500   | Good  | 0     | 0     | 0     |
| 4     | F     | S     | Primary | 500   | Good  | 0     | 0     | 0     |
| 5     | M     | D     | Primary | 500   | Good  | 0     | 0     | 0     |
| 6     | F     | D     | Primary | 500   | Good  | 0     | 0     | 0     |
| 7     | M     | W     | Primary | 500   | Good  | 0     | 0     | 0     |
| 8     | F     | W     | Primary | 500   | Good  | 0     | 0     | 0     |
| 9     | M     | U     | Primary | 500   | Good  | 0     | 0     | 0     |
| 10    | F     | U     | Primary | 500   | Good  | 0     | 0     | 0     |

only showing top 10 rows

root

```

|-- _col0: integer (nullable = true)
|-- _col1: string (nullable = true)
|-- _col2: string (nullable = true)
|-- _col3: string (nullable = true)
|-- _col4: integer (nullable = true)
|-- _col5: string (nullable = true)
|-- _col6: integer (nullable = true)
|-- _col7: integer (nullable = true)
|-- _col8: integer (nullable = true)

```

The dataframe has 1920800 rows.

## Testing with Parquet

**Data File: Sample.parquet**

```
package com.evenkat;
```

```

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Session;

```

```

public class ParquetToDataframeApp {
    public static void main(String[] args) {

```

```

ParquetToDataframeApp app = new ParquetToDataframeApp();
app.start();
}

private void start() {
    Logger.getLogger("org").setLevel(Level.ERROR);
    SparkSession spark = SparkSession.builder()
        .appName("Parquet to Dataframe")
        .master("local")
        .getOrCreate();

    Dataset<Row> df = spark.read()
        .format("parquet")
        .load("in/Sample.parquet");

    df.show(10);
    df.printSchema();
    System.out.println("The dataframe has " + df.count() + " rows.");
}
}

```

| id | bool_col | tinyint_col | smallint_col | int_col | bigint_col | float_col | double_col |
|----|----------|-------------|--------------|---------|------------|-----------|------------|
| 4  | true     | 0           | 0            | 0       | 0          | 0.0       | 0.0        |
| 5  | false    | 1           | 1            | 1       | 10         | 1.1       | 10.1       |
| 6  | true     | 0           | 0            | 0       | 0          | 0.0       | 0.0        |
| 7  | false    | 1           | 1            | 1       | 10         | 1.1       | 10.1       |
| 2  | true     | 0           | 0            | 0       | 0          | 0.0       | 0.0        |
| 3  | false    | 1           | 1            | 1       | 10         | 1.1       | 10.1       |
| 0  | true     | 0           | 0            | 0       | 0          | 0.0       | 0.0        |
| 1  | false    | 1           | 1            | 1       | 10         | 1.1       | 10.1       |

| date_string_col       | string_col | timestamp_col       |
|-----------------------|------------|---------------------|
| [30 33 2F 30 31 2...] | [30]       | 2009-03-01 05:30:00 |
| [30 33 2F 30 31 2...] | [31]       | 2009-03-01 05:31:00 |
| [30 34 2F 30 31 2...] | [30]       | 2009-04-01 05:30:00 |
| [30 34 2F 30 31 2...] | [31]       | 2009-04-01 05:31:00 |
| [30 32 2F 30 31 2...] | [30]       | 2009-02-01 05:30:00 |
| [30 32 2F 30 31 2...] | [31]       | 2009-02-01 05:31:00 |
| [30 31 2F 30 31 2...] | [30]       | 2009-01-01 05:30:00 |
| [30 31 2F 30 31 2...] | [31]       | 2009-01-01 05:31:00 |

```

root
|-- id: integer (nullable = true)
|-- bool_col: boolean (nullable = true)
|-- tinyint_col: integer (nullable = true)
|-- smallint_col: integer (nullable = true)
|-- int_col: integer (nullable = true)
|-- bigint_col: long (nullable = true)
|-- float_col: float (nullable = true)

```



```
|-- double_col: double (nullable = true)
|-- date_string_col: binary (nullable = true)
|-- string_col: binary (nullable = true)
|-- timestamp_col: timestamp (nullable = true)
```

The dataframe has 8 rows.