Example solutions for CW1

Task1

Split-by. It splits the table horizontally. The command --*split*-by is used to specify the column of the table used to generate splits for imports. This means that it specifies which column will be used to create the *split* while importing the data into the cluster. Basically it is used to improve the import performance to achieve faster parallelism. If the actual values for the primary key are not uniformly distributed across its range, then this can result in unbalanced tasks. In this case, the --split-by command can be used to explicitly choose a different column with the `--split-by` argument. If that selected column has values that have roughly similar frequencies, then performance can be improved.

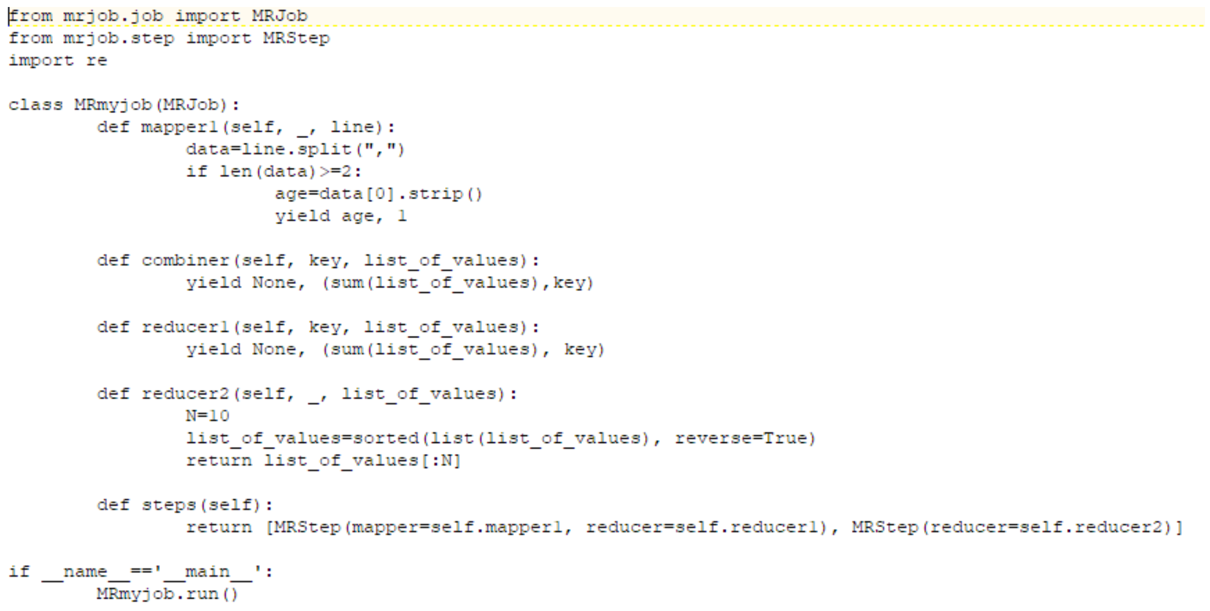
m. This parameter is a hint, and Sqoop may not use exactly the number specified. By default, the value is 4. The parameter m controls the parallelism -- how many mappers will be used e.g., in a specific import command. For example, if you are importing data from a database this parameter specifies the concurrent connections Sqoop will make to the database to pull and execute data transfer in parallel. A larger m will lead to more parallelism and complete the data transfer faster, but it will incur some overhead. By correctly setting m, we can improve efficiency.

direct. By default, Sqoop uses JDBC to connect to the database when importing or exporting data. However, depending on the database, there may be a faster, database-specific connector available, which you can use by using the --direct option.

Other possible answers are “--incremental append” or “--incremental lastmodified”, which can improve performance by not inserting an entire table, or using “query” to specify what will be imported, for example.

