ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Laurea Magistrale in Ingegneria e Scienze Informatiche

USED CARS ANALYSIS

Elaborato in
BIG DATA

Presentata da
GABRIELE GUERRINI

Anno Accademico 2019 – 2020

Table of contents

Chapter 1

Dataset and query

1.1 Dataset

The dataset can be downloaded at:

https://www.kaggle.com/austinreese/craigslist-carstrucks-data.

It contains data about more than 400k used cars for sale in US. Each car is described by several info such as price, odometer, manufacturer, year, sale region and so on.

1.2 Query

1.2.1 Description

The query to be executed is the following one:

"For each region, it must be found the OPI of the most widespread brand in such region, considering cars that use gas fuel only.

OPI is an acronym for Odometer-Price Index and represents the average ratio odometer/price. It is calculated upon all cars of a given brand in the country, regardless of other car features (fuel type, number of cylinders...)."

1.2.2 Workflow

Figure 1.1 shows the adopted workflow[1].
The pipeline is made up of four different jobs, that is:

- **Job 1: Preprocessing**: It executes all preprocessing operation needed to correctly use the dataset in next jobs. Firstly, it cleans the raw dataset by droppinig records that contain missing values and by eliminating useless columns. Then, it executes the dataset growth operation, namely, each remaining record is copied by a certain replication factor so that the output dataset contains enough amount of data. The default replication factor is 100.
- **Job 2a: Opi**: It calculates the OPI for each brand.
- **Job 2b: Region**: It calculates the most widespread brand for each region.
- **Job 3: Join**: It merges partial results of jobs 2a and 2b.

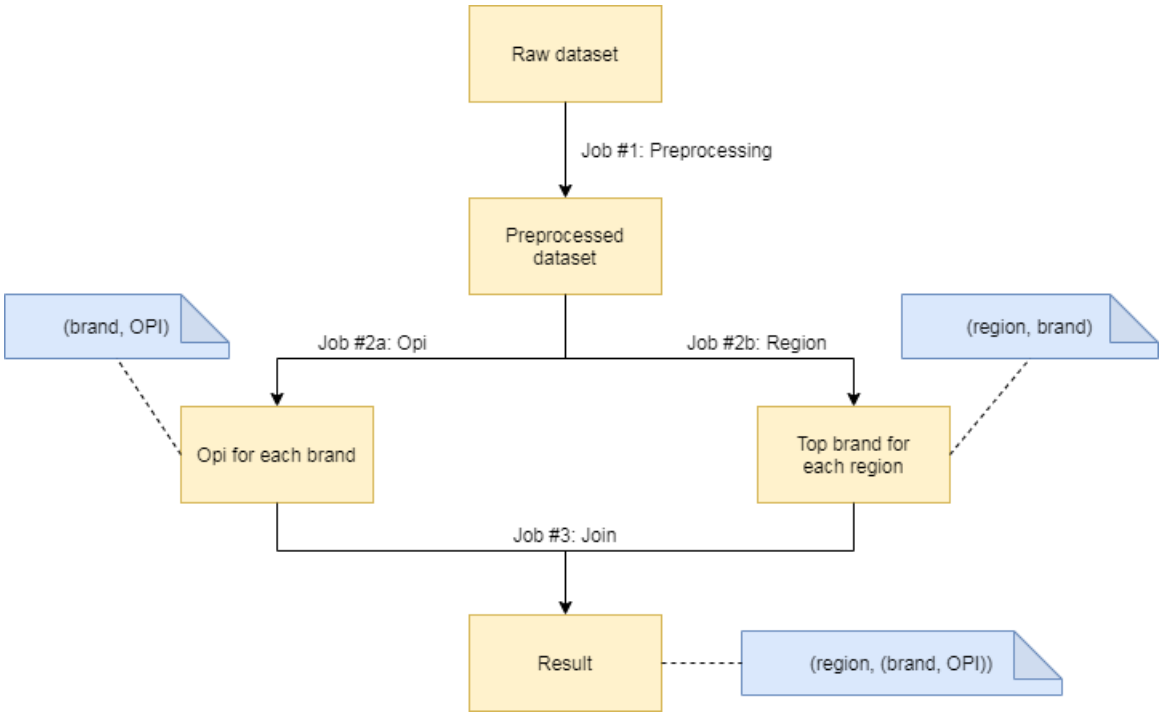


Figure 1.1: Adopted workflow for query execution.

[1]It is not the optimal one since the same query could be executed in less than four job. The idea is to structure the workflow this way so that few different algorithms (e.g. filtering, projection, summarization, join) and optimizations (e.g. caching) can be applied.

Chapter 2

MapReduce

2.1 Jobs

2.1.1 Job 1: Preprocessing

Job execution and interfaces are described in figure 2.1.

Each raw record is read and parsed into a "Car", i.e. a custom "Writable" object, during map stage. Each "Car" stores data about:

- Region
- Price
- Brand
- Fuel
- Odometer

The mapper output is a pair where the key is the default one used by Hadoop when reading text files and the value is the "Car" itself.

The reduce stage replaces the default key with a "NullableWritable" so that the whole job output is a set of records yet. Then, it publishes each record several times as specified by the replication factor.

Actually, the two phases could be collaped into one and reduce stage could be obmitted. However, a better separation of concerns between map related and reduce related responsabilities is preserved by keeping them separated.

