DTS Exercise 2

For exercise 2, we are going to use the following technologies and tools:

- Hadoop 3.2.1
- Intellij IDEA

1. Hadoop

Hadoop is an open source platform aimed at processing large amounts of data in a distributed manner. There are several vendors on the market that offer Hadoop distributions (e.g. Apache, Cloudera, Hortonworks, ...). The most commonly used distribution is the Apache Hadoop distribution, which represents a framework within the Hadoop ecosystem of different technologies and tools for data streams or documents (in a file system) processing, databases, cluster management etc. Apache Hadoop consists of four modules: (1) HDFS, (2) Hadoop MapReduce, (3) YARN and (4) Hadoop Common.

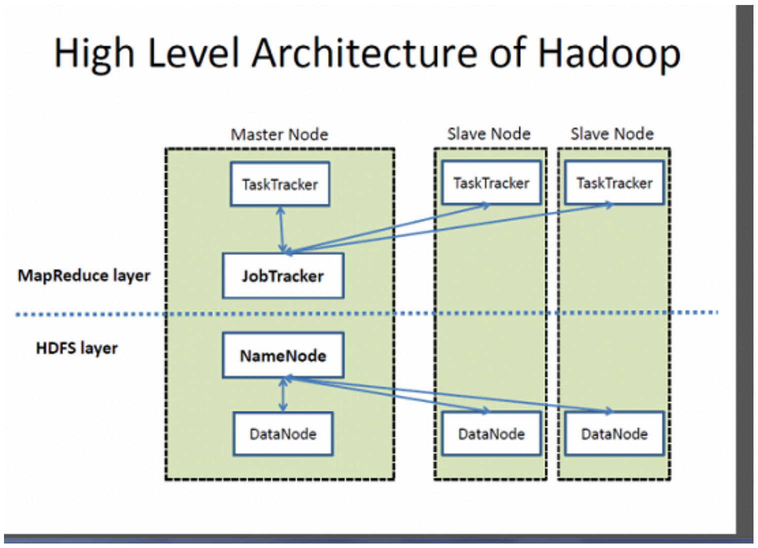


Figure 1: Simplified representation of Hadoop architecture. [1]

Hadoop cluster management

An important system requirement to successfully install and run the Hadoop cluster is Java 8. We can check if it is installed in the system with the following statement: *java -version*.

Hadoop cluster configuration

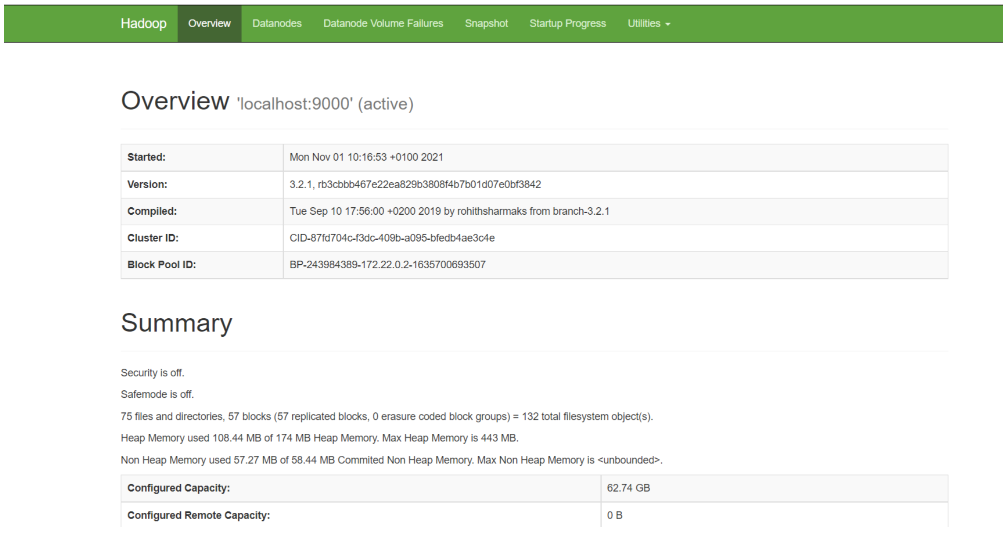
There are three ways to run a Hadoop cluster:

1. Standalone - cluster is run as a single Java process,
2. **Pseudo-distributed** - cluster is run on only one physical node, but each Hadoop daemon is run within a separate Java process, and
3. Distributed - cluster is run on separate physical nodes (machines).

