Multiple MR Jobs

In the context of distributed computing and big data processing, MapReduce is a programming model that allows for processing and generating large datasets in parallel across a distributed cluster of computers. A typical MapReduce job consists of two main phases: the Map phase and the Reduce phase.

If your data processing task is complex and cannot be accomplished with a single MapReduce job, you may need to chain or sequence multiple MapReduce jobs. Each MapReduce job takes input data, performs a specific operation, and produces output data that can be used as input for the next job.

Here's a general overview of how you might use multiple MapReduce jobs:

First MapReduce Job (Job1): This job performs an initial processing step on the input data, and the output is stored in an intermediate location.

Intermediate Processing (Optional): Depending on the complexity of your task, you might perform additional processing or transformations on the output of the first job before proceeding to the next step. This could involve non-MapReduce processing.

Second MapReduce Job (Job2): This job takes the output of the previous job as input and performs another set of operations. The final output is typically the result of your overall processing task.

Final Processing (Optional): If necessary, you may perform any final processing or analysis on the output of the second job.

This sequence can be extended to include more MapReduce jobs as needed for your specific data processing requirements.

Keep in mind that newer technologies, like Apache Spark, have become popular for big data processing due to their ability to handle more complex workflows and iterative processing more efficiently than traditional MapReduce.

Task 1: For the word count problem, Sort the words by their frequency by using multiple MR steps.

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re
WORD_REGEXP = re.compile(r"[\w']+")

class MRWordFrequencyCount(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   reducer=self.reducer_count_words),
            MRStep(mapper=self.mapper_make_counts_key,
                   reducer = self.reducer_output_words)
        ]
    def mapper_get_words(self, _, line):
        words = WORD_REGEXP.findall(line)
        for word in words:
            yield word.lower(), 1

    def reducer_count_words(self, word, values):
        yield word, sum(values)

    def mapper_make_counts_key(self, word, count):
        yield str(count).rjust(4,'0'), word

    def reducer_output_words(self, count, words):
        for word in words:
            yield count, word

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

Task 2: Write the MR job for multiplication of matrices

Step1: Given two matrices (A, B), we need to convert the matrices into a text file.

M,row,column,value

A 0 0 1

A 0 1 2

A 1 0 3

A 1 1 4

B 0 0 10

B 0 1 20

B 1 0 30

B 1 1 40

Step 2: Then program the mapper and reducer functions

```
from mrjob.job import MRJob
from mrjob.step import MRStep
class MatrixMultiplicationJob(MRJob):
	def steps(self):
		return [MRStep(mapper=self.mapper, reducer=self.reducer)]

	def mapper(self, _, line):
		matrix, row, col, value = line.strip().split()
		row, col, value = int(row), int(col), int(value)

		if matrix == 'A':
			for k in range(2):
				yield (row, k), ('A', col, value)
		elif matrix == 'B':
			for k in range(2):
				yield (k, col), ('B', row, value)

	def reducer(self, key, values):
		result = 0
		a_values = {}
		b_values = {}

		for matrix, index, value in values:
			if matrix == 'A':
				a_values[index] = value
			elif matrix == 'B':
				b_values[index] = value
		for k in range(2):
			result += a_values.get(k) * b_values.get(k)
		yield key, result
```

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