Concepts and Models for Parallel and Data-centric Programming, Summer 2021

Exercise 4: MapReduce and Spark

Task 1. MapReduce and Maximum Temperature
You have a huge set of temperature measurements (in °C) for different cities. The input data consists of key-value pairs *(city name, temperature values)* where the city name is a `String` and the temperature values as a `String` consisting of comma-separated floating point values. There are multiple key-value pairs for each city. You should use the MapReduce programming model to determine the *maximum temperature* for each city. Consider the following example key-value pairs:

```
(Aachen, "9.3, 15.2, 24.0")
(Berlin, "4.1, 6.4, 100.4")
(Cologne, "-5.5, 14.2")
(Aachen, "2.0, 16.0, 9.0")
(Berlin, "6.4, 100.4")
(Cologne, "-2.0, -137.3")
```

Unfortunately, some measurement values are inconsistent such that you have to filter out those values where the temperature is greater than 60 °C or lower than −90 °C.

Task 1.1. Give the required `Map()` and `Reduce()` functions to filter out the invalid values and compute the maximum temperature for each city in pseudocode. Also denote the type signatures of both functions. You should not pre-combine values in the `Map()` function. **Hint:** You may use the function `extractFloats(String valString)` which produces a list of `float` values out of a given comma-separated string of floating point values.

Task 1.2. Perform the MapReduce computation on the given example dataset manually: Apply the defined `Map()` function on each input key-value pair, group the corresponding outputs by key and apply the `Reduce()` function on it to generate the final output.

Task 1.3. Which (combination of) MapReduce design patterns did you use to solve the task?

Task 1.4. Is using a `Combiner()` function useful for this kind of computation? If so, give the corresponding pseudocode of the `Combiner()` function and its type signature. If not, state why a combiner is not applicable here.

Task 2. MapReduce and Friends
Beside counting words and calculating temperatures, another typical use case well-suited for the MapReduce programming model is finding common friends. Assume you are a developer of a social network service. If some user visits the profile of a friend, then the friends they both have in common should be displayed. Since recomputing the list of common friends on each profile visit is expensive, you want to precompute this list for each possible combination of two friends and store it.
Each user and his friends list are stored as key-value pairs. Note that the friend relation is in general symmetric, i.e., if Alice is a friend of Bob, then Bob is also a friend of Alice. Consider the following example key-value pairs:

```
Alice -> [Bob, Dave, Eve]
Bob   -> [Alice, Carol, Dave, Eve]
Carol -> [Bob, Dave]
Dave  -> [Alice, Bob, Carol]
Eve   -> [Alice, Bob]
```

The MapReduce computation should finally output for each combination of two friends their common friends, e.g., for Alice and Bob, it should output the following key-value pair:

```
(Alice, Bob) -> [Dave, Eve]
```

Hint: It might be useful to read through all of the following subtasks to get the right idea before starting with the first subtask.

Task 2.1. Give the required `Map()` function for the MapReduce computation in pseudocode. You may assume that the signature of the function is

```
Map: (String, List<String>) -> List(Pair<String, String>, List<String>).
```

The input *key* argument (k_1) is the user name and the input *value* argument (v_1) the user's friends list. For example, the key-value pair `Alice -> [Bob, Dave, Eve]` leads to a function call

```
Map(Alice, [Bob, Dave, Eve]).
```

Furthermore, you may assume that an element of type `Pair<String, String>` has no order, so the pairs `(Alice, Bob)` and `(Bob, Alice)` are equal and will be summarized into the same key group.
Hint: For each input `Map(user, friends list)`, the number of output elements should be the same as the length of the friends list.

Task 2.2. Apply the `Map()` function to each key-value pair given in the example. For simplicity, you might only write down the first letter of each user name. After that, group the output by key to perform the shuffle step.

Task 2.3. Give the required `Reduce()` function in pseudocode. You may assume that the signature of the function is

```
Reduce: (Pair<String, String>, List<List<String>>)
   -> List(Pair<String, String>, List<String>).
```

Hint: You should exploit that the nested input list always consists of two sublists.

Task 2.4. Apply the `Reduce()` function to the output you got in Problem 2.2 to generate the final output. Again, you might only write down the first letter of each user name.

Task 3. RDD Dependencies in Apache Spark
The dependencies between RDDs (see 55-Apache Spark slides 19 ff.) can have huge impact on the performance of transformations. An important property of an RDD is the partitioner which partitions the dataset into distinct groups. In this task, you will investigate the effect of the partitioner on narrow and wide dependencies further.