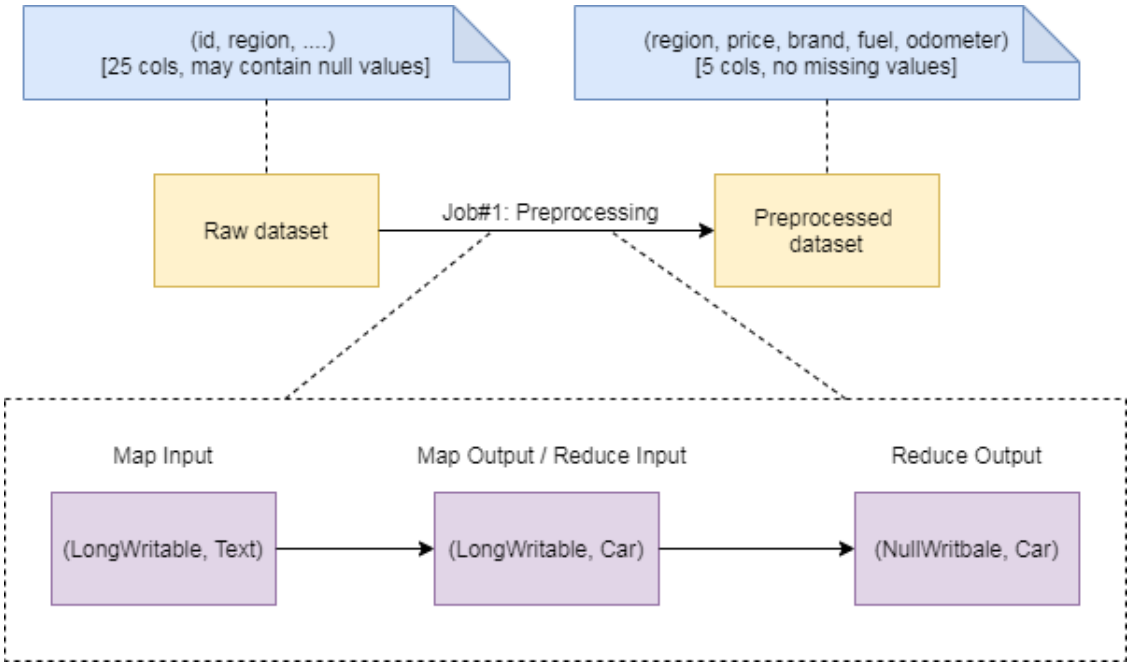


Figure 2.1: Description of "job 1: preprocessing" workflow and interfaces.

2.1.2 Job 2a: Opi

Job execution and interfaces are described in figure 2.2.

The map stage input value is a "Car" encoded as text. The key is the Hadoop default one (it won't be used). The mapper calculates the OPI for a given car using price and odometer fields and gives a pair consisting of (brand, OPI) as output.

The reduce stage just calculates average values using single OPIs and reports it as output.

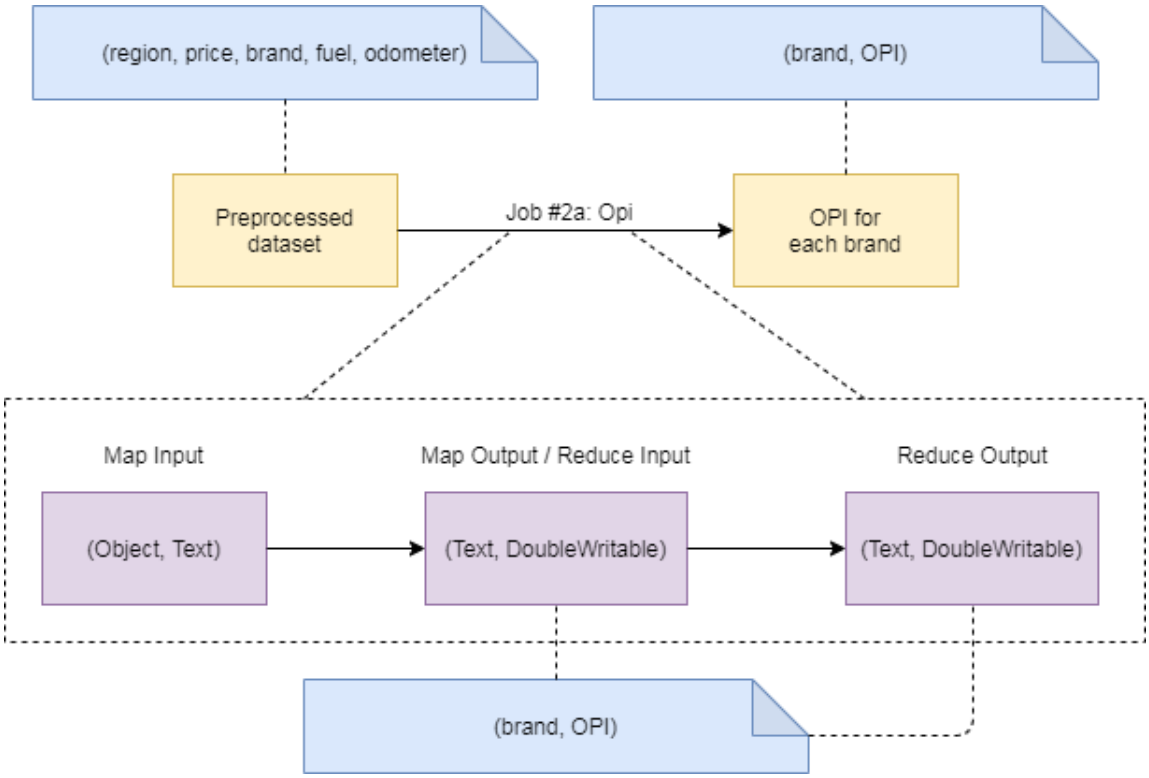


Figure 2.2: Description of "job 2a: opi" workflow and interfaces.

