CSE 344 Homework 6: Parallel Data Processing and Spark

Objectives: To practice writing queries that are executed in a distributed manner. To learn about Spark and running distributed data processing in the cloud using AWS.

Assigned date: Wednesday, May 9, 2018

Due date: Wednesday, May 16, 2018, 11:30 pm

What to turn in:

A single Java file for entire assignment, `HW6.java` in the `submission` directory, along with the text outputs from AWS. A skeleton `HW6.java` has been provided for you in the starter-file. **Make sure you copy this over to `submission` before working on the assignment to avoid repo conflicts.** You should not change any of the method's signatures. You should not need to create any other class to complete this assignment, although you are free to do so.

Resources

- Spark programming guide
- Spark Javadoc
- Amazon web services EMR (Elastic MapReduce) documentation
- Amazon S3 documentation
- Small dataset for local testing
- Prof. Cheung's research group has been working on a tool called Casper that translates sequential Java to Hadoop/Spark. Feel free to use it to learn Spark syntax. This is entirely optional.

Assignment Details

In this homework, you will be writing Spark and Spark SQL code, to be executed both locally on your machine and also using Amazon Web Services.

We will be using the same flights dataset from HW2 and HW3. This time, however, we will be using the *entire* data dump from the US Bereau of Transportation Statistics, which consists of information about all domestic US flights from 1987 to 2011 or so.

The data is stored in a columnar format called Parquet and is available publicly on Amazon S3. S3 is Amazon's cloud storage service. The data is around 4GB compressed (79GB uncompressed), and contains 162,212,419 rows. Hence you probably don't want to be crunching on this dataset using your laptop! Each row contains 109 different attributes (see `HW6.java` for details), although you will only need to use a few of them in this homework. Note that the attribute names are different from those in HW2/HW3.

To help you debug, we provide a subset of the data on the course website. This dataset is a dump of all the flights that happened on November 4, 2010. We strongly encourage you to run your code using this small dataset locally before trying it out on AWS. See instructions below.

A. Sign up on Amazon Web Services

Follow these steps to set up your Amazon Web Services account. **You should have already done this since the second week of the quarter, but we just repeat the instructions here for your reference.**

1. If you do not already have an Amazon account, go to their website and sign up. Note: Amazon will ask you for your credit card information during the setup process. This is normal. Then sign in to your AWS console, go to "Support -> Support center" in the navigation bar, and locate your **account number**.
2. To get $$$ to use Amazon AWS, you must apply for credits by going to their education website. **You must use your UW email address, @uw.edu, when registering for the credits, as they use this to verify your identity.** Leave the promo code blank, and enter your AWS account number on the next page. Make sure you don't check the starter account option on the final page as that has limited permissions which may cause problems.
3. After applying, you will have to wait to be approved. You should get an email when your application has been approved, which gives you a credit code. Make sure you check the spam folder. Once you have it, go to the AWS website and apply the credit. We have no control / idea how long this can take, but was told it

can range from minutes to days. Hence, it is crucial that you apply ASAP!

IMPORTANT: if you exceed the credit you are given, Amazon will charge your credit card without warning. If you run AWS in any other way rather than how we instruct you to do so below, you must remember to manually terminate the AWS clusters when you are done. While the credit that you receive should be more than enough for this homework assignment, but you should still monitor your billing usage by going to their billing website and clicking on "Bills" (upper left).

You should get $100 from AWS once your application is approved. The credits that you have left over after the assignment are for you to keep, but if you exceeded the credits due to forgetting to turn on your clusters / mining bitcoins etc then you will be responsible for paying the extra bill.

Now you are ready to run applications using Amazon cloud. But before you do that let's write some code and run it locally.

B. Download Spark

Run `sudo ./installSpark.sh` in your `starter-code` directory in your VM. This will download Spark 2.2.0 and unzip it into `/usr/local` on your VM.

For this assignment, we will be running your submissions from the home VM, and will only support those platforms. **Please make sure your code compiles and runs there!** You are on your own if you decide to use your own platform.

Here are some hints in case you like to install Spark on your own Windows machine.

