Data Intensive Computing – Review Questions 2

Adriano Mundo, Riccardo Colella (Group 29)

1. **Imagine you are working in a big company, and your company is planning to launch the next big Blogging platform. Tomorrow morning you go to your office and see the following mail from your CEO regarding a new work. How do you answer to this email? Hint: use MapReduce to solve it, and explain what you do in Map and Reduce phases.**

```
Dear <Your Name>,
As you know we are building a blogging platform, and I need some statistics.
I need to find out, across all blogs ever written on our blogging system, how
many times 1 character words occur (e.g., "a", "I"), How many times two
character words occur (e.g., "be", "is"), and so on. I know its a really big
job. I am going on a vacation for one week, and its really important that I've
this when I return. Good luck.
Regards,
The CEO
```

Dear CEO, to obtain the statistics you need and solve the problem, all the texts of our blogging systems should become the input of a MapReduce process for word counting.
The algorithm needed is the following one:

- *Mapping:* starting from the input files, we transform it in the right input format for the map function. Hence, we can read the text line by line and obtain (key, value) pairs for the input having stored in the value the content of a single line. Once that, we have as output intermediate key-value pairs where the key is the number of characters in each word *{(3, "CEO"), (4, "blog"), (4, "hello")}*.
  Then, in the shuffle phase the pairs just created are grouped together according to their key values and passed to next phase *{(3, ("CEO")), (4, ("blog", "hello")}*.

- *Reduce*: at this stage we can simply runs the reduce function to count the number of values of each key, so we are counting how many words have the same number of characters.

In this way we obtain the frequency for every word length required.

2. **What is the problem of skewed data in MapReduce? (when the key distribution is skewed)?**

The problem is the skew in the distribution of value associated with intermediate keys which leads to resource stragglers. It often happens that the task or tasks responsible for processing the most frequent few elements will run much longer than the typical task. To better explain, in the case of a word counter a reducer that's responsible for computing the count of a frequently occurring word will have a lot more work than the typical reducer and therefore will likely be a straggler. This means that the load on the reducers becomes unbalanced and the job time longer. A better local aggregation is a possible solution to the problem.

3. **Briefly explain the differences between Map-side join and Reduce-side join in Map-Reduce?**

Joins are constructs used to combine relations together based on foreign keys. They exist and can be used in MapReduce in situations where you have two or more datasets to combine. There are two types of join: reduce-side and map-side join.

Reduce-side join, which is also called repartition join, is usually used to join two or more large datasets together. The map works over both datasets and emit the join key as the intermediate key, and the tuple itself as the intermediate value. Because MapReduce guarantees that all values with the same key are brought together, all tuples will be grouped by the join key, that's what we need to perform the join. The approach isn't particularly efficient since it requires shuffling both datasets across the network. But, on the other side, it's flexible to implement because it does not need structured data and it's easier to implement.

Map-side join, which is also called replication join, is usually used when one dataset is small because it is adequate only when one of the tables fit into the memory; hence, it is not suitable for operations with huge amount of data. All the tasks are performed by the mapper before data are consumed by the map-function. In practice, the map works over one of the datasets (the larger one) and inside the mapper read the corresponding part of the other dataset to perform merge join. No reducer is required unless the developer specify it and it is far more efficient than a reduce-join since there's no need to shuffle the datasets over the network.

4. **Explain briefly why the following code does not work correctly on a cluster of computers. How can we fix it?**

```
val uni = sc.parallelize(Seq(("SICS", 1), ("KTH", 2)))
uni.foreach(println)
```

The problem with this piece of code arises when we have a cluster of computers since when the function *foreach(println)* on one computer the output has no particular problems. With a cluster of computers, the behavior of the code is the following: the output of the function is executed by the executor on every cluster and there's no result on the driver. Hence it won't show what is expected.
To avoid the problem data must be collected before using the foreach statement; in this way data is collected, RDD is on driver mode and the results is the expected one.

```
val uni = sc.parallelize(Seq(("SICS", 1), ("KTH", 2)))
uni.collect().foreach(println)
```

5. **Assume you are reading the file campus.txt from HDFS with the following format:**

```
SICS CSL
KTH CSC
UCL NET
SICS DNA
. . .
```

Draw the lineage graph for the following code and explain how Spark uses the lineage graph to handle failures.

```
val file = sc.textFile("hdfs://campus.txt")
val pairs = file.map(x => (x.split(" ")(0), x.split(" ")(1)))
val groups = pairs.groupByKey()
val uni = sc.parallelize(Seq(("SICS", 1), ("KTH", 2)))
val joins = groups.join(uni)
val sics = joins.filter(x => x.contins("SICS"))
val list = sics.map(x => x._2)
vali result = list.count
```

The image below represents the lineage graph for the previous piece of code, which represents the computations done on the RDD. Anyway, lineages are the key for handling fault tolerance since Spark records each transformation in its lineage and DataFrames are immutable between transformations. Hence, it can reproduce its original state by simply recomputing only the lost partitions of an RDD, giving resiliency in the event of failures. In fact, if one of the partitions fails, only the data needed for that partition have to be recomputed to get back on track.