Lecture 4: MapReduce Algorithm Design

CSCI4180 (Fall 2013)

Patrick P. C. Lee

MapReduce Recap

- Two basic functions of MapReduce:

- **Map** `(k1,v1) → list(k2,v2)`
  - Takes an input key/value pair
  - Produces a set of intermediate key/value pairs

- **Reduce** `(k2,list(v2)) → list(k3,v3)`
  - Takes a set of values for an intermediate key
  - Produces a set of output values
  - *MapReduce framework guarantees that all values associated with the same key are brought together in the reducer*

MapReduce Recap

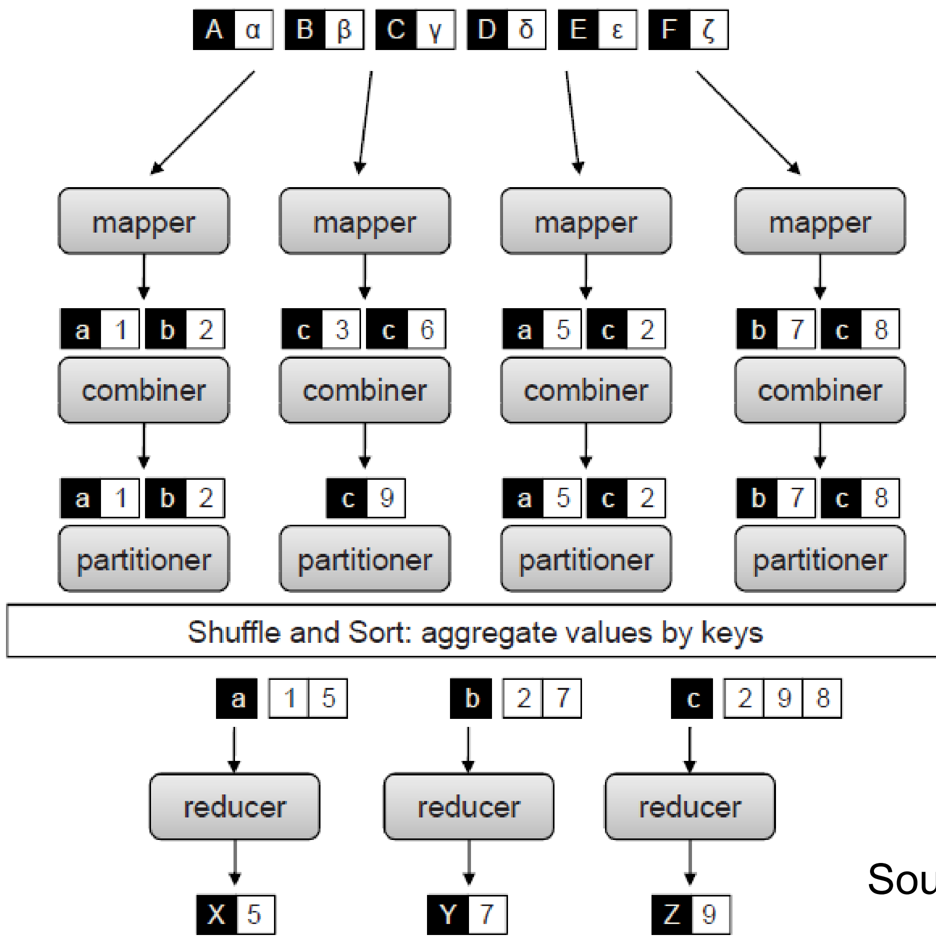
- Optional functions:

- **partition** (k’, number of partitions) → partition for k’
  - dividing up the intermediate key space and assigning intermediate key-value pairs to reducers
  - Often a simple hash of the key, e.g., hash(k’) mod n

- **combine** `(k2,list(v2))` → `list(k2’,v2’)`
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic
  - Will be discussed later

MapReduce Recap

MapReduce Architecture



Source: LD Ch.2 page 28

Goal of this Lecture

- **Key question**: MapReduce provides an elegant programming model, but how should we recast a multitude of algorithms into the MapReduce model?

