CSE 587 Spring 2023

Name: Pavana Lakshmi Venugopal

UBIT: pavanala

Analysis

Answer the following questions based on your WordCount application.

1. In the PySpark REPL, run your basic word count program on a single text file.
    a. What are the 25 most common words? Include a screenshot of program output to back-up your claim.
    b. How many stages is execution broken up into? Explain why. Include a screenshot of the DAG visualization from Spark's WebUI to back-up your claim.

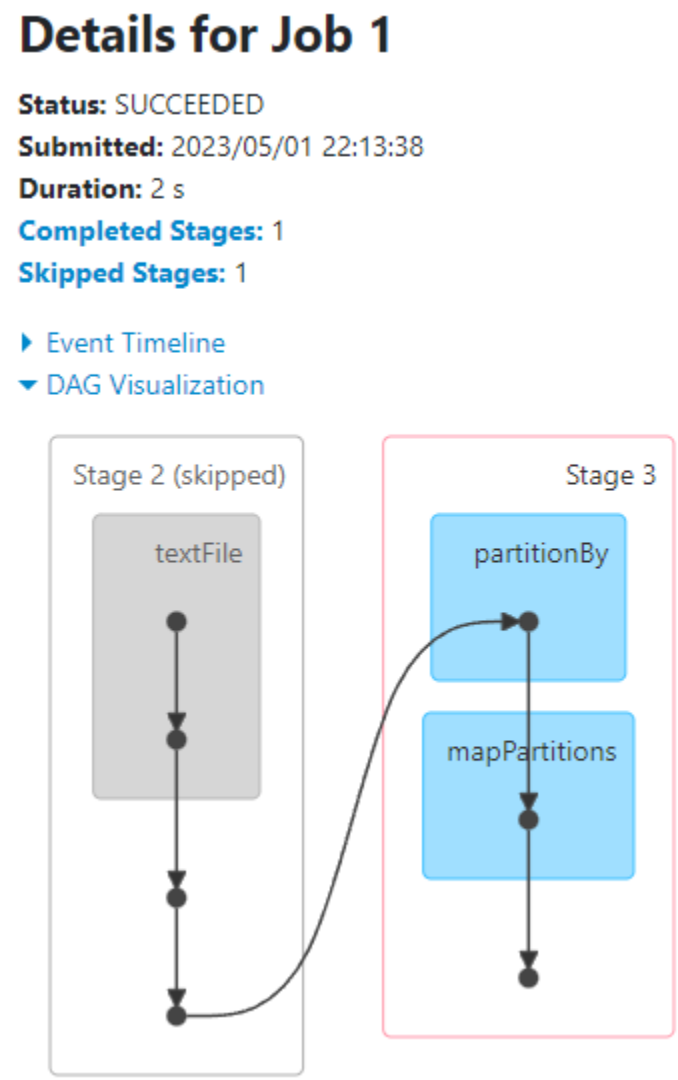
Solution:

```
def countWords(sc):
    import re, nltk

    from nltk.corpus import stopwords

    stopwords = stopwords.words('english')

    file = 'book1.txt'

    lines = sc.textFile(file)
    counts = lines.flatMap(lambda x: x.split(' ')) \
            .map(lambda x: re.sub(r'[^\w\s]', '', x)) \
            .map(lambda x: (x.lower(),1)) \
            .filter(lambda x: x[0] not in stopwords + ['']) \
            .reduceByKey(lambda a,b: a+b) \
            .sortBy(lambda pair: pair[1], ascending=False)

#     counts.saveAsTextFile("C:/DIC-Project/HW2/output")
    return counts

if __name__ == '__main__':
    from pyspark.context import SparkContext
    sc = SparkContext('local', 'test')

    results.saveAsTextFile("C:/DIC-Project/HW2/output")
```

a) The 25 most common words are shown below.

```
('co', 3323)
('biograph', 1949)
('edison', 1827)
('american', 1714)
('mutoscope', 1641)
('thomas', 1363)
('scenes', 844)
('mfg', 574)
('14', 566)
('lubin', 514)
('lc', 483)
('america', 438)
('vitagraph', 429)
('inc', 379)
('new', 341)
('york', 205)
('méliès', 189)
('see', 184)
('love', 173)
('parade', 155)
('scene', 154)
('fight', 148)
('city', 146)
('picture', 145)
('war', 145)
```

b) The execution is broken into 2 stages.



