ID2221 - Data Intensive Computing - Review Questions 2

Deepak Shankar Reethika Ambatipudi
deepak | reethika @kth.se

September 15, 2023

Q1. Briefly explain how you can use MapReduce to compute the multiplication of a M $\times$N matrix and N $\times$1. You don't need to write any code and just explain what the map and reduce functions should do.

Map function should break the matrix into smaller pieces (rows) and the vector into individual components, Calculate the multiplication result for each element in a row of the matrix and its corresponding element in the vector and label each of these results with the corresponding column (or row) number in the final result.

Shuffling part is taken care by the framework.

Reduce function groups the labeled results together based on their column (or row) numbers. For each group, it adds up all the labeled results. This gives us the final answer for each column (or row) in the result.

In simpler terms, map function divides the matrix and vector into smaller parts, doing some math on those parts, and then reduce function combines the results to get the answer.

Q2. What is the problem of skewed data in MapReduce? (when the key distribution is skewed)

Skewed data reduces the efficiency of MapReduce since it might lead to Load imbalance, Reducer Parallelism or unequal Reducer which can lead to Task Failures. It might also lead to Resource bottlenecks in some nodes which can delay the task execution or even fail.

To mitigate the problems of skewed data in MapReduce, various techniques and strategies can be employed, such as data skew detection, key redistribution, custom partitioning schemes, and combiners. These approaches help distribute

the workload more evenly and improve the overall performance and efficiency of MapReduce jobs when dealing with skewed data distributions.

Q3. 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 for combining data from multiple sources, such as two or more datasets, based on a common key. The choice between Map-side join and Reduce-side join depends on the characteristics of the data, the size of the datasets, and the specific requirements of MapReduce job. Each approach has its advantages and trade-offs, and the selection should be based on the nature of the data and the join operation we need to perform.

Map-side Join - In a Map-side join, the data from multiple sources (e.g., two datasets) is combined and processed in the map phase of MapReduce. This means that the join operation is performed before the data reaches the reduce phase.

This works well when both datasets are partitioned or sorted in the same way based on the join key. This is typically more suitable when the input datasets are relatively small and can comfortably fit in memory, memory limitations can be a concern for larger datasets. Since the join operation occurs within the map phase, network traffic is reduced.

Reduce-side Join - In a Reduce-side join, the data from multiple sources is processed separately in the map phase and then brought together and joined during the reduce phase. The actual join operation occurs in the reduce phase. This is used when joining two or more large datasets with different partitioning or sorting schemes, where the join operation is complex and resource-intensive.

This is more flexible in terms of data distribution. The datasets can have different partitioning or sorting schemes, and the join is performed during the reduce phase by matching keys from the sorted and grouped input. These are better suited for larger datasets because the join operation is performed during the reduce phase, which can handle larger amounts of data. However, it may require more disk I/O as intermediate data is written and read between map and reduce phases. This may involve more network traffic as intermediate data is shuffled between map and reduce phases. However, this approach allows for greater flexibility in handling different data distributions.

Q4. 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 provided code attempts to parallelize a sequence of tuples using Apache Spark's 'parallelize' method and then calls 'foreach' to print each element of the RDD (Resilient Distributed Dataset) on the cluster. However, this code may not work correctly on a cluster of computers because Spark's 'foreach' operation is not guaranteed to run the provided function on the driver node. Instead, it may execute the function on the worker nodes in a distributed manner. In this case, the 'println' function would execute on the worker nodes, and the output would not be visible in the driver node's console.

To fix this and ensure that the 'println' function is executed on the driver node, you can collect the RDD back to the driver and then apply 'foreach' on the collected data. Here's the modified code:

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

In this updated code, 'collect()' retrieves all the data from the RDD and brings it back to the driver node as an array, and then 'foreach(println)' is applied to the array, ensuring that the output is printed on the driver node's console.

Q5. 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.?

Handling of failures: If any partition of an RDD (e.g., groups) is lost due to a node failure or any other reason, Spark can recover that partition by recomputing it from the original data (in this case, from the "pairs") and applying the same sequence of transformations recorded in the lineage graph. Spark keeps track of these dependencies in the lineage graph, allowing it to recompute only the lost partitions efficiently, minimizing data loss and ensuring fault tolerance.

So, if a failure occurs at any point in the execution, Spark can use the lineage information to recompute the necessary partitions and continue the computation from the point of failure.

