Lab 05: Processing Big Data with Spark

Department of Computer Science and IT, La Trobe University

Submissions close 1pm Monday 11th of May

Apache Spark is a relatively new system for processing Big Data which is rapidly growing in popularity. A large part of Spark's appeal is that programmers are able to create jobs involving multiple MapReduce stages in only a few lines of clearly readable code. Furthermore, Spark boasts better execution times than existing MapReduce implementations such as Hadoop, giving developers a big incentive to adopt this new technology. In this lab we will be using the Scala programming language to perform various tasks in Spark.

At various points in this lab you will be directed to write your own code. You should keep track of your answers where indicated in the provided text file `lab05.scala` so that you can be marked for the lab.

Spark has many API calls and many of them are very powerful. It is highly recommended that you get familiar with the web site that lists all the Spark API calls. You can look at the API calls here`https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package`. In particular look at the left hand column of the web page and see the two entries, RDD and PairRDDFunctions. These contain by far the most useful Spark functions.

The API on the spark web site does not have many examples at all, so although you can see the full list of API calls, you may not know how to use them. Fortunately, Matthias and Zhen have gone over the entire Spark RDD API and have written examples for almost all of the API calls. You can see the examples here: Spark API Examples

Just like in the previous lab there are a lot of code examples already given to you. Please type them all into the Spark shell and see what happens. This is the best way to learn. Don't worry there are still plenty of exercises left that will require you to apply your knowledge. In fact there are 6 exercises for you to do by yourself. Once again, there's a helper video available in ECHO360, so watch that before attempting the lab.

Task A: Creating RDDs and performing operations on them

You like the sound of a faster-and-easier-MapReduce-thingermejigger or "Spark" as it's normally referred to. You also know a bit about coding in Scala so you're ready to jump right in!

1. In Spark, data is stored in RDDs (Resilient Distributed Datasets). You can think of RDDs as immutable Scala collections whose data is spread across multiple cluster nodes. Each RDD is divided into multiple partitions and each partition is processed by one worker thread. There are two main ways for creating RDDs. The first way is to use the key word `parallelize` to create an RDD from a Scala sequence collection. The second way is to load data from an input file. We will focus on the first way in this task and we will show you the second way in Task C.

2. Open a terminal, then start a Spark shell.

```
$ spark-shell
```

3. During the initialization process a Spark context was created for us and made available in the variable `sc`. We can use `sc` to create a new Spark RDD from data which currently only exists outside of Spark. Enter the following line of code to create an RDD from a Scala list.

```
scala> val numbers = sc.parallelize(List(1, 5, 10, 15, 20, 25, 30, 35))
```

The above command creates a new RDD called `numbers` which contains the data in the list, spreading the contents across Spark worker nodes. The variable `sc` is a `SparkContext` object. A `SparkContext` object stores all the configuration information necessary to launch the Spark job on the cluster.

4. Now lets take a look at what is inside the `numbers` RDD. We can do this using the `collect` command which gathers all the RDD contents from the worker into an `Array` at the master node.

```
scala> val returnedArray = numbers.collect
```

If you just call `numbers.collect` without storing it into a variable it will just display the output to the screen only.

5. Let's create a large RDD with 10,000 numbers, then try to multiply each number by 5 and keep only numbers greater 5000.

```
scala> val manyNumbers = sc.parallelize(1 to 10000)
scala> val largeNumbers = manyNumbers.map(_ * 5).filter(_ > 5000)
```

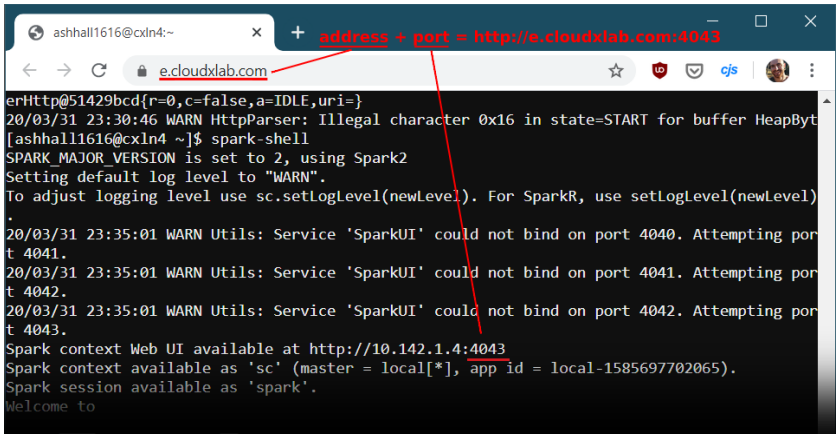
You will notice there is no output! The reason is that Spark performs lazy evaluation. This means Spark will not output something until an action command is issued. Some action commands include `collect`, `count`, `take`, `reduce`, `lookup`, and `saveAsTextFile`. Let's use `take` to get the first 10 numbers in the `largeNumbers` RDD.