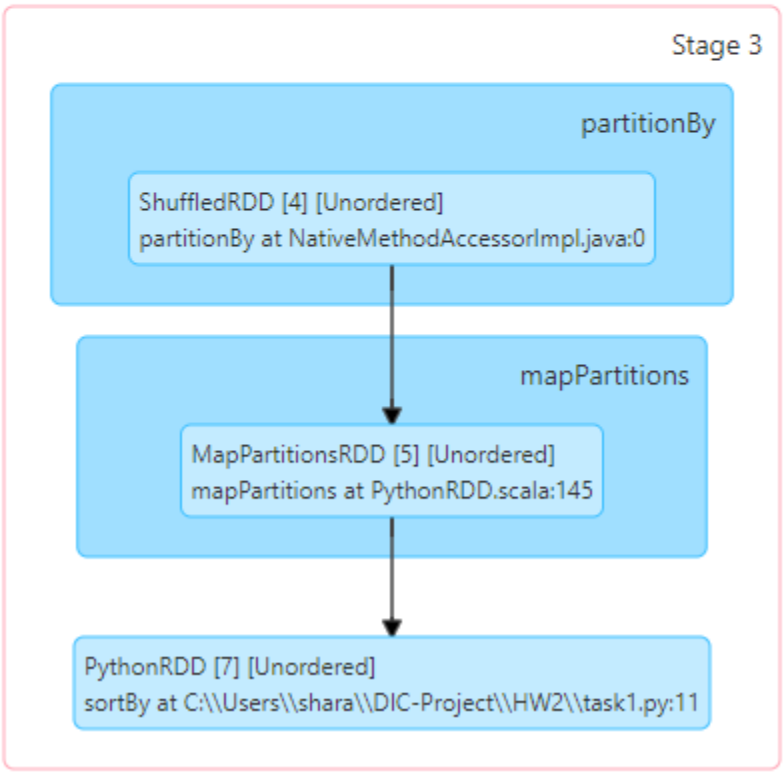
Stage2 : Started with book1.txt. Spark uses in memory caching. It would have the output of those stages already, so it skipped it. Basically this is for reading the text file using `sc.textFile(file)` creates a RDD that needs to be stored in memory and partitioned for parallel processing, thereby creating the first stage of the execution.

Stage 3: It basically picks up where it left off. All this stage really does is shuffle the data around and remap the partitions. The transformations "flatMap," "map," and "filter" are applied to the RDD produced in the previous stage. This stage includes the `reduceByKey` and `sortBy` transformations and the take action on the RDD. The transformations are not executed immediately but are instead evaluated and executed together when they are needed. Then, using the 'sortBy' transformation to order the resulting word counts in descending order, see the top 25 frequently occurring words.

Details for Stage 3 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 3 s
Locality Level Summary: Node local: 2
Shuffle Read Size / Records: 170.1 KiB / 56
Associated Job Ids: 1

▼ DAG Visualization



2. In the PySpark REPL, run your extended word count program on all 10 text files.
    a. What are the 25 most common words? Include a screenshot of program output to back-up your claim.
    b. How many stages is execution broken up into? Explain why. Include a screenshot of the DAG visualization from Spark's WebUI to back-up your claim.