- **Goal of this lecture**: provide a guide to MapReduce algorithm design:
  - **Design patterns**, which form the building blocks of many problems

Challenges

- MapReduce execution framework handles most complicated details
  - e.g., copy intermediate key-value pairs from mappers to reducers grouped by key during the shuffle and sort stage
- Programmers have little control over MapReduce execution:
  - *Where* a mapper or reducer runs
  - *When* a mapper or reduce begins or finishes
  - *Which* input key-value pairs are processed by a specific mapper
  - *Which* intermediate key-value pairs are processed by a specific reducer

Challenges

- Things that programmers can control:
  - Construct complex data structures as keys and values to store and communicate partial results
  - Execute user-specified initialization/termination code in a map or reduce task
  - Preserve state in both mappers and reducers across multiple input or intermediate keys
  - Control sort order of intermediate keys, and hence the order of how a reducer processes keys
  - Control partitioning of key space, and hence the set of keys encountered by a reducer

Challenges

- What we really want?
  - It depends on datasets and applications

- Fundamental principle: No inherent bottlenecks as algorithms are applied to increasingly large datasets
  - Linear scalability: an algorithm running on twice the amount of data should take only twice as long
  - An algorithm running on twice the number of nodes should only take half as long

Design Patterns

- Combing and in-mapper combining
  - Aggregate map outputs to reduce data traffic being shuffled from mappers to reducers
- Pairs and stripes
  - Keep track of joint events
- Order inversion
  - Sort and control the sequence of computation
- Value-to-key conversion
  - Allow secondary sorting

Local Aggregation

- In Hadoop, intermediate results (i.e., map outputs) are written to local disk before being sent over the network
  - Network and disk latencies are expensive
- Local aggregation of intermediate results reduces the number of key-value pairs that need to be shuffled from the mappers to the reducers
- Default combiner:
  - Provided by the MapReduce framework
  - Aggregate map outputs with the same key
  - Acts like a mini-reducer

Word Count: Baseline

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)

class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        Emit(term t, count s)
```

- What is the number of records being shuffled
  - Without combiners?
  - With combiners?

Word Count: Version 1

```
class Mapper
    method Map(docid a, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                    ▷ Tally counts for entire document
        for all term t ∈ H do
            Emit(term t, count H{t})
```

- In-mapper combining:
  - Emits a key-value pair for each unique term per document

Word Count: Version 2

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                    ▷ Tally counts across documents
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

Setup() in Java

Cleanup() in Java

- In-mapper combining
  - Recall a map object is created for each map task
  - Aggregate all data appearing in the input block processed by the map task

Combiners vs. In-Mapper Combiners

- Advantages of in-mapper combiners:
  - Provide control over where and how local aggregation takes place. In contrast, semantics of default combiners are underspecified in MapReduce.
  - In-mapper combiners are applied inside the code. Default combiners are applied to the map outputs (after being emitted by the map task).
- Disadvantages:
  - States are preserved within mappers → potentially large memory overhead
  - Potential order-dependent bugs

Combiner Design

- Combiner and reducer must share the same signature
  - Combiner is treated as mini-reducer
  - Combiner input and output key-value types must match the reducer input key-value type
- Remember: combiner are optional optimizations
  - With/without combiner should not affect algorithm correctness
  - May be run 0, 1, or multiple times, determined by the MapReduce execution framework
- In Java, you specify the combiner class as:
  - `public void setCombinerClass(Class<? extends Reducer> cls)`
  - Exactly the Reducer type

Computing the Mean: Version 1

class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)

class Reducer
    method Reduce(string t, integers $[r_1, r_2, \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all integer $r \in$ integers $[r_1, r_2, \ldots]$ **do**
            $sum \leftarrow sum + r$
            $cnt \leftarrow cnt + 1$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string t, integer r_{avg})

- Any drawback?
- Can we use reducer as combiner?
  - i.e., set combiner class to be reducer class

Computing the Mean: Version 1

- Mean of the means is not the original mean…

- e.g., Mean(1,2,3,4,5)
  ≠ Mean(Mean(1,2), Mean(3,4,5))

- It’s not a problem for WordCount, but it’s a problem here.

Computing the Mean: Version 2

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)

class Combiner
    method Combine(string t, integers [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        Emit(string t, pair (sum, cnt))                ▷ Separate sum and count

class Reducer
    method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
            sum ← sum + s
            cnt ← cnt + c
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
```

