CS 6240 Study Guide for the Exam

You should be well-prepared since you studied the online modules and practiced your knowledge with the end-of-module quizzes, attended the lectures, did the homework where we practiced many of the concepts, and were reading along in the textbooks where needed. At this point, we recommend you browse through all modules again to see the "big picture" of what we covered and how material in different modules relates to each other. Then go over your own lecture notes and try to explain the most important material (ideally to somebody else). The learning objectives for each module tell you what you should definitely be able to explain. Expect questions of the type a more technical interviewer might ask you when you apply for a job in a team that develops MapReduce programs for large-scale data analytics:

1. Questions similar to the check-your-knowledge quizzes, but more emphasis on writing MapReduce programs. Instead of just yes/no answers, you will usually have to write brief justifications.
2. What are Mapper, Reducer, Combiner, and Partitioner? Give examples for each.
3. Given a small MapReduce program and some data, explain how the data is processed and derive the final output. Practice by picking some of the programs we discussed and running them manually (on paper) on some small input.
4. Write a MapReduce program for a given problem. This is very important. Expect problems of the size and complexity like the ones we discussed in class. Write in pseudo-code (like in the modules), not detailed Java code. A good way to prepare is to take problems we discussed (or a new one) and try to write the MapReduce pseudo-code without looking at the solution.
5. Know the MapReduce design patterns and basic algorithms and their tradeoffs, e.g., how does in-mapper aggregation improve performance and what are its disadvantages? How does order inversion work and what does it achieve? Prepare by solving the problems we discussed in class without looking at the solution.
6. Understand important non-trivial algorithms, e.g., sorting, equi-join, PageRank, matrix product. Walk through Map and Reduce calls for small data samples.
7. Know the basics of the MapReduce system implementation. What are the roles of master and workers? How does data get to Mappers and from Mappers to Reducers? What are Mappers working on? Where do Reducers write their output? What happens if any machine crashes?
8. Given a new problem scenario, argue why (not) MapReduce would be a good match for it. To do this, you should understand its basic properties, e.g., compared to relational databases.
9. You do not need to know details about HDFS, but you should be aware of its general functionality.

Specific suggestions for writing pseudo-code

- Keep function definitions compact. E.g., instead of *public void map(IntWritable key, Text value, Context context)*, just write *map(matrix cell c)*.

- Don't worry about details like writing a complete string parser just to access the fields of a record. E.g., instead of *String[] index = value.toString().split(",")* etc., just refer to the different fields of a record by their name: c.row, c.column, c.value.
- Use *for all records r in file/list/etc* (or *for each*—same thing) when going through all records in a list or file. This is easier to read than statements like *while (tokenizer.hasMoreTokens())*.

Example question and answer

Problem:

We are given a large input file (word1, word2, word3,…) containing words. Write the best possible MapReduce program (pseudo-code) you can think of to find the word with the most occurrences of letter "y". (If there are multiple such words, output at least one of them.) You do not need to write any text parsers. In a couple of sentences, explain the main idea and design decisions of your program.

Example solution:

Brief summary of solution idea: A Map task receives a split of the file, usually containing many words. We can find the word with most "y" in such a split by using in-mapper combining. All Mappers then send their "local winner" to a single Reduce call, which determines the global winner. Instead, we could also globally sort the words by their number of y's, but this would move a lot more data from Mappers to Reducers.

```
Class Mapper {

   localWinner = (NULL, 0)  // localWinner is a pair (word, count), keeping track of the word with most y's
in the task

   Map( …, word w) {
      if (localWinner = NULL or yCount(w) > localWinner.count)
         localWinner = (w, yCount(w))
   }

   Cleanup()
      emit( dummy, localWinner.word )

}

// There is only a single Reduce call: for key=dummy
Reduce( dummy, [ word1, word2,…]) {
   globalWinner = (word1, yCount(word1))

   for each word w in input list do
      if (yCount(w) > globalWinner.count)
         globalWinner.set( w, yCount(w) )
```

```
    emit( globalWinner.word )
}
```

Possible improvement: We can use the secondary sort design pattern to avoid having to go through the entire Reduce input list. To implement it, we make localWinner.count part of the key in the Map output. (Notice that this increases the amount of data sent from Mappers to Reducers due to the additional count field, but overall the amount of data is still very small because only a single record per Mapper is emitted.) We also need to add a custom Partitioner that partitions on the dummy field only, ignoring the counter field of the key. Similarly, we have to add a grouping comparator that ignores the counter field of the key.

For the Reduce function to receive the global winner as the first input record, we need to define a regular key comparator that sorts on the dummy field first (not really needed here because all keys have the same dummy value), then breaks ties by sorting in decreasing order of the count field.

With secondary sort, the Reduce function simply emits the first input record.