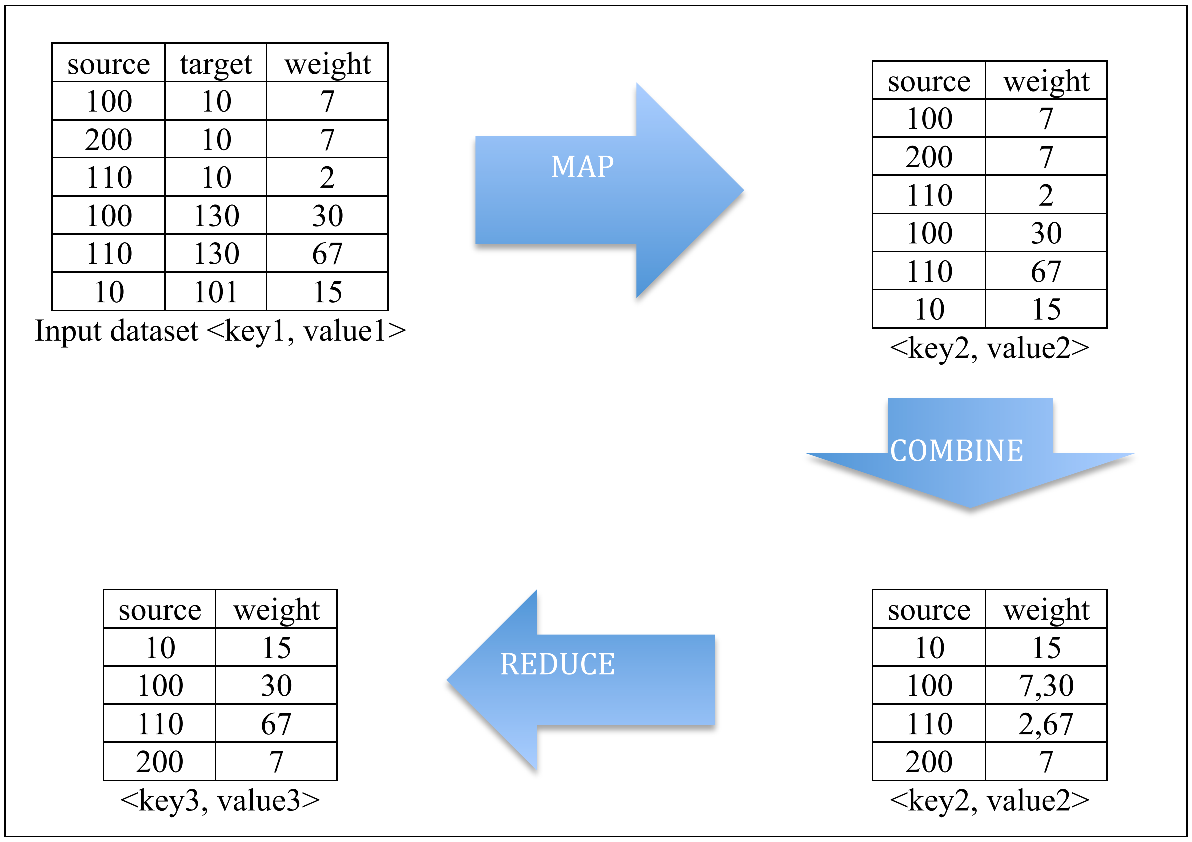
Q1a) A MapReduce procedure consists of two main steps, map and reduce processes, and operates on key-value pairs. The mapper function takes the initial <key, value> pairs from the input file. At the beginning a key is pointing to a position in the input file and each row is considered as the corresponding value <Object key, Text value>. In this example, rows data are tab separated; therefore, each row needs to be tokenized to obtain the source, target, and weight values. Moreover, the key and value variables (in this case, source and weight) should be defined IntWritable (implementing Writable interface to be serializable). Then, they will be stored as IntWritable <key, value> pairs in the context. In this way, the input data set splits into independent chunks, which can be processed by the map tasks. Then, the pairs are combined in which values are aggregated by keys and prepared as Iterable values. In this problem the combiner class is the same as the reducer. The output of this step will be the input to the next step which is reducer. In the reducer, the reduce function takes the <Iterable key, Iterable<IntWritable> value> pairs and find the largest value for each key in a for loop over each value list. For instance, for key=100, the values list is [7,30]. After finding 30 as the maximum value for this key, the output pair <100,30> will be stored in the context. All these map and reduce procedures can be processed in a completely parallel manner.



