CS 6240: Assignment 2

Goals: For MapReduce, compare Combiners to the in-mapper combining design pattern. Use keys, comparators, and Partitioner to implement secondary sort. Then explore how this would be implemented in Spark Scala.

This homework is to be completed individually (i.e., no teams). You have to create all deliverables yourself from scratch. In particular, it is not allowed to copy someone else's code or text and modify it. (If you use publicly available code/text, you need to cite the source in your code and report!)

Please submit your solution through Blackboard by the due date shown online. For late submissions you will lose one percentage point per hour after the deadline. This HW is worth 100 points and accounts for 15% of your overall homework score. To encourage early work, you will receive a 10-point bonus if you submit your solution on or before the early submission deadline stated on Blackboard. (Notice that your total score cannot exceed 100 points, but the extra points would compensate for any deductions.) Always package all your solution files, including the report, into a single standard ZIP file. Make sure your report is a **PDF** file.

For each program submission, include complete source code, build scripts, and small output files. Do not include input data, output data over 1 MB, or any sort of binaries such as JAR or class files.

The following is now *required*: To enable the graders to run your solution, make sure you include a standard Makefile with the same top-level targets (e.g., *alone* and *cloud*) as the one Joe presented in class (see the Extra Material folder in the Syllabus and Course Resources section). You may simply copy Joe's Makefile and modify the variable settings in the beginning as necessary. For this Makefile to work on your machine, you need Maven and make sure that the Maven plugins and dependencies in the pom.xml file are correct. Notice that in order to use the Makefile to execute your job elegantly on the cloud as shown by Joe, you also need to set up the AWS CLI on your machine. (If you are familiar with Gradle, you may also use it instead. However, we do not provide examples for Gradle.)

As with all software projects, you must include a README file briefly describing all of the steps necessary to build and execute both the standalone and AWS Elastic MapReduce (EMR) versions of your program. This description should include the build commands and fully describe the execution steps. This README will also be graded and you will be able to reuse it on all of this semester's assignments with little modification (assuming you keep your project layout the same).

You have 2 weeks for this assignment. Section headers include recommended timings, e.g., "complete in week 1", to help you schedule your work. Of course, the earlier you work on this, the better.

Set up Github (Week 1; skip if you did this for HW 1 already)

You will now have to use the CCIS Github when developing your code. Find the **CCIS Github** (not the public Github!) server and create a project for this homework. Use an IDE like Eclipse with the corresponding Github plugin to pull and push your code updates. Make sure you do the following:

- Set all your projects for this course so that they are private in general, but accessible to the TAs and instructor. (We posted our CCIS logins in week 3.)
- Make sure you commit and push changes regularly. As a rule of thumb, the "delta" between consecutive snapshots of your source code should be equivalent to about 2 hours' worth of coding. We do not want to see you committing large, complete chunks of code that look like you just copied from someone else.

Climate Analysis in MapReduce

We continue working with the weather data from HW 1, located at ftp://ftp.ncdc.noaa.gov/pub/data/ghcn/daily/by_year/. You must use *unzipped* files as input to your MapReduce program.

1. Complete in Week 1: Write three MapReduce programs that calculate the **mean minimum temperature** and the **mean maximum temperature, by station, for a single year of data**.
   Reducer Output Format (lines do *not* have to be sorted by StationID):
   StationId0, MeanMinTemp0, MeanMaxTemp0
   StationId1, MeanMinTemp1, MeanMaxTemp1
   …
   StationIdn, MeanMinTempn, MeanMinTempn
   a. NoCombiner: This program should have a Mapper and a Reducer class with no custom setup or cleanup methods, and no Combiner or custom Partitioner.
   b. Combiner: This version of the program should use a Combiner. Define the Combiner in the best possible way you can think of.
   c. InMapperComb: This version of the program should use in-mapper combining to reduce data traffic from Mappers to Reducers. Think of the best possible way to achieve this and make sure the *Combiner is disabled*.
2. Complete in Week 2: Create time series of temperature data. Using 10 years of input data (1880.csv to 1889.csv), calculate **mean minimum temperature** and **mean maximum temperature, by station, by year**. Use the secondary sort design pattern to minimize the amount of memory utilized during Reduce function execution. (Do *not* tinker with data types, e.g., short versus long integers, but focus on exploiting sort order in the Reduce function input list to reduce the need for many local variables.)
   Reducer Output Format (lines do *not* have to be sorted by StationID):
   StationIda, [(1880, MeanMina0, MeanMaxa0), (1881, MeanMina1, MeanMaxa1) … (1889 …)]
   StationIdb, [(1880, MeanMinb0, MeanMaxb0), (1881, MeanMinb1, MeanMaxb1) … (1889 …)]
   …

Climate Analysis in Spark

While working on the MapReduce part above, explore an equivalent Spark Scala implementation of the same programs. Start by setting up your Spark development environment already. You can set it up from scratch, following one of the tutorials on the Web. Or you can use an existing virtual machine (VM) with a readily installed Spark environment. For example, the authors of "Spark in Action" provide such a VM (see end of Section 1 in the book for instructions). Cloudera also offers environments with Hadoop and Spark.

