CSE 4/587
Data Intensive Computing

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Dr. Shamshad Parvin
shamsadp@buffalo.edu
313 Davis Hall

Word Co-Occurrence

Announcements

- HW 1 released due Monday 4/3
    - No late submissions will be accepted

Additional References for Today

Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer, Synthesis Lectures on Human Language Technologies, 2010, Vol. 3, No. 1, Pages 1-177, (doi: 10.2200/S00274ED1V01Y201006HLT007).

An online version of this text is also available through UB Libraries since UB subscribes to Morgan and Claypool Publishers.

Online version available at: http://lintool.github.com/MapReduceAlgorithms/index.html

Dealing with Intermediate Data

- In distributed applications (ie MapReduce), one of the most important parts of synchronization is ***exchange of intermediate results***
  - This usually involves communication of data over the network
  - In Hadoop/MR intermediate results are also written to disk
- Network and disk latencies are much more expensive compared to most other operation

Reducing the amount of intermediate data translates to better performance and efficiency

Local Aggregation

- One way to address the intermediate data problem is to perform local aggregation **before** the data gets written to disk/sent over the network
- Two basic approaches:
  - Combiners
  - In-Mapper Combining

Basic Word-Count Example

```
class Mapper
  method Map(docid id, doc d)
    for all term t in d do
      emit(t, 1)
```

```
class Reducer
  method Reduce(term t, int [c1,c2,…])
    sum ← 0
    for all int c in [c1,c2,…] do
      sum ← sum + c
    emit(t, sum)
```

Combiners

- Process to aggregate the output of Mappers
- Can be thought of as "mini-reducers"
- Must preserve the types of <key, value> pairs
  - Must take the output from Mapper as its input, and output something in the same form for the reducers
- It is a completely **optional** optimization in the eyes of MapReduce
  - It may not be run, it may be run once, it may be run many times
  - ***Correctness cannot rely on the Combiner***

Combiners

- Process to aggregate the output of Mappers
- Can be thought of as "mini-reducers"
- Must preserve the types of <key, value> pairs
    - Must take the output from Mapper as its input, and output something in the same form for the reducers
- It is a completely **optional** optimization in the eyes of MapReduce
    - It may not be run, it may be run once, it may be run many times
    - ***Correctness cannot rely on the Combiner***

For WordCount, we can use the same Reducer class for the Combiner

In-Mapper Combining

Another option is to do local aggregation in the Mapper itself

Makes the mapper more complex, but…

```
class Mapper
  method Map(docid id, doc d)
    for all term t in d do
      emit(t, 1)
```

In-Mapper Combining

Another option is to do local aggregation in the Mapper itself

Makes the mapper more complex, but…

…also gives direct control over aggregation

```
class Mapper
  method Map(docid id, doc d)
    for all term t in d do
      emit(t, 1)
```

In-Mapper Combining

Another option is to do local aggregation in the Mapper itself

Makes the mapper more complex, but…

…also gives direct control over aggregation

```
class Mapper
  method Map(docid id, doc d)
    map ← new AssociativeArray
    for all term t in d do
      map[t] ← map[t] + 1
    for all term t in map do
      emit(t, map[t])
```

In-Mapper Combining

Another option is to do local aggregation in the Mapper itself

Makes the mapper more complex, but…

…also gives direct control over aggregation

```
class Mapper
  method Map(docid id, doc d)
    map ← new AssociativeArray
    for all term t in d do
      map[t] ← map[t] + 1
    for all term t in map do
      emit(t, map[t])
```

Store intermediate result

In-Mapper Combining

Another option is to do local aggregation in the Mapper itself

Makes the mapper more complex, but...

...also gives direct control over aggregation

```
class Mapper
  method Map(docid id, doc d)
    map ← new AssociativeArray
    for all term t in d do
      map[t] ← map[t] + 1
    for all term t in map do
      emit(t, map[t])
```

Only emit after we've processed the whole input

In-Mapper Combining

What if our mapper is run on multiple <key, value> pair inputs?

In-Mapper Combining

What if our mapper is run on multiple <key, value> pair inputs?

We can utilize the `initialize` and `close` methods of our mapper!