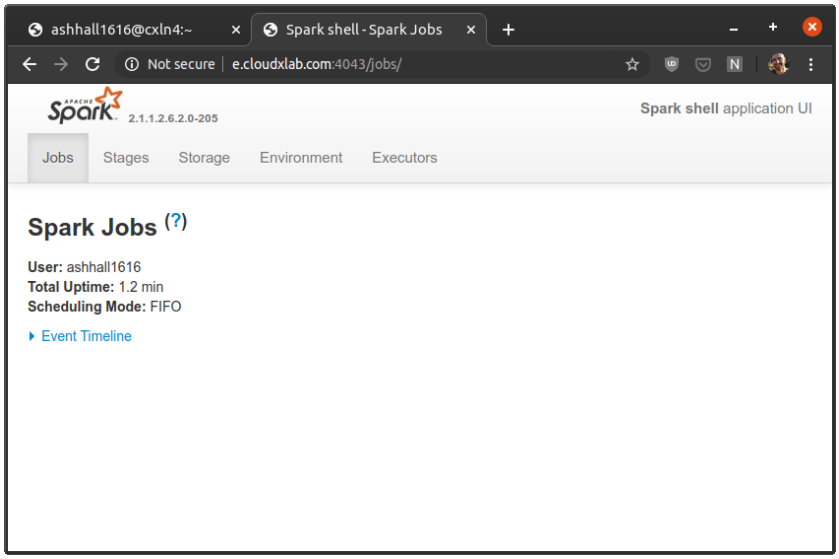
```
scala> largeNumbers.take(10)
```

Active Spark sessions

If you have a `spark-shell` or compiled Spark application which has not yet exited, you can find Spark metrics for it by accessing the Spark web interface.

1. When you started the Spark shell in the previous task, you will have been provided an address like in the screenshot below. This IP address is internal to the CloudxLab network, so replace it with the domain name found in the address bar. In this example the constructed path will be `http://e.cloudxlab.com:4043` - your port number may be different. Open this in a new tab to load the Spark web interface.
   Important: you must explicitly type `http://`, or it may default to `https://` and not work.





2. Job information and statistics (such as those we have explored previously using Hue) are available for Spark in this page. Under "Completed Stages" you should see items corresponding to the previous steps you've executed. Notice that compared to Hue the information provided is a bit sparse—with each new version of Spark more metrics are being added, so in time this page will grow considerably.

Exercise 1. Now you should know enough to do something on your own. Write code that first finds the square root of each number in an RDD and then sums all the square roots together. Hint the following math library function `math.sqrt` will be useful. Note you just need to write 1 line of code. Please test your program using the following template. Check to make sure the result is same as the that displayed below.

```
scala> val someNumbers = sc.parallelize(1 to 1000)
scala> val result = ...TODO...
result: Double = 21097.455887480734
```

Task B: Playing around with PairRDDFunctions

The creators of Spark have created a sizable list of functions specifically devoted to working with key-value pairs. Let's play around with them.

