Big Data Assignment 5: *Standalone applications & Spark Structured Streaming*

Instructions

In this assignment, we will get some hands-on experience with Spark Streaming. We will start off by discussing standalone applications for Spark, and then we will dive deeper in a couple of Spark Streaming applications.

Throughout this document you will find questions which you will have to answer. These answers must be handed in as a PDF document on Brightspace.

The deadline for this assignment is **May 14th, 23:59**.

After completing (and handing in) the assignment, you can go to Brightspace Quizzes to test if you understood the assignment. Making the quiz is voluntary, but recommendable, as you can check your answers from the assignment there. If you have any questions (you find out your answer is incorrect and you do not know why), do not hesistate to come discuss it with us in the practicum (or send us an email / contact us in the matrix room, if going to campus is not an option for you).

Part 1: Spark standalone applications

Up until this point, we have executed our all Spark code in interactive Zeppelin notebooks. However, a cluster is usually shared by many people, and this model does not work that well with the concept of interactive notebooks - they tend to claim too many resources for a long time, and it is unclear when a job will be complete. Also, query optimization may be hindered when the program logic is distributed over multiple cells that are submitted as isolated operations.

As a result, "real" big data processing rarely happens directly from within your notebook. You mostly start and test your code on a small sample of data in a notebook, and then you do the *big* data processing in a standalone program that is deployed on the cluster. The `redbad` Docker container has been setup to simplify this process, so you can go through the full procedure on your own machine. For this assignment, though, we will not deploy our Spark applications on the cluster (yet). Instead, we will be compiling and executing them in the local Spark environment found in the `rubigdata/course` Docker container. (We will be using the `redbad` container in the final project, though.)

Scala Build Tool

We will use the Scala Build Tool to reduce the effort necessary to create standalone applications that are bundled with all their dependencies and can be submitted to multiple worker nodes without problem.

Briefly skim this blog post on `sbt` to get an idea what it is like; it is somewhat similar to using `maven` for java development.

RUBigDataApp

We have put a template directory structure ready to use in `/opt/hadoop/rubigdata` in your course container (I will assume you named it `big-data` in assignment 3, but you might of course have used a different name

when you issued `docker create`).

Glance over the Spark project's documentation on creating self-contained Spark apps and subsequently submitting applications to Spark. Eventually we will return to using the Yarn resource manager as in assignment two (see `spark-submit` for a yarn cluster), but for now we keep it simple using only Spark itself without Hadoop and Yarn.

The official documentation remains verbose and incomplete, so let us walk through the process with an actual (but *overly simple*) example of a Spark application written in Scala.

You find the sample application in a sub-directory of `/opt/hadoop/rubigdata`, the working directory when entering the container:

```
docker start big-data
docker exec -it big-data /bin/bash
```

The two inputs to building trivial sample app `RUBigDataApp` are the `build.sbt` file (the SBT equivalent of maven's `pom.xml`) and the actual code, provided in `src/main/scala/org/rubigdata/RUBigDataApp.scala`. **The directory structure is important** (do not change anything); there are many implicit assumptions about what is located where - for the details, look into the `sbt` documentation.

Look at the provided `RUBigDataApp.scala` code so you know what to expect to see as output!

Build the stand-alone application (the very first time you execute this in your container, it will take longer than subsequent commands):

```
sbt package
```

The result is a **jar** file generated in the `target/scala-2.12` directory, that we execute using Spark's `spark-submit` command:

```
spark-submit target/scala-2.12/rubigdataapp_2.12-1.0.jar
```

In your console, you will now nicely see the output of the Spark program (somewhere at the end of the logs).

```
########## OUTPUT ##########
Lines with a: 3, Lines with e: 4
############ END #############
```

The `rubigdata/course` Docker image is setup to run Spark applications locally on your computer. In technical terms, this means that Spark has a default *master* of `local[*]`. This indicates to Spark that we are running a local application instead of an application on a cluster using Yarn. In this assignment, we will only be

running our Spark applications locally. In the final project, we will see how we can execute applications on the cluster from the `redbad` container.

We have told Spark it can use as many threads as there are cores available on the machine by using the asterisk `[*]`. If you want to use fewer cores, you can specify the amount of cores as `local[K]` in the `master` parameter to the `spark-submit` call.

```
spark-submit --master 'local[4]' target/scala-2.12/rubigdataapp_2.12-1.0.jar
```

You can even leave out the amount of cores entirely, to run Spark without parallelism at all. **Do not do this for this assignment, though.** Spark needs at least one thread to handle the incoming stream data and one to actually execute the streaming application. For more information on the `master` parameter, see the Spark documentation.

