REPORT

1)

- **Did you receive any help whatsoever from anyone in solving this assignment?**
  No

- **Did you give any help whatsoever to anyone in solving this assignment?**
  No

2) Consider the code below:

```
def tokenize(doc):
        doc = doc.strip()
        words = doc.split(' ')
        yield (words[0], words[1:])

def extractCounts(row):
        word = row[0]
        fgCount = 0
        bgCount = 0
        for counts in row[1]:
                if counts[0] == '1960':
                        fgCount += int(counts[1])
                else:
                        bgCount += int(counts[1])
        result = word + ", fg=" + str(fgCount) + ", bg=" + str(bgCount)
        yield result

class Phrase(Planner):
        doc = ReadLines('input.txt')
        unigram = Flatten(doc, by=tokenize)
        phrasing = Group(unigram, by=lambda w:w[0], retaining=lambda w:w[1])
        output = Flatten(phrasing, by=extractCounts)
```

There are 3 steps involved in generating the foreground and background counts using guinea pig.

- Stream through the input data and flatten the document. In this step get the word count

for each decade. The output of this step is as below:

```
('apple', ['1960', '60'])
('apple', ['1970', '79'])
('apple', ['1980', '934'])
('apple', ['1990', '3875'])
('banana', ['1960', '3'])
('banana', ['1970', '2'])
('banana', ['1990', '10'])
('carrot', ['1970', '3'])
('carrot', ['1980', '484'])
('carrot', ['1990', '492'])
```

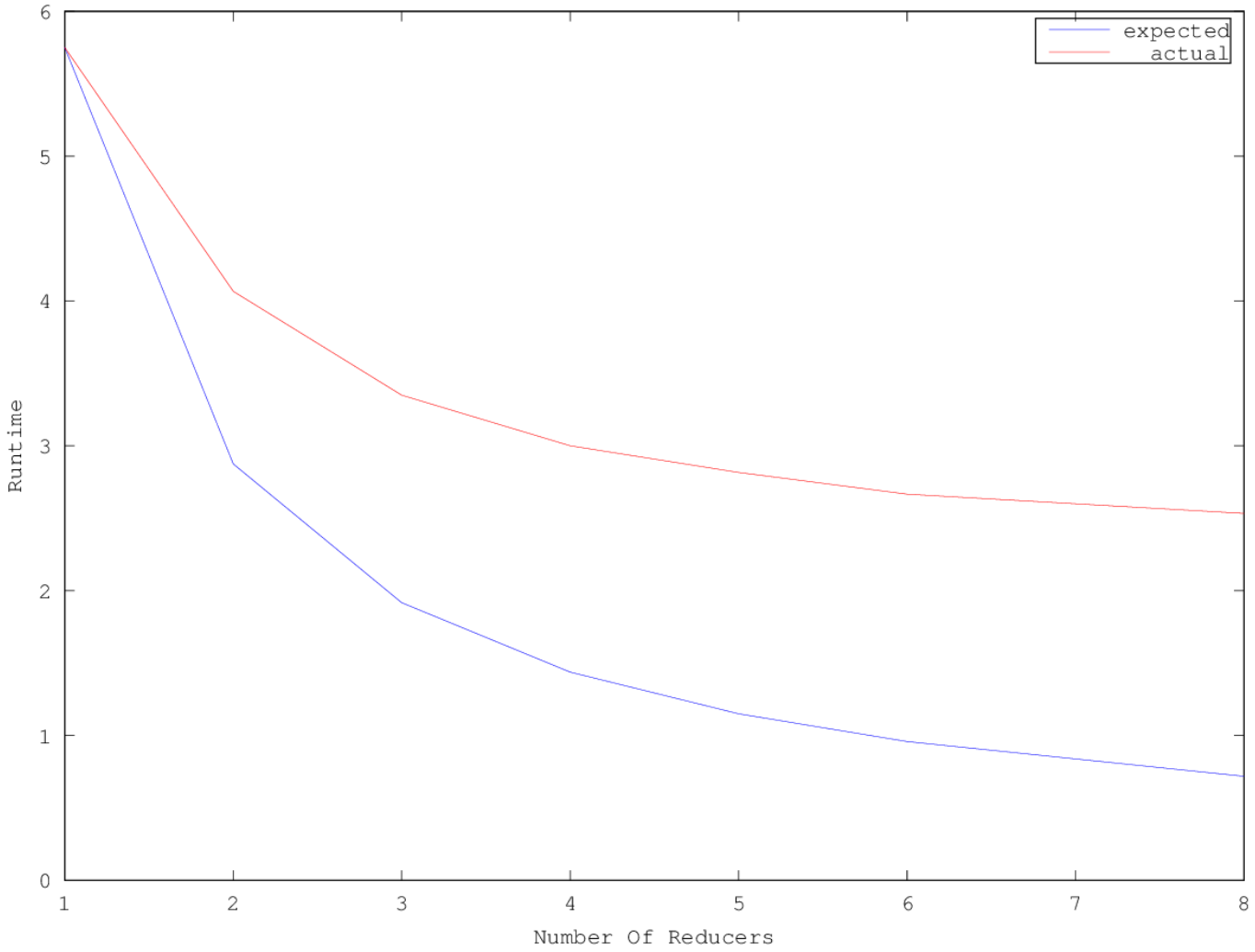
- Group the data based on the unigram, so that all the counts related to that unigram are in 1 place. Output of this step is as below:

```
('apple', [['1960', '60'], ['1970', '79'], ['1980', '934'], ['1990',
'3875']])
('banana', [['1960', '3'], ['1970', '2'], ['1990', '10']])
('carrot', [['1970', '3'], ['1980', '484'], ['1990', '492']])
```