This part of the HW is exploratory, because we have not covered Spark in detail in class. We want you to learn about relevant Scala commands and slowly become familiar with them. In the end, you do not need to have a fully working Scala program to receive full credit here, but try to get as close as possible. Make sure you create the appropriate (pair) RDDs or DataSets, and apply meaningful map, reduce, and/or aggregate functions to them.

Like other functional languages you are familiar with, Scala provides functions that transform a list of data objects into another. As a first step, find out the difference between RDD, pair RDD, DataSet, and DataFrame. Then try to work only with DataSets. (We will also accept solutions using RDD and pair RDD, but strongly encourage DataSets—that is where Spark is heading.) The pseudo-code for the first Spark program (mean min and max temperature for each station in a single year) is as follows:

1. Load the input data into a DataSet.
2. Use the appropriate mapping function (explore functions like map, flatMap, mapValues, flatMapValues) to create a data representation with columns (stationID, minTempValue, maxTempValue).
3. Group the data by stationID, computing the required temperature averages per group. For the average computation per group, explore functions such as aggregate, reduce, fold, groupByKey, reduceByKey, foldByKey, combineByKey, and aggregateByKey.

For the second problem (time series per stationID), find out how Spark Scala supports secondary sort. Hint: make sure to take a look at groupByKeyAndSortValues. Then implement program 2 (10-year time series per station) in Spark Scala.

Report

Write a brief report about your findings, using the following structure.

Header

This should provide information like class number, HW number, and your name.

Map-Reduce Algorithms (40 points)

For each of the three programs in part 1 above, write compact pseudo-code. Look at the online modules and your lecture notes for examples. Remember, pseudo-code captures the essence of the algorithm and avoids wordy syntax.

For program 2 above, also show the pseudo-code and briefly (in a few sentences) explain which records a Reduce function call will process and in which order they will appear in its input list. This should also be explained in comments in the source code.

Spark Scala Programs (16 points)

Show the Spark Scala programs you wrote for part 1 (mean min and max temperature for each station in a single year), omitting any imports. (6 points)

Discuss the choice of aggregate function for the first problem (see step 3 above). In particular, which Spark Scala function(s) implement(s) NoCombiner, Combiner, and InMapperComb; and why? (6 points)

Show the Spark Scala programs you wrote for part 2 (10-year time series per station), omitting any imports. (4 points)

Performance Comparison (24 points total)

Run all three MapReduce programs from 1 above in Elastic MapReduce (EMR) on the unzipped climate data from 1991.csv, using six m4.large machines (1 master, 5 workers).

Report the running time of each program execution. (Find out how to get the running time from a log file. It does not have to be down to a tenth of a second.) Repeat the time measurements one more time for each program, each time starting the program from scratch. Report all 3 programs * 2 independent runs = 6 running times you measured. (12 points)

Look at the syslog file. It tells you about the number of records and bytes moved around in the system. Try to find out what these numbers actually mean, focusing on interesting ones such as Map input records, Map output bytes, Combine input records, Reduce input records, Reduce input groups, Combine output records, Map output records. Based on this information, explain as much as you can the following, backing up your answer with facts/numbers from the log files: (4 points each)

- Was the Combiner called at all in program Combiner? Was it called more than once per Map task?
- Was the local aggregation effective in InMapperComb compared to NoCombiner?

Run the program from part 2 (secondary sort) above in Elastic MapReduce (EMR), using six m4.large machines (1 master, 5 workers). Report its running time. (4 points)

Deliverables

Submit the following in a **single zip file**:

1. The report as discussed above. (1 PDF file)
2. The syslog files for one successful run for each MapReduce program. (4 plain text files) (4 points)
3. All output files produced by that same one successful run for each MapReduce program on the full input on AWS (4 sets of part-r-… files) (4 points)

Make sure the following is easy to find in your **CCIS Github** repository:

4. The source code for each of the four MapReduce programs, including build scripts. (8 points)
5. The source code for the two Spark Scala programs, even if they do not run. (4 points)

IMPORTANT: Please ensure that your code is properly documented. In particular, there should be comments concisely explaining the role/purpose of a class. Similarly, if you use carefully selected keys or custom Partitioners, make sure you explain their purpose (what data will be co-located in a Reduce call; does input to a Reduce function have a certain order that is exploited by the function, etc.). But do not over-comment! For example, a line like “SUM += val” does not need a comment. As a rule of thumb, you want to add a brief comment for a block of code performing some non-trivial step of the computation. You also need to add a brief comment about the role of any major data structure you introduce.