Solution:

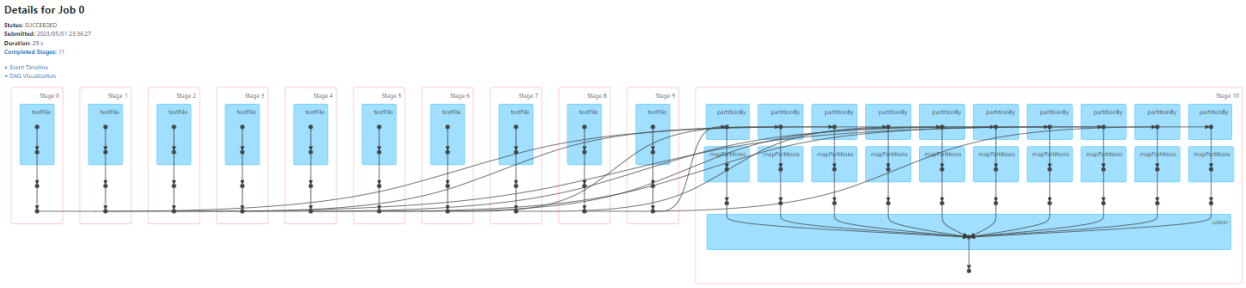
a)

```
def task2(sc, files):
    import re, nltk

    from nltk.corpus import stopwords

    stopwords = stopwords.words('english')

    output = None

    counts_list = []


    count = [sc.textFile(file) \
        .flatMap(lambda x: x.split(' ')) \
        .map(lambda x: re.sub(r'[^\w\s]', '', x)) \
        .map(lambda x: (x.lower(),1)) \
        .filter(lambda x: x[0] not in stopwords + ['']) \
        .reduceByKey(lambda a,b: a+b) for file in files]


    counts = sc.union(count) \
        .sortBy(lambda pair: pair[1], ascending=False)

#     counts.saveAsTextFile("C:/DIC-Project/HW2/output")
    return counts

if __name__ == '__main__':
    from pyspark.context import SparkContext
    sc = SparkContext('local', 'test')

    results = task2(sc, ['book1.txt', 'book2.txt', 'book3.txt', 'book4.txt', 'book5.txt','book6.txt', 'book7.txt', 'book8.txt', 'book9.txt',
'book10.txt'])

    results.saveAsTextFile("C:/DIC-Project/HW2/output2")
```

The 25 most common words are as shown below.

