LAB 10: MapReduce + Spark

1. Motivation: Nowadays, an enormous amount of data is gathered, stored, and processed every day. Such data that is very large, difficult to analyse because of its volume, variety, and velocity, but at the same time is valuable in terms of data analysis, is referred to as **Big Data**. How to store and process very large amount of data efficiently is probably one of the most important questions of computer science today. The following examples give a hint about the numbers:

- Google: processing 10 terabytes (TB) of data daily in 2004 while in 2008 about 20 petabytes (PB) and in 2009 about 24PB. In 2017 about 10-15 exabytes (EB) of data but this is only a rough estimation.
- eBay: 6.5PB of users' data and 150 million new users daily (2009).
- Facebook: 2.5PB of data in 2009 while in 2014 about 300PB with 600TB of yearly increase. In January 2013 Facebook users uploaded around 240 billion photos.
- LHC (Large Hadron Collider): in 2007 LHC passed a milestone of having about 200PT of data.
- Large Synoptic Survey Telescope (3200 mega pixels): 15TB of data every night.
- EBI (European Bioinformatics Institute (EBI): in 2015 the total disc capacity of EMBL-EBI was around 75 PB.

b) How to process such amount of data efficiently? Distributed architecture and parallel computing is obviously an answer but we want to:

- maintain **really** large amount of data,
- and process it efficiently.

Traditional distributed systems are not designed to handle problems of this scale. The following list consists of several characteristics of a distributed system and makes a comparison of traditional (we do not want to) and devoted to process Big Data (we want to) systems:

Hardware	We do not want to: use very expensive and sophisticated hardware because it is still prone to failures. E.g., imagine a cluster consisting of 10000 good machines, each having mean time between failures (MTBF) of about 1000 days. Still it is, on average, about 10 failures per day. The components must be repaired/replaced anyway.
	We want to: use relatively inexpensive, widely available, and interchangeable components (commodity hardware).
Architecture	We do not want to: use traditional distributed architecture with computers (nodes) having their own local memory and passing messages in order to synchronise computing and transfer data.
	We want to: move processing to the data instead of transferring large amount of data to different nodes. Tasks should be executed on these processors that have direct access to disks having required data (data locality).

Data access	We do not want to: use random (direct) memory.
	We want to: use sequential memory which read/write speed is much faster but, obviously, records must be read one by one and a particular record cannot be accessed directly. However, in such systems data is usually organised sequentially and rewriting a large part of memory instead of accessing them directly is in that case faster.
How vs. What	We do not want to: focus on how the system works, e.g., handling parallel memory access.
	We want to: focus on the task (what we want to do) instead of details of implementation.
Parallelization	We do not want to: divide resources to solve a single task.
	We want to: divide tasks, use *divide and conquer* approach.

Considering the above characteristics, several questions arises:

- How to divide a program into smaller tasks such that they may be run in parallel?
- How to assign these tasks into machines (processors) such that the program is executed efficiently?
- How to make required data available for tasks to obtain?
- How to synchronize tasks?
- How to transfer data from one machine/task to another?
- How to handle hardware and software failures?

2. **Answer: MapReduce (+ Apache Hadoop MapReduce).** MapReduce is a simple programming model designed for processing Big Data in parallel on multiple nodes. The main idea of MapReduce is to hide implementation details of parallel execution and let user to focus on implementation of data processing strategy. What is more, MapReduce model scales linearly with the amount of data and is fault tolerant. Obviously, since it is impossible to avoid data transfer between different units, implementation of processing strategy that minimizes communication is essential to benefit from MapReduce.