- Does it work? Why?

Computing the Mean: Version 3

class Mapper
    method Map(string t, integer r)
        Emit(string t, pair $(r, 1)$)

class Combiner
    method Combine(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        Emit(string t, pair (sum, cnt))

class Reducer
    method Reduce(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string t, pair (r_{avg}, cnt))

- Does it work? Why?

Computing the Mean: Version 4

```
class Mapper
    method Initialize
        S ← new AssociativeArray
        C ← new AssociativeArray
    method Map(string t, integer r)
        S{t} ← S{t} + r
        C{t} ← C{t} + 1
    method Close
        for all term t ∈ S do
            Emit(term t, pair (S{t}, C{t}))
```

- Does it work?
- Do we need a combiner?

Pairs and Stripes

- To illustrate how constructing complex keys and values improves the performance of computation

A New Running Example

- Problem: building a word co-occurrence matrix over a text collection
  - M = n x n matrix (n = number of unique words)
  - m_{ij} = number of times word w_i co-occurs with word w_j within a specific context (e.g., same sentence, same paragraph, same document)
    - It is easy to show that $m_{ij} = m_{ji}$
- Why this problem is interesting?
  - Distributional profiles of words
  - Information retrieval
  - Statistical natural language processing

Challenge

- Space requirement: $O(n^2)$.
  - Too big if we simply store the whole matrix with billions of words in memory
  - A single machine typically cannot keep the whole matrix
- How to use MapReduce to implement this large counting problem?
- Our approach:
  - Mappers generate partial counts
  - Reducers aggregate partial counts

Pairs

- Each mapper:
  - Emits intermediate key-value pairs with each co-occurring word pair and integer 1
- Each reducer:
  - Sums up all values associated with the same co-occurring word pair
  - MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer

Pairs

- Pseudo-code:

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            for all term u ∈ Neighbors(w) do
                Emit(pair (w, u), count 1)          ▷ Emit count for each co-occurrence