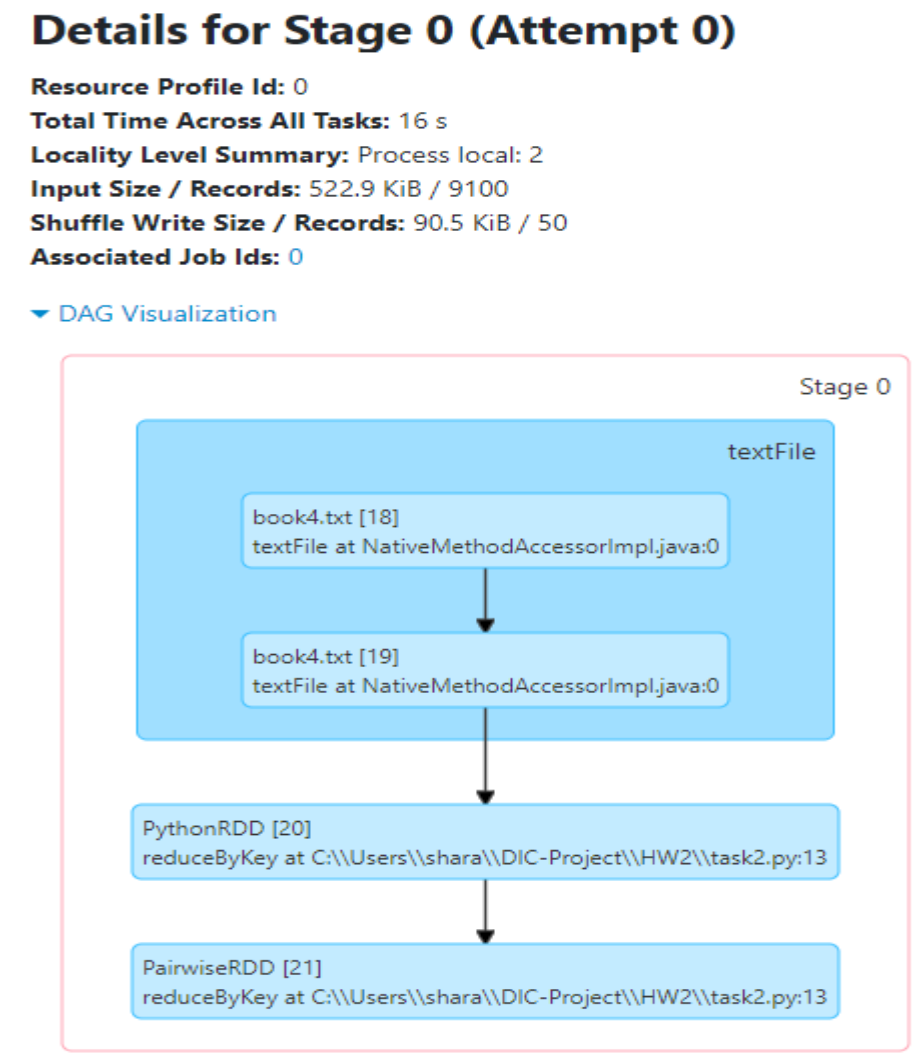
```
('co', 3323)
('biograph', 1949)
('edison', 1827)
('american', 1714)
('mutoscope', 1641)
('thomas', 1363)
('mrs', 958)
('scenes', 844)
('mr', 805)
('upon', 785)
('fig', 754)
('plant', 747)
('plants', 732)
('leaves', 675)
('one', 640)
('elizabeth', 603)
('water', 597)
('mfg', 574)
('14', 566)
('cells', 564)
('would', 560)
('could', 531)
('said', 529)
('illustration', 519)
('would', 517)
```

b) The execution is broken down into 11 stages.

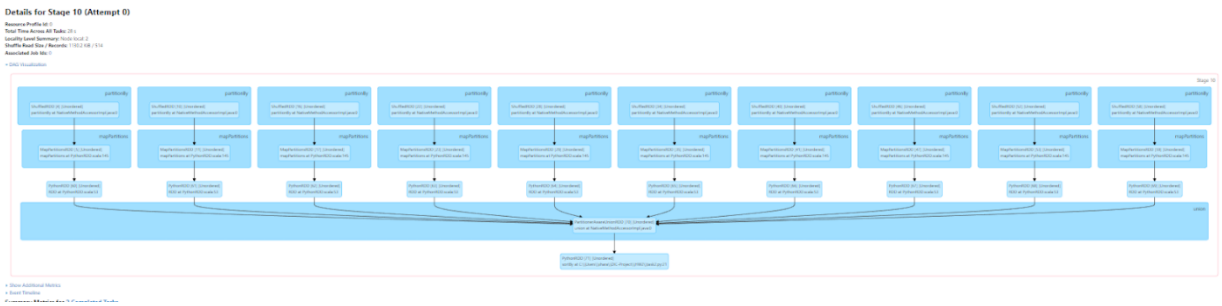


Stage0 to Stag9 : Reads all text files. Spark uses in memory caching. It would have the output of those stages already, so it skipped it. Basically this is for reading the text file using `sc.textFile(file)` creates a RDD that needs to be stored in memory and partitioned for parallel processing, thereby creating the first

stage of the execution. This transformation takes a line of text as input and separates it into individual words, resulting in a list of words. (flatmap)



Stage10: Each word is changed to lowercase and all punctuation is removed in this transition. (map). This transformation counts the number of occurrences of each word. (reduceByKey). It includes take action on the RDD.