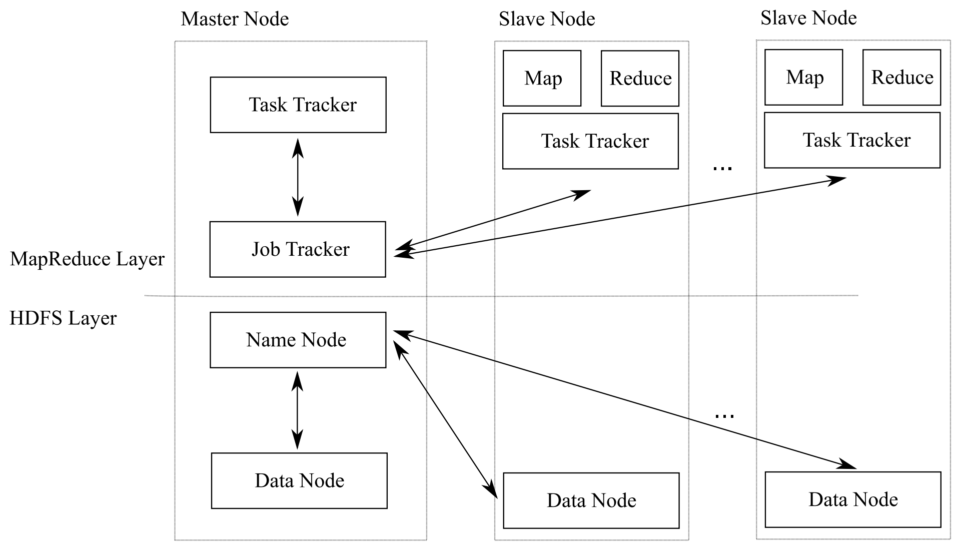

Figure 1: MapReduce architecture.

Figure 1 depicts architecture of MapReduce program. It consists of the following components:

- **Name Node + Data Node**. Makes a HDFS (Hadoop Distributed Files System), store data distributed among computers (cluster).
- **Name Node**. Metadata of HDFS (list of files, references to computers Data Nodes).
- **Data Node**. Stores data files
- **Job Tracker**. It is responsible for:
    - **Scheduling**: job is divided into smaller tasks that are to be run in parallel.
    - **Data/code co-location**: not data distribution! Task is assigned to a machine that has required data.
    - **Synchronisation**: MapReduce program consists of steps that have to be executed one by one (**map** and **reduce)**. Next step may be performed only if all required data is computed.
    - **Error and fault handling**: When an error occurs (e.g., because a hard drive is broken), JobTracker detects is and redistributes tasks among available (working) nodes.
- **Task Tracker**. Run tasks (map, reduce) and communicate with Job Tracker in order to report a status.

Considering a well-known MapReduce implementation **Apache Hadoop MapReduce**, all the above-mentioned elements except map and reduce steps are handled/maintained/ performed by the framework by default. When writing a program, except providing a configuration, the user is usually expect to write **map** and **reduce** procedures which perform transformation of given chunks of data. All other system processes such as scheduling or fault handling are done without user knowledge.

<u>How does MapReduce program look like? An example:</u> Figure 1 depicts an exemplary task where the problem is to count the number of words occurrences in a collection of 3 documents. Please note, that this Figure depicts only a single strategy and the problem could be solved in another way (e.g., by implementing different map and reduce functions). The steps of the depicted process (Figure 2) are as follows:

a) Data (here, only 3 short documents) is distributed among a cluster of computers using Distributed Files System (DFS), e.g., HDFS (Hadoop).

b) Next, data has to be split over available nodes. This step is not about transferring data but about how to run tasks (Task Tracker) on different nodes such that each task has all required data. In the provided example each task is given a single document. If each document is on different node, no data transfer is needed. However, in case when, e.g., a document data is distributed between two nodes, some data has to be transferred.

c) **<u>Map</u>:** One of the two the most important steps of MapReduce program (MapReduce = Map + Reduce). It is usually responsible for data preparation. This functionality (map function) has to be implemented by the user. TaskTracker (mapper) is given a task do perform. It works on a single node and executes a **map** method on **local data**. Using only **local** data means that this step is strongly parallelized. Having multiple nodes, each mapper performs its independently of others. One mapper may e.g., process a single file, small collection of files starting on letter ‘a’, and so on. Exemplary task may be, e.g., to tokenize a document or to count the number of words occurrences in a collection of documents. In Figure 2 the latter is depicted. Each mapper loads locally stored document. Map function processes data and, as well as other steps of MapReduce, is expected to output pairs in a form of (key, value). In this example, mapper processes a document (tokenize, normalise, and so on) and for each single occurrence of a term it returns a pair (term, 1). This temporary data is written to the local disc.

