# Exercise 7

In this exercise, you will learn how to work with RDDs.

## Examples from class

In the exercises today we would like you to attempt the Example 1 & 2 from today's lecture (see the slides!)

As a matter of fact, we've already prepared for you to perform these. In the file 'WordCountRDD.scala' you can work with Example 1 and in the files 'BikeShareData.scala'/'BikeShareAnalysis.scala' we'd like you to work with Example 2.

All these examples are intended to be implemented and tested locally (because of the nature of the sbt project structure, it is not possible to execute these on the AWS cluster).

Therefore the implemented solutions to the examples should be evaluated using the SBT console ('sbt console').

## For Windows users (using CMD/PowerShell/IntelliJ)

**NOTE: if you use BASH/CygWin or you are a MAC/Linux user: skip this section!**

Since you'll be working with file input/output in a local Spark instance (through SBT) you may run into issues where it'll tell you that 'HADOOP_HOME' is not set. This often leads to the SBT console being unable to properly perform read and write of files on your local system when running Spark.

Most of you may run into this issue and therefore it may be necessary to install the Hadoop binaries in order to avoid this issue.

In order to set these up, perform the following steps:

1. Download the binaries from https://github.com/steveloughran/winutils/releases/download/tag_2017-08-29-hadoop-2.8.1-native/hadoop-2.8.1.zip and unzip into a folder on your drive. Preferably name it something like 'hadoop'. These files are sourced from https://github.com/steveloughran/winutils and the repository owner is one of the developers behind the Hadoop project.
2. Open the folder and create a new folder here called 'bin'. Then move all of the unzipped files into the 'bin' folder.
3. Open up a command prompt (cmd) / PowerShell terminal.
4. Type in the following command 'setx HADOOP_HOME [path-to-folder]'. So if for example you put the unzipped data into the 'hadoop' folder as in step 1 (where this folder then contains the 'bin' folder) and this folder is placed on your C drive, you'll type in 'setx HADOOP_HOME C:\hadoop'.

What this does is to declare a variable in your system that binds 'HADOOP_HOME' to the directory that you defined in the 'setx' command.

If you ever wish to remove this binding, you can do so by typing in 'setx HADOOP_HOME ""'. However, you will likely have use for both the files downloaded and this variable throughout the remainder of the course.

## Loading Data to RDD

A file can be read directly to an RDD by using the SparkContext class. The following Scala code creates a SparkContext object and uses the object to load the file called "medium_dataset.csv" from the path "data":

```
val sc = new SparkContext()
val mediumDatasetRDD:RDD[String] = sc.textfile("data/medium_dataset.csv")
```

You can see the "textfile" method in the Scala Spark documentation: https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkContext Notice that the return type is "RDD[String]".

You can also load data from files into DataFrames by using the SparkSession class. Create a SparkSession object "spark" with the app name "MyApp" running locally:

```
val spark: SparkSession = SparkSession.builder
        .appName("MyApp")
        .master("local") //Remove this line when running in cluster
        .getOrCreate
```

Use the SparkSession object to read from the file:

```
val mediumDatasetDF = spark.read.format("csv").load("data/medium_dataset.csv")
```

Calling "read" on the SparkSession returns a DataFrameReader on which the functions "format" and "load" are called. See this in the documentation for SparkSession: https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.SparkSession

The example loads a CSV-file, but other file formats can be loaded as well by changing the argument to "format". Some common formats has their own function in DataFrameReader, such as CSV:

```
val mediumDatasetDF = spark.read.csv("data/medium_dataset.csv")
```

A DataFrame can be converted to an RDD by calling the rdd method on the DataFrame:

```
val mediumDatasetRDD = mediumDatasetDF.rdd
```

When reading a file, different options can be set by calling "option" on the DataFrameReader. Examples of this can be seen in https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html and in the documentation https://spark.apache.org/docs/latest/api/scala/#org.apache.spark.sql.DataFrameReader

## Writing Data from RDD to Disk

An RDD can be saved by calling saveAsTextFile:

```
mediumDatasetRDD.saveAsTextFile("outputpath/mediumDatasetRDD")
```

## RDD Functions

Look at the functions listed in https://spark.apache.org/docs/latest/api/scala/#org.apache.spark.rdd.RDD.

Take a look at the following functions:

- map
- filter
- groupBy
- reduce
- count

Additionally, the following can be called on DataFrames/Datasets:

- schema (returns the schema of the Dataset)
- show (prints the top 20 rows of the Dataset)

## RDD Persistence Levels

Read about the persistence level of RDDs: https://data-flair.training/blogs/apache-spark-rdd-persistence-caching/

To change the storage level you have to import the Storage level package:

´´´scala import org.apache.spark.storage.StorageLevel._ ´´´

To persist an RDD value called "rddval", do the following:

´´´scala rddval.persist() // Memory only rddval.persist(MEMORY_AND_DISK) // Look at the link above to find out what this persistence level means ´´´

## Application History in AWS

When you have a cluster on EMR there are multiple monitoring tools directly in the AWS Console in your web browser that can help you understand the execution of your applications.

You find the application history tab by going to your AWS Console:

1. From the front page, click on "EMR".
2. Click the "Application History" tab.

The tab includes execution times and lots of other information about your application executions. IT EVEN SAVES THE EXECUTIONS AFTER YOU TERMINATE YOUR CLUSTER.

## Problems

The following problems can be solved on your local machine.

1. Load medium_dataset.csv into an RDD and print the top 20 rows of the dataset.
2. Count the number of rows of the dataset.
3. Count the number of times the character 'X' is in the column "first" and the column "second" respectively.
4. Generate an RDD containing the "id", "first" and "second" values of the rows with the most frequent ID. This means that if 132 is the "id" that occurs most frequently, then all rows with "id" 132 are returned.
5. Try to add an RDD persistency level where you intend to reuse that RDD in the code written for the above problems.

## Further reading and guides if interested

Links Spark Dataset, DataFrames and RDD Programming guide:

https://spark.apache.org/docs/latest/sql-programming-guide.html

Spark Scala documentation:

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package
https://spark.apache.org/docs/latest/api/scala/#org.apache.spark.rdd.RDD

AWS Spark Guide:

https://aws.amazon.com/blogs/big-data/submitting-user-applications-with-spark-submit/