Usage of combiners

The job can be optimized by the usage of combiners. New job execution and interfaces are described in figure 2.3.

The combiner stage calculates a partial average. Since the mean function is not associative, it is required the usage of a support variable so that a weighted mean can be calculated in reduce phase. Hence, interfaces between the different stages are fixed[1] and a new custom "Writable" that stores info about partial mean and the number of items used when calculating such mean is created ("OpiAveragePair").

[1]Map interface is changed too since output from combiners and mappers must have the same format.

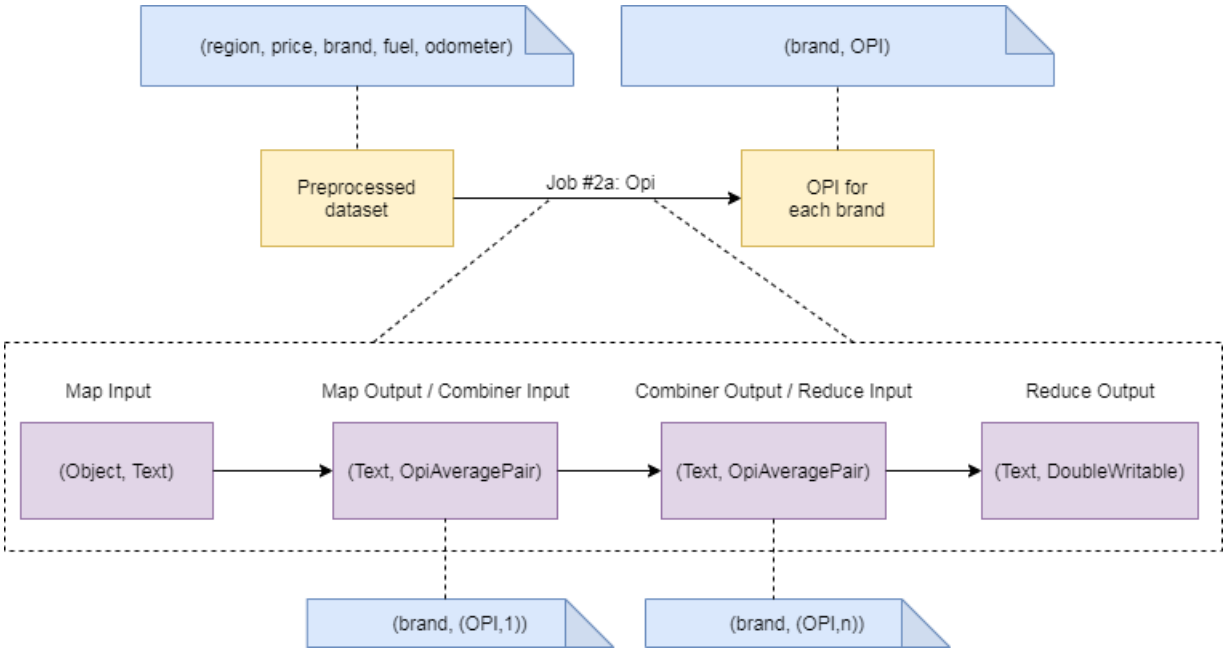


Figure 2.3: Description of "job 2a: opi" workflow and interfaces with addition of combiners.

2.1.3 Job 2b: Region

Job execution and interfaces are described in figure 2.4.

The map stage input value is a "Car' encoded as text. The key is the Hadoop default one for text files (it won't be used). If the fuel type is correct, the mapper generates as output a pair (region, brand) so that reduce stage can operate per region basis.

The reducer, given a region, firstly calculates the cardinality for each brand by collecting the input pairs. Then, it calculates the maximum and gives it as output in the form of (region, brand).

