Advanced MapReduce

CMPSC/DS 410

MapReduce Components

- Mapper
  - Can be written in Java
  - Can be written in language of your choosing with hadoop-streaming
- Reducer
  - Can be written in Java
  - Can be written in language of your choosing with hadoop-streaming
- Combiner (optional)
- Partitioner
  - Write it in java
  - Can tell hadoop-streaming (or mrjob) to use it
- Sorter
  - Write it in java

Remember the WordCount!

```
from mrjob.job import MRJob

class WordCount(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield (w, 1)

    def reducer(self, key, values):
        yield (key, sum(values))

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

But we want more

- Suppose we want to know some Statistics:
  - total number of words
  - number of words starting with A
  - number of words starting with B
  - etc.
- How would we do it?

Option 1

- Run a mapreduce job on output of WordCount (using Steps api of MRJOB)

```
    def mapper(self, key, value):
        yield "Total", value
        first_letter = key[0].upper()
        yield first_letter, value

    def reducer(self, key, value_list):
        yield key, sum(value_list)
```

Option 1

- Run a mapreduce job on output of WordCount (using Steps api of MRJOB)

```
    def mapper(self, key, value):
        yield "Total", value
        first_letter = key[0].upper()
        yield first_letter, value

    def reducer(self, key, value_list):
        yield key, sum(value_list)
```


- "With big data comes big responsibility" -- Ben Parker (Spiderman's Uncle)
- This code is wrong! (But why? It gives me the right answer).
  - Inefficient
    - Getting the right answer **slowly**
    - Second mapreduce job is an extra transfer of data
    - Can we get wordcount and statistics in 1 mapreduce job?

Option 2

```
class WordCountStats(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield ("_"+w, 1)
        yield "Total_", 1
        first_letter = key[0].upper()
        yield first_letter +"_", 1

    def reducer(self, key, values):
        yield (key, sum(values))
```


- Why the underscore?
- Yucky output. Why?

Option 2

```
class WordCountStats(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield ("_" + w, 1)
        yield "Total_", 1
        first_letter = key[0].upper()
        yield first_letter +"_", 1

    def reducer(self, key, values):
        yield (key, sum(values))
```


- Where is the total? Where are the A words?
  - Scattered on different part files (shards)
  - Within a part file, mixed in with the normal word count information
  - How do we fix this?

Wait until someone says "Partitioner"

Option 3

```python
from mrjob.job import MRJob

class WordCountStats(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield ("_" + w, 1)
        yield "Total_", 1
        first_letter = key[0].upper()
        yield first_letter +"_", 1

    def reducer(self, key, values):
        yield (key, sum(values))

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

```java
int getPartition(Key key, Value value, int numPart) {
    if key.startsWith("_") { //normal word
       partition = 1 + (key.hash() % (numPart - 1))
    }
    else { //Summary
       partition = 0
    }
    return partition
}
```


- Statistics go to first partition
- Wordcounts distributed across the rest of the partitions
- Good?

Big Data

- Big data can easily overwhelm big resources
- Have to use resources efficiently
- In this sense the WordCountStats program is still wrong.
- To see why, let us look at LetterCount

Counting letters

```
class LetterCount(MRJob):
    def mapper(self, key, line):
        for symbol in line:
            if isletter(symbol):
                yield (symbol, 1)

    def reducer(self, key, values):
        yield (key, sum(values))
```


- What is the size of the data being shipped from mappers to reducers?

Counting letters

```
class LetterCount(MRJob):
    def mapper(self, key, line):
        for symbol in line:
            if isletter(symbol):
                yield (symbol, 1)

    def reducer(self, key, values):
        yield (key, sum(values))
```


- What is the size of the data being shipped from mappers to reducers?
  - L number of characters in input
  - B is size of a character
  - N is size of an integer
  - Roughly L(B+N) being shipped across network
  - Could be larger than original input file
  - What to do?

Combining

```
class LetterCount(MRJob):
    def mapper(self, key, line):
        for symbol in line:
            if isletter(symbol):
                yield (symbol, 1)

    def reducer(self, key, values):
        yield (key, sum(values))
```


- Why wait until reducer to start adding up values?
- We can **combine** messages with the same key by adding up their values.