Before starting the Hadoop cluster, we need to specify certain configuration properties of the cluster and data storage location. All the following steps are performed within the Hadoop root map (*/opt/hadoop*). To successfully install Hadoop in the system, we need to configure the following files:

- *hadoop-env.sh* - path to the JAVA installation map;
- *core-site.xml* - location of the folders for the HDFS NameNode and DataNode and the URL to HDFS;;
- *hdfs-site.xml* - HDFS replication factor (3 by default, but will be set to 1 in the pseudo-distributed mode), system folders for data storage on NameNode and DataNode and temporary folder for storing HDFS data;;
- *mapred-site.xml* - YARN framework for running jobs in the cluster;
- *yarn-site.xml* - system information for successfull job execution (list of environment variables).

Now we can start the cluster (**HDFS and YARN**). We run the script *start-all.sh* located in the */sbin* Hadoop directory. If everything is run sucessfully, then we can open the browser and check the HDFS (`localhost:9870`) and YARN GUIs (`localhost:8088`).



Overview 'localhost:9000' (active)

Started:	Mon Nov 01 10:16:53 +0100 2021
Version:	3.2.1, rb3cbbb467e22ea829b3808f4b7b01d07e0bf3842
Compiled:	Tue Sep 10 17:56:00 +0200 2019 by rohithsharmaks from branch-3.2.1
Cluster ID:	CID-87fd704c-f3dc-409b-a095-bfedb4ae3c4e
Block Pool ID:	BP-243984389-172.22.0.2-1635700693507

Summary

Security is off.

Safemode is off.

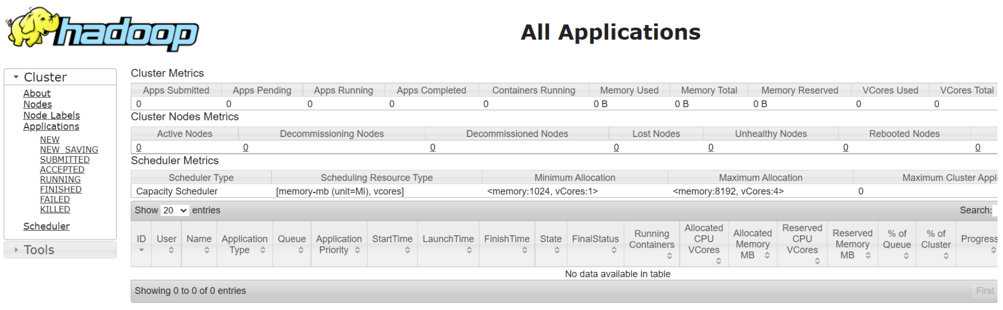
75 files and directories, 57 blocks (57 replicated blocks, 0 erasure coded block groups) = 132 total filesystem object(s).

Heap Memory used 108.44 MB of 174 MB Heap Memory. Max Heap Memory is 443 MB.

Non Heap Memory used 57.27 MB of 58.44 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	62.74 GB
Configured Remote Capacity:	0 B

Figure 2: Home page of the HDFS GUI.



All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total
0	0	0	0	0	0 B	0 B	0 B	0	0

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	
0	0	0	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	Maximum Cluster Appl
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:1024, vCores:1>	<memory:8192, vCores:4>	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress
No data available in table																		

Showing 0 to 0 of 0 entries First

Figure 3: Home page of the YARN GUI.

2. HDFS

HDFS (Hadoop Distributed File System) is a file system for distributed data storage, while enabling high system availability. It comprises of two components: DataNode and NameNode. DataNode is responsible for managing data blocks. The default replication factor of an individual data block is 3, which enables achieving high availability. When replicating blocks, NameNode defines an ID of a given block and stores the metadata about its location, whereas the block itself is forwarded to be stored on a DataNode.

HDFS statements

All HDFS statements begin with *hadoop fs* or *hdfs dfs* (these two statements are synonyms if we use HDFS as a distributed storage system). The statements are very similar to those used in Unix systems (e.g. mkdir, ls, rm, cat, mv). The list of basic HDFS statements can be found at `https://images.linoxide.com/hadoop-hdfs-commands-cheatsheet.pdf`.

In the beginning, by using the *-ls* option we can display the list of all directories in the system. The replication factor for each file is also printed (*hdfs dfs -ls /*).