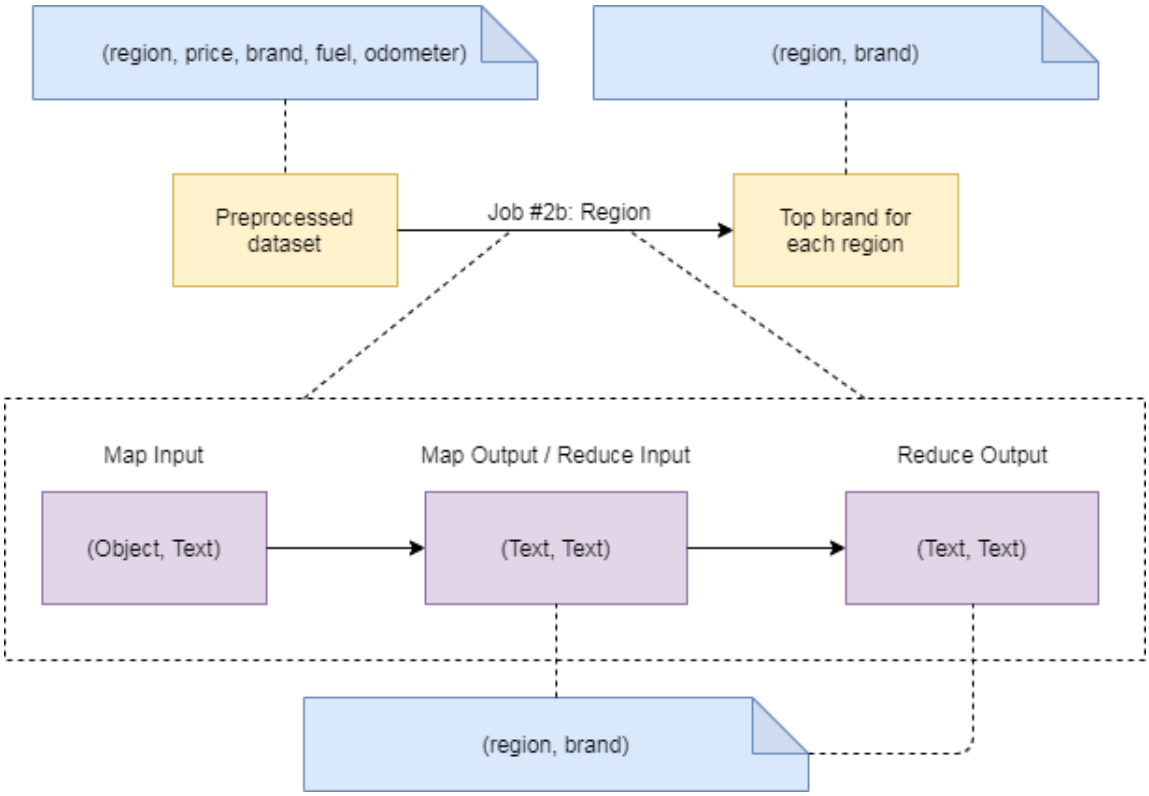


Figure 2.4: Description of "job 2b: region" workflow and interfaces.

Usage of combiners

The job can be optimized by the usage of combiners. New job execution and interfaces are described in figure 2.5.

The combiner stage calculates partial aggregations on brand frequencies. A support variable is added to save the counting so that partial results can be merged in reduce phase. Hence, interfaces between the different stages are fixed[2] and a new custom "Writable" that stores info about brand and its frequency is created ("BrandQuantityPair").

[2]Map interface is changed too since output from combiners and mappers must have the same format.

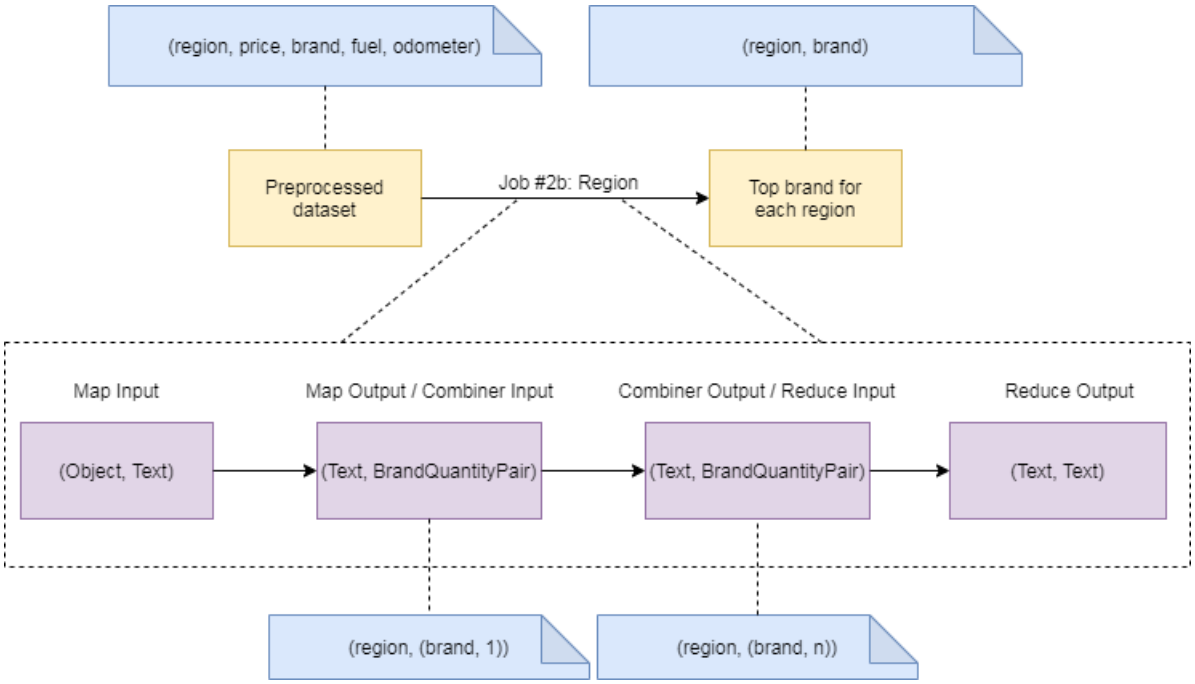


Figure 2.5: Description of "job 2b: region" workflow and interfaces with addition of combiners.

2.1.4 Job 3: Join

Job execution and interfaces are described in figure 2.6.

The map stage takes as input pairs structured as (Text, Text). The source of a given pair can be from job 2a output or from job 2b output. By examinating a given input record, it is discovered its source and the pair is marked with a flag. Since the join key will be the brand, the output of map stage is a pair (brand, (flag, region)), where flag and region are encapsulated into a "JoinPair" (a custom "Writable").