Setup and Teardown functions

```
def mapper_init(self):
    # initialization code

def mapper_final(self):
    # cleanup code

def reducer_init(self):
    # initialization code

def reducer_final(self):
    # cleanup code
```


- You can override them
- They are called automatically by mrjob
- Call sequence for each mapper:
  - first mapper_init() is called
  - then for every line, mapper(key, value) is called
  - then mapper_final() is called
  - similar sequence for reducers

Setup and Teardown

- mapper_init()
  - can initialize class variables (self.cache={})
  - these variables are available when the **same** mapper processes subsequent lines
  - i.e. a mapper has its own state
    - it can modify its own state
    - it cannot modify the state of other mappers (they are on different machines and may run at different times)
  - can output key,value pairs (using **yield**)
- mapper_final()
  - cleans up at the end
  - can output key, value pairs
  - can close any files that were opened

How to use for letter count

```
    def mapper(self, key, line):
        for symbol in line:
            if isletter(symbol):
               yield (symbol, 1)

    def reducer(self, key, values):
        yield (key, sum(values))
```

Hint:

```
    def mapper(self, key, line):
        for symbol in line:
            if isletter(symbol):
               yield (symbol, 1)

    def reducer(self, key, values):
        yield (key, sum(values))
```


- mapper_init() initializes a dictionary (key=letter, value=how many times seen so far)
- mapper() updates the counts, yields nothing but sends heartbeat messages periodically
  - call **set_status(message)** to send heartbeat message
  - why is it needed? what happens if heartbeat message is not sent?
- mapper_final() outputs key value pairs from the dictionary
- Comibing messages inside mapper is called **in memory combining**
- Combiners in hadoop also specify how to combine messages
  - But there are no guarantees on when/if/how-many-times a combiner will run
  - Hence in memory combining is preferred
- Why should we use only 1 reducer for this specific problem?

So why is this wrong?

```python
from mrjob.job import MRJob

class WordCountStats(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield ("_" + w, 1)
        yield "Total_", 1
        first_letter = key[0].upper()
        yield first_letter +"_", 1

    def reducer(self, key, values):
        yield (key, sum(values))

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

```java
int getPartition(Key key, Value value, int numPart) {
    if key.startsWith("_") { //normal word
       partition = 1 + (key.hash() % (numPart - 1))
    }
    else { //Summary
       partition = 0
    }
    return partition
}
```

What was wrong

- Reducer 0 gets message "Total_", 1 for **every** word in input
  - Assumes 1 reducer can store something as big as the entire input
  - That is small data programming
  - For big files your job will crash
- No in memory combining

How to add in memory combining?

```python
from mrjob.job import MRJob

class WordCountStats(MRJob):
    def mapper(self, key, line):
        words = line.split()
        for w in words:
            yield ("_" + w, 1)
        yield "Total_", 1
        first_letter = key[0].upper()
        yield first_letter +"_", 1

    def reducer(self, key, values):
        yield (key, sum(values))

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

```java
int getPartition(Key key, Value value, int numPart) {
    if key.startsWith("_") { //normal word
       partition = 1 + (key.hash() % (numPart - 1))
    }
    else { //Summary
       partition = 0
    }
    return partition
}
```

Note:

- Don't let the dictionary (or other data) stored by mapper get too big
  - Otherwise it might not fit in memory
  - Avoid small data solutions!
  - Flush the cache
    - output key value pairs when the cache gets too big
    - then clear the cache

What we now know:

- Using partitioners to send messages where we want
- Using in memory combining to reduce the number of messages sent
  - Reduces network traffic
  - Makes the Shuffle and Sort phase faster (less data to shuffle and sort)
  - Helps avoid overloading a reducer with too many messages

Next problem: Word rates

- WordCount output is **word, count**
- WordRates output is **word, rate**
  - rate iis (number of times word appears) / (total number of words in input)
- How to do it?

General Strategy

- Mapper emits two kinds of messages:
  - word, count
  - Total_0, count
    - emitted by mapper_final()
    - represents total number of words seen by this mapper
    - this message is sent to Reducer 0 (via partitioner)
  - Total_1, count
    - emitted by mapper_final()
    - represents total number of words seen by this mapper
    - this message is sent to Reducer 1 (via partitioner)
  - etc.