3. Your WordCount application should compute the same results as your WordCount application from Homework #1. Answer the following based on your knowledge of both MapReduce and Spark:
    a. If you were running your WordCount programs in a large cluster or cloud environment, and one of the nodes you were running on died mid computation, how would your MapReduce and Spark programs handle this?
    b. Explain one concrete benefit you experienced when writing the Spark version of WordCount compared to the MapReduce version.

Solution:

a) MapReduce offers linear scalability and fault tolerance for processing very large data sets
Spark maintains this revolutionary approach brought about by MapReduce.

In MapReduce, if a node fails during computation, the framework identifies the failed task and schedules its re-execution on another available node. The fact that the data is copied and stored on HDFS makes this possible. The intermediate results that were generated by the node that failed are recovered using the Hadoop Distributed File System (HDFS).

Sparks fault tolerance involves recovering partitions when there is a failure. In Spark, a node failure can be handled by its lineage mechanism. Each RDD's lineage, which shows the data transformations carried out on the RDD, is tracked by Spark. If a partition of a Resilient Distributed Dataset (RDD) is lost due to a node failure, the lineage of the RDD can be used to recompute the lost partition. Spark performs recovery automatically and ensures that the transformed RDDs are reprocessed only when needed, thereby minimizing the computation cost.

b) Spark transitions seamlessly between exploratory analytics and operational analytics.

Related to productivity, we saw wordcount in MapReduce had multiple java classes we had to define, we had a configuration to setup. Our mappers were very finely grained, explaining to read the file, split that into words go through each word, emit a key value pair and the reducer will take those key value pairs and it will reduce them into a single key value pair.

```java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  // Mapper
  // takes the text, tokenize the words
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      // tokenize the lowercased text string
      String line = value.toString().toLowerCase(); // convert line to lowercase and to string variable
      StringTokenizer itr = new StringTokenizer(line, "!\"#$%&()*+,-./:;<=>?@[]^_`{|}~ "); // remove possible punctuations
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

```java
// Reducer
// counts the number of times the word is in data and writes the word and sum as key value pair
public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}

public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "word count");
  job.setJar("WordCount.jar");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  job.waitForCompletion(true);
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