The reducer executes the join algorithm on input items and creates output pairs in the form (region, (brand, OPI)).

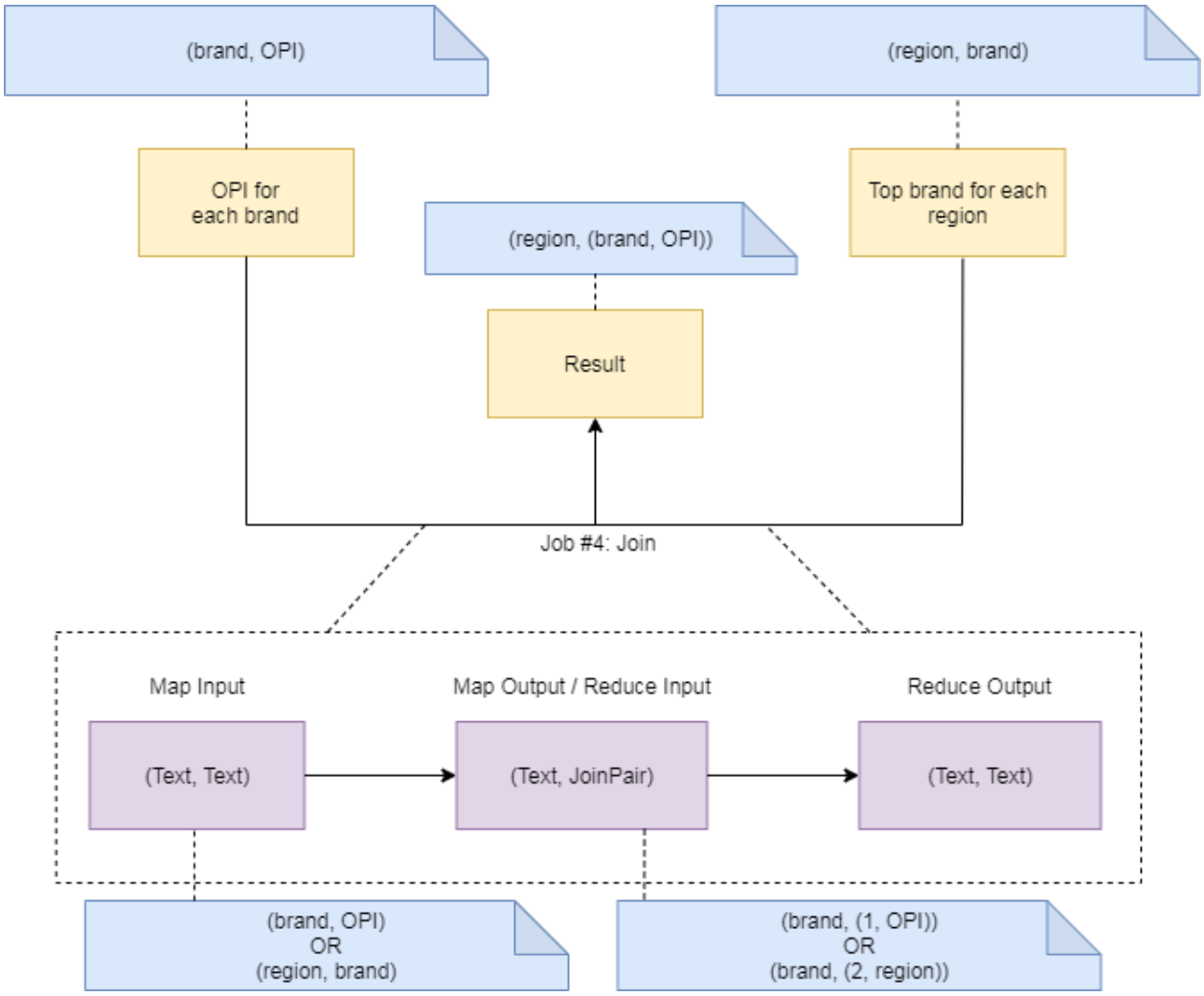


Figure 2.6: Description of "job 3: join" workflow and interfaces.

2.2 Performance evaluation

Performance are evaluated by varying the number of reducers, by enabling the use of combiners and by considering different replication factors.

Each casuistry result is the average value upon 3 runs. Results are shown in table 2.1.

Number of reducers

As we can notice, the optimal number of reducers is 5[3]: too many reducers are an overkill due to task bootstrap and to the slender amount of data; on the other hand, too few reducers can't parallelize the work in the best way.

Combiners

The usage of combiners improves performances independently from the number of reducers used. Obviously, the increase of performance decreases by incrementing the numbers of reducers. Indeed, the best increase is about 24% when using just one reducer (parallelize aggregation work that would be totally sequential otherwise).

Replication factor

By compairing results in tables 2.1 and 2.2, we can see that execution times for query on the bigger dataset are slightly higher than the other ones. Hence, if we consider that there are two orders of magnitude between the size of the two datasets (8.2MB against 820MB) , it is evident how such an architecture is quite an overkill for small datasets.

		Reducers						
		1	2	5	6	8	10	100
Combiner	No	292	244	223	235	251	280	1235
	Yes	221	215	210	226	228	270	1176
Improvement		24.3%	11.8%	5.8%	3.8%	9.1%	3.5%	4.8%

Table 2.1: Performance results expressed as elapsed time in seconds using **replication factor = 100**.