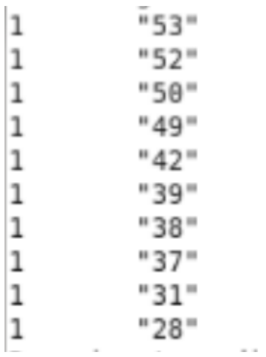
Task 2

The mapper gets as input key/value pairs. The key is empty. The value is a line of the input file. We now comment on the lines of the mapper. The first line of the mapper splits the input line into attributes, based on commas. If the length is at least 2, then we get the first attribute (word before first comma) into a variable age. Then, we return the age as key and 1 as value. Then, we use a combiner. The combiner gets as input the key/value pairs of the mapper and returns an empty key (None) and a value that is a tuple. The first element of the tuple is the sum of the input values. The second element of the tuple is the key of the input key/value pairs. The first reducer, reducer1, is similar to the combiner. The second reducer gets as input key/value pairs with empty key and value the tuple of reducer1. It first initializes a variable N to 10, because we are interested in top-10 values. Then, it sorts the list of values that is provided as input with reverse=True (to start from the most frequent elements), gets the output into list_of_values and returns this list of values. The steps function tells MRJob to have two tasks; the first has mapper1 as its mapper and reducer1 as its reducer, while the second one has empty mapper and reducer2 as reducer. The last lines execute the run method of the Mrmyjob, to execute the class.



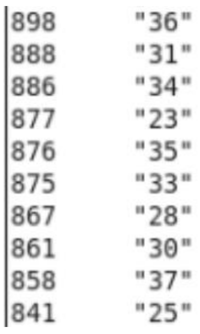
```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

class MRmyjob(MRJob):
        def mapper1(self, _, line):
                data=line.split(",")
                if len(data)>=2:
                        age=data[0].strip()
                        yield age, 1

        def combiner(self, key, list_of_values):
                yield None, (sum(list_of_values),key)

        def reducer1(self, key, list_of_values):
                yield None, (sum(list_of_values), key)

        def reducer2(self, _, list_of_values):
                N=10
                list_of_values=sorted(list(list_of_values), reverse=True)
                return list_of_values[:N]

        def steps(self):
                return [MRStep(mapper=self.mapper1, reducer=self.reducer1), MRStep(reducer=self.reducer2)]

if __name__=='__main__':
        MRmyjob.run()
```

Output in 10 first lines



```
1       "53"
1       "52"
1       "50"
1       "49"
1       "42"
1       "39"
1       "38"
1       "37"
1       "31"
1       "28"
```

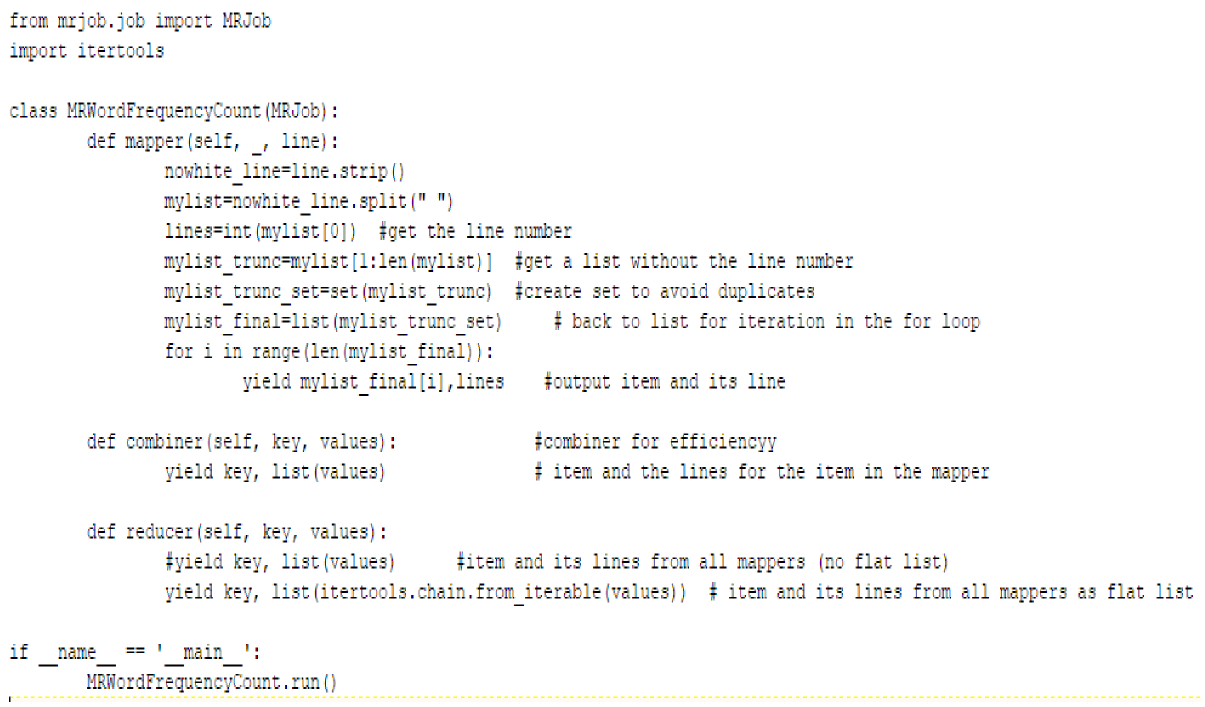
Output in entire file



```
898     "36"
888     "31"
886     "34"
877     "23"
876     "35"
875     "33"
867     "28"
861     "30"
858     "37"
841     "25"
```

