ISIT312/ISIT912 Big Data Management

Spring 2023

Spark Practice I

In this practice, you will perform basic operations and develop basic data processing applications in Spark.

Warning: DO NOT attempt to copy the Linux commands in this document to your working Terminal, because it is error-prone. Type those commands by yourself.

Laboratory Instructions.

(0) Start Hadoop services

Start the five Hadoop services in a Terminal window and start Zeppelin (if you use Zeppelin).

(1) How to start and interact with Spark in the Shark Shell?

There are two ways to interact with Spark: the Spark shell and Zeppelin.

To start the Spark shell, process the following command in a Terminal window (not Zeppelin):

```
$SPARK_HOME/bin/spark-shell --master local[*]
```

The above will run Spark in a local mode with a standalone cluster manager. (The * symbol means using multiple threads in the VM to process a Spark job.) You can also run it in a pseudo-distributed mode with YARN as the cluster manager, by processing:

```
$SPARK_HOME/bin/spark-shell --master yarn
```

See the lecture note for more information about the two modes. *Recommendation: Use the local mode for better efficiency.*

Spark-shell runs on top of the Scala REPL. To quit Scala REPL, type

```
:quit
```

If you use Zeppelin as the default interface to Spark, you need to specify the Spark interpreter `%spark` at the first line of your Scala commands. For example, process the following:

```
%spark
spark
```

You will see:

```
res0: org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@xxxxxx
```

Here a SparkSession instance named `spark` is the entry points to your Spark application.

IMPORTANT: *Do NOT* use the Spark shell and Zeppelin's %spark AT THE SAME TIME; just use either one of the two.

(2) Create and process DataFrames, and retrieve data from DataFrames

To create a simple DataFrame, process

```
val myRange0 = spark.range(20).toDF("number")
myRange0.show()
val myRange1 = spark.range(18).toDF("number")
myRange1.show()
myRange0.except(myRange1).show()
```

You can also create a DataFrame on the data in HDFS. First, load the file `README.txt` in `$SPARK_HOME` to HDFS in, say, `/user/bigdata`. Then read it into a DataFrame:

```
val text = spark.read.textFile("/user/bigdata/README.txt")
text.count()
text.first()
```

The following command counts how many lines contain the word "Spark":

```
text.filter(line => line.contains("Spark")).count()
```

The following command gets the length of the longest line:

```
text.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
```

The following command implements a (naïve) word count application:

```
val wordCounts = text.flatMap(line => line.split(" ")).
     groupByKey(identity).count()
wordCounts.show()
```

(3) DataFrame/Dataset transformations and actions

Use a link `Resources` to download the files `people.json`, `people.txt` and `employees.json` from Moodle. Create a folder `week10` on HDFS and upload the files `people.json`, `people.txt` to HDFS into a folder `/user/bigdata/week10`.

Process the following DataFrame/Dataset operations in Spark-shell:

```
// read a json file into a dataframe
val df = spark.read.json("/user/bigdata/week10/people.json")
df.show()
df.printSchema()

//some basic relational operations
df.select($"name", $"age" +1 ).show()
df.filter($"age" > 21).show()
df.groupBy("age").count().show()
df.createOrReplaceTempView("people")
val sqlDF = spark.sql("select * from people")
sqlDF.show()
```

```
//create a Dataset
case class Person(name: String, age: Long)
val ccDS = Seq(Person("Andy", 32)).toDS()
ccDS.show()
ccDS.select($"name").show()

// another way to create DataFrame
val peopleDF = spark.sparkContext.
  textFile("/user/bigdata/week10/people.txt").
  map(_.split(",")).
  map(attributes => Person(attributes(0), attributes(1).trim.toInt)).
  toDF()
peopleDF.show()

// convert DataFrame to Dataset
case class Employee(name: String, salary: Long)
val ds = spark.read.
       json("/user/bigdata/week10/employees.json").as[Employee]
```

(4) Implementation and processing of a self-contained application

In the following example, we implement a self-contained application and we submit it as a Spark job. Open a new document in Text Editor, input the following code and save it as `SimpleApp.scala`.

```
import org.apache.spark.sql.SparkSession
object SimpleApp {
  def main(args: Array[String]) {
    val text = "<YOUR_HDFS_PATH>/README.md"
    val spark = SparkSession.builder
      .appName("Simple Application")
      .config("spark.master", "local[*]")
      .getOrCreate()
    val data = spark.read.textFile(text).cache()
    val numAs = data.filter(line => line.contains("a")).count()
    val numBs = data.filter(line => line.contains("b")).count()
    spark.sparkContext.setLogLevel("ERROR")
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

Use Terminal or `%sh` interpreter on Zeppelin to compile an application `SimpleApp.scala` in the following way:

```
scalac -classpath "$SPARK_HOME/jars/*" SimpleApp.scala
```

Then create a `jar` file in the following way:

```
jar cvf app.jar SimpleApp*.class
```

Quit Spark Shell or stop Zeppelin before you submit it to Spark.

Use Terminal to process the application in the following way:

```
$SPARK_HOME/bin/spark-submit --master local[*] --class SimpleApp app.jar
```

The output is:

```
Lines with a: 62, Lines with b: 30
```

(5) Shakespeare wordcount exercise

Complete the following exercise (a sample solution will be released on Moodle later):

Use `Resources` on link on Moodle to download the datasets `shakespeare.txt`, and `stop-word-list.csv`.

An objective of the exercise is to count the frequent words used by William Shakespeare in a file `shakespeare.txt` but remove the known English stops words (such as "the", “and” and “a”) available `stop-word-list.csv`. Return top 20 most frequent non-stop words in Shakespeare’s works.

The first few lines of code are provided:

```
val shakes = spark.read.textFile("<your path>/shakespeare.txt")
val swlist = spark.read.textFile("<your path>/stop-word-list.csv")
val shakeswords = shakes.
      flatMap(x => x.split("\\W+")).
      map(_.toLowerCase.trim).
      filter(_.length>0)
shakeswords.createOrReplaceTempView("shakeswords")
val stopwords = swlist.flatMap(x=>x.split(",")).map(_.trim)
stopwords.createOrReplaceTempView("stopwords")

// your Scala code goes here...//
```

A hint is to create views that can be accessed with Spark SQL and of course … use SQL.

The final output is as follows:

```
result.show(20)
+-----+-----+
|value|count|
+-----+-----+
|    d| 8608|
|    s| 7264|
| thou| 5443|
|  thy| 3812|
|shall| 3608|
| thee| 3104|
|    o| 3050|
| good| 2888|
|  now| 2805|
| lord| 2747|
| come| 2567|
|  sir| 2543|
|   ll| 2480|
| here| 2366|
| more| 2293|
| well| 2280|
| love| 2010|
|  man| 1987|
| hath| 1917|
| know| 1763|
+-----+-----+
only showing top 20 rows
```