C. Run Code Locally

Use your favorite text editor / IDE and open `HW6.java` to check out the code. You have provided an example method, `warmup`, that you can run after compiling the code. To compile, add all the jar files that come with Spark to classpath as follows: (Make sure that you use the fully-qualified directory path) (On Unix or OSX you need to put " " in the classpath, i.e., `"/usr/local/spark-2.2.0-bin-hadoop2.7/jars/*"`)

```
$ javac -cp /usr/local/spark-2.2.0-bin-hadoop2.7/jars/* HW6.java
```

You can now run the code as (use ";" instead of ":" as the classpath separator on Windows):

```
$ java -cp /usr/local/spark-2.2.0-bin-hadoop2.7/jars/*:. HW6 <path to the flights data directory> <path to output directory wher
e you want the output to be stored>
```

If you are using Eclipse, you can do the same thing by right clicking on the project name → "Build Path" → "Configure Build Path" → Libraries → "Add External JARs". To run you program with arguments, go to "Run" → "Run Configurations".

You might get some warning messages such as
`WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable`.
You can safely ignore those.

The output will be stored in a file (called `part-XXXX`) in the output directory you specified (the directory structure may depend on your platform). That is just a plain text file that you can open that file with any text editor. **You will get an error if the output directory already exists, so delete it before running your program.** Be sure you read through the code and understand what is going on.

D. Run Code on EMR

We will use Amazon's Elastic Map Reduce (EMR) to deploy our code on AWS. Follow these steps to do so, after you have set up your account and received credits as mentioned above.

1. Toggle the `SparkSession` initialization on line 32-36 of `HW6.java` to allow it to run on AWS. Then create a jar file containing your class files. On the home VM you can do this with:

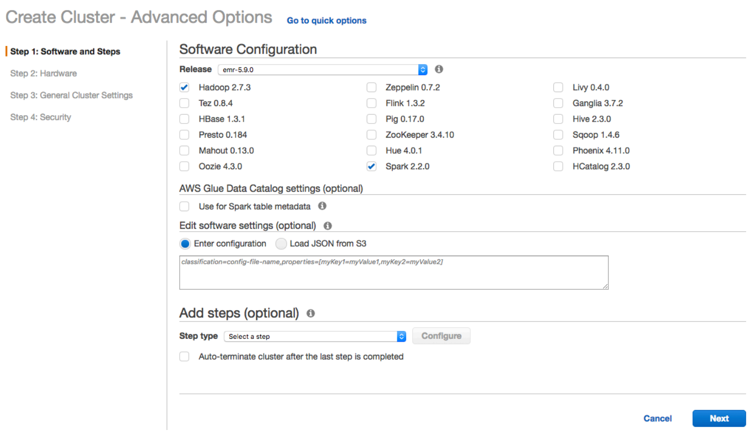
   ```
   $ jar cf HW6.jar *.class
   ```

   This creates `HW6.jar` that includes all `.class` files inside your current directory.

2. Log in to S3 and create a "bucket." S3 is Amazon's cloud storage service, and a bucket is similar to a folder. Give your bucket a meaningful name, and leave the settings as default. Upload the jar file that you created in Step 1 to that bucket by selecting that file from your local drive and click "Upload" once you have selected the file.

3. Log in to the EMR website. Make sure you select `US East (N. Virginia)` or `US East (Ohio)` on the upper right --- the full data file that we will use is stored in a bucket there, so it will be faster to access from a machine located on the east coast.

4. We will now launch a three node m3.xlarge EMR cluster with Spark to execute the code. m3.xlarge is an "instance type" on AWS. You can find out about other instances on the AWS website. Go to the **Create Cluster – Advanced Options** in the Amazon EMR console. Scroll down to the **Software Configuration** section to add Spark as an application. Select the proper options and make sure that your screen looks like this:



Create Cluster - Advanced Options

Step 1: Software and Steps

Software Configuration

Release: emr-5.9.0

✓ Hadoop 2.7.3	Zeppelin 0.7.2	Livy 0.4.0
Tez 0.8.4	Flink 1.3.2	Ganglia 3.7.2
HBase 1.3.1	Pig 0.17.0	Hive 2.3.0
Presto 0.184	ZooKeeper 3.4.10	Sqoop 1.4.6
Mahout 0.13.0	Hue 4.0.1	Phoenix 4.11.0
Oozie 4.3.0	✓ Spark 2.2.0	HCatalog 2.3.0