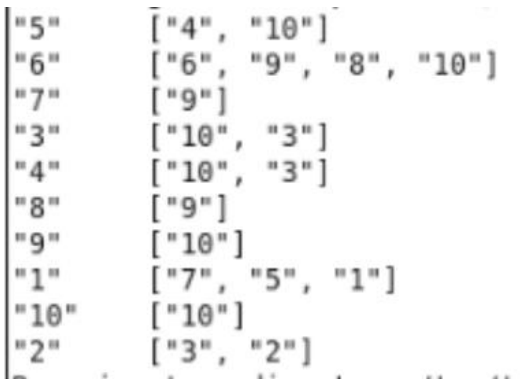
Task 3

The mapper gets as input key/value pairs. The key is empty. The value is a line of the input file. We now comment on the lines of the mapper. The first line strips white characters. The second line splits the input line “line” into words. The next line gets the line number (first attribute/word of the input line) and stores it into lines. The next line creates a set my_list_trunc_set. This is used to avoid duplicate line numbers. The next line converts the set back to a list, and the for look goes over the list an yields a key/value pair with key equal to the element and value equal to the line number. Then, we use a combiner for efficiency. The combiner gets as input a key/value pair, which contains the key and value from the mapper. It returns a key/value pair where the key is the same as the key of the input key/value pair, and the value is a list of values. This list contains all the line numbers of the key. The reducer gets as input key/value pairs from the combiner and outputs key/value pairs, with key equal to the key of the combiner and value a list. The first (commented out) line returns key/value pair where the value is a list that is not flat. The second line returns a key/value pair where the value is a list that is flat -- this is done with itertools.chain.from_iterable(). Both are correct. The last lines execute the run method of the MRWordFrequencyCount, to execute the class.



```
from mrjob.job import MRJob
import itertools

class MRWordFrequencyCount(MRJob):
        def mapper(self, _, line):
                nowhite_line=line.strip()
                mylist=nowhite_line.split(" ")
                lines=int(mylist[0])  #get the line number
                mylist_trunc=mylist[1:len(mylist)]  #get a list without the line number
                mylist_trunc_set=set(mylist_trunc)  #create set to avoid duplicates
                mylist_final=list(mylist_trunc_set)     # back to list for iteration in the for loop
                for i in range(len(mylist_final)):
                        yield mylist_final[i],lines    #output item and its line

        def combiner(self, key, values):            #combiner for efficiencyy
                yield key, list(values)             # item and the lines for the item in the mapper

        def reducer(self, key, values):
                #yield key, list(values)      #item and its lines from all mappers (no flat list)
                yield key, list(itertools.chain.from_iterable(values))  # item and its lines from all mappers as flat list

if __name__ == '__main__':
        MRWordFrequencyCount.run()
```

Output in first 10 lines



```
"5"      ["4", "10"]
"6"      ["6", "9", "8", "10"]
"7"      ["9"]
"3"      ["10", "3"]
"4"      ["10", "3"]
"8"      ["9"]
"9"      ["10"]
"1"      ["7", "5", "1"]
"10"     ["10"]
"2"      ["3", "2"]
```