In-Mapper Combining

```
class Mapper
  method Initialize()
    map ← new AssociativeArray

  method Map(docid id, doc d)
    for all term t in d do
      map[t] ← map[t] + 1

  method Close()
    for all term t in map do
      emit(t, map[t])
```

In-Mapper Combining

```
class Mapper
  method Initialize()
    map ← new AssociativeArray

  method Map(docid id, doc d)
    for all term t in d do
      map[t] ← map[t] + 1

  method Close()
    for all term t in map do
      emit(t, map[t])
```

Create the AssociativeArray for intermediate aggregation before processing any data

In-Mapper Combining

```
class Mapper
  method Initialize()
    map ← new AssociativeArray

  method Map(docid id, doc d)
    for all term t in d do
      map[t] ← map[t] + 1

  method Close()
    for all term t in map do
      emit(t, map[t])
```

Create the AssociativeArray for intermediate aggregation before processing any data

Don't emit and <key, value> pairs until after we've seen all of our input

Trade-Offs

Combiner

+ Simple mapper code
+ Let MapReduce manage the optimization
- No direct control
- Overhead of generating intermediate <k,v> pairs

In-Mapper Aggregation

+ More efficient aggregation
+ Direct control
- Scalability bottleneck requires memory management
- No "purity" of functional programming
- May introduce ordering bugs

Correctness with Local Aggregation

Example: <key, value> pairs associate a string with a number, we want to compute the mean value for each key.

```
class Mapper
  method Map(str s, int i)
    emit(s, i)
```

```
class Reducer
  method Reduce(str s, int [i₁,i₂,...])
    sum ← 0; count ← 0
    for all int i in [i₁,i₂,...] do
      sum ← sum + i
      count ← count + 1
    avg = sum / count
    emit(t, avg)
```

Correctness with Local Aggregation

Example: <key, value> pairs associate a string with a number, we want to compute the mean value for each key.

```
class Mapper
  method Map(str s, int i)
    emit(s, i)
```

```
class Reducer
  method Reduce(str s, int [i_1,i_2,...])
    sum ← 0; count ← 0
    for all int i in [i_1,i_2,...] do
      sum ← sum + i
      count ← count + 1
    avg = sum / count
    emit(t, avg)
```

Can this reducer also be the combiner?

Correctness with Local Aggregation

Observation: Mean(1,2,3,4,5) ≠ Mean(Mean(1,2),Mean(3,4,5))

Count was associative and commutative, Mean is not!

How can we write a combiner to do local aggregation?

Combiner Attempt #1

```
class Combiner
  method Combine(str s, int [i₁,i₂,...])
    sum ← 0; count ← 0;
    for all int i in [i₁,i₂,...] do
      sum ← sum + i
      count ← count + 1
    emit(s, (sum, count)) // Emit a pair
```

Combiner Attempt #1

```
class Combiner
  method Combine(str s, int [i₁,i₂,...])
    sum ← 0; count ← 0;
    for all int i in [i₁,i₂,...] do
      sum ← sum + i
      count ← count + 1
    emit(s, (sum, count)) // Emit a pair
```

Will this work for local aggregation?

Combiners

- Process to aggregate the output of Mappers
- Can be thought of as "mini-reducers"
- Must preserve the types of <key, value> pairs
    - **Must take the output from Mapper as its input, and output something in the same form for the reducers**
- It is a completely **optional** optimization in the eyes of MapReduce
    - It may not be run, it may be run once, it may be run many times
    - ***Correctness cannot rely on the Combiner***

Combiner Attempt #2

```
class Mapper
  method Map(str s, int i)
    emit(s, (i, 1))
```

```
class Combiner
  method Combine(str s, pair [(s1,c1),...])
    sum ← 0; count ← 0
    for all pair (s,c) in [(s1,c1),...] do
      sum ← sum + s
      count ← count + c
    emit(t, (sum, count))
```

```
class Reducer
  method Reduce(str s, pair [(s1,c1),...])
    sum ← 0; count ← 0
    for all pair (s,c) in [(s1,c1),...] do
      sum ← sum + s
      count ← count + c
    avg = sum / count
    emit(t, avg)
```

Combiner Attempt #2

```
class Mapper
  method Map(str s, int i)
    emit(s, (i, 1))
```