AWS Glue Data Catalog settings (optional)

Use for Spark table metadata

Edit software settings (optional)

[✓ Enter configuration] Load JSON from S3

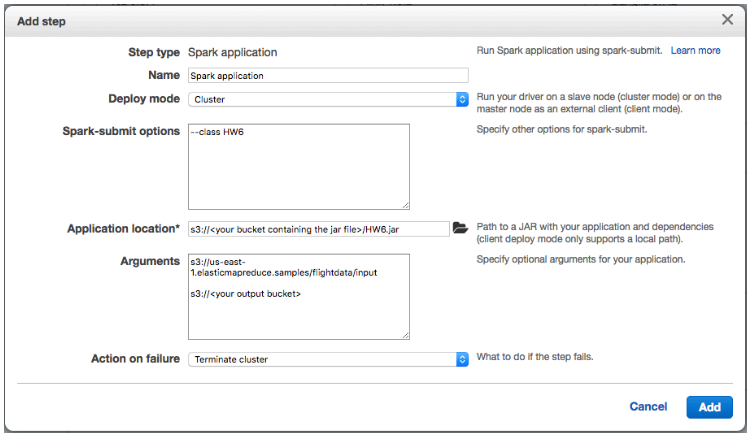
classification=config-file-name,properties=[myKey1=myValue1,myKey2=myValue2]

Add steps (optional)

Step type: Select a step

Auto-terminate cluster after the last step is completed

5. Next, scroll to the **Steps** section near the bottom of the page and select **Spark application**. A "step" is a single Spark job to be executed. You can specify multiple Spark jobs to be executed one after another in a cluster. Add a Spark application step by filling in the textboxes so that your screen looks like this:



Add step

Step type	Spark application	Run Spark application using spark-submit.
Name	Spark application	
Deploy mode	Cluster	Run your driver on a slave node (cluster mode) or on the master node as an external client (client mode).
Spark-submit options	--class HW6	Specify other options for spark-submit.
Application location*	s3://<your bucket containing the jar file>/HW6.jar	Path to a JAR with your application and dependencies (client deploy mode only supports a local path).
Arguments	s3://us-east-1.elasticmapreduce.samples/flightdata/input s3://<your output bucket>	Specify optional arguments for your application.
Action on failure	Terminate cluster	What to do if the step fails.

The `--class` option under Spark-submit options tells Spark where your `main` method lives (in this case inside `HW6.class`).

Make sure you fill out the correct bucket names. There are two arguments listed (and separated by space, as if you were running the program locally). The first one specifies where the input data file directory is (the full data file is located in the bucket `s3://us-east-1.elasticmapreduce.samples/flightdata/input` so you should put that there), and the second argument specifies where you want the output to be written. EMR can only write to a bucket, but you can download the output from the bucket afterwards. You can use the same bucket where you uploaded the HW6 jar, but we recommend using a separate folder each time (there will be failures if file already exists).

Change **Action on failure** to **Terminate cluster** (otherwise you will need to terminate the cluster yourself). Then click **Add**.

6. Back to the main screen, now check the **Auto-terminate cluster after the last step is completed** option at the bottom of the page, so the cluster will automatically shut down once your Spark application is finished. Click **Next**.

7. On the next screen you will get to choose how many machines you want in your cluster. For this assignment **1 master instance and 2 core (i.e., worker) instances of m3.xlarge should be enough**. (You are free to choose others but make sure you check their price tags first...) Click **Next**.

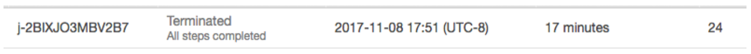
8. Under **General Options** uncheck the **Termination protection** option. We recommend that you can also specify a S3 bucket to store the logging information (easiest if you just specify the same bucket as where you uploaded your HW6 jar file). Click **Next**.

