University of Augsburg, Institute of Computer Science
Prof. Dr. P. Fischer
J. Kastner, L. Rudenko

SS 2019
10. May 2019
Sheet 3

Analyzing Massive Data Sets

Exercise 1: MapReduce - Joins (live)

Map/Reduce does not provide a direct implementation of the join operator. Instead, multiple approaches have been proposed to express joins in an efficient manner. In this exercise, we will focus on equijoins, where the join predicate states equality between the join attributes.

a) The most obvious choice would be a **reduce-side join**, where the map stage just prepares data, the shuffle phase groups items with the same keys and reduce performs the actual joins. Provide the Python code for the *map* and *reduce* functions to compute **reduce-side join**.

Note: In order to avoid the complexity of setting up an actual Map-Reduce environment for the exercises, but still have consistent answers, we expect you to work with the following function signatures in Python:

```
def map(key, val): # key, val are each a single value
        return [] # A List of 2-values tuples [(k1,v1),(k2,v2),...]
def reduce (key, val): # key single value, val a list
         return [] # A List of 2-values tuples [(k1,v1),(k2,v2),...]
```

The framework performing the invocations and the processing does not have to be modelled.

b) Would it also be possible to perform a job purely on the **map side**? What conditions would have to be fulfilled, and how would the map and reduce functions look like?

c) What would be the respective benefits and disadvantages of **reduce-side** and **map-side** joins?

Exercise 2: Understanding Spark (homework)

The following sequences of Spark operations (see Figures 1, 2 and 3) are given, where A-J are RDDs with partitions (grey rectangles). The transformations to generate B-D for the first sequence, D-F for the second one and D-J for the third one are stated at each step.

- Specify and explain the type of dependencies for each transformation in the sequence.
- The nodes containing the partitions crossed out in red have failed and data was lost. In order to restore this data, these partitions need to be recomputed using lineage. Which RDDs and which partitions need be to do so? Also consider that partitions framed in green are persisted - what effect does this have on the recovery?
- Explain why in Figure 2 the join operation has **narrow** dependency on the A RDD-side and the **wide** one on the B RDD-side.

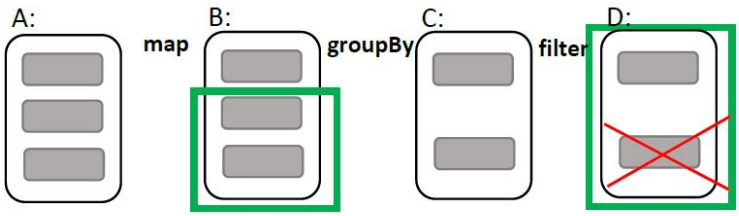

Abbildung 1: Sequence of Spark Operations a)

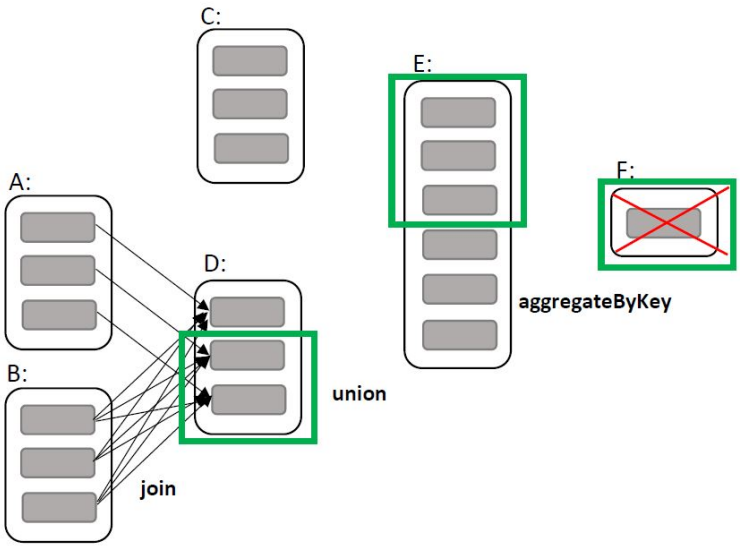

Abbildung 2: Sequence of Spark Operations b)

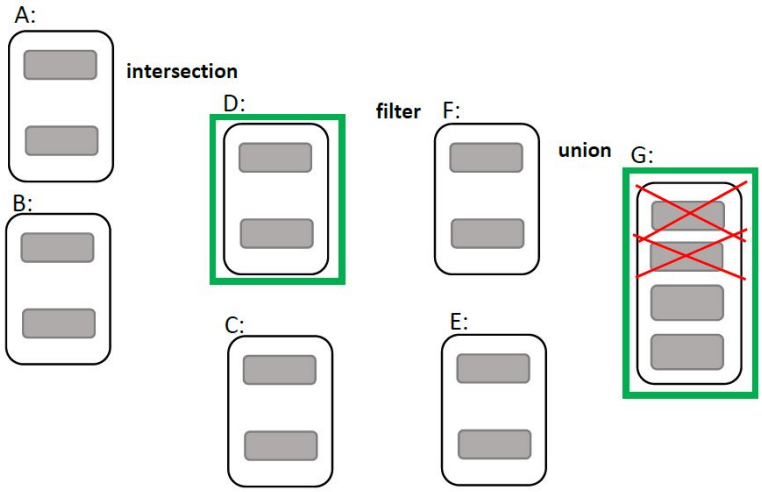

Abbildung 3: Sequence of Spark Operations c)

Exercise 3: Spark (live)

a) In this exercise we want to *find and output duplicate files* again (compare sheet 2, exercise 1). Given is a .csv-file *filehash.csv* (available in Digicampus). Each line of this file is a record that consists of <filename><md5hash>, separated by a comma. Find and report the names of duplicate files. You should report only **distinct file names**. Two files are duplicate if their md5hash values are equal.

   Solve this exercise with Spark's RDD API (documentation: Spark RDD or Pysparkling RDD). You can use *Pysparkling* to test your code locally without actually installing Spark. More information on Pysparkling is available here: Pysparkling.

b) In this exercise we want to implement *word count* using Spark's RDD. You can use the example from Pysparkling as the starting point for your solution. The example *word count*-code has to be extended with the following features:

   - All words should be turned to lower case before being counted.
   - Sort your output by the word frequency in descending order (the most common words come at the beginning).
   - Before couting, remove stop words (e.g., "the", "a", ...), as they are very frequent, but have little relevance. The list of stop words should be entered directly in the code.
   - Count the number of words (without stop words). The number of words should be outputted separately.