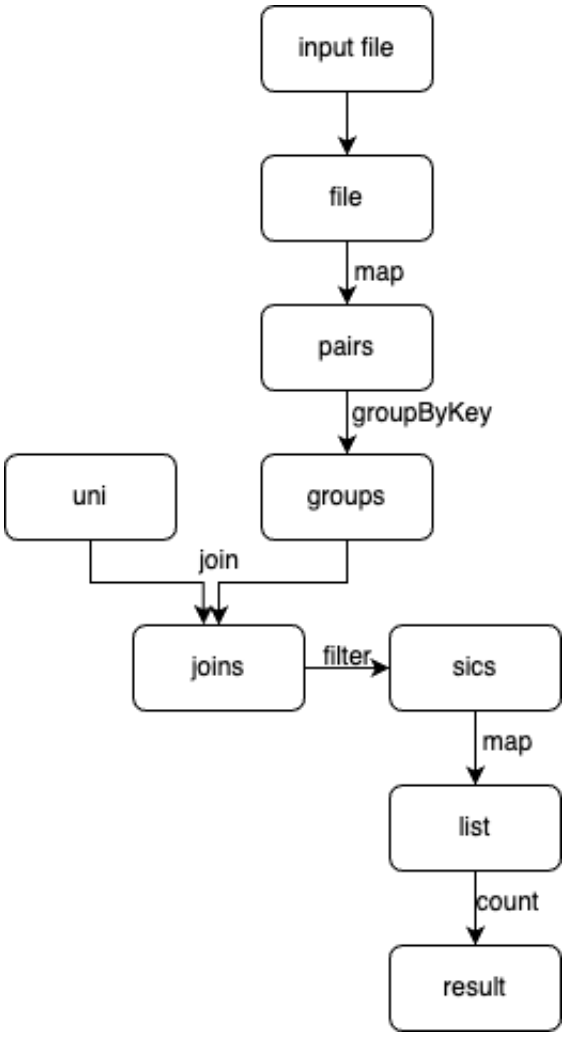

Figure 1: lineage graph - version 1

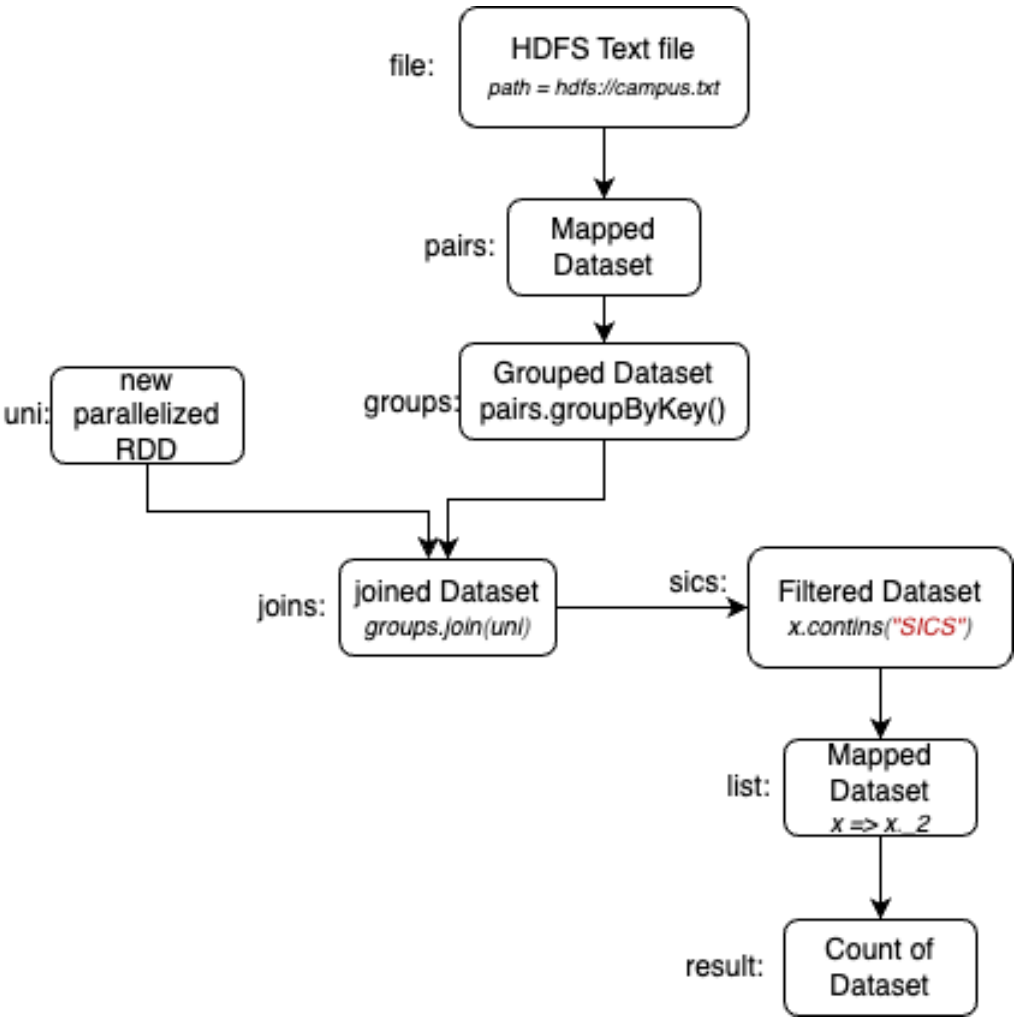

Figure 2: lineage graph - version 2