- Sorter
  - makes sure Total_0 appears before all normal words
    - So Reducer 0 will receive key=Total\0 and value_list as very first key, value_list pair
    - Reducer sums them up, now knows total and can generate word rate for every word it sees
  - make sure Total_1 appears before all normal words
    - So Reducer 1 can do its job.
  - etc.
- Sorter + Partitioner in java can control delivery destination and order

Sorter

- Write a sorter for word rate

```
int compare(WritableComparable w1, WritableComparable w2) {
   //return 1 if w1 is "greater" than w1
   //return 0 if they are equal
   //return -1 if w1 is "less" than w2
}
```


- Sorter requirements:
  - sorter works on keys
  - only return 0 if the keys are the same!
  - antisymmetric: compare(w1, w2) == -compare(w2, w1)
  - transitive property:
    - if compare(w1,w2) == 1
    - and compare(w2, w3) == 1
    - then compare(w1,w3)==1

Exercise

- Use mapreduce to output word,count
- but only for words that appear more frequently than "like"

Joins

- File 1 (Customers):
  - key = None
  - value = customer id, city
- File 2 (Orders):
  - key = None
  - value = customer id, item, amount
- Need to do a join
  - Find the total amount of purchases for each city
  - SELECT city, SUM(amount) FROM Customers C, Orders O WHERE C.id=O.id
  - Can specify multiple input files for mrjob (just list them)

Joins

```
def mapper(self, key, value):
   parts = value.split(",")
   if len(parts) == 2: # Customers table
      id = parts[0]
      city = parts[1]
      yield id, (city, "Customers")
   else: # Orders table
      id = parts[0]
      item = parts[1]
      amount = parts[2]
      yield id, (amount, "Orders")
```


- Joining Customers and Orders on id
- Mapper ensures co-location of file
  - for both files, id is the key
  - partitioner decides which reducers get assigned which id
  - records from both files with same id go to same reducer (co-location)
  - records tagged with info about file they belong to

Joins

```
def reducer(self, key, value_list):
    amount = 0
    for (a,b) in value_list:
        if b == "Customers":
            city = a
        else:
            amount = amount + int(a)
    yield city, amount
```


- Reducer finishes the join
  - Here we assume a customer does not appear multiple times in Customers table
    - In practice we will need to check this.
    - How do we check? What to do if check fails?
  - What if customer has no orders?
  - What if order has invalid customer id (no customer exists?)
- Are we done?
- How do we make it faster?

Joins

```
def reducer(self, key, value_list):
    amount = 0
    for (a,b) in value_list:
        if b == "Customers":
            city = a
        else:
            amount = amount + int(a)
    yield city, amount
```


- Are we done?
  - No
  - Second mapreduce step must add amounts for same city
- How do we make it faster?
  - in-memory combining inside reducer
- Main point: mapper co-locates records from two files by using join value as the key

PageRank

- One of the initial motivations for mapreduce.
- Every web page (or site) has a reputation score.
- Every web page has outgoing links
- Initially every page has the same score (1).
- Input file:
  - key = website
  - value = (score, list of websites)
- Algorithm operates in rounds. In each round
  - A site takes its score,
  - divides it evenly among outgoing links
  - the new score of a page is sum of scores it receives
- Original algorithm has random restarts which we ignore

PageRank

```
def mapper(key, value):
    score = value[0]
    neighbors = value[1]
    yield key, ("neighbors", neighbors) #why?
    for site in neighbors:
        yield site, ("score",  score/len(neighbors))
```


- mapper sends the neighborlist
- mapper sends a share of the score to each neighbor

PageRank

```
def mapper(key, value):
    score = value[0]
    neighbors = value[1]
    yield key, ("neighbors", neighbors) #why?
    for site in neighbors:
        yield site, ("score",  score/len(neighbors))
```

```
def reducer(key, value_list):
   total = 0
   for v in value_list:
      if v[0] == "neighbors":
         neighbor_list = v[0]
      else:
          total = total + int(v[1])
   yield key, (total, neighbor_list)
```


- In reducer: one of the values will be neighbor list
- The other values will be scores received
- In actual code:
  - in mrjob, each value is a string
  - in java it is more flexible