Task 3.1. Consider the following two lists of key-value pairs you want to store as RDDs:

```
List 1: [(a, 1), (b, 2), (a, 3), (c, 42), (d, 43), (e, 5))]
List 2: [(b, 3), (b, 6), (d, 10), (d, 13), (e, 12), (a, 41), (c, 4), (c, 0)]
```

When creating the RDDs, a custom hash partitioner is used to split the data. We define the hash function h_i for $i \in \mathbb{N} \setminus \{0\}$ as

$$h_i : \mathbb{N} \to \{0, \ldots, i-1\}, \qquad n \mapsto n \bmod i.$$

The hash functions h_3 and h_4 should be used to partition the data. Since the key entries are characters, use for the hash computation an ASCII table to convert them to their *decimal* representation.
Compute the partitions of the following three RDDs:

- `RDD1` out of `List 1` with hash function h_3
- `RDD2` out of `List 2` with hash function h_3
- `RDD3` out of `List 2` with hash function h_4

Use the following notation to write down the partitions of an RDD:

```
RDD1:  P0: (key1, value1), (key2, value2), ...
       P1: (key1, value1), (key2, value2), ...
       P2: (key1, value1), (key2, value2), ...
       ...
```

Task 3.2. Perform the following transformations on the RDDs by writing down the resulting RDD (use the same notation as in Task 3.1). State for each resulting RDD whether it has a *narrow* or *wide* dependency on its parent(s).

- `mappedRDD = RDD1.mapValues(n -> n + 1)`
  Note: `mapValues` is the same as `map`, but works on key-value pairs and just modifies the value.
- `unionRDD2 = mappedRDD.union(RDD2)`
  Note: `union` merges the resulting partitions only if the parent RDDs are co-partitioned. Otherwise, it just combines the partitions in a new RDD without merging them and the partitioning is lost.
- `unionRDD3 = mappedRDD.union(RDD3)`
- `joinRDD = mappedRDD.join(RDD2)`
- `joinUnionRDD = mappedRDD.join(unionRDD3)`

- `reducedRDD2 = RDD2.reduceByKey((a, b) -> a + b)`
- `reducedUnionRDD3 = unionRDD3.reduceByKey((a, b) -> a + b)`

Task 3.3. Draw the lineage graph of the defined transformations in Task 3.2. It is sufficient to just use a single box for each RDD (see 55-Apache-Spark slide 18 for a reference).

Task 4. Using a Hadoop Cluster
In this task, you will set up a Hadoop cluster consisting of 16 nodes (1 master, 15 workers) on the CLAIX supercomputer. Each node of CLAIX18 has a local storage of 400 GB that we will use to set up HDFS. Since the cluster is small and the availability of data is not critical, the HDFS replication factor is set to 1 (no replication). In the following, you will create and submit some MapReduce programs on this cluster.
Before you can start with this task, you have to establish a connection to a frontend node of CLAIX. You may preferably use FastX (`https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/25f576374f984c888bb2a01487fef193/`) or an SSH connection (`https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/10c8f0d9b0064013aa439f0b504cc806/`). Furthermore, only students that have sent their RWTH user id to `contact@hpc.rwth-aachen.de` and are part of the `lect0053` group will be able to spawn the Hadoop cluster. If you have not done it yet, please do so.
After logging in, you have to create the cluster first. We prepared a script that does all the work for setting it up (requesting compute nodes, starting containers with Hadoop, putting data to the cluster and so on). Execute the following command in a terminal to run the script:

```
sh /home/lect0053/allocatecluster.sh
```

Note that it might take a while until 16 nodes are assigned by the scheduler. If you have to wait a long time, then stop the script with `CTRL-C` and try to run the script in the morning (there are typically more spare nodes available than in the afternoon). As soon as the resources are assigned, the Hadoop cluster will be initialized with some data and will output many log messages, this will take a few minutes. In the end, the script prints out some URLs to the web UIs of the ResourceManager and the NodeManager. You can optionally have a look at them.
The allocation of your cluster will stay at maximum 2 hours, then it is stopped. When you have finished the exercise or you want to stop the cluster in the end, just leave the terminal by running the `exit` command.
For the course of the whole task, execute all following commands in the terminal where you invoked the script to start the cluster.
The example programs are available in the archive file `/home/lect0053/mapreduce-exercise.tar.gz`. Please decompress the archive into your local home directory using the command

```
tar -xzf /home/lect0053/mapreduce-exercise.tar.gz -C $HOME
```

This will create a folder named `mapreduce` with two subfolders `wordcount` and `invertedindex` in your home directory.