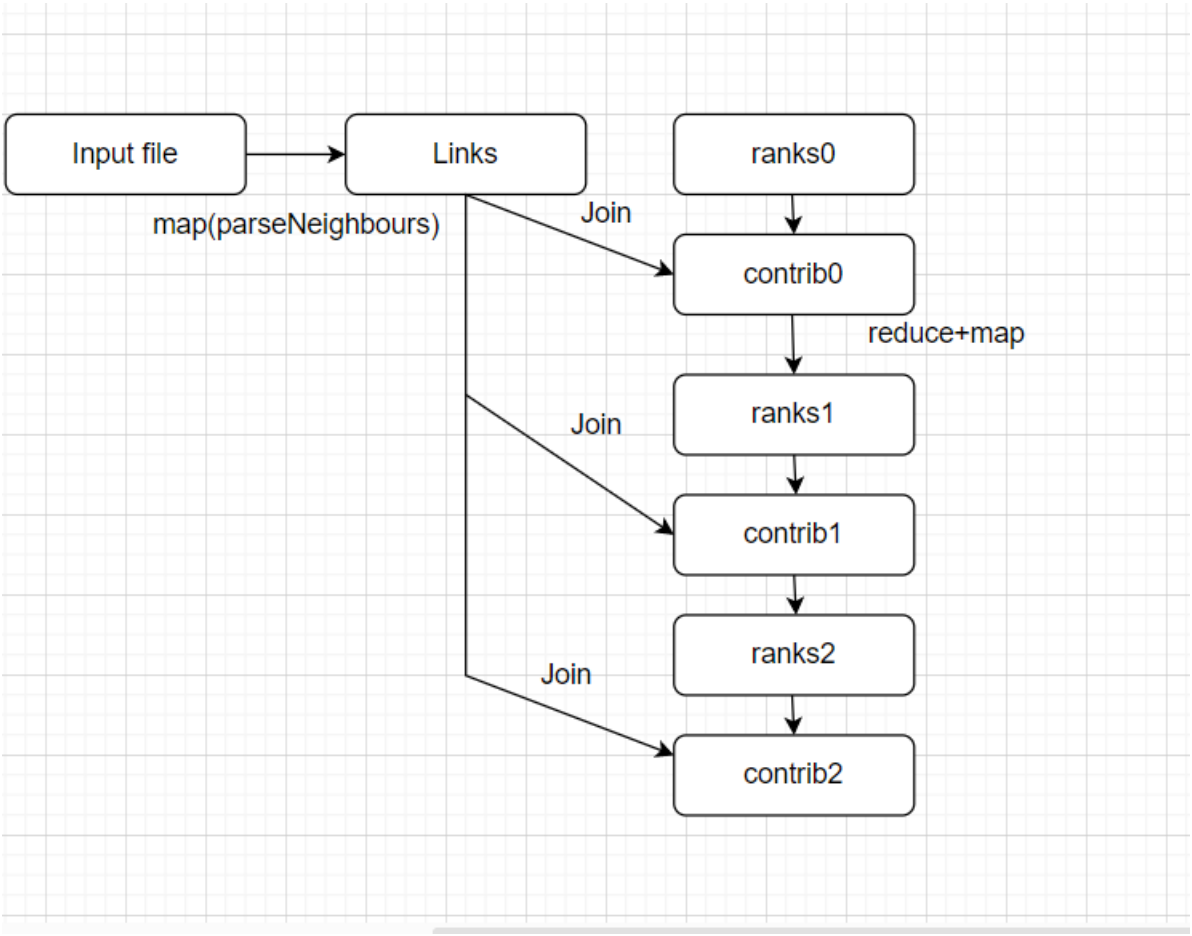
In Spark its very streamlined into something that much more resembles something like a more basic script. There is again a series of operations like flat map, map and reduce by key. How those operate on data, how the data is communicated and how that data is distributed all that is handled by Spark for us.

```python
def countWords(sc, file):
    import re, nltk

    nltk.download('stopwords')

    from nltk.corpus import stopwords

    stopwords = stopwords.words('english')


    lines = sc.textFile(file)
    counts = lines.flatMap(lambda x: x.split(' ')) \
              .map(lambda x: re.sub(r'[^\w\s]', '', x)) \
              .map(lambda x: (x.lower(),1)) \
              .filter(lambda x: x[0] not in stopwords + ['']) \
              .reduceByKey(lambda a,b: a+b) \
              .sortBy(lambda pair: pair[1], ascending=False) \
              .take(25)
    return counts

if __name__ == '__main__':
    from pyspark.context import SparkContext
    sc = SparkContext('local', 'test')

    counts = countWords(sc, "*.book1.txt")

    for word, count in counts:
        print("{} : {}".format(word, count))
```

Also, the Spark implementation of WordCount is more brief and readable than the MapReduce implementation, while also providing better performance due to its ability to store intermediate data in memory and perform operations in parallel.

4. Given the above spark application, draw the lineage graph DAG for the RDD **`ranks`** on line 12 when the iteration variable **`i`** has a value of 2. Include nodes for all intermediate RDDs, even if they are unnamed.