[3]Actually, a more fine-grained analysis could be done by tuning the number of reducers on sigle jobs (instead of the same for each one). This could lead to find a more efficint setup but the number of combinations is quite huge and this argument will not be covered.

		Reducers				
		1	2	5	10	100
Combiner	No	223	162	168	216	1400
	Yes	154	160	162	210	1210

Table 2.2: Performance results expressed as elapsed time in seconds using **replication factor = 1**.

Chapter 3

Spark

3.1 Job description

A unique Spark file will execute all jobs specified in section 1.2.2. The dataset is loaded as a Spark "DataFrame" and every operation will lean on "SparkSQL" layer.

Figure 3.1 shows the complete DAG corresponding to the main job created by Spark. Figure 3.2 shows Spark DAG accompained with few extra info.

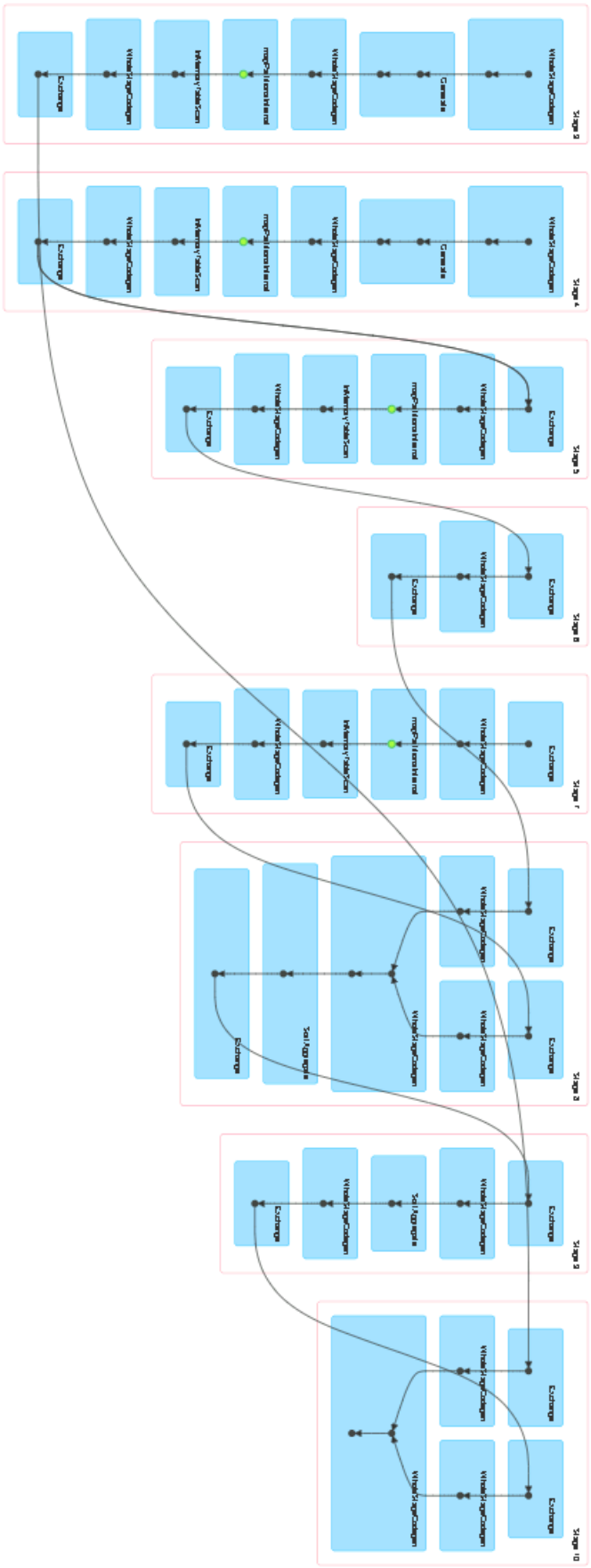
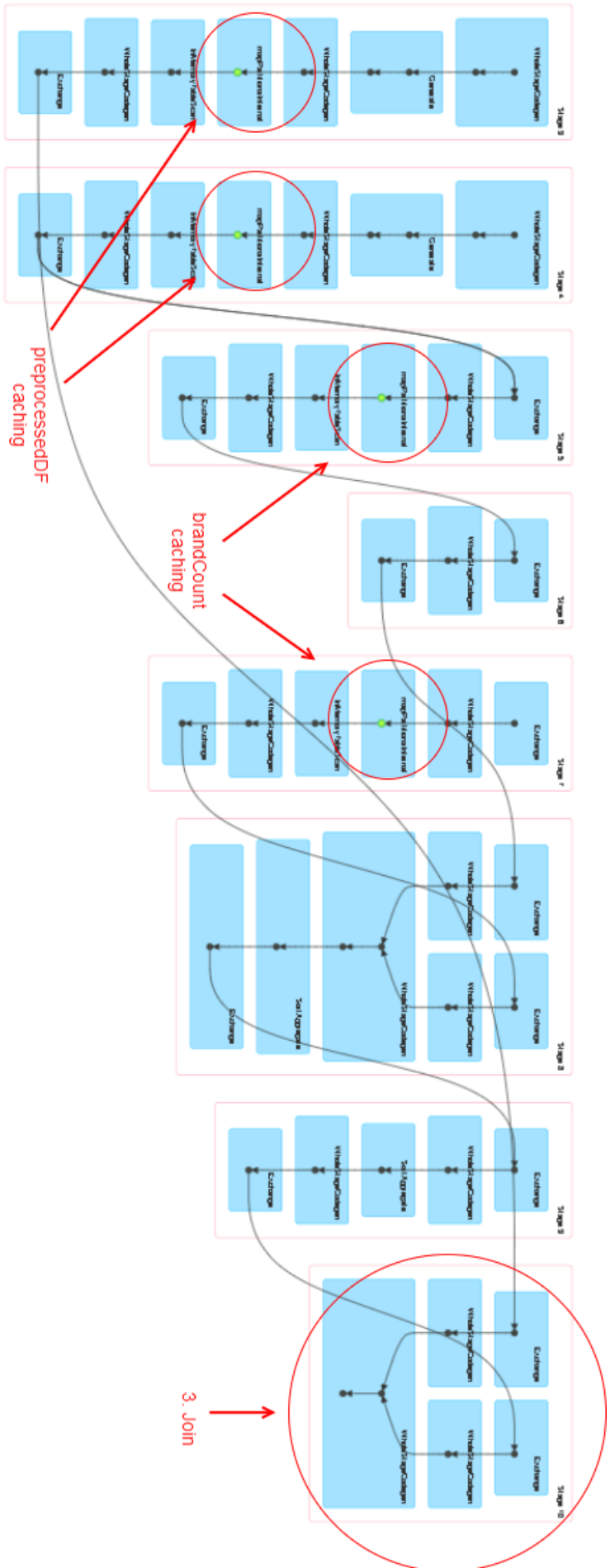