9. Finally, you can optionally create an EC2 pair if you want to ssh into the machines you are about to create. Doing so is optional. Click **Create cluster** once you are done and your cluster will start spinning up!

It will take a while for AWS to start the machines and run your Spark job. As a reference, it took 10 mins to run the `warmup` job on EMR using 1 master and 2 core nodes. You can monitor the status of your cluster on the EMR homepage. You might also find the "Clone" cluster functionality useful.

Once you are done, make sure you terminate the entire cluster (it should do so if you selected the options above). You can check cluster status on the EMR homepage). You can now check the output by clicking on the bucket you have created on S3. Text written to standard output, if any (e.g., from `System.out.println`), are located in the `containers/application<ID>/container<ID>/stdout.gz`, if you have enabled logging when you launched the EMR cluster.

It's fine if you see warning (or even occasional error) messages in the logs. If your EMR job finishes successfully you should see something similar to the below in the main EMR console screen:



j-2BIXJO3MBV2B7	Terminated All steps completed	2017-11-08 17:51 (UTC-8)	17 minutes	24

S3 charges by downloading/uploading data from/to the buckets. So once you are done with the assignment you might want to delete all the buckets that you have created (in addition to shutting down any EMR clusters that you have created).

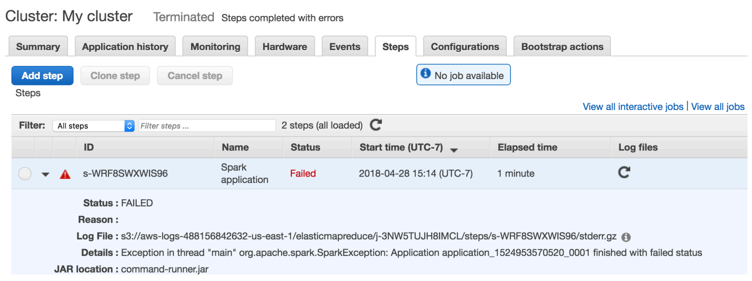
The amount you are allocated from Amazon should be more than enough to complete the assignment. And every year we have students forgetting to shut down their cluster / clean up their buckets and that can result in substantial charges that they need to pay out of pocket. **So be warned!!!**

Debugging AWS jobs

Debugging AWS jobs is not easy. Obviously, you should first make sure your program works locally before running on AWS. Here are some general tips:

- Make sure that you set the job details (i.e., options, arguments, bucket names etc) correctly! 80% of problems are due to that.

- The easiest way to debug is to look at the output / logging files. Spark generates a lot of log files, the most useful ones are probably the `stderr.gz` files listed under `containers/application.../container/stderr.gz`. You will have one `container` folder per machine instance. So make sure you check all folders. You should also check out the log files that you have specified when you created the job in Step 7 above. You can also see the names of those files listed as "Log File" under "Steps":

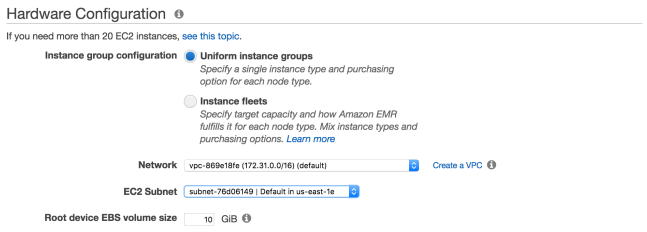


Cluster: My cluster Terminated Steps completed with errors

	ID	Name	Status	Start time (UTC-7)	Elapsed time	Log files
	s-WRF8SWXWIS96	Spark application	Failed	2018-04-28 15:14 (UTC-7)	1 minute	

Status : FAILED
Reason :
Log File : s3://aws-logs-488156842632-us-east-1/elasticmapreduce/j-3NW5TUJH8IMCL/steps/s-WRF8SWXWIS96/stderr.gz
Details : Exception in thread "main" org.apache.spark.SparkException: Application application_1524953570520_0001 finished with failed status
JAR location : command-runner.jar

- If you can't find available instances in a region, try changing to a different *EC2 subnet*, like so:



Hardware Configuration

If you need more than 20 EC2 instances, see this topic.