Outputs pairs

```
class Combiner
  method Combine(str s, pair [(s1,c1),...])
    sum ← 0; count ← 0
    for all pair (s,c) in [(s1,c1),...] do
      sum ← sum + s
      count ← count + c
    emit(t, (sum, count))
```

Inputs AND outputs are pairs

```
class Reducer
  method Reduce(str s, pair [(s1,c1),...])
    sum ← 0; count ← 0
    for all pair (s,c) in [(s1,c1),...] do
      sum ← sum + s
      count ← count + c
    avg = sum / count
    emit(t, avg)
```

Inputs are pairs

In-Mapper Aggregation

```
class Mapper
  method Initialize()
    sumMap ← new AssociativeArray
    countMap ← new AssociativeArray
  method Map(str s, int i)
    sumMap[s] ← sumMap[s] + i
    countMap[s] ← countMap[s] + 1
  method Close()
    for all key in sumMap do
      emit(key, (sumMap[key], countMap[key]))
```

Word Co-Occurrence

- Word Co-Occurrence counts the number of times pairs of words occur in the same context, ie a sentence
- Involves constructing an ***N*** x ***N*** matrix, ***M***, where ***N*** is the total number of words in the vocabulary
  - M_{ij} is the number of times words w_i and w_j occurred in the same context
  - Very simple to compute…if the matrix fits in memory

Word Co-Occurrence

Can come up in a number of different domains, not just in text processing

Some Examples:

- Information retrieval, NLP, text mining, etc
- Co-Occurrence in consumer purchases (can help with inventory mgmt)
- Finding associations between recurring financial transactions

Word Co-Occurrence - Pairs Method

```
class Mapper
  method Map(docid id, doc d)
    for all w in d do
      for all u in Neighbors(w) do
        emit((w,u), 1)
```

```
class Reducer
  method Reduce(pair p, int[] cnts)
    sum ← 0
    for all c in cnts do
      sum ← s + c
    emit(p, sum)
```

Word Co-Occurrence - Pairs Method

```
class Mapper
  method Map(docid id, doc d)
    for all w in d do
      for all u in Neighbors(w) do
        emit((w,u), 1)
```

The key in our <key,value> pair is a pair of words. Represents a single entry in our co-occurrence matrix.

```
class Reducer
  method Reduce(pair p, int[] cnts)
    sum ← 0
    for all c in cnts do
      sum ← s + c
    emit(p, sum)
```

Word Co-Occurrence - Pairs Method

```
class Mapper
  method Map(docid id, doc d)
    for all w in d do
      for all u in Neighbors(w) do
        emit((w,u), 1)
```

The key in our <key,value> pair is a pair of words. Represents a single entry in our co-occurrence matrix.

```
class Reducer
  method Reduce(pair p, int[] cnts)
    sum ← 0
    for all c in cnts do
      sum ← s + c
    emit(p, sum)
```

How many possible keys are there? Any issues with this method?

Word Co-Occurrence - Stripes Method

What else could we use as a key?

Word Co-Occurrence - Stripes Method

What else could we use as a key?

Could we use a single word as a key?

Word Co-Occurrence - Stripes Method

What else could we use as a key?

Could we use a single word as a key?

If so, what would be the value?

Word Co-Occurence - Stripes Method

```
class Mapper
  method Map(docid id, doc d)
    for all w in d do
      map ← new AssociativeArray
      for all u in Neighbors(w) do
        map[u] ← map[u] + 1
      emit(w, map)
```

```
class Reducer
  method Reduce(str w, stripes)
    map ← new AssociativeArray
    for all s in stripes do
      Sum(map, s)
    emit(w, map)
```

Word Co-Occurence - Stripes Method

```
class Mapper
  method Map(docid id, doc d)
    for all w in d do
      map ← new AssociativeArray
      for all u in Neighbors(w) do
        map[u] ← map[u] + 1
      emit(w, map)
```

Build and emit a map containing the counts of all neighbors of w

```
class Reducer
  method Reduce(str w, stripes)
    map ← new AssociativeArray
    for all s in stripes do
      Sum(map, s)
    emit(w, map)
```

Word Co-Occurence - Stripes Method

```
class Mapper
  method Map(docid id, doc d)
    for all w in d do
      map ← new AssociativeArray
      for all u in Neighbors(w) do
        map[u] ← map[u] + 1
      emit(w, map)
```