Figure 3.1: Complete Spark DAG.


preprocessedDF
caching
brandCount
caching
3. Join

Since the size of tables used in the two join operations is not excessive, the usage of the broadcast feature may be used on one table. Such an operation makes Spark split the job in three pipelined different jobs (few stages are skipped though).

The broadcast version results on an overall of 800 tasks (against the 1200 of the basic one), a more efficient usage of memory (garbage collection time) and a decrease of execution time close to 8/9%.

DAGs for the broadcast version are reported in figures 3.3, 3.4 and 3.5.

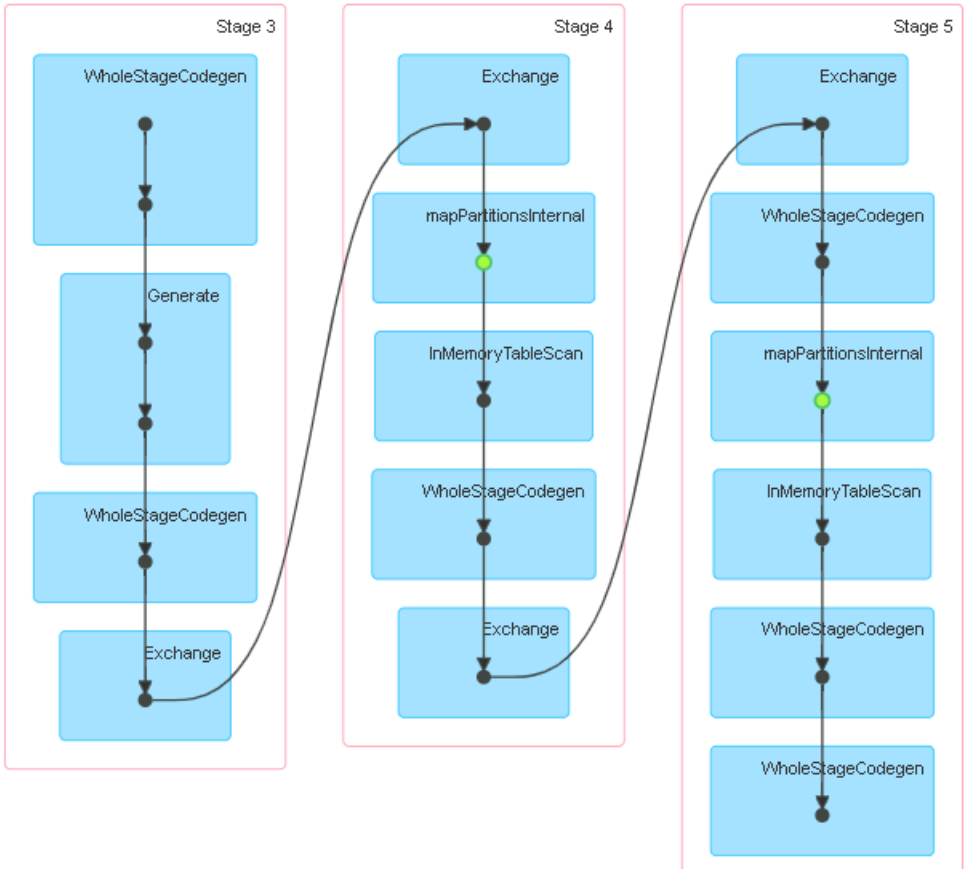


Figure 3.3: DAG for first job created by Spark when using broadcast on join.