Side note: it is also possible to specify the master from within your Spark application, using the `.master(...)` method on the `SparkSession.builder` instance. However, it is generally more useful to supply the master as parameter to `spark-submit`, as this allows you to change the master without recompiling the entire application.

Part 2: Spark Structured Streaming

Spark Structured Streaming offers query processing over dataframes in a streaming fashion. Before you start, read the excellent introduction to Spark Structured Streaming. The assignment further assumes that you have seen the Structured Streaming lecture and have read the background paper (posted on Brightspace).

The exercise uses the abstractions offered by Spark Structured Streaming to analyze the data from an online marketplace - inspired by a popular online game.

In that game:

- players sell various items; whenever an item is sold the transaction is reported;
- every item has a material (e.g., Iron, Steel), a type (Sword, Shield), and a price.

In order to get live updates on the economy, a Spark Streaming application would be perfect!

By the way, did you spot the easter-egg?

Input stream

Okay, let's start! For the sake of the assignment, we will be running a simulation that generates a stream of events inside the course container. This is a simplification (on a simulated cluster, really); in reality you'd be reading your data from a Kafka input stream over an internet connection to a remote server.

The data generator is a python program that writes RuneScape-"like" output to port 9999. Once you start the course container, the generator is automatically started in the background. To see what the data looks like, we can use `netcat` (from inside the `big-data` container) to connect to the stream and gather some data.

```
nc localhost 9999
```

You should now see a large amount of transactions scrolling by in your terminal. To disconnect from the stream, press CTRL+C.

Our first streaming application

In order to create our streaming application, we will be building from the `RUBigDataApp.scala` we just executed. Refer to assignment 2 for a refresher on how to edit files inside your Docker containers. Now, from our letter-counting program, you can remove all lines of code in the `main`, except the lines where the `spark` instance is created and stopped. These are needed for every stand-alone application to build (and stop) the SparkSession, so we need them in our new program as well.

Then, we need to implement the Spark *implicits* to work with Datasets. This needs to be added after the creation of the `spark` instance, so in the `main` function (on top of the file like other imports does not work for this one).

```
import spark.implicits._
```

Then, we can create a dataframe tied to the TCP/IP stream of our data generator on `localhost:9999`, using the `readStream` operation.

```
val socketDF = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()
```

This is by any means the same as the regular dataframes we have worked with up until this point. However, internally, it identifies itself as a *Streaming Dataframe*. You could verify this by printing the value `socketDF.isStreaming`, which would evaluate to `true`.

Streaming dataframes also make use of Spark's lazy evaluation paradigm. Nothing has actually happened at this point. Only when we define a query and start the streaming job, Spark will try to connect to the data generator and read its data.

To verify that the streaming application works correctly, we will start by using a very basic query. We simply output all values that Spark has collected from the data generator.

```
val query = socketDF
  .writeStream
  .outputMode("append")
  .format("console")
  .option("truncate", false)
  .start()
```

A couple of things are noteworthy here:

- We define an output mode of `append`. Spark works with a *micro-batch* approach, meaning it processes all output up until a certain point, generates the corresponding output, and then repeats this process with whatever new data has been collected in the meantime. By specifying the output mode `append`, we tell Spark it only has to print out the lines it hasn't output yet. In other words, it will only print the values corresponding to the latest micro-batch, instead of all values it has ever received. *Note: this only works if rows in the output are not updated, i.e. if we do not perform any grouping or aggregation.*

  Click here to read more about output modes.

- We use `console` as output format, also known as output *sink*. This tells Spark output should simply be printed to the console, instead of written to a file or other application. This should only be used for debugging purposes or simple use cases such as this assignment. In reality, you would perform some operations on your streaming data and then make the data available to other applications (e.g. a dashboard frontend).

  Click here to read more about output sinks.

- We set the `truncate` option to `false`. This only works for the `console` output, and is used to that values are not truncated when they are printed to the console.

- The `start` method is used to actually kick off the streaming task.

Finally, if we want our application to keep on running, we will need to tell Spark to wait until the query is terminated before the application is exited.

```
query.awaitTermination()
```

That's all! We have now successfully written a full Spark Structured Streaming application. Piece of cake, right?

To run the application and see it in action, we first compile it with `sbt` and then submit it using `spark-submit`.

```
sbt package
spark-submit target/scala-2.12/rubigdataapp_2.12-1.0.jar
```

After submitting the jar file, we can first see the default Spark logging information. After a few seconds, the streaming job should start, and you should see the micro-batchs coming in. It should look something like this:

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-----+
|value|
+-----+
+-----+
```

```
-------------------------------------------
Batch: 1
-------------------------------------------
+-----------------------------------------+
|value                                    |
+-----------------------------------------+
|Steel Mace was sold for 869gp            |
|Mithril Mace was sold for 2317gp         |
|Bronze Two-handed sword was sold for 2343gp|
|...                                      |
+-----------------------------------------+
```

To stop the application, you can press CTRL+C.

Note: if you do not wish to see so many logs around your output batches, you can set the log level to something like `"WARN"`. This displays less log information, however, less logs also make debugging harder. So, choose your settings wisely.