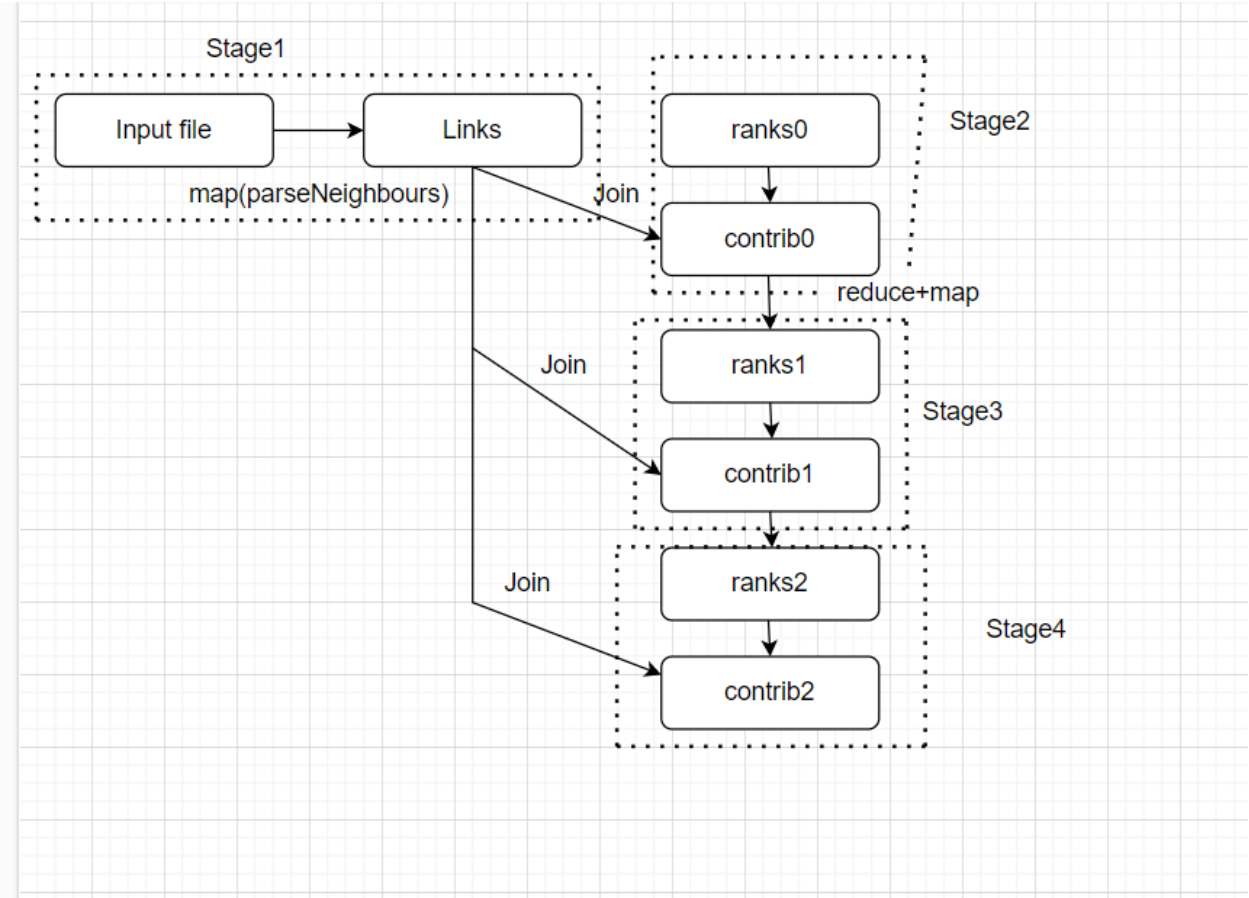
Solution:



5. How many stages will the above DAG be broken into? Give the number of stages AND draw stage boundaries on your diagram.

Solution:

Above DAG will be broken down into 4 stages.



6. Identify in the above code (by function name AND line number) one instance of:
    a. A transformation that results in a wide dependency
    b. A transformation that results in a narrow dependency
    c. A transformation that may result in a narrow dependency OR a wide dependency
    d. An action

Solution:

a) groupByKey()

Line 3

b) map(func)

Line 2, Line 6

c) join()

Line 9

d) count()

Line 5

7. How many "jobs" will the above code run if **`iters`** has value 10?

Solution:

Number of jobs will still be the same that is one.

8. What algorithm is the above code an implementation of?

Solution:

The above code is an implementation of Page rank algorithm.