Data Intensive Computing - Review Questions 2

Deadline: Sep. 13, 2025

Group AKA
Ahmad Al Khateeb
Aleksandra Burdakova
Kusumastuti Cahyaningrum

1. **What are the three main stages of a MapReduce job? Which stage is usually the bottleneck, and why?**
   3 main stages:
   a. Map: input splits are processed (by a mapper) into intermediate key-value pairs, that are stored locally, and data is partitioned.
   b. Shuffle: Moves intermediate data across the network to reducers, after sorting and grouping them by key (so that all values with the same key end up in the same cluster).
   c. Reduce: Applies the reduce function to grouped keys and writes final output.

   Bottleneck: Shuffle.

   The Shuffle stage is the bottleneck mainly because of how intermediate data is handled:
   - Disk I/O: Map outputs are first buffered in memory, then spilled to local disk. Reduce workers must fetch these files from many map workers' disks. Disk reads/writes are much slower than in-memory operations, especially when intermediate data is huge.
   - Sorting: sorting a large amount of key-value pairs can be expensive in time.

2. **What is the problem of skewed data in MapReduce? (when the key distribution is skewed)**
   When some keys are more frequent than the others, these keys are called skewed keys in MapReduce. The problem with skewed data is that the data distribution across the reducers becomes imbalanced. In other words, certain reducers end up receiving a much larger volume of data than other reducers. Additionally, it leads to an increased job execution time, since the overall job must wait for the slowest reducer to finish.

   In terms of 'critical' aspect: skewed data in MapReduce will be more pronounced, such that the majority of data will be processed first and the minority data will have to wait until the majority data process finishes (assuming majority data is ordered before the minority data).

3. **Why are iterative algorithms like PageRank inefficient in MapReduce, and how does Spark improve on this?**
   Iterative algorithms like PageRank are inefficient in MapReduce because each iteration writes results to disk and re-reads them, causing heavy disk I/O, repeated shuffles, and

job startup overhead. MapReduce also lacks in-memory data sharing, so data is always reloaded. Spark improves this by keeping data in memory across iterations (via RDDs), reducing disk, and making iterative algorithms much faster.

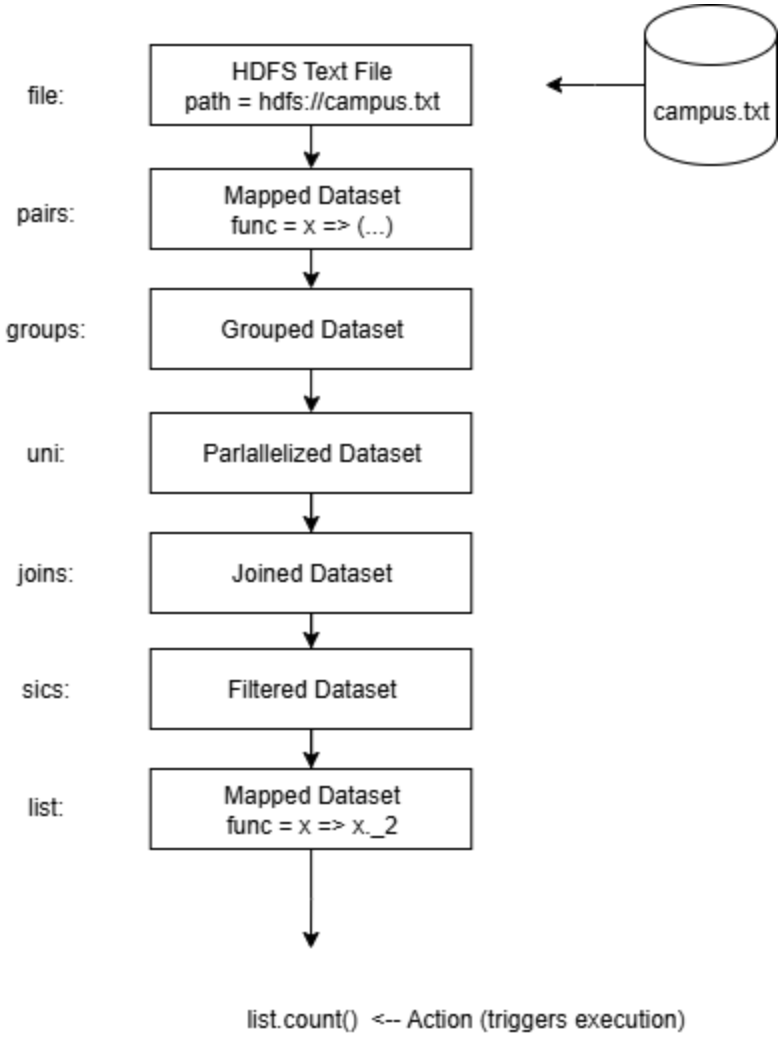
4. **Spark is often praised for its speed compared to MapReduce. What technical feature gives Spark this advantage, and what trade-off does it introduce?**
   Spark introduces RDDs which keep intermediate outputs in memory instead of writing them to disk after each step.

   Trade-off: RDDs are immutable. Re-creating new RDDs for updates or transformations can consume more memory and storage. Very long lineage chains may need checkpointing, adding some overhead.

5. **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 function foreach executes on the worker nodes in the cluster. However, println writes output to each worker’s local stdout, not to the driver program, so on a real cluster the output would not be printed on the terminal because each worker prints to its own log file. To fix it, we need for example to collect the data using uni.collect (in that case we will have an array), and then print it out in the terminal.

   RDDs cannot be printed -> **uni.foreach(println)** is invalid
6. **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)
   val result = list.count
   ```



Explanation of Lineage:
Spark records this DAG (Directed Acyclic Graph) of transformations instead of materializing results at each step (lazy operation). Each RDD knows how it was derived from parent RDDs (e.g., pairs → groups → joins).
If a partition is lost due to a node failure, Spark does not recompute everything. It looks at the lineage: only the missing partition of the affected RDD is recomputed from its parent(s).
For example, if a partition of joins is lost, Spark re-runs the join on the corresponding partition(s) of groups and uni, not the entire pipeline.

7. **Both MapReduce and Spark were designed for speed and scale in corporate settings. From a Data Feminism perspective, whose needs to do these systems serve, and whose knowledge or contexts might be excluded by their design?**
The main purpose MapReduce and Spark were designed for is to handle massive data processing at scale and supporting corporate and scientific use cases, while optimizing for speed, performance, and scalability. These systems primarily serve large

corporations (e.g., Google, Facebook, Amazon), governments and research labs, and highly technical users. The systems may, from a feminist perspective, exclude or marginalize culturally contextual knowledge since “big data” often ignores small and localized data that can’t be easily aggregated or parallelized. Additionally, non-technical or community-based actors may also be excluded.