1. In Spark key-value pairs are just modeled using two-element tuples. Let's create an RDD of animals and their corresponding age in years. Below is code that returns the total age of animals of each type.

```
scala> val animalAges = sc.parallelize(List(("cat", 7), ("dog", 5),
  ("monkey", 3), ("cat", 6), ("dog", 10), ("bird", 1), ("bird",1)))
scala> val result = animalAges.reduceByKey(_ + _)
scala> result.collect
```

2. The `groupByKey` function does what you might expect—it groups together values with the same key. Try it out and see what happens.

```
scala> val result = animalAges.groupByKey()
scala> result.collect
```

3. Another useful `PairRDDFunction` is the `join` function. Let's see how that works. Suppose we have another RDD which stores the number of legs each type of animal has. We can join the two RDDs together to get the number of legs and age for each animial. Let's try it and see what happens.

```
scala> val animalLegs = sc.parallelize(List(("monkey", 2), ("dog", 4),
  ("cat", 4), ("bird", 2)))
scala> val result = animalAges.join(animalLegs)
scala> result.collect
```

4. Now let's do something a little bit fancy. Suppose we want to sort all the animals by `age` in ascending order (ie. sort `animalAges` by *value*). How can we do that? We use the `sortBy` function to sort the animals by their age. The first argument of the `sortBy` function specifies

which attribute you want to sort by. In this case it is the second attribute. The second argument in the `sortBy` function specifies whether the data should be sorted in ascending or descending order (`true` means in ascending order).

```
scala> val sortedAnimals = animalAges.sortBy(_._2, true)
scala> sortedAnimals.collect
```

Exercise 2. You are given an RDD of 3-element tuples (<name>, <occupation>, <salary>). Your job is to sum up the total `salary` for each `occupation` and then report the output in ascending order according to `occupation`. You need to do all of this using just one line of code, by chaining together three Spark functions: `map`, `reduceByKey`, and `sortBy`. The purpose of the `map` is to create pairs from the three-element tuples, so think about which field should be the key, which field should be the value, and which field will not be used.

Use the template below to test your code:

```
scala> val people = sc.parallelize(Array(("Jane", "student", 1000),
  ("Peter", "doctor", 100000), ("Mary", "doctor", 200000),
  ("Michael", "student", 1000)))
scala> val result = ...TODO...
scala> result.collect
res19: Array[(String, Int)] = Array((doctor,300000), (student,2000))
```

Task C: Loading and saving files

The ability to load and save files is essential knowledge when working with big data. In this task we will learn to load and save text files and binary files (object files) from HDFS.

1. The dataset we will be working with is a collection of census information collected in 1994, and has already been copied to HDFS for you. `census_description.txt` describes the values stored in each record's fields and `census.txt` contains the data itself. Take a look at the first few entries of the dataset to familiarize yourself with the input format using the following command (ignore the warning about writing to the output stream):

   ```
   $ hdfs dfs -cat /user/ashhall1616/bdc_data/lab_5/census.txt | head
   ```

2. Once we have loaded the data and split the data, each line of input is stored as an `Array`. To make it a bit easier we have included the following table with each field name, its associated array index and its possible values.

Index	Column name	Possible values
0	age	continuous
1	workclass	Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, etc.
2	fnlwgt	continuous
3	education	Bachelors, Some-college, 11th, HS-grad, Prof-school, etc.
4	education-num	continuous
5	marital-status	Married-civ-spouse, Divorced, Never-married, Separated, etc.
6	occupation	Tech-support, Craft-repair, Other-service, Sales, etc.
7	relationship	Wife, Own-child, Husband, Not-in-family, Other-relative, etc.
8	race	White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black
9	sex	Female, Male
10	capital-gain	continuous
11	capital-loss	continuous
12	hours-per-week	continuous
13	native-country	United-States, Cambodia, England, Puerto-Rico, Canada, etc.
14	income	>50K, <=50K

3. We will begin by loading our data set into Spark. Using the method `textFile` we can read the census data into an RDD.

```
scala> val censusLines = sc.textFile("/user/ashhall1616/bdc_data/lab_5/census.txt")
```

The above line of code loads the contents of `census.txt` into an RDD called `censusLines`.

4. Print the first element of the censusLines RDD.

```
scala> censusLines.first()
```

You should see a string representing a line from the `census.txt` file.

5. Currently our RDD is a big collection of strings, one for each line. To make the data easier to work with we can split each line into fields using the `map` function. Notice how Spark makes a complex parallel operation look like a standard Scala operation you might perform on a local `Array`.

```
scala> val censusSplit = censusLines.map(_.split(", "))
```

After entering that line of code you may be thinking "wow, Spark is really quick!". Well, not quite. Remember that Spark uses lazy evaluation, which means that it hasn't actually split any lines yet—it will do that only when it needs to. We can force Spark to actually do some splitting by asking for a result as follows.

```
scala> censusSplit.first()
```

You should be presented with an `Array` containing the fields of the first record in the census data. It is important to note each line is converted into an `Array`. Since this means we need to get the data out by using the `Array` notation.