```
spark.sparkContext.setLogLevel("WARN")
```

Parsing the input stream

In a previous assignment, we have seen that we can use the Spark Dataset API to structure the data in our dataframes. This, in turn, allows us to work with the typed dataset API as opposed to the untyped dataframe API. Let's take a look at how to do this for our RuneScape data.

First off, we will define the type each row in our dataset will have. For now, we will only record the type of the item and the sale price. We denote the type using `tpe`, as `type` is a reserved value in Scala. To register the type, add the following line to `RUBigDataApp.scala` (outside the `RUBigDataApp` object).

```
case class RuneData(tpe: String, price: Int)
```

Since each line follows the same generic pattern of "[material] [type] was sold for [price]gp", we can use a regular expression (regex) to parse the data. To do so, we will first define the following regex:

```
val regex = "^[A-Z].+ ([A-Z].+) was sold for (\\d+)gp$"
```

The parts inside the parentheses are called *groups*, and we can use the `regexp_extract` function of the Spark SQL API to execute the regex on the input value and extract the groups. We first need to import the relevant Spark SQL functions.

```

```

```
// Add these to the top of the file, with the other imports
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.scalalang.typed
```

Then, we can use the `select` and `regexp_extract` functions to transform the input stream into a structured output stream.

```
val sales = socketDF
  .select(
    regexp_extract($"value", regex, 1) as "tpe",
    regexp_extract($"value", regex, 2).cast(IntegerType) as "price"
  )
  .as[RuneData]
```

If we now edit the output stream to read from `sales` instead of `socketDF`, we can see our newly structured data in action.

```
val query = sales
  .writeStream
  .outputMode("append")
  .format("console")
  .option("truncate", false)
  .start()

query.awaitTermination()
```

And then we compile and run the application again:

```
sbt package
spark-submit target/scala-2.12/rubigdataapp_2.12-1.0.jar
```

The output should now look something like this:

```
-------------------------------------------
Batch: 0
-------------------------------------------
+---+-----+
|tpe|price|
+---+-----+
+---+-----+

-------------------------------------------
Batch: 1
```

```
-----------------------------------------
+---------------+------+
|tpe            |price |
+---------------+------+
|Sword          |678   |
|Halberd        |12847 |
|Warhammer      |21821 |
|...            |...   |
+---------------+------+
```

This is already a lot easier to work with! However, we still don't have access to the material of the item, and we do want to run some analyses on that as well. So, we'll have to include that in our parsing of the data.

Update the `RuneData` class, the value `regex` and the dataframe `sales` such that we can also use the material of the item. The remainder of the assignment assumes you have implemented this.

Running queries on the data

We've seen that the streaming dataframes work in a similar way as the normal dataframes from previous assignments. This means we also have the same ways of performing queries on our streaming data:

- Using the untyped DataFrame API
- Using the typed Dataset API
- Using the Spark SQL API

We will take a look at each of them using a basic example. Suppose we want to find out how much gold is being spent in total on daggers of each specific material type. To do so, we need to:

- Filter on the "Dagger" type
- Group on the material
- Sum over all prices

To do this using the DataFrame API, we can use the following query:

```
val counts = sales
  .filter($"tpe" === "Dagger")
  .groupBy("material")
  .agg(sum($"price") as "total_sales")
```

To do it with the typed Dataset API, we can use the following:

```
val counts = sales
  .filter(_.tpe == "Dagger")
  .groupByKey(_.material)
  .agg(typed.sum(_.price))
  .toDF("material", "total_sales")
```

And finally, we could also use the following query to make use of the Spark SQL API:

```
sales.createOrReplaceTempView("sales")
val counts = spark.sql("SELECT material, SUM(price) AS total_sales " +
                       "FROM sales " +
                       "WHERE tpe = 'Dagger' " +
                       "GROUP BY material")
```

To see these in action, we need to define the output write stream from `counts`. However, because the output contains an aggregation over groups, we need to overwrite the entire output for each mini-batch. We can do this by using the output mode `complete` instead of `append`.

