ID2221 Data Intensive Computing - Review Questions 2 - Group 19

by Artem Sliusarenko, Mihailo Cvetkovic and Eugen Lucchiari Hartz

1.) Briefly explain how you can use MapReduce to compute the multiplication of a M ×N matrix and N ×1. You don't need to write any code and just explain what the map and reduce functions should do.

Let us consider a matrix multiplication with M x N and N x 1 to visualize MapReduce.
For the map phase, each element of the matrix A and the vector v is provided as input to the mappers. Each mapper takes an element from matrix A and pairs it with the corresponding element from vector v. The mapper emits a key-value pair where the key is the row index i, and the value is the partial product.

For the reducer phase, each reducer receives all the partial products corresponding to a specific row i of the matrix A. The reducer sums all the partial products for the corresponding row. The final output will be the M×1 result vector u. This way, each mapper handles the multiplication, and the reducers handle the summing to get the result vector.

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

In MapReduce, keys are essential for partitioning and grouping data. When the distribution of keys is skewed which means that a few keys dominate the dataset, it can disrupt the balance of work distribution and affect the parallelism that MapReduce relies on. This skew can lead to load imbalance, where some reducers are assigned far more work than others which causes resource contention and slows down the overall process.

When a small number of keys have a disproportionately large number of values, the reducers handling these keys end up with much more data to process compared to others. This leads to load imbalance, where some reducers finish quickly, while others are overloaded which causes delays. As a result, the overall parallelism of the job is reduced, as the entire job can only complete once the slowest reducer finishes its work. This imbalance also causes a lack of utilisation of cluster resources because many reducers remain idle, waiting for the overloaded reducers to finish.

Additionally, this load imbalance can lead to resource contention. The reducers handling the skewed keys may consume excessive memory and CPU resources, further exacerbating performance issues and delaying the job's completion.

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

Map-side join and Reduce-side join are two techniques used in MapReduce to combine data from multiple datasets, but they are suitable for different scenarios based on data size and system resources.

Map-side join is more efficient when one of the datasets can fit comfortably in memory. In this approach, the data is pre-partitioned and sorted by a common key before the map phase. Each Mapper processes a portion of the data and performs the join directly during the map phase which eliminates the need for a Reducer. This approach reduces the overhead of data shuffling and minimizes network transfers which makes it faster and more efficient when the datasets are small enough.

On the other hand, Reduce-side join is designed for scalability and is more effective when both datasets are too large to fit into memory. In this method, the datasets are processed separately during the mapping phase and the key-value pairs are shuffled and grouped by key before being sent to the Reducer. The actual join operation is performed during the reduce phase where records with the same key from different datasets are combined. This approach is preferred for joining large datasets as it avoids the need to load entire datasets into memory at once.

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 println is executed on worker nodes, but the driver does not collect the results from workers for display. This means the output will not be visible in the driver console where you are likely expecting to see it. Since the foreach action only executes on the workers, it does not collect or aggregate the results back to the driver node, and the output is lost unless you look at the logs on individual worker nodes.

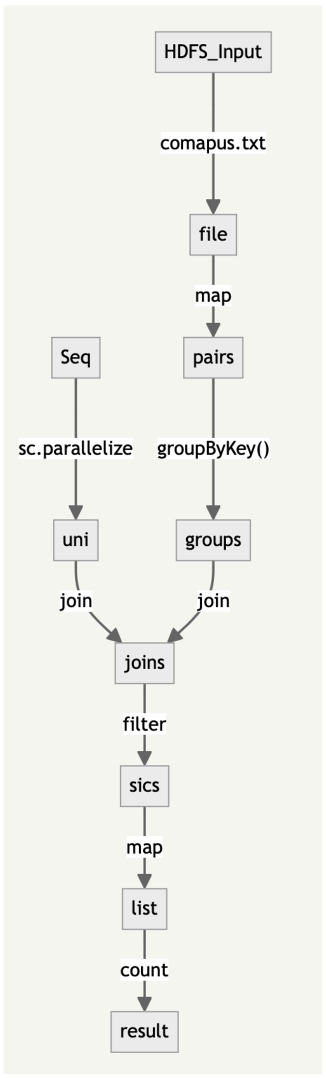
In order to fix it, you can use collect() which collects the data back to the driver before printing. collect() gathers the data from all worker nodes and brings it to the driver, where it can then be printed. Here, collect() retrieves the RDD to the driver and prints the results there, so you will see the output in the driver's console.

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
```



A lineage graph in Spark is a Directed Acyclic Graph (DAG) that represents the sequence of transformations applied to an RDD. It shows how each RDD is derived from previous RDDs through a series of transformations. Spark uses this lineage graph to handle failures by tracking the transformations that led to the current state of the data. If a partition of an RDD is lost due to a failure, Spark can use the lineage graph to recompute only the lost data by replaying the necessary transformations from the original data, instead of recomputing the entire dataset. This makes Spark fault-tolerant and efficient in handling large-scale data processing.