Build and emit a map containing the counts of all neighbors of w

```
class Reducer
  method Reduce(str w, stripes)
    map ← new AssociativeArray
    for all s in stripes do
      Sum(map, s)
    emit(w, map)
```

Combine all maps for w into one map and output it

Analysis of Stripes

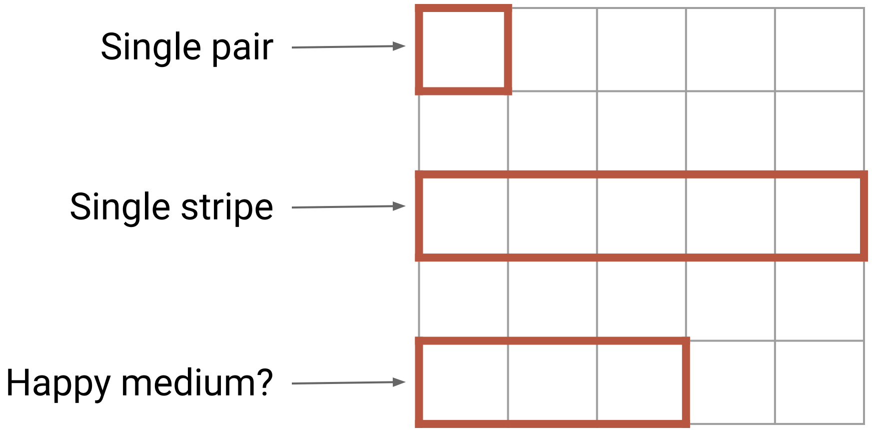
- \+ Stripes generate far fewer <key, value> pairs
- \+ Stripes are much more compact (the pairs approach duplicates the left word in the pair for every pair)
- \+ Fewer and shorter keys means less sorting
- \+ Better for local aggregation
- \- Values are larger and more complex with more serialization overhead
- \- Scalability concerns similar to In-Mapper combining (memory overflow)

Pairs vs Stripes in General

In a pairs approach, each key corresponds to a ***single entry*** of our matrix

In a stripes approach, each key corresponds to a ***single row*** of our matrix

We can also aim to find a middle ground somewhere between the two extremes



Local Aggregation

- Both combiners and In-Mapper combining can be used with either the pairs or stripes method
  - The pairs method has less opportunity for combining (it is less likely to find many occurrences of a specific pair of words)
  - The stripes method may run into scalability issues as the maps get larger

Performance Study

Performance study from Lin and Dyer Chapter 3

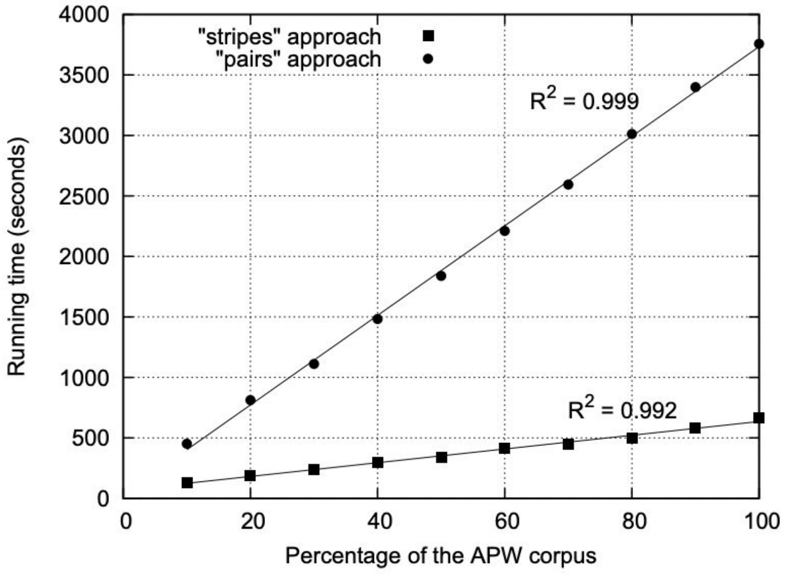


Figure 3.10: Running time of the "pairs" and "stripes" algorithms for computing word co-occurrence matrices on different fractions of the APW corpus. These experiments were performed on a Hadoop cluster with 19 slaves, each with two single-core processors and two disks.