```
val query = counts
  .writeStream
  .outputMode("complete")
  .format("console")
  .start()
```

After compiling and running these different queries, you should see that each of the different ways of defining the query ultimately results in the same output.

Now that you know how to handle streaming data in Spark, you can combine it with the knowledge you gathered on Spark in the previous weeks to answer the following questions:

Question 1: *Which of the materials is (on average) most expensive? (Also write down your added code.)*

Question 2: *Which of the weapons is (on average) most expensive? (Also write down your added code.)*

Question 3: There is one item that costs roughly 9850 GP. *Which item is this? (Also write down your added code.) Hint: Spark might truncate output tables with more than 20 rows. To mitigate this, you might have to filter your output using the SQL HAVING clause, or you can play with the `numRows` option of the output stream.*

Question 4: *How much gold is (on average) spent on Mithril swords? (Also write down your added code.)* Note that there are multiple types of swords. For the purposes of this assignment, we will consider every item type that contains the string "sword".

Joining with external data

Suppose we want to enrich the streaming data from above with some more information on each item. From the RuneScape wiki, we have gathered some extra information on each type of item, e.g. whether it is a one-handed or two-handed weapon, and what its damage type/method is. You can find it in the `weapons.csv` file (in the Brightspace assignment).

First, let's copy this file over to the Docker container.

```
docker cp weapons.csv big-data:/opt/hadoop/rubigdata/
```

Then, inside our streaming application, we will include a `case class` for this data, and read it from the CSV.

```
// Put this outside the RUBigDataApp object
case class Weapon(name: String, num_hands: Int, speed: String, method: String)

// Add this inside the main function
val weapons = spark.read
  .option("header", "true")
  .option("inferSchema", "true")
  .csv("/opt/hadoop/rubigdata/weapons.csv")
  .as[Weapon]

weapons.createOrReplaceTempView("weapons")
```

By inspecting the data, we can see that the `tpe` of a transaction is equal to the `name` of a weapon. Hence, we can execute a `JOIN` on the two datasets, joining on these values. For our use case, an `INNER JOIN` should suffice.

```
val combined = spark.sql("SELECT * FROM sales " +
                         "INNER JOIN weapons " +
                         "ON sales.tpe = weapons.name")
```

Using this combined data, answer the following questions:

Question 5: *Which weapons are, on average, most expensive? One-handed or two-handed weapons? (Also write down your added code.)*

Question 6: *On which weapons are, in total, most gold pieces spent? One-handed or two-handed weapons? (Also write down your added code.)*

Question 7: Suppose we have the fixed `weapons` dataset, and the streaming `sales` dataset. We want to enrich the weapons dataset by including the total amount of sales for each weapon type. *Will the following query work? Briefly explain why/why not.*

```
SELECT weapons.*, sales_agg.total_sales
FROM weapons
LEFT OUTER JOIN
    (
        SELECT tpe, SUM(price) AS total_sales
        FROM sales
        GROUP BY tpe
    ) AS sales_agg
ON weapons.name = sales_agg.tpe
```

Checkpointing

As you know, Spark is built with resilience and fault-tolerance in mind. But how do you provide data certainty with streaming data? For our current streaming application, it does not matter if previously processed data is lost if the application crashes or is exited. However, you can imagine that large-scale production streaming applications have different needs, and we don't want to lose any data when the application suddenly crashes.

To support this, Spark Structured Streaming makes use of "checkpointing". Checkpointing basically boils down to saving intermediate states of the application, which can be used as starting point if the application needs to be restarted. We will not go into a lot of detail on checkpointing in this assignment, so it is fine if you take it for granted for now. If you are interested in learning more, you can read about it in the Spark Structured Streaming guide.

Also, you can enable checkpoints yourself by proving the `checkpointLocation` option to the output write stream, as follows:

```
val query = counts
  .writeStream
  .outputMode("complete")
  .format("console")
  .option("checkpointLocation", "/opt/hadoop/rubigdata/.sparkCheckpoints")
  .start()
```

If you do this, and then you start, exit and restart the application, you will see that it picks up where it left off and starts with the next batch in line.

Note: the socket input stream does not provide end-to-end fault-tolerance guarantees, and might crash if you decide to use checkpointing. Experiment at your own risk.

Wrapping up

In this assignment, you have hopefully seen that Spark Structured Streaming provides us with a nice interface for streaming applications, built on top of the existing DataFrame, Dataset and SQL APIs. You have developed a couple of small standalone streaming applications which have helped you get a basic feeling for programming with Spark streaming.

Hand in your answers to each of the questions as a PDF document on Brightspace. The deadline for this assignment is included at the top of this document. Use the quiz (under Brightspace - Activities) to check your answers.

Feel free to play around a bit more with Spark Streaming and our marketplace datastream. From here on, the initiative is yours; time for some creativity! Try to create your own dashboard, that you might have used to take decisions in the original game. Can you think of related queries, or have a go at an even more advanced report that is continuously updated? What we are after is a real-time agent that would buy and sell on the market and make you rich while you study!