6. Now let's extract the `native-country` (array index 13) and `occupation` (array index 6) columns and stored them into a separate text file called `countryOccupation.txt`. .

```
scala> val countryOccupation = censusSplit.map(r => (r(13), r(6)))
scala> countryOccupation.saveAsTextFile("lab_5/countryOccupation")
```

After entering the above command a directory called `countryOccupation` will be created. Inside the directory you will find the output text files.

7. The `textFile` is usually not a very efficient way for storing large amounts of data. Let's stored the data in a binary file instead. Use the following command to stored the data into a binary file.

```
scala> countryOccupation.saveAsObjectFile("lab_5/countryOccupationBin")
```

8. Next let's load the stored object file.

```
scala> val loadedCountryOccupation =
  sc.objectFile[(String,String)]("lab_5/countryOccupationBin")
```

Notice that we must provide a type parameter, `[(String, String)]`, which corresponds to the type of each record. This is required for Spark to correctly read the data.

9. Now we'll write this RDD to the local filesystem so we can inspect it in Jupyter. In order to do so, you'll need to replace **USERNAME** below with your **CloudxLab** username.

```
scala> loadedCountryOccupation.saveAsTextFile(
        "file:///home/USERNAME/lab_5_countryOccupationText"
)
```

When this operation has finished, navigate to the root (top-level) directory in Jupyter. You'll find a folder there named `lab_5_countryOccupationText`. Open one of the text files in this directory to see the contents of the RDD we just wrote. This may take a moment to open as the files are quite large.

Task D: Performing simple analytics

Now that you have a taste for what Spark can do you wish to do something more real. You know, a program that actually does something useful.

Exercise 3. Starting with the `censusSplit` RDD from the previous task, use the `map` method to create a new pair RDD where the key is `native-country` (keep as a string) and the value is `age` (convert to an integer using the `toInt` method). Some of the data is missing, which is indicated by `native-country` having a value of `"?"`. Use the `filter` function to remove all records with missing data, and call the resulting RDD `countryAge`. Use `countryAge.count()` to confirm that there are 31978 rows in the RDD.

Exercise 4. Starting with the `countryAge` RDD from the previous exercise, find the age of the oldest person from each country. So the resulting RDD should contain one `(country, age)` pair per country. In order to solve the task efficiently, use the following two steps:

1. Remove all duplicate records by using the `distinct` function. This will reduce the amount of data going into the next step without affecting the results.
2. Find the oldest person from each country using the `reduceByKey` function. Hint: you might find the Scala function `math.max` useful.

Your resulting RDD should look something like the following (note: the order maybe different).

```
(Japan,61),
(Outlying-US(Guam-USVI-etc),63),
(Taiwan,61),
(Portugal,78),
(Guatemala,66),
...
```

Exercise 5. Continue to modify this program so that it now outputs the top 7 countries in terms of having the oldest person. Hint: the functions `sortBy` and `take` will be very useful here. The output should again be the `country` followed by the `age` of the oldest person. Your output should be the following. The countries that have 90 year old people can appear in arbitrary order among themselves.

```
(England,90)
(Puerto-Rico,90)
(Philippines,90)
(United-States,90)
(Ecuador,90)
(South,90)
(Poland,85)
```

Task E: Using Spark for decision support

Exercise 6. You are the head of a new North American union which spans both USA and Canada. You decide to appoint a team of two representitives from amongst your members. In order to maintain fairness you must select one Canadian and one American. Furthermore, you would like for your representatives to be in the same occupation. As a step towards making this selection

process easier you would like to produce a list of possible candidate representative pairs using Spark.