class Reducer
    method Reduce(pair p, counts [c1, c2, ...])
        s ← 0
        for all count c ∈ counts [c1, c2, ...] do
            s ← s + c                               ▷ Sum co-occurrence counts
        Emit(pair p, count s)
```

- Can we use the default combiner here?

Stripes

- Each mapper:
  - For each particular word, stores co-occurrence information in an associative array
  - Emits intermediate key-value pairs with words as keys and corresponding associative arrays as values
- Each reducer:
  - Sums all the counts in the associative arrays
  - MapReduce execution framework guarantees that all associative arrays with the same key are brought together in the reducer

Stripes

- Example:

$(a, b) \rightarrow 1$
$(a, c) \rightarrow 2$
$(a, d) \rightarrow 5$
$(a, e) \rightarrow 3$
$(a, f) \rightarrow 2$

$a \rightarrow \{ b{:}\ 1, c{:}\ 2, d{:}\ 5, e{:}\ 3, f{:}\ 2 \}$

- Each mapper emits $a \rightarrow \{ b{:}\ count_b, c{:}\ count_c, d{:}\ count_d \ldots \}$
- Reducers perform element-wise sum of associative arrays

$$
\begin{array}{lllllll}
 & a \rightarrow \{ & b{:}\ 1, & & d{:}\ 5, & e{:}\ 3 & \} \\
+ & a \rightarrow \{ & b{:}\ 1, & c{:}\ 2, & d{:}\ 2, & & f{:}\ 2 \} \\
\hline
 & a \rightarrow \{ & b{:}\ 2, & c{:}\ 2, & d{:}\ 7, & e{:}\ 3, & f{:}\ 2 \}
\end{array}
$$

Stripes

- Pseudo-code:

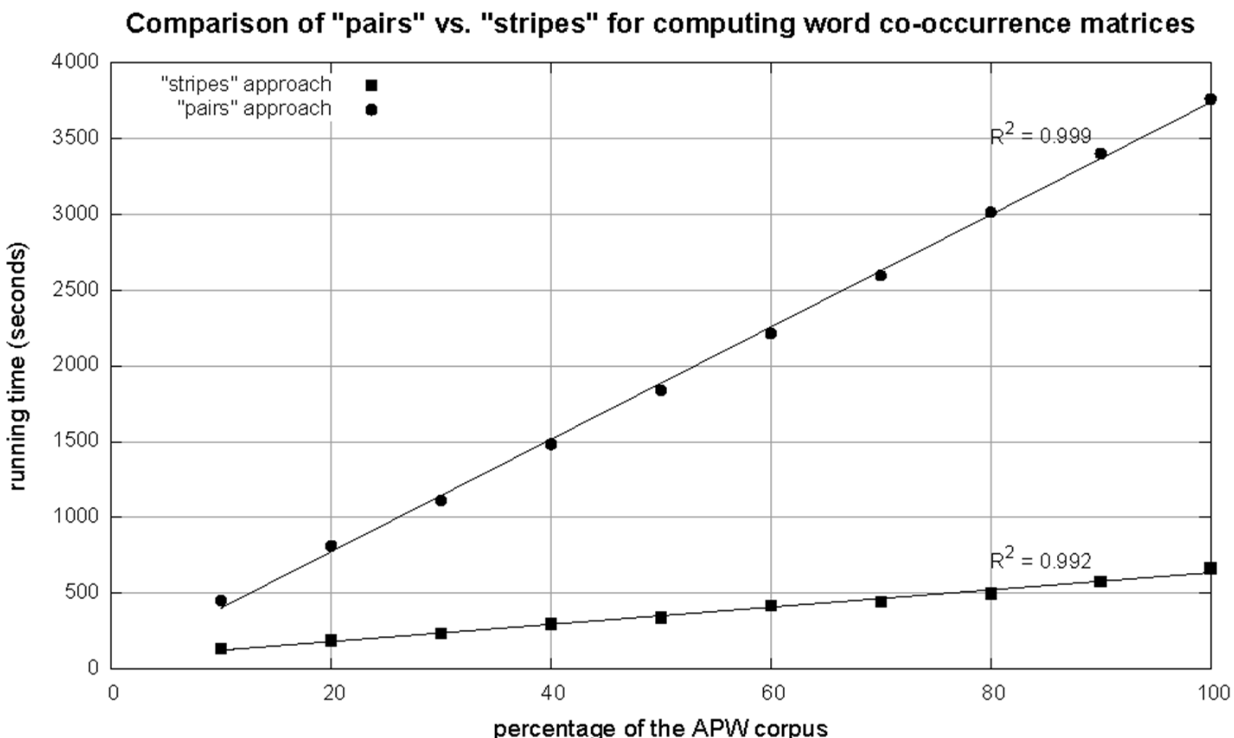
```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            H ← new AssociativeArray
            for all term u ∈ Neighbors(w) do
                H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
            Emit(Term w, Stripe H)

