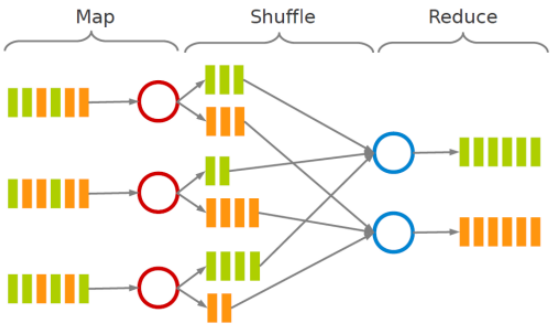
Group: Group Therapy
Casper Kristiansson
Nicole Wijkman

RQ 2: Data-Intensive Computing

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.

MapReuced is an architecture/technique where the goal is to provide a better solution to process big data sets using parallel and distributed algorithms. The general approach of MapReduce is that it divides a task into smaller tasks and runs those tasks in parallel. There are two main parts of a MapReducer, the mapper which handles the input data, and the reducer which handles the processing of the data. The first task of the problem is to find a good way to represent the matrixes which could be done by using a three-depth mapping (i, j, value) [1]. Note that the N x 1 matrix can be represented using only (j, value). We then with the help of the mapper (mapping function) map key-value pairs. For this specific problem mapping the matrix A (M x N) using column is the best way meaning (j: (i, value)).

The next step is the shuffle and sort where all values with the key row will be grouped by j. The next step of the MapReduce is the reduce part. Each reducer will receive a key for j which contains a list of pairs (i, value) from matrix A and the value from matrix B. The reducer needs to calculate the product of them and then emit the result. The result will then be compiled as the final matrix.



Slide Page 22 -
https://id2221kth.github.io/slides/2023/05_parallel_processing_part1.pdf

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

In MapReduce, keys play a central role in data partitioning and grouping. Having a few keys dominate the dataset can disrupt the natural parallelism and distribution of work. This can lead to *load imbalance*, and then in turn *reduced parallelism* as well as *resource contention*.

If a few keys have a disproportionally large number of values associated with them, *load imbalance* can occur. This is because the MapReduce framework assigns more work to the reducers that handle the affected keys, which in turn results in some reducers having a lot more data to process than others. A load imbalance among reducer tasks can cause delays and will reduce the overall job efficiency.

The load imbalance will in turn limit the achievable parallelism in a MapReduce job, leading to *reduced parallelism*. Due to the load imbalance causing delays with the reducers processing skewed keys taking longer, there is a lack of utilization of cluster resources. This is because the reducers must wait for the slowest reducers to finish, which will increase the execution time and reduce the parallelism.

Due to the load balance, there can also occur *resource contention* with the small number of reducers handling a large amount of data potentially consuming a lot of memory and CPU resources in the process.

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 different techniques, both of which are used in MapReduce to combine data from multiple datasets. However, they are efficient for different scenarios.

Map-side join focuses on efficiency and is therefore efficient in scenarios where one dataset can comfortably fit in memory. When using Map-side join, data from multiple datasets are pre-partitioned and sorted based on a common key before entering the mapping phase. Each Mapper processes a partition of data and can perform the join operation without the need for a Reducer, which in turn eliminates the overhead of data shuffling, reducing network transfers.

Reduce-side join is on the other hand designed for scalability and is therefore more efficient in situations where both datasets are too large to simultaneously fit in memory. When using Reduce-side join, data from multiple datasets are instead processed separately in the mapping phase and key-value pairs are shuffled and grouped by key. The actual join operation takes place in the Reducer phase, where records with the same key from different datasets are combined. It is the preferred method for joining large datasets, as it does not require loading entire datasets into memory.

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 provided code should work correctly in a standalone Spark environment, or on a local machine. However, it will not work as expected on a cluster of computers because of a few problems.

Firstly, creating an RDD using parallelize without any specification on the number of partitions will result in the default number of partitions being used. This could, in a cluster environment, not be optimal for efficient parallel processing and in turn, lead to bad performance. To fix this problem, all one needs to do is specify the number of partitions.

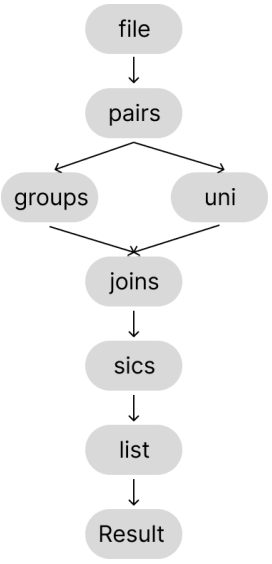
```
val uni = sc.parallize(Seq((“SICS”, 1), (“KTH”, 2)), numSlices = 4)
```

Secondly, the foreach-action will print the output locally on each worker node in the distributed cluster, leading to the output not being seen in a centralized location. To solve this problem, one could save the RDD’s content to a file that is stored in a location that is accessible to all cluster nodes, instead of using “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
```

A job can be described based on directed acyclic graphs (DAG) data flow. The represented computations by DAG that have been done on the RDD are what we call a lineage graph. A lineage graph displaying the above instructions can be shown as.



Spark uses a lineage graph to handle failures where it uses it to recover lost data without needing to re-compute the entire dataset (job). If a failure has happened for example in groups spark will check the lineage graph and see that groups are calculated based on the pairs and that it then only needs to recompute those partitions (RDD is divided into smaller chunks called partitions).

References

[1] https://www.geeksforgeeks.org/matrix-multiplication-with-1-mapreduce-step/