source	target	weight
100	10	7
200	10	7
110	10	2
100	130	30
110	130	67
10	101	15

Input dataset <key1, value1>

source	weight
100	7
200	7
110	2
100	30
110	67
10	15

<key2, value2>

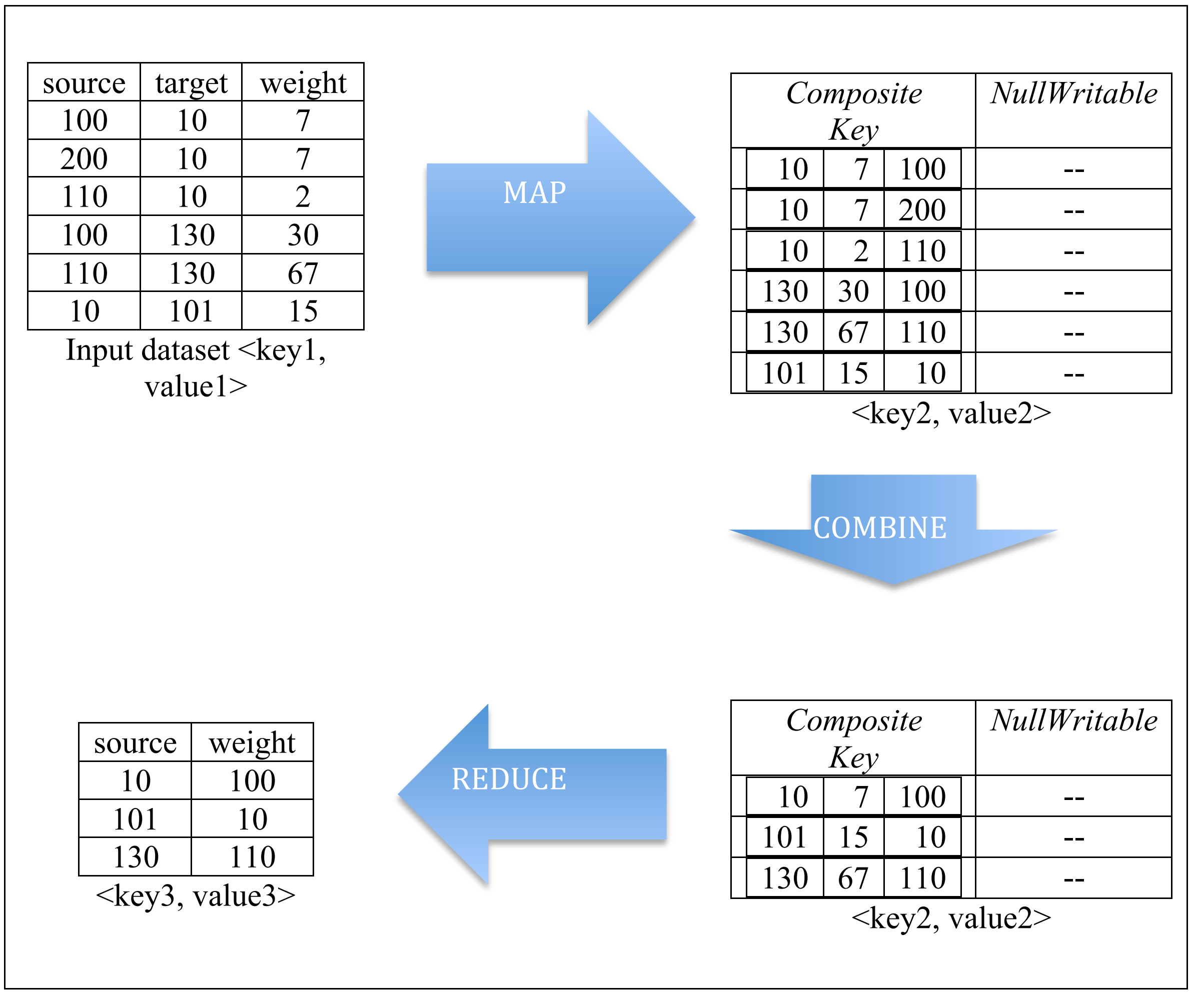
source	weight
10	15
100	7,30
110	2,67
200	7

<key2, value2>

source	weight
10	15
100	30
110	67
200	7

<key3, value3>

Q1b) In this section, the Composite Key concept has been implemented to perform the required secondary sorts. In this case, in addition to targets (as the natural key), weights and sources need to be taken into account. Therefore, instead of defining the conventional <key, value> pairs, all the input data will be stored in a composite key which consists of [target (natural key) -> weight -> sources]. The main procedure is the same as the section a; however, defining the Composite Key class and overriding the required methods are the key parts. Moreover, since all the input data need to be sorted as a composite key, the intermediate value is NullWritable type.
The key part of the Composite Key class is overriding the compareTo function (lines 91 to 103). In this function, first the input pairs are compared with the natural key (line 95), then if the natural keys (target node IDs) are equal, then the composite keys will be compared by their weights (in a descending order to pick the one with higher weight). Finally, they will be compared by the source values, if both targets and weights are equal.
The whole procedure are demonstrated in the following figure:



source	target	weight
100	10	7
200	10	7
110	10	2
100	130	30
110	130	67
10	101	15

Input dataset <key1, value1>

Composite Key			*NullWritable*
10	7	100	--
10	7	200	--
10	2	110	--
130	30	100	--
130	67	110	--
101	15	10	--

<key2, value2>

Composite Key			*NullWritable*
10	7	100	--
101	15	10	--
130	67	110	--

<key2, value2>

source	weight
10	100
101	10
130	110

<key3, value3>