Relative Co-Occurrence Problem

- Individual words occur with different frequency.
  - In English, we expect to come across "the" much more often than "zebra"
- Using absolute counts can be deceiving – more frequent words may have a higher co-occurrence count simply due to being more common
  - "the" and "stripe" may have more co-occurrences than "stripe" and "zebra" simply because "the" is way more common than "zebra"

How can we determine the "relative" co-occurrence?

Relative Co-Occurrence Problem

$$f(w_j|w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

Relative Co-Occurrence Problem

Number of times w_i co-occurs with w_j

$$f(w_j|w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

Relative Co-Occurrence Problem

Number of times w_i co-occurs with w_j

$$f(w_j | w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

Number of times w_i co-occurs with anything else

Relative Co-Occurrence Problem

Number of times w_i co-occurs with w_j

$$f(w_j|w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

Number of times w_i co-occurs with anything else
This is called the **marginal**

Relative Co-Occurrence Problem

- Computing relative co-occurrence with stripes is trivial
    - Reducer can sum all counts for a particular key to get the **marginal**

Relative Co-Occurrence Problem

- Computing relative co-occurrence with stripes is trivial
  - Reducer can sum all counts for a particular key to get the **marginal**

Can we compute relative co-occurrence with the pairs approach?

Reducer-Side Aggregation

- The reducer in the pairs method reduces single pairs at a time
    - Just having counts for a single pair is not enough to compute relative co-occurrence, we can't compute the **marginal**

Reducer-Side Aggregation

- The reducer in the pairs method reduces single pairs at a time
  - Just having counts for a single pair is not enough to compute relative co-occurrence, we can't compute the **marginal**
- Just like the Mapper, our Reducers can preserve state across calls to **`Reduce(...)`**, by using **`Initialize()`** and **`Close()`**
  - To do this we need a few modifications…

Reducer-Side Aggregation

Given the following co-occurrence pairs, what assumptions do we need in order to do reducer side aggregation on, for example, the word dog?

(dog, aardvark), (dog, zebra), (dog, apple), (cat, tail), (dog, fur), (fly, banana), (dog, banana), …

Reducer-Side Aggregation

Given the following co-occurrence pairs, what assumptions do we need in order to do reducer side aggregation on, for example, the word dog?

(dog, aardvark), (dog, zebra), (dog, apple), (cat, tail), (dog, fur), (fly, banana), (dog, banana), ...

1. We need to ensure that all pairs starting with dog go to the same reducer.

Reducer-Side Aggregation

Given the following co-occurrence pairs, what assumptions do we need in order to do reducer side aggregation on, for example, the word dog?

(dog, aardvark), (dog, zebra), (dog, apple), (cat, tail), (dog, fur), (fly, banana), (dog, banana), …

1. We need to ensure that all pairs starting with dog go to the same reducer.
2. We need to be able to tell when we have reduced all the dog pairs.

Reducer-Side Aggregation

Given the following co-occurrence pairs, what assumptions do we need in order to do reducer side aggregation on, for example, the word dog?

(dog, aardvark), (dog, zebra), (dog, apple), (cat, tail), (dog, fur), (fly, banana), (dog, banana), ...

1. We need to ensure that all pairs starting with dog go to the same reducer.
2. We need to be able to tell when we have reduced all the dog pairs.

This can be accomplished with a custom partitioner/sort order

Reducer-Side Aggregation

(dog, aardvark),
(dog, apple),
(dog, banana),
(dog, fur),
…
(dog, zebra),
(door, open),
…

Reducer-Side Aggregation

(dog, aardvark),
(dog, apple),
(dog, banana),
(dog, fur),
…
(dog, zebra),
(door, open),
…

If we sort our keys by the first word, we know as soon as we encounter (door, open) we are done with all of the dog keys.