Task 4.1. Change to the `wordcount` folder with `cd`. The folder contains a word count example project realized with Maven. You can find the source code in the following folder:

```
src/main/java/de/hpc/pdp/wordcount
```

You are free to use your own desired text editor or IDE to edit the source code. The folders `input-local1` and `input-local256` contain local test data sets which you can use to perform a test run locally. You can run

```
mvn compile exec:java -Dexec.args="input-local1 1"
```

to perform word count on the example text file in folder `input-local1`. The first input argument of the word count program is the input folder with the corresponding text files. The second input argument is the number of reduce tasks which should be used. For the local tests, you can set the value to 1 as depicted above.
When running the computation, a folder named `output-yyyyMMdd-HHmmss` will be created. It contains the generated output files (here: counts of each word) of the computation. The number of output files depends on the number of the reduce tasks.
After launching your first MapReduce program locally, you can now submit your first job to the Hadoop cluster. First, have a look at the (root) folders available in HDFS:

```
hdfs dfs -ls /
```

You will find the folders `words50G`, `words100G`, `words200G`, containing text files of accumulated sizes 50 GB, 100 GB, 200 GB, respectively. There are also larger folders with a size of 400 GB and 800 GB, but the execution with those data sets may take longer than just a few minutes. Keep in mind that the Hadoop cluster only can access data which has been made available in HDFS.
Package the word count program in a JAR file:

```
mvn package
```

This will create a JAR file in the `target` directory of the project. Then, submit the program to the cluster with the `words[50|100|200]G` folders and 100 reduce tasks as parameters:

```
hadoop jar target/wordcount-1.0.jar /words[50|100|200]G 100
```

Try out the different dataset sizes. Running a job might take a few minutes. The current state of the MapReduce computation will be reported continuously on the terminal. If you think that there is something going wrong with your job, then you can kill it by escaping from the run with `CTRL-C` and then killing the application:

```
yarn application -list
yarn application -kill <Application ID>
```

As soon as the computation has finished, some statistics about the MapReduce computation are printed out. Save the final statistics output to some text file. The output of the computation itself is written to a local user folder on HDFS. You can have a look at your output folders with

```
hdfs dfs -ls
```

and copy the output data folder to the local file system with

```
hdfs dfs -get output-yyyyMMdd-HHmmss .
```

whereby you have to replace the date placeholders.
As a next step, change the `WordCount.java` file such that no combiner is used (remove the `setCombiner` call). Repackage the JAR file using `mvn package` and submit the job again to the Hadoop cluster. Save again the statistics of the MapReduce computation.
After finishing all these steps, answer the following questions:

1. Examine the effect of the combiner: How much network I/O and time is saved compared to a run without combiner for the different data sizes?
2. In case of a run with combiner: Why is the total number of input records of the combiner larger than the total number of map output records (see statistics "Map-Reduce Framework")?
3. Why is there (most probably) a number of killed map and reduce tasks (see statistics "Job Counters") shown in the statistics?

Task 4.2. Another typical example of a MapReduce program is the calculation of an inverted index (see 44-MapReduce, slide 9). The main goal is to calculate for each word in a given set of documents a list of documents that contain this word. Using an inverted index, you can for example look up all Wikipedia articles containing a certain word very easily.
Change to the `invertedindex` folder and have a look at the source code. The source code files `InvertedIndex(Mapper|Combiner|Reducer).java` contain some `TODO`s where you have to add some code in order to make the program run correctly.
After resolving the `TODO`s, first test your program with the local test data set in the folder `input-local` with 1 reduce task

```
mvn compile exec:java -Dexec.args="input-local 1"
```

and check the output by looking at the output file. If it is running successfully, then you can run your data set on a dump of the Simple English Wikipedia (folder `input-simplewiki` in HDFS): Package a JAR with `mvn package` and submit the job:

```
hadoop jar target/invertedindex-1.0.jar /input-simplewiki 10
```

After that, you can again retrieve the output with

```
hdfs dfs -get output-yyyyMMdd-HHmmss .
```

and have a look at the corresponding output files. As a next processing step, you would put the output files in a data structure or database with efficient access to a given search word. However, this is not part of this exercise. Nevertheless, you can stick with a simple `grep` command to search for a word in the output files (execute it in the output folder):

```
grep -RPi "^mapreduce\t"
```

This will search for a line beginning with the word "mapreduce" and should output

```
mapreduce       Google Search, Pig (disambiguation)
```

where the values on the right side are the pages of the Simple English Wikipedia which contain this word.

Bonus Task: Run your program on the English Wikipedia data set (`input-enwiki`). Before you can do that, you first have to send the file to the HDFS file system. Do that by running

```
hdfs dfs -put /hpcwork/lect0053/input-enwiki /
```

Beware that moving this large file ($\approx$ 70 GBs) to HDFS takes roughly 10 minutes.

Task 4.3. We discussed architectural differences in the file system of a supercomputer and a big data cluster in the lecture (see 48-MapReduce, slide 6). In general, it is also possible to run HDFS on the compute nodes of a supercomputer and access data via a parallel file system like Lustre. In other words, a file access on HDFS is translated to a file access on Lustre. Explain how this affects the data locality feature of HDFS and the data accesses in the three phases (map, shuffle, reduce) of the MapReduce computation.

Discussion of this exercise on July 1st, 2021.