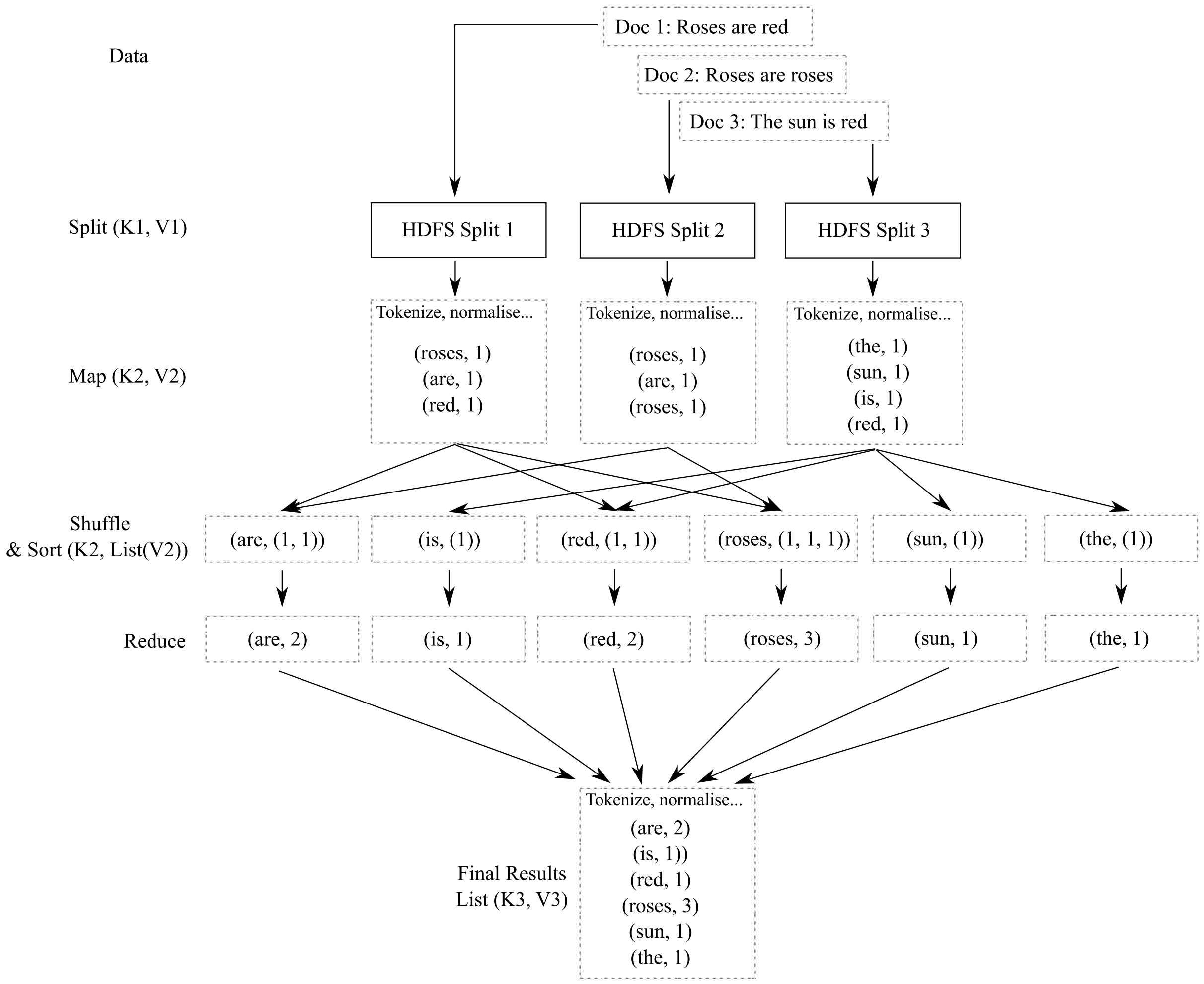

Figure 2: Map Reduce: Word Count example.

d) **<u>Shuffle & Sort</u>**: two steps that do not need to be implemented by the user. Outputs of **map** step have to be “aggregated” before processed to **reduce** step. During shuffle, the program merges all outputs of mappers which share the same key and prepares an input data for reduce step in a form of (key, [list of values of mappers]) (in this example (term, [1,1,1,...]). During this step some data may be transferred between nodes, however, MapReduce implementations minimise transfer rate by assigning tasks closer to where the data is present. Nonetheless, system efficiency also depends on how the user implements processing strategy. After **shuffle** step, data may be **sorted** by key but this step is not obligatory.