Reducer-Side Aggregation

```
class Reducer
  method Initialize()
    currWord ← ""
    map ← new AssociativeArray
  method Reduce(pair p, int[] cnts)
    if pair.first != currWord then
      computeAndEmitRelative()
      currWord ← pair.first
      map ← new AssociativeArray
    for all c in cnts do
      map[pair.second] = map[pair.second] + c
```

Reducer-Side Aggregation

```
class Reducer
  method Initialize()
    currWord ← ""
    map ← new AssociativeArray
  method Reduce(pair p, int[] cnts)
    if pair.first != currWord then
      computeAndEmitRelative()
      currWord ← pair.first
      map ← new AssociativeArray
    for all c in cnts do
      map[pair.second] = map[pair.second] + c
```

Since we sorted based on the first word in the pair, we can assume all keys with the same first word will appear in a row…once we encounter a different word, we can output the result for our previous word

Reducer-Side Aggregation

```
class Reducer
  method Initialize()
    currWord ← ""
    map ← new AssociativeArray
  method Reduce(pair p, int[] cnts)
    if pair.first != currWord then
      computeAndEmitRelative()
      currWord ← pair.first
      map ← new AssociativeArray
    for all c in cnts do
      map[pair.second] = map[pair.second] + c
```

The map holds the total number of co-occurrences with our current word

Reducer-Side Aggregation

```
class Reducer
  method computeAndEmitRelative()
    marginal ← 0
    for all key in map do
      marginal ← marginal + map[key]
    for all key in map do
      N ← map[key]
      relative = N / (marginal - N)
      emit(currWord, relative)
```

Reducer-Side Aggregation

```
class Reducer
  method computeAndEmitRelative()
    marginal ← 0
    for all key in map do
      marginal ← marginal + map[key]
    for all key in map do
      N ← map[key]
      relative = N / (marginal - N)
      emit(currWord, relative)
```

Compute marginal across all words co-occurring with our current word

Reducer-Side Aggregation

```
class Reducer
  method computeAndEmitRelative()
    marginal ← 0
    for all key in map do
      marginal ← marginal + map[key]
    for all key in map do
      N ← map[key]
      relative = N / (marginal - N)
      emit(currWord, relative)
```

Compute relative co-occurrence for all words

Scalability

Observation: This method has the same scalability issues as the stripes method In-Mapper aggregation…as our vocabulary gets bigger, the map may not fit in memory

…but we need it to compute the marginal…or do we?

Order Inversion

- Can we compute the marginal **before** we compute the individual co-occurrences?

Order Inversion

- Can we compute the marginal **before** we compute the individual co-occurrences? **YES!**
- Emit to a special pair that contains total occurrences of a word
  - ie: (dog, *) would count co-occurrences of dog with **any** word
  - In our sorting, make sure this pair comes before all other dog pairs

<(dog,*),[10,2,147]>, <(dog, aardvark),[2,1])>, …, <(dog, zebra), [3,1,1]>, <(cat,*),[31,491,6]>, …

Order Inversion

- Can we compute the marginal **before** we compute the individual co-occurrences? **YES!**
- Emit to a special pair that contains total occurrences of a word
  - ie: (dog, *) would count co-occurrences of dog with **any** word
  - In our sorting, make sure this pair comes before all other dog pairs

<(dog,*),[10,2,147]>, <(dog, aardvark),[2,1])>, ..., <(dog, zebra), [3,1,1]>, <(cat,*),[31,491,6]>, ...

This is the first "dog" pair our reducer will encounter, and can be used to compute the marginal for dog **before** we compute the individual co-occurrences for dog. This is called **order inversion.**

Word Co-Occurrence - Pairs Method

```
class Mapper
  method Map(docid id, doc d)
    for all w in d do
      for all u in Neighbors(w) do
        emit((w,u), 1)
        emit((w,*), 1)
```

```
class Reducer
  method Initialize()
    marginal ← 0
  method Reduce(pair p, int[] cnts)
    N ← 0
    for all c in cnts do
      N ← N + c
    if p.second == * then
      marginal ← N
    else
      emit(p, N / (marginal - N))
```

Word Co-Occurrence - Pairs Method

```
class Mapper
  method Map(docid id, doc d)
    for all w in d do
      for all u in Neighbors(w) do
        emit((w,u), 1)
        emit((w,*), 1)
```

```
class Reducer
  method Initialize()
    marginal ← 0
  method Reduce(pair p, int[] cnts)
    N ← 0
    for all c in cnts do
      N ← N + c
    if p.second == * then
      marginal ← N
    else
      emit(p, N / (marginal - N))
```

Special pair can be used to compute the marginal and indicates we are starting a new word in the reducer