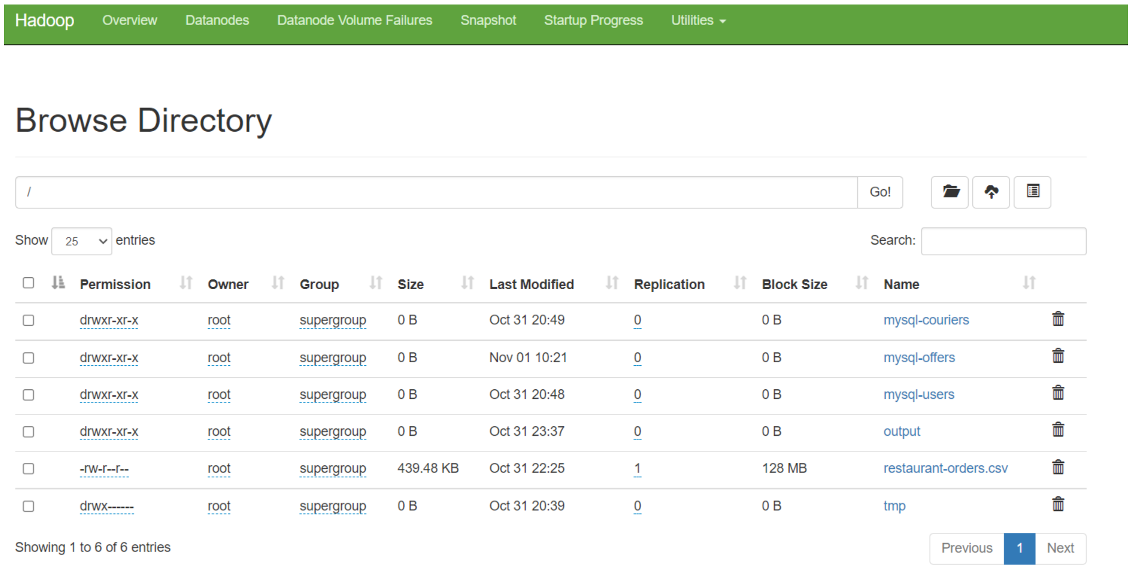
Now, we are going to add a new file *restaurant-orders.csv* into HDFS, which contains data about orders (download from eŠtudij). The file must first be added from the host OS to the Docker container with Ubuntu OS. In the terminal, position yourself into the folder that contains the file, and run:

```
docker cp restaurant-orders.csv pts-hadoop:/etc
```

This query will copy the file into the */etc* folder inside the Docker container. The next step is to add this file from the OS file system into HDFS. To add files from the local file system into HDFS, we can use options *-copyFromLocal* and *-put*, where we need to specify the file path in the local FS and the target HDFS directory.

```
hdfs dfs -put /etc/restaurant-orders.csv /
```

This query will add the file from the local */etc* folder to the HDFS root directory.



Browse Directory

/

Show 25 entries Search:

	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
☐	drwxr-xr-x	root	supergroup	0 B	Oct 31 20:49	0	0 B	mysql-couriers
☐	drwxr-xr-x	root	supergroup	0 B	Nov 01 10:21	0	0 B	mysql-offers
☐	drwxr-xr-x	root	supergroup	0 B	Oct 31 20:48	0	0 B	mysql-users
☐	drwxr-xr-x	root	supergroup	0 B	Oct 31 23:37	0	0 B	output
☐	-rw-r--r--	root	supergroup	439.48 KB	Oct 31 22:25	1	128 MB	restaurant-orders.csv
☐	drwx------	root	supergroup	0 B	Oct 31 20:39	0	0 B	tmp

Showing 1 to 6 of 6 entries

Previous 1 Next

Figure 4: Successfully copied file from the local file system to HDFS.

3. MapReduce

MapReduce framework was introduced in Hadoop v2. It enables parallel processing of large amounts of data. The framework is comprised of two parts (steps): **Map** in **Reduce**. The Map step is responsible for analyzing information from input data, using a given algorithm and generating results as key-value pairs. These results are then forwarded to the Reduce step, which aggregates data received from the Map step. Figure shows components included in the execution of a MapReduce job: Job Tracker - master and Task Tracker - slave.

