Exercise Sheet 2, 2019

6.0 VU Advanced Database Systems

Gent Rexha (11832486), Princ Mullatahiri (11846033)

08.05.2019

Exercise 1 (MapReduce)[4 Punkte]

a)

For this, we take the input and map the author to the row in the `checkouts-by-title.csv` file. After that most of the heavy lifting is being done in the reducer, where a hashmap of all the titles of the current creator is created and for each value in our values list the number of checkouts is updated for the corresponding title. After having iterated all the values, the title with the maximum amount of checkouts is written out with it's corresponding author.

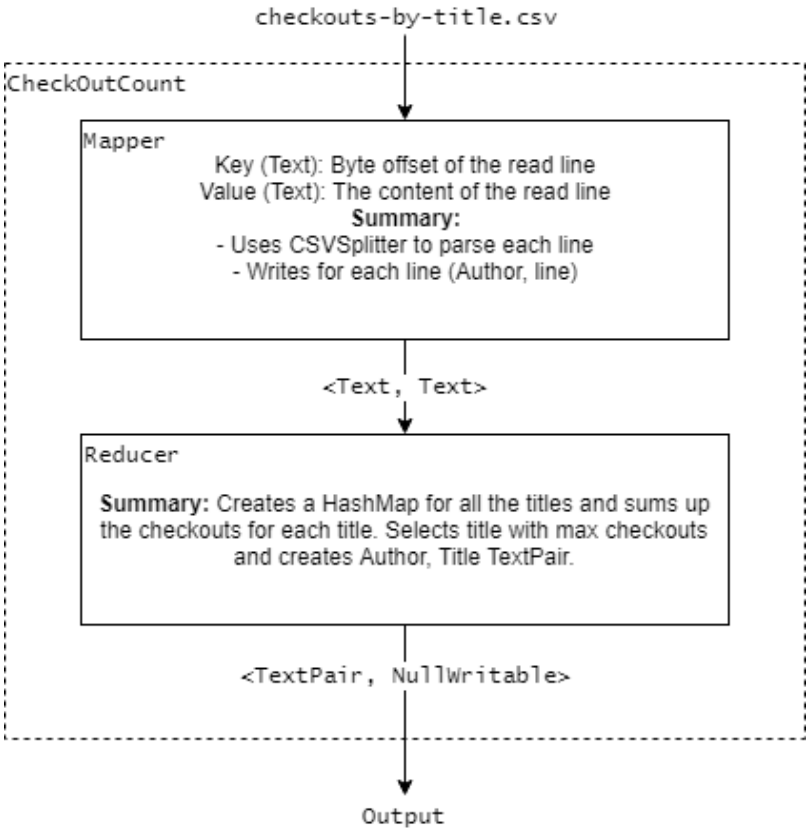

Figure 1: Diagram of the Map Reduce job to solve Task A

b)

For this task we used the MultipleInputs class. It allows you to create MapReduce jobs with multiple mapper functions, each for a different input file. The results of the mappers is then sent to the same reducer function. For the two datasets we've created two separate mappers which send the row with the author as key to the reducer. The reducer then creates two HashMaps, for both checkouts and inventory. Checkout HashMap includes all the titles and sums up the checkouts for each title. After that it selects the title with max

checkouts and creates Author, Title TextPair and adds PublicationYear, Subjects, ItemLocation from the second HashMap where all the locations for all titles of said Author are stored.

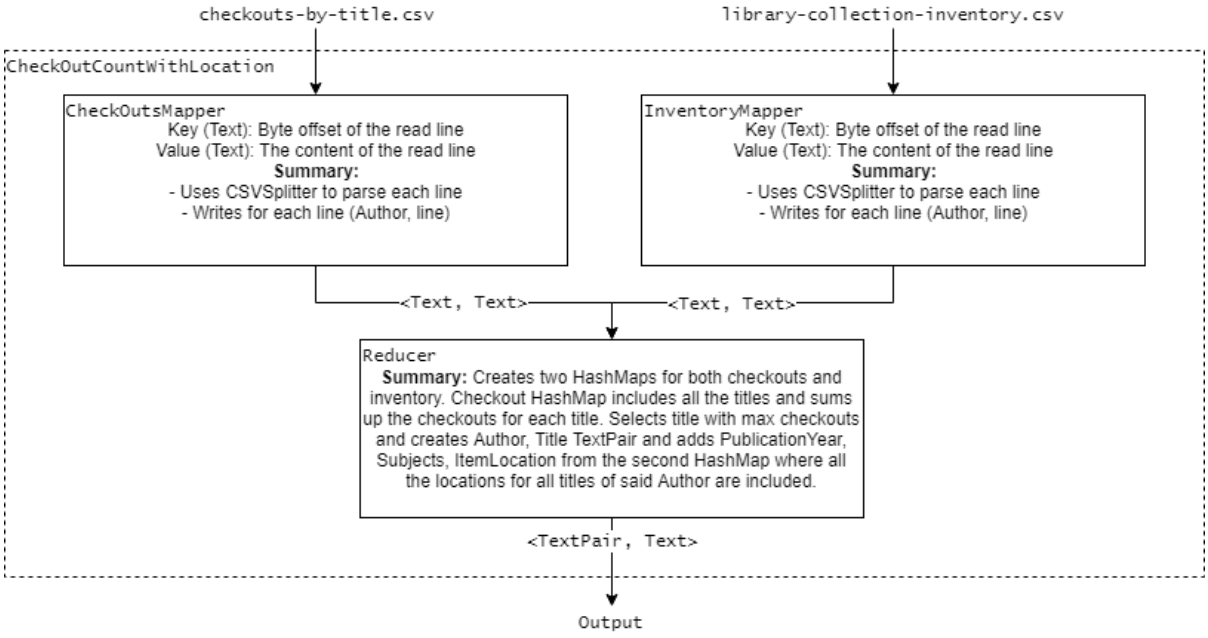

Figure 2: Diagram of the Map Reduce job to solve Task B

Exercise 2 (Costs of MapReduce)[1 Punkte]

a)

If not using a combiner, there's some skew to be expected in the Reducer considering different keys can receive different amount of values and therefore result into different processing times. For example there can be a lot of values for key_A but very few or none for key_B, resulting in more calculation for the values in key_A and therefore more processing time.

