# Data Intensive Computing - Review Questions 2

Sri & Nik - 15th Sept, 2019

## Question 1: Explain how to do word count filtered on amount of characters with MapReduce:

The Map part will take a set of blog posts, and by looping over each word, create a list of tuples in the form (number of characters, 1).

The Reduce part will sum these (number of characters, 1) tuples to a final list of counts for each number of characters.

## Question 2: Explain the difference between map-side join and reduce-side join:

Reduced-side join, the commonly used join has a map, shuffle and reduce stages. The map gives the key-value pair for joining the 2 tables. The shuffle stage takes these pairs and then sort and merge them. The reducer then finished the task to join them to form a new table.
In contrast, the map-side join is performed before data is passed through the map and thus work is performed entirely by the mapper without the shuffle and reduce phase. Thus faster and efficient. But this is ideal only if one of the tables is small enough to fit into the memory. To perform map-side join each input should be partitioned into the same number of partitions and sorted with the same key.

## Question 3: Why does the following code not work on a cluster of computers?

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

You should first collect data before you use it.
It works on a single computer as the data is only in once place and hence collect() is not required.

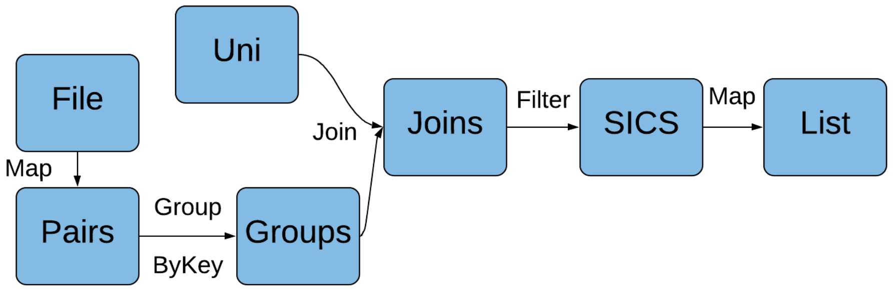
```
sc.parallelize(Seq(("SICS", 1), ("KTH", 2)))
uni.collect().foreach(println)
```

## Question 4: Draw the lineage graph and explain how spark uses it to handle failure.

Corrected Code

```
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(_._1.contains("SICS"))
val list = sics.map(x => x._2)
val result = list.count
      file = campus.txt MapPartitionsRDD[1] at textFile at <console>:27
      pairs = MapPartitionsRDD[2] at map at <console>:28
      groups = ShuffledRDD[3] at groupByKey at <console>:29
      uni = ParallelCollectionRDD[4] at parallelize at <console>:30
      joins = MapPartitionsRDD[7] at join at <console>:31
      sics = MapPartitionsRDD[8] at filter at <console>:32
      list = MapPartitionsRDD[9] at map at <console>:33
      result = 1
```

```
list.toDebugString
(2) MapPartitionsRDD[9] at map at <console>:33 []
 |  MapPartitionsRDD[8] at filter at <console>:32 []
 |  MapPartitionsRDD[7] at join at <console>:31 []
 |  MapPartitionsRDD[6] at join at <console>:31 []
 |  CoGroupedRDD[5] at join at <console>:31 []
 |  ShuffledRDD[3] at groupByKey at <console>:29 []
 +-(2) MapPartitionsRDD[2] at map at <console>:28 []
    |  campus.txt MapPartitionsRDD[1] at textFile at <console>:27 []
    |  campus.txt HadoopRDD[0] at textFile at <console>:27 []
 +-(4) ParallelCollectionRDD[4] at parallelize at <console>:30 []
```



You can also see the top-down order of the RDDs in the todebugString output of list RDD.

Spark uses the lineage graph which has all the dependencies between the RDDs stored and as spark RDDs are immutable, in the event of fault occurrence the Spark looks up this lineage graph and resumes the remaining tasks without having to start from scratch.