DTIM
Database Technologies and Information Management

Big Data Management
Hands-on session: MapReduce

Lecturer: Petar Jovanovic

1 Required Tools

- `eclipse` IDE with JDK 8 and Maven plugin installed

Be aware that to locally execute MapReduce jobs you should use your own laptop and you should be able to work in both Linux and Windows OS. Refer to the enclosed manual for details on how to work with both settings.

2 Part A: Introduction and examples of MapReduce jobs

First, we will clarify the main objectives of this Hands-on session. We will then recall the necessary knowledge on MapReduce for the session and answer the possible questions.

2.1 MapReduce Java project

To work during this Hands-on session, you need to access and download the MapReduce Java projects from:

`https://github.com/dtim-upc/MapReduce-PATC`

If you are familiar with git, you can use directly git command set. Otherwise, just go to `Code->Download ZIP`, unpack it, and import it as existing Maven project in your IDE (Eclipse). Check Eclipse-manual.pdf for instructions on how to import and compile the MapReduce project.

2.2 Exercise 1: Examples of MapReduce jobs implementing relation algebra operators

In this exercise you will get familiar with MapReduce by analyzing the code and executing the provided MapReduce jobs. To this end, you are provided a file `wines.1000.sf` in the diretory `src/main/resources` of the Java project. In the following subsections we will first provide you the rationale behind the implementation of some of the most typical *relational algebra operators* with MapReduce.

2.2.1 Projection

The `Projection.java` job selects a subset of the attributes in the dataset. Precisely, we are projecting the attributes `alc`, `col` and `hue`. If such dataset was stored in a relational table, we would be performing the following SQL query:

```
SELECT alc, col, hue FROM wines
```

Configuration First, take a look at the method `configureJob`. In this method we first define the mapper class and its output types for the key and the value. Next, as the projection does not require a reduce step, we already provide the same information for the output of the job. The next lines specify the format of the input (SequenceFile for all the cases) and the input/output paths. Finally, we pass parameters to the mapper. In this case, we send the list of `projection` attributes (i.e., `alc`, `col` and `hue`). Note that a projection does not require an aggregation phase, thus the combiner or reducer are not configured.

Map task Take a look at the `ProjectionMapper` class, which extends `Mapper` and its consistently typed with the configuration (i.e., its input key and value types are `Text`, as well as for the map output). The `map` method starts fetching the projection parameters from the context and splits the comma-separated input. The `Utils.getAttribute` method returns the projection for a specific attribute in the splitted line. To this end, we iterate on as many projection attributes as requested by appending the output in `newValue`.

Now, let's execute the MapReduce job locally (using MainLocal.java).

To do so, we need to additionally create a `Run Configuration` of our Java project specifying the following input arguments:

```
-projection src/main/resources/wines.1000.sf src/main/resources/output/projection.out
```

Notice that you need to create an ***`output`*** *folder inside the* ***`resources`*** *directory of your project, where all the outputs will be stored.*

Take a look at the output in console and analyze the provided information about map tasks and reduce tasks.

Then, open the output file from the executed MapReduce job from the previously specified location:

```
src/main/resources/output/projection.out/part-r-00000
```

Analyze the results of the executed job and relate it to the specifications of the projection operation in the code.

Also, remember you can run the job as many times as you want, however you will need to delete the output folder every time from the previously specified location:

`src/main/resources/output/projection.out`

2.2.2 Aggregation sum

The `AggregationSum.java` job performs a grouping on the attribute `type` and aggregates using the sum the attribute `col`. If such dataset was stored in a relational table, we would be performing the following query:

```
SELECT type, SUM(col) FROM wines GROUP BY type
```

Configuration The method `configureJob` is similar as before. However, now the output key-value are respectively the classes text and double. Also, note now that, oppositely as the projection, we specify the combiner and reducer classes. The last two lines of the method pass the required parameters to the map task (i.e., the `groupBy` attribute and the `aggregation` attribute).

Map task The map task starts fetching the parameters from the context, as well as splitting the input comma-separated value. Next, it extracts the key and value and writes them to the context.

Combiner and reduce task The only processing that the combiner and reduce tasks perform is to sum the list of values for the same key. This is finally written to the output.

Now, let's execute the MapReduce job, by creating another `Run Configuration` with the following input arguments:

```
-aggregationSum src/main/resources/wines.1000.sf src/main/resources/aggregationSum.out
```

Like before, take a look at the output in console and analyze it.

2.2.3 Cartesian product

The `CartesianProduct.java` performs the cross product of all elements that have `type` with values `type_1` and `type_2`. If such dataset was stored in a relational table, we would be performing the following query:

```
SELECT external.*, internal.*
FROM wines as internal, wines as external
WHERE external.type = "type_1" AND internal.type = "type_2"
```

Configuration The method `configureJob` is similar as before. Note that now we do not configure a combiner phase to maintain the semantics of the operator.

Map task The map task applies a different logic depending on whether the input value contains the `type` for external or internal. If the value corresponds to external, we generate a key with the formula $random\%N$ (with $N = 100$) and write it to the context. Otherwise, if the value corresponds to internal, we write to the context the same value for all keys in the range $1..N$.

Reduce task The reduce task defines the internal and external lists, and adds to them the elements in the list of values accordingly to the `type`. Afterwards, a double loop iterates on them which causes the generation of all combinations.

Now, let's execute the MapReduce job, by creating another `Run Configuration` with the following input arguments:

```
-cartesian src/main/resources/wines.1000.sf src/main/resources/cartesian.out
```

Take a look at the output in console and analyze it.

2.3 Exercise 2: Implement your first MapReduce jobs

In this exercise you will implement two MapReduce jobs. To help you, we provide the `MainLocal.java` method, where you will be able to run MapReduce jobs locally in eclipse. For simplicity, we provide you with a local `wines.1000.sf` file in the resources folder of the project.

2.3.1 Selection

Implement the selection (i.e., filtering) of the dataset. Precisely we want only to output those tuples where the attribute `type` has a value `type_1`. If such dataset was stored in a relational table, we would be performing the following query:

```
SELECT * FROM wines WHERE type = "type_1"
```

Write the required classes (i.e., map, combiner and/or reduce) in the file `Selection.java` and run it following the same instructions as with the MapReduce jobs in the previous exercise.

2.3.2 Aggregation average

Implement the aggregation with an average function of the dataset. If such dataset was stored in a relational table, we would be performing the following query:

```
SELECT type, AVG(col) FROM wines GROUP BY type
```

Write the required classes (i.e., map, combiner and/or reduce) in the file `AggregationAvg.java` and run it following the same instructions as with the MapReduce jobs in the previous exercise.