- Now to calculate the foreground and background counts, we stream through each row and calculate the counts for which the decade is less than 1960 and also wherein decade is more than 1960.

```
'apple, fg=60, bg=4888'
'banana, fg=3, bg=12'
'carrot, fg=0, bg=979'
```

3) Below is the graph of Runtime Vs Parallel Mode used. The blue line represents our expectation that the optimal speedup gained through parallelization is linear with respect to the number of tasks running in parallel. The red line represents the actual relationship after conducting the experiment in different parallel modes. The possible reasons for this to happen are:
   - When parallelism increases, an extra overhead is involved in shuffling the data among the various reducer and mapper nodes.
   - There is no guarantee that the dataset is uniformly distributed among the different mappers/reducers. Many times, a few of the nodes are much more computationally occupied then other nodes. This happens because distribution of the data among the nodes depends on the keys (i.e. the words) and some keys are always more common than others.
   - Each of the nodes may not have the same capacity. Hence, some of the nodes may be slower/faster as compared to the other nodes.



4) Consider the below workflow:

```
class JoinHadoop(Planner):
        purchaseData = ReadLines('joinID.txt') | Flatten(by=tokenizeID)
        nameData = ReadLines('joinName.txt') | Flatten(by=tokenizeName)
        mergedData = Union(purchaseData, nameData)
        grouping = Group(mergedData, by=lambda (w1,w2):w1) | ReplaceEach(by=lambda
(w1,w2):w2)
        output = ReplaceEach(grouping, by=getDistinctCounts)
```

Consider the below sample dataset:

1 3,4,1,2,2,5	1 Ejaz
2 4,4,1,2,5	2 Zeean
3 3,4,1,6,9	3 John

We can implement the hadoop join operation in basically 3 steps: The only assumption is that the input data is already streamed and flattened as below:

```
('1', ['3', '4', '1', '2', '2', '5'])        ('1', 'Ejaz')
('2', ['4', '4', '1', '2', '5'])             ('2', 'Zeean')
('3', ['3', '4', '1', '6', '9'])             ('3', 'John')
```

- First we need to concatenate this data. A normal union operation would suffice

```
('1', 'Ejaz')
('1', ['3', '4', '1', '2', '2', '5'])
('2', 'Zeean')
('2', ['4', '4', '1', '2', '5'])
('3', John)
('3', ['3', '4', '1', '6', '9'])
```

- Now we take the union output and group it based on the 'id' attribute. Also we can then remove the 'id' group using the replaceEach construct.

```
[('1', 'Ejaz'), ('1', ['3', '4', '1', '2', '2', '5'])]
[('2', 'Zeean'), ('2', ['4', '4', '1', '2', '5'])]
[('3', 'John'), ('3', ['3', '4', '1', '6', '9'])]
```

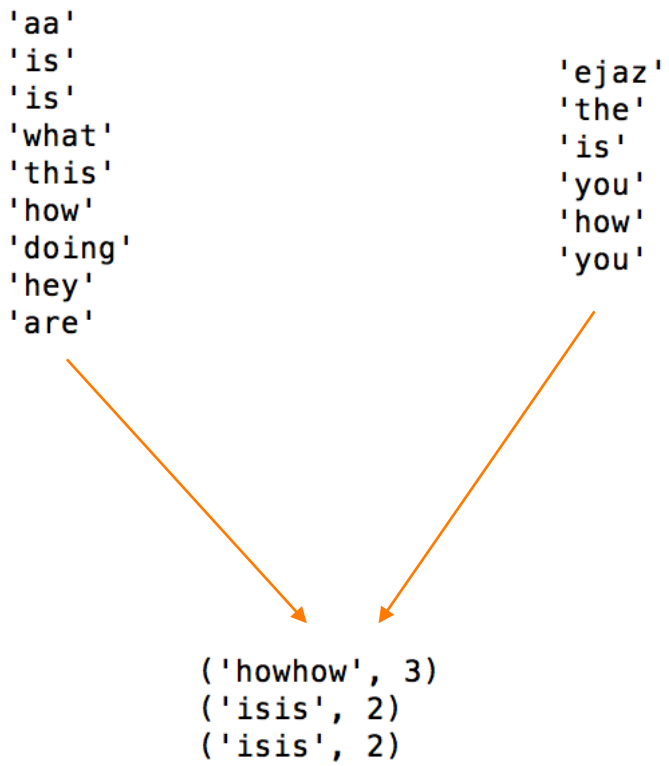
- Now the final step is to remove duplicates and take the total count. This can be done in one reduce step

```
('Ejaz', 5)
('Zeean', 4)
('John', 5)
```

5) Let the input file 'data.txt' be represented as below

```
aa ejaz is the is
what is this
how you doing
hey how are you
```

This script combines all the similar words in the file that are located at odd location from a given word. For example, for the above input file we first flatten all even and odd placed words together as shown below:



As seen above, the script first splits the input file into 2 parts:

- A list of words located at even locations on a line.
- A list of words located at odd locations on a line.

The above 2 lists are then combined based on common words in the 2 lists. We then keep track of the length of the common word.