e) **Reduce**: **reduce** function is executed by Task Trackers after **map** step is done (usually, the truth is that reduce may be executed even though all mappers have not finished their tasks). The data returned by **map** is "summarized" in reduce step by applying some functions. This process has to be implemented by the user. Please note that reducers are executed (if possible) on the same processing units where mappers were run. However, the number of reducers may be different than mappers. Input of the reducer is, as previously stated, in a form of (key, [list of values]). The output should be (key, value). In this example reducer counts how many times a particular term occurred in a collection of documents. Thus, for a given key (term) it computes the sum of given "1" ([1,1,1,1,1...]). The output is (term, number of occurrences).

Please note that a MapReduce program may consists of **many** map and reduce steps that would be run sequentially. Other steps that may be implemented:

f) **Combine:** Combiner, also referred to as "mini reducer", performs the local reduce task. If the network bandwidth is limited it may be advantageous to perform some computation before running reduce (and transferring data between nodes). A combiner is executed after map but before reduce step and it is run locally. In Figure 2 a combiner could, e.g., sum the number of occurrences of a term and return such aggregated result instead of returning pairs (term, 1). For instance, considering the second document, combiner (after mapper) would return (roses, 2) instead of (roses, 1) twice.

g) **Partition:** Partitioner may be used to partition intermediate data produced by mappers. The data is divided under user-defined conditions. Each partition is then processed into only one reducer (the number of reducers equals the number of such partitions). For instance, considering Figure 2, one may wish to compute the number of occurrences of terms starting with letters A-D, E-I, and so on. In such case the user may implement a partitioner that would direct intermediate results (terms, [list of occurrences]) towards a specified reducer.

3. Apache Spark: There are some aspects of **Apache Hadoop MapReduce** which may be of a concern and were motivation to make Spark:

- Batch processing. Hadoop MapReduce was not introduced to perform real-time computation. It uses a lot of disk read/write operations in order to handle huge amount of data. **SPARK:** Spark follows different paradigm. It is designed to handle batch and stream processing, preferring in-memory computation. Instead of HDFS it uses RDD (Resilient Distributed Datasets) that enable multiple map operations in memory. For that reason Spark is about 100x faster than Hadoop MapReduce. On contrary, the later is definitely a better option when it comes to really huge data and not real-time processing.
- CPU usage: Job Tracker of Apache Hadoop consists of "slots" which are **map** and **reduce** jobs. However, these slots are not generic which means that only one type of task (map, reduce) may be run within a single slot. So when it comes to performing, e.g., a map step, reduce slot does nothing and mean that some of CPU computing

power is wasted. In addition, map and reduce tasks are processes (heavy) which consume resources. **SPARK:** Uses generic slots, which means that a slot can be used to perform map or reduce task depending on the situation. What is more, comparing to Hadoop MapReduce, Spark spawns single (light) threads in order to process a task.

Programming assignment (Python notebook, deadline +1 week)

Download apache spark 2.2.1 (pre-built) from www.cs.put.poznan.pl/mtomczyk/ir/lab10/spark-2.2.1-bin-hadoop2.7.tgz (200MB). Next, download a notebook containing exercises http://www.cs.put.poznan.pl/mtomczyk/ir/lab10/Lab10_exercises.ipynb. Usually Spark has to be configured, however, if you are working with Java 1.8, you need to specify some environmental variables only. For this purpose, modify **os.environ['SPARK_HOME']** and **os.environ['JAVA_HOME']** strings in the provided notebook. Also, you should install **findspark** python library **(**use pip or other tool). Thanks, to findspark.init(), you can run spark through the notebook (Python + Spark = PySpark). To complete this assignment, finish all the exercises and present and discuss the results. You can download a nicely written **PySpark Cheat Sheet** http://www.cs.put.poznan.pl/mtomczyk/ir/lab10/PySpark_Cheat_Sheet_Python.pdf, by Data Camp (www.datacamp.com).