

# **Lecture 4: MapReduce Algorithm Design**

CSCI4180

Patrick P. C. Lee

# References

- Slides are adapted from <http://lintool.github.io/UMD-courses/bigdata-2015-Spring> (Session 4-5)
- Source code: Hadoop definitive guide: <https://github.com/tomwhite/hadoop-book>

# MapReduce Recap

➤ Two basic functions of MapReduce:

➤ **Map**  $(k1, v1) \rightarrow list(k2, v2)$

- Takes an input key/value pair
- Produces a set of intermediate key/value pairs

➤ **Reduce**  $(k2, list(v2)) \rightarrow list(k3, v3)$

- 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**  $(k', \text{number of partitions}) \rightarrow \text{partition for } k'$

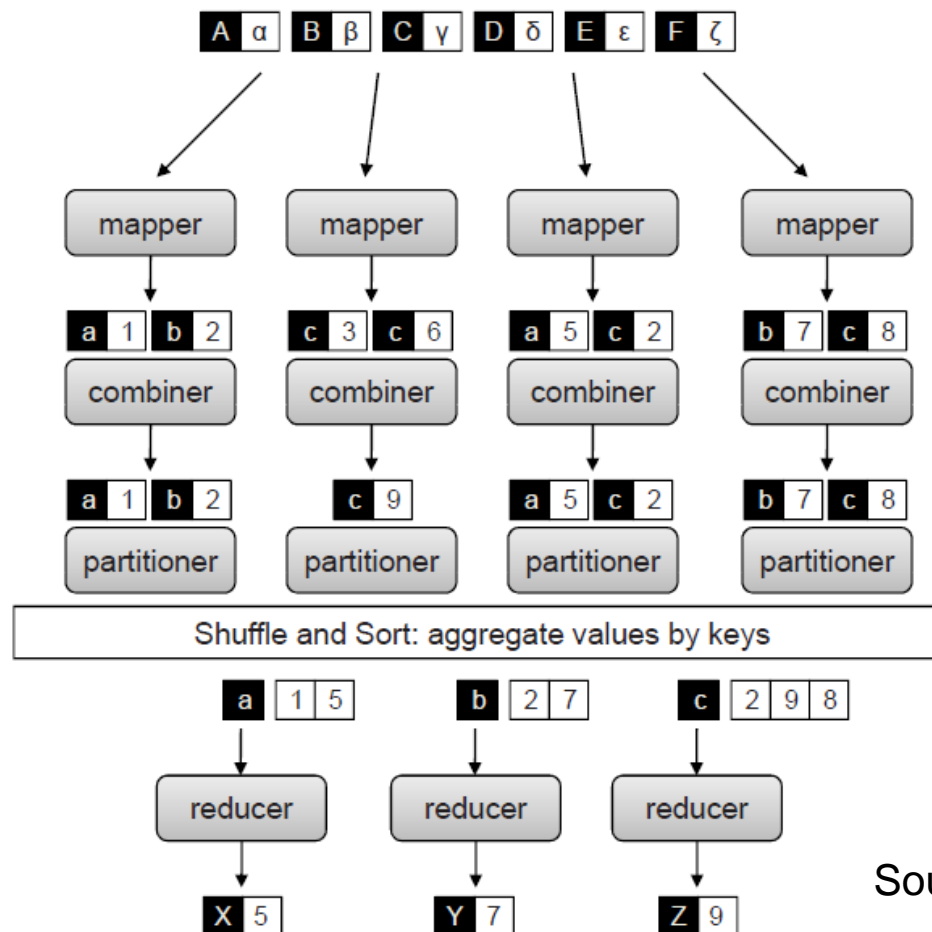
- dividing up the intermediate key space and assigning intermediate key-value pairs to reducers
- Often a simple hash of the key, e.g.,  $\text{hash}(k') \bmod n$

➤ **combine**  $(k2, \text{list}(v2)) \rightarrow \text{list}(k2', v2')$

- 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

- Combining 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

```
1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:     for all term  $t \in \text{doc } d$  do
4:       EMIT(term  $t$ , count 1)

1: class REDUCER
2:   method REDUCE(term  $t$ , counts  $[c_1, c_2, \dots]$ )
3:      $sum \leftarrow 0$ 
4:     for all count  $c \in \text{counts } [c_1, c_2, \dots]$  do
5:        $sum \leftarrow sum + c$ 
6:     EMIT(term  $t$ , count  $s$ )
```

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

# Word Count: Version 1

```
1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:      $H \leftarrow$  new ASSOCIATIVEARRAY
4:     for all term  $t \in$  doc  $d$  do
5:        $H\{t\} \leftarrow H\{t\} + 1$  ▷ Tally counts for entire document
6:     for all term  $t \in H$  do
7:       EMIT(term  $t$ , count  $H\{t\}$ )
```

## ➤ In-mapper combining:

- Emits a key-value pair for each **unique** term **per document**

# Word Count: Version 2

```
1: class MAPPER
2:   method INITIALIZE
3:      $H \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:   method MAP(docid  $a$ , doc  $d$ )
5:     for all term  $t \in \text{doc } d$  do
6:        $H\{t\} \leftarrow H\{t\} + 1$ 
7:   method CLOSE
8:     for all term  $t \in H$  do
9:        $\text{EMIT}(\text{term } t, \text{count } H\{t\})$ 
```

Setup() in Java

▷ Tally counts *across* documents

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

```
1: class MAPPER
2:   method MAP(string  $t$ , integer  $r$ )
3:     EMIT(string  $t$ , integer  $r$ )

1: class REDUCER
2:   method REDUCE(string  $t$ , integers  $[r_1, r_2, \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all integer  $r \in$  integers  $[r_1, r_2, \dots]$  do
6:        $sum \leftarrow sum + r$ 
7:        $cnt \leftarrow cnt + 1$ 