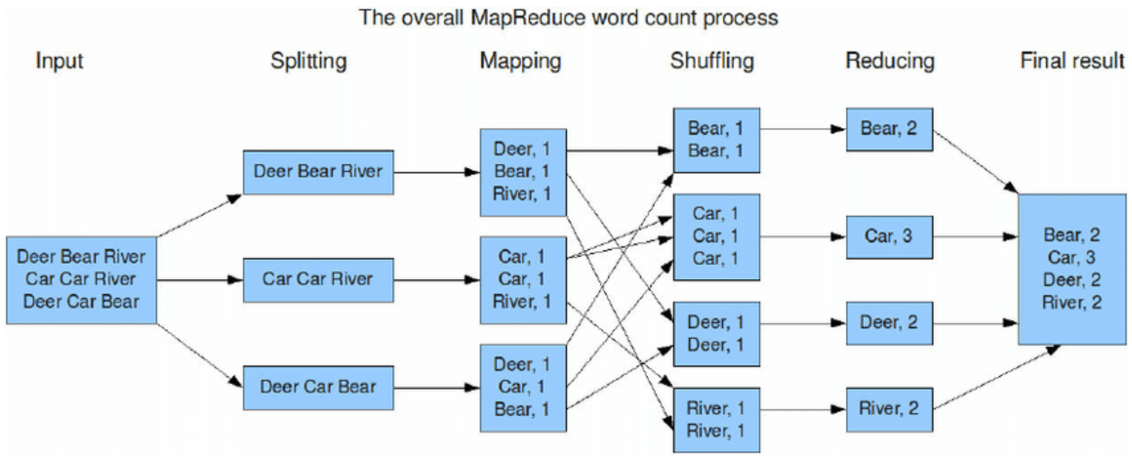


Figure 5: MapReduce job execution (WordCount example).

Throughout the exercises, we will implement MR jobs in Java by building executable JAR files in Intellij IDEA. The Java code of the MR job is available for download at `https://github.com/msestak2/`

`MapReduceExample/`. The Java code will include three parts: main class, Mapper and Reducer classes. The main class includes the main method and parameters necessary to execute MR job. Function map() includes the algorithm for input data processing, whereas function reduce() includes and algorithm for aggregating results of the map() function.

The objective of our first MR job is to process input file "restaurant-orders.csv", which includes data about customer orders in a restaurant, where the result is a list of items and how many times they were ordered.

If we look at the structure of "restaurant-orders.csv" (note: inserted into HDFS in the previous step), we can that it included columns: *Order Number, Order Date, Item Name, Quantity, Product Price and Total products*. In the result of the MR job we only want to print *Item Name* and the aggregated number of orders.

We create a new Maven project in Intellij. We declare three dependencies in the *pom.xml* file: *hadoop-common, hadoop-core and mapred-client-core*, which can be found at `https://mvnrepository.com/`. We create a new package *um.si*, where we will create three classes. Under *File > Project Structure > Project Settings > Artifacts > Jar > from modules with dependencies...* we must specify that we want to build a JAR file from our classes.

The OrderMapper class includes the following:

```
package um.si;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class OrderMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedExc
        String row = value.toString();

        String itemName = row.split(",")[2];
        int quantity = Integer.valueOf(row.split(",")[3]);

        context.write(new Text(itemName), new IntWritable(quantity));
    }
}
```

The OrderReducer includes the following:

```
package um.si;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class OrderReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
    IOException, InterruptedException {
        int sum = 0;
        for(IntWritable val: values){
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

These two classes must be registered in the main class:

```
package um.si;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class OrderProcessing {
    public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedExcept
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration, "CalculateOrderedItemQuantity");

        job.setJarByClass(OrderProcessing.class);
        job.setMapperClass(OrderMapper.class);
        job.setReducerClass(OrderReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));

        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

When we build artifacts, under the root project map in the local FS, we will see a newly added /out map, where the JAR file *MapReduce21.jar* is available. We copy it to the Docker container with: *docker cp MapReduce21.jar pts-hadoop:/etc.*

Once this is done, we can run our MR job with the following statement:

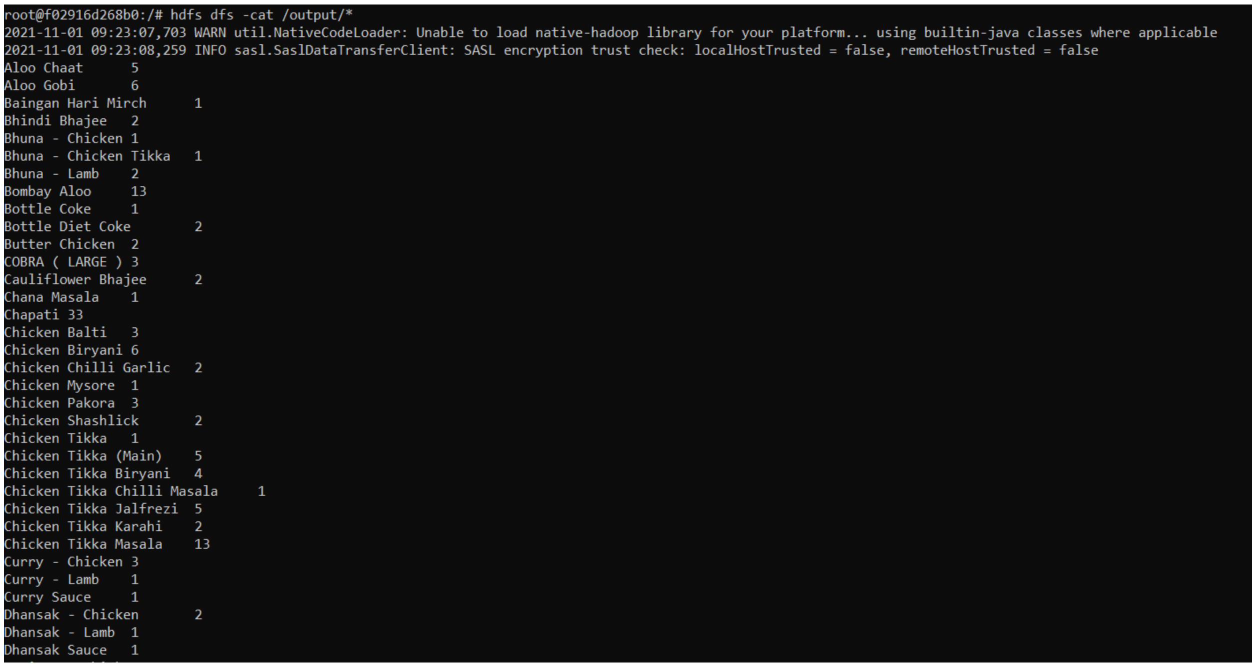
```
$HADOOP_HOME/bin/hadoop jar /etc/MapReduce21.jar
/restaurant-orders.csv /output
```

Note: The "output" directory does not need to be created in advance, YARN will create it automatically - but, we can specify this map as a target directory to store results only once, i.e. for only one MR job execution. If we want to re-use it, we must first delete the existing "output" directory with:

```
hdfs dfs -rm -r /output
```

If everything runs successfully, you can use the YARN GUI to see our MR job as an application with status "SUCCEEDED". The results of the MR job can be printed with:

```
hadoop fs -cat /output/*
```



```
root@f02916d268b0:/# hdfs dfs -cat /output/*
2021-11-01 09:23:07,703 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2021-11-01 09:23:08,259 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
Aloo Chaat      5
Aloo Gobi       6
Baingan Hari Mirch      1
Bhindi Bhajee   2
Bhuna - Chicken 1
Bhuna - Chicken Tikka   1
Bhuna - Lamb    2
Bombay Aloo     13
Bottle Coke     1
Bottle Diet Coke        2
Butter Chicken  2
COBRA ( LARGE ) 3
Cauliflower Bhajee      2
Chana Masala    1
Chapati 33
Chicken Balti   3
Chicken Biryani 6
Chicken Chilli Garlic   2
Chicken Mysore  1
Chicken Pakora  3
Chicken Shashlick       2
Chicken Tikka   1
Chicken Tikka (Main)    5
Chicken Tikka Biryani   4
Chicken Tikka Chilli Masala     1
Chicken Tikka Jalfrezi  5
Chicken Tikka Karahi    2
Chicken Tikka Masala    13
Curry - Chicken 3
Curry - Lamb    1
Curry Sauce     1
Dhansak - Chicken       2
Dhansak - Lamb  1
Dhansak Sauce   1
```

Figure 6: The result of a MR job.

When you are finished, stop all Hadoop cluster components (HDFS, YARN) that you used by executing the *stop-all.sh* script.