This is till dependent on the `key,value` data distribution, because in the extreme case where all of your values are of the same key, the number of reducers doesn't matter. In this case a custom partitioner might make more sense.

b)

Considering a small number of reducers, we expect more randomized key values at different reducers and a more even distribution of high processing values of one key. This also results in less parellesim of reducers and might take longer.

If the numbers of reducers is increased, the probability of only one reducer ending up with with a key,value pair where there's a lot of values is also higher, therefore increasing the skewness to a very significant level. On the other hand a maximum of parellelism is achieved, but with this comes the problem of computation overhead.

c)

If we use a combiner in the WordCount MapReduce program then we would have less computation for reducer and more for mapping phase, basicly we would calculate sum of word occurences in a document for each mapper node. And because reducer would have less work to do we do not expect to be a significant skew. The best way to avoid skew it would be using combiner and also partitioning where we determine which

reduce task receives which key and its list of values, for partitioning we could apply hash function to the key and this would guarantee that all pairs with same key go to same reducer.

d)

Communication cost is effected by input file size, sum of the sizes of all files passed from map process to reduce process and the sum of output sizes of reducer processes, so basicly it doesnt depend on the number of reducers that we have, because even if we have 1 reducer or if we have 100000 reducers we would still have the same input file size, same sum of all file sizes passed from map to reduce, and sum of all outputs from reducers.

Exercise 3 (Relational Operations)[2 Punkte]

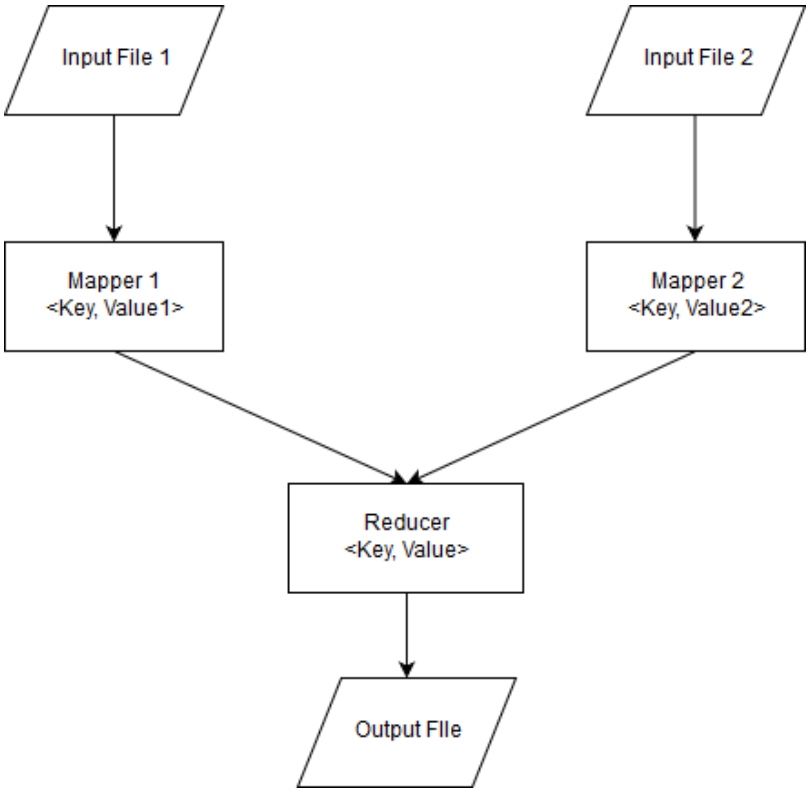

Figure 3: Mapreduce Sketch for Bag Union and Difference

a: Bag Union

Pseudo-code:

```
sum = 0
foreach val:value
    sum = sum + 1
context.write(Key, sum)
```

First we get all values from the same key in reducer, then we iterate through each value and increment sum variable, then we simply write to output key and sum as value.

b

Pseudo-code:

```
first group by table origin and seperate them in diffrent vales : value 1 and
value 2 corresponding to input file 1 and input file 2
sum_value1 = 0
sum_value2 = 0
foreach val: value 1
    sum_value1 = sum_value1 + 1
foreach val: value 2
    sum_value2 = sum_value2 + 1

if sum_value1 >= sum_value2
    context.write(Key, sum_value1 - sum_value2)

First we group by table origin, then sum up values into two different variables sum_value1
and sum_value2, then we substract these values if value is positive,
we write to output else just skip that Key

dif = 0
foreach val:value
    sum = sum + 1
context.write(Key, sum)
```

Exercise 4 (Hive Exercise)[4 Punkte]

Placeholder

Exercise 5 (Spark in Scala)[4 Punkte]

Placeholder

References