1. To begin with we want to select a smaller set of data columns from the `censusSplit` RDD. Let's keep only the following columns: `native-country`, `education`, `occupation`, and `sex`. To see if the program worked print out the first 5 rows. Note the exact rows reported by your program maybe different since we have not explicitly ordered the data, but the format should be the same.

```
scala> val allPeople = **TODO**
scala> allPeople.take(5)
(United-States,Bachelors,Adm-clerical,Male)
(United-States,Bachelors,Exec-managerial,Male)
(United-States,HS-grad,Handlers-cleaners,Male)
(United-States,11th,Handlers-cleaners,Male)
(Cuba,Bachelors,Prof-specialty,Female)
```

2. Remove all people whose `occupation` is `"?"` (meaning `occupation` data is missing). Use the following to test your code. You should get the following count.

```
scala> val filteredPeople  = allPeople.**TODO**
scala> filteredPeople.count
30718
```

3. Next you need to extract two datasets from the census data to represent our union members—`canadians` and `americans` (people from United States). Put all rows of the reduced data containing Canadians into the RDD `canadians`. Do likewise for the `americans`. The following tells you the number of people you should get for each country.

```
scala> val canadians = filteredPeople.**TODO**
scala> val americans = filteredPeople.**TODO**
scala> canadians.count
107
scala> americans.count
27504
```

4. Now let's create all candidate pairs (`canadian`, `american`) which have the same occupation. Hint: the `join` function only works on pair RDDs, so you need to turn the rows into appropriate key-value pairs first. After doing the join, use the `values` function to discard the join key (see the template below).

```
scala> val repCandidates = **TODO**.join(**TODO**).values
scala> repCandidates.count
```

```
325711
```

We now have an RDD containing all possible representative pairs which could be chosen.

5. 325,711 is a lot of options! Let's add one more constraint, namely *at least one of the* employees must have a `Doctorate` for `education`. The count you should get is as follows.

```
scala> val includingDoctorate = repCandidates.**TODO**
scala> includingDoctorate.count
31110
```

There are still many candidates left, but at this stage you should have a feel for how repeatedly filtering data can support the decision making process.

Task G: Compiling Spark programs

Typing commands into an interactive Spark shell is a great way to quickly experiment with short snippets of code. But then you think about typing a full, production program with hundreds or thousands of lines into the shell and almost have a heart-attack. Surely there's a better way?

Since Scala is a programming language that runs on the Java Virtual Machine (JVM), it is possible to compile one or more Scala source files into a `.jar` file. This `.jar` file can then be submitted to the Spark master node to be run, similar to what we've done in the past with Hadoop MapReduce. We will now go through the process of building and running a Spark application `.jar`. The code for this application is in the `Task_G` directory supplied with this lab document.

1. In order to automate the build process we will be using a tool called Maven. Maven is a general Java build tool that can be used for all sorts of programs, not just Spark applications. In order to configure Maven to build a Spark application, we need to provide an appropriate `pom.xml` file. If you have a brief look at the file `Task_G/pom.xml` provided with this lab document, you should be able to see Spark and Scala dependencies listed in the configuration. Don't worry about the specific details of the configuration, just understand that it is necessary to build Spark applications.

2. Open a terminal in the `Task_G` directory, and invoke Maven to attempt building the application:

```
$ mvn package
```

The build will fail with a **compilation error**, which means that something is wrong with the source code we are trying to compile. We will need to fix this problem before the application can be built.

3. The application's source code is contained in `Task_G/src/main/Main.scala`, so open that file up in Jupyter. The purpose of the application is to calculate the average hours worked by people in each occupation. Read through the code and try to understand how it works.

4. The reason for the compilation error is that a line of code is missing. Look for the TODO comment in the file, and add code which **collects** the `sorted` RDD into a variable called `results`. Once you have added the missing code, use Maven to build the application.

5. If all went well, Maven should have produced a file called `spark_example-1.0-SNAPSHOT-uber.jar` in a directory called `target`. This is the `.jar` file which contains our compiled application (along with some dependencies). Now that we have a `.jar` file for our application, we can submit it to the Spark master to be executed as a job:

```
$ spark-submit target/spark_example-1.0-SNAPSHOT-uber.jar
```

6. When the job finishes, you can check that everything worked correctly by looking for the results in the console output. You should see the average hours worked in each occupation:

```
(46.98994,Farming-fishing)
(44.9877,Exec-managerial)
(44.65623,Transport-moving)
...
```

Well done for making it this far! We covered a lot of new ideas in this lab, so you should feel good about learning the fundamentals of Spark.