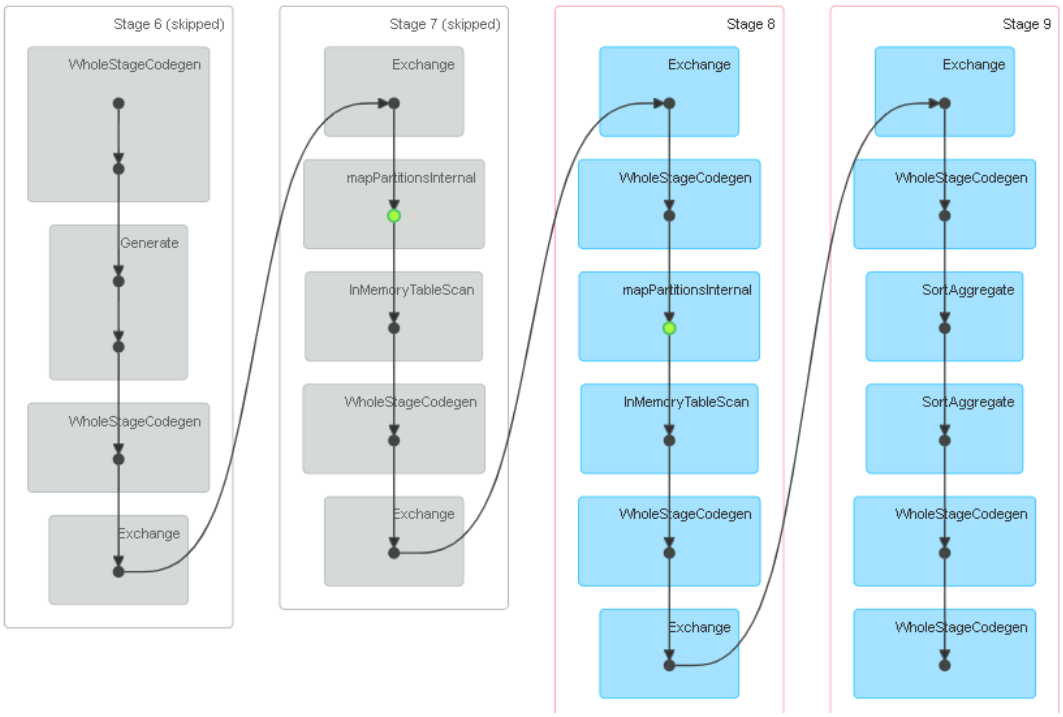


Figure 3.4: DAG for second job created by Spark when using broadcast on join.

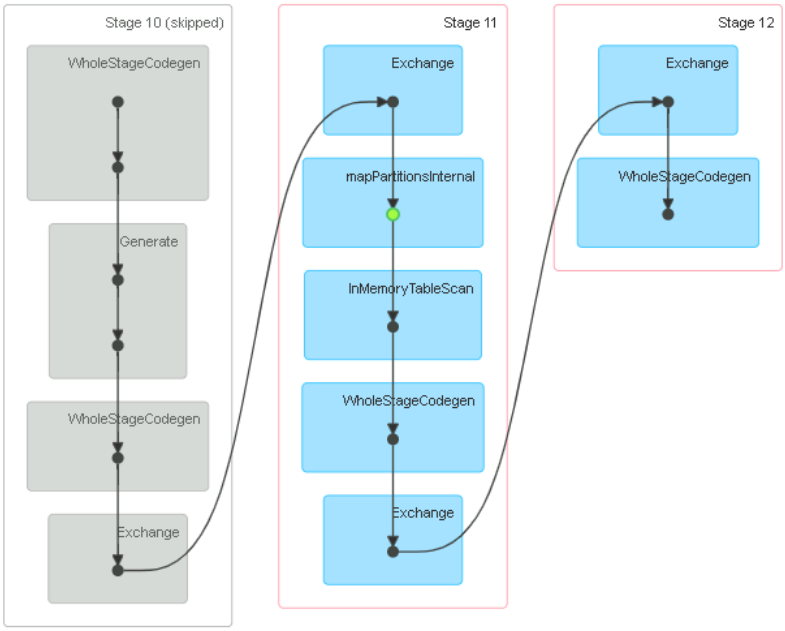


Figure 3.5: DAG for third job created by Spark when using broadcast on join.

3.2 Performance evaluation

Performance are evaluated by varying the number of executors and cores per executor.

Let's consider cluster characteristics (10 nodes, 4 cores per node) and given constraints (usage of 8 CPUs at most). Each casuistry result is the average value upon 3 runs. Results are shown in table 3.1.

As we can notice, results confirm common best practice rules: best performances among all more promising combinations are obtained using the maximum amount of resources (8 CPUs); moreover, the whole application speeds up when assigning more than one core to each executor.

		Executors				
		1	2	3	4	8
Cores	1					88
per	2			94	79	
executor	3		96			

Table 3.1: Performance results expressed as elapsed time in seconds using **replication factor = 100** for most promising combinations.

Chapter 4

Conclusions

As it could be foreshadowed, empirical results confirm theorical results and common best practice rules, that is:

- *MR vs Spark*: by using best setup found for both programs, Spark (60s ca.) performs better than MR (210s ca.).
- *Architecture overkill*: It is confirmed that architecture based on cluster, HDFS, Hadoop and so on are designed to work with huge amount of data. Indeed, by varying replication factor, the cluster scales wery well. Rather, such an architecture results to be an overkill for small datasets.
- *MR reducers and combiners*: Optimal number of reducers makes sense if considering number of nodes and amount of data; moreover, the usage of combiners in right points can improve performances.
- Spark executors: number of executors and core assigned to each one follow common best practice rules when considering available resources and cluster characteristics.