Instance group configuration: [✓ Uniform instance groups] *Specify a single instance type and purchasing option for each node type.* / Instance fleets *Specify target capacity and how Amazon EMR fulfills it for each node type. Mix instance types and purchasing options. Learn more*

Network: vpc-869e18fe (172.31.0.0/16) (default)

EC2 Subnet: subnet-76d06149 | Default in us-east-1e

Root device EBS volume size: 10 GiB

- Spark have a web UI that you can set up to check on job progress etc. You can check out their webpage for details. But these are more involved so you are probably better off to first try examining the logs.

E. Problems

We have created empty method bodies for each of the questions below (`Q1`, `Q2`, etc). Please don't change any of the method signatures. You are free to define extra methods (or even classes) if you need to. Run all of the following problems on the full dataset.

Except for problem 1, you should use the methods from the JavaRDD API. Optionally you can also use `reduceByKey`, `toRDD` from JavaPairRDD, and `toJavaRDD` from JavaRDD classes to implement your solution. Save the output from EMR to `q1.txt`, `q2.txt` etc and **make sure you add those to your `submission` directory on gitlab**.

1. Select all flights that leave from Seattle, WA, and return the destination city names. Only return each city name once. Implement this using the `Dataset` API and writing a SQL query. This should be trivial and is intended for you to learn about the `Dataset` API. You can either use the functional form (i.e., `join(...)`, `select(...)`) or write a SQL query using `SparkSession.sql`. Check the corresponding Spark Javadoc for the parameters and return values. Save the EMR output as `q1.txt` and add it to your repo. (10 points) [Result Size: 79 rows (50 rows on the small dataset), 10 mins on EMR]

   Hint: If you decide to write a SQL query, note that you can use single quotes inside your query for string literals (e.g., `'Seattle'`). Also, it does not matter what you name the output column as, since that information is not dumped to the output.

2. Implement the same query as above, but use the `RDD` API. You can convert a `Dataset` to a `JavaRDD` by calling `javaRDD()`, which we did for you in the skeleton code. Save the EMR output as `q2.txt` and add it to your repo. (20 points) [Result Size: 79 rows (50 rows on the small dataset), 15 mins on EMR]

3. Find the number of non-cancelled flights per month that departs from each city, return the results in a RDD where the key is a pair (i.e., a `Tuple2` object), consisting of a `String` for the departing city name, and an `Integer` for the month. The value should be the number of non-cancelled flights. Save the EMR output as `q3.txt` and add it to your repo. (20 points) [Result Size: 4383 rows (281 rows on the small dataset), 17 mins on EMR]

4. Find the name of the city that is connected to the most number of other cities within a single hop flight. Return the result as a pair that consists of a `String` for the city name, and an `Integer` for the total number of flights to the other cities. Save the EMR output as `q4.txt` and add it to your repo. (25 points) [Result Size: 1 row, 19 mins on EMR]

   Hint: check out this post if you get a "Task Not Serializable exception in Spark" exception.

5. Compute the average delay from all departing flights for each city. Flights with NULL delay values (due to cancellation or otherwise) should not be counted. Return the results in a RDD where the key is a `String` for the city name, and the value is a `Double` for the average delay in minutes. Save the EMR output as `q5.txt` and add it to your repo. (25 points) [Result Size: 383 rows (281 rows on the small dataset), 17 mins on EMR]

Submission Instructions

Turn in your `q1.txt`, `q2.txt`, ..., `HW6.java` and any other Java files that you created.

Important: To remind you, in order for your answers to be added to the git repo, you need to explicitly add each file:

```
$ git add HW6.java ...
```

Again, just because your code has been committed on your local machine does not mean that it has been submitted -- it needs to be on GitLab!

Use the same bash script `turnIn_Hw6.sh` in the root level directory of your repository that commits your changes, deletes any prior tag for the current lab, tags the current commit, and pushes the branch and tag to GitLab.

If you are using Linux or Mac OSX, you should be able to run the following:

```
$ ./turnIn_Hw6.sh
```

Like previous assignments, make sure you check the results afterwards to make sure that your file(s) have been committed.