class Reducer
    method Reduce(term w, stripes [H_1, H_2, H_3, ...])
        H_f ← new AssociativeArray
        for all stripe H ∈ stripes [H_1, H_2, H_3, ...] do
            Sum(H_f, H)                        ▷ Element-wise sum
        Emit(term w, stripe H_f)
```

Pairs vs. Stripes

- Pairs:
  - Pro: Easy to understand and implement
  - Con: Generate many key-value pairs
- Stripes:
  - Pro: Generate fewer key-value pairs
  - Pro: Make better use of combiners
  - Con: Memory size of associative arrays in mappers could be huge
- Both pairs and stripes can apply in-mapper combining

Pairs vs. Stripes



Cluster size: 38 cores
Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)

- Stripes much faster than pairs
- Linearity is maintained

Relative Frequencies

- Drawback of co-occurrence counts:
  - Absolute counts doesn't consider that some words appear more frequently than others
  - e.g., "is" occurs very often by itself. It doesn't imply "is good" occurs more frequently than "Hello World"
- Estimate relative frequencies instead of counts:

$$f(B \mid A) = \frac{\text{count}(A, B)}{\text{count}(A)} = \frac{\text{count}(A, B)}{\sum_{B'} \text{count}(A, B')}$$

- How do we apply MapReduce to this problem?

Relative Frequencies

- Computing relative frequencies with the stripes approach is straightforward
  - Sum all the counts in the associative array for each word
  - Why is it possible in MapReduce?
  - Drawback: assuming that each associative array fits into memory
- How to compute relative frequencies with the pairs approach?

Relative Frequencies with Pairs

$(a, *) \rightarrow 32$ **Reducer holds this value in memory**

$(a, b_1) \rightarrow 3$	$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12$	$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7$	$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow 1$	$(a, b_4) \rightarrow 1 / 32$
…	…

- Mapper emits (a,*) for every word being observed
- Mapper makes sure same word goes to the same reducer (use partitioner)
- Mapper makes sure (a,*) comes first, before individual counts (how?)
- Reducer holds state to remember the count of (a,*), until all pairs with the word "a" have been computed

Order Inversion

- Why order inversion?
  - Computing relative frequencies requires marginal counts
  - But marginal cannot be computed until you see all counts
  - Buffering is a bad idea!
  - Trick: getting the marginal counts to arrive at the reducer before the joint counts
- MapReduce allows you to define the order of keys being processed by the reducer
  - Shuffle and sort

Order Inversion: Idea

- How to use the design pattern of order inversion to compute relative frequencies via the pair approach?
  - Emit a special key-value pair for each co-occurring word for the computation of marginal
  - Control the sort order of the intermediate key so that the marginal count comes before individual counts
  - Define a custom partitioner to ensure all pairs with the same left word are shuffled to the same reducer
  - Preserve state in reducer to remember the marginal count for each word

Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
- What if want to sort value also?
- Scenario:
  - Sensors record temperature over time
  - Each sensor emits (id, time t, temperature v)

Secondary Sorting

- Naive solution:
  - Each sensor emits:
    - id → (t, v)
  - All readings of sensor id will be aggregated into a reducer
  - Buffer values in memory for all id, then sort
  - Why is this a bad idea?

Secondary Sorting

- Value-to-key conversion:
  - Each mapper emits:
    - (id, t) → v
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else?
- Main idea: sorting is offloaded from the reducer (in naive approach) to the MapReduce framework

Tools for Synchronization

- Cleverly-constructed data structures
  - Bring data together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values

Issues and Tradeoffs

- Number of key-value pairs
  - Object creation overhead
  - Time for sorting and shuffling pairs across the network
- Size of each key-value pair
  - De/serialization overhead
- Local aggregation
  - Opportunities to perform local aggregation varies
  - Combiners make a big difference
  - Combiners vs. in-mapper combining
  - RAM vs. disk vs. network

Debugging at Scale

- Works on small datasets, won't scale… why?
  - Memory management issues (buffering and object creation)
  - Too much intermediate data
  - Mangled input records
- Real-world data is messy!
  - Word count: how many unique words in Wikipedia?
  - There's no such thing as "consistent data"
  - Watch out for corner cases
  - Isolate unexpected behavior, bring local

Summary

- Design patterns:
  - In-mapper combining
  - Pairs and stripes
  - Order inversion
  - Value-to-key conversion