CS 6240: Assignment 2

Goals: Compare Combiners to the in-mapper combining design pattern. Use keys, comparators, and Partitioner to implement secondary sort.

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 start with unzipping the original submission, 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.

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 <u>secondary sort</u> 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.) Memory-inefficient solutions will lose points.
   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 …)]
   …
   StationIdz, [(1880, MeanMinz0, MeanMaxz0), (1881, MeanMinz1, MeanMaxz1) … (1889 …)]

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 Algorithm (40 points)

For each of the three programs in 1 above, write compact <u>pseudo-code</u>. 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 <u>pseudo-code</u> and briefly (in a few sentences) <u>explain</u> 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.

Performance Comparison (32 points total)

Run all three 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?
- What difference did the use of a Combiner make in Combiner compared to NoCombiner?
- Was the local aggregation effective in InMapperComb compared to NoCombiner?
- Which one is better, Combiner or InMapperComb? Briefly justify your answer.
- How do the running times and accuracy of these MapReduce programs compare to the sequential implementation of per-station mean temperature? Modify, run, and time the *sequential* version of your HW1 program on the 1991.csv data. Make sure to change it to measure the *end-to-end* running time by including the time spent reading the file. Tip: Modify your code to read and process the data line by line (i.e., instead of reading it all into memory). Finally, compare the MapReduce output to the sequential program output to verify and report on its correctness.

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.

Deliverables

1. The report as discussed above. (1 PDF file)
2. The source code for each of the four programs, including build scripts. (12 points)
3. The syslog files for one successful run for each program. (4 plain text files) (8 points)
4. All output files produced by that same one successful run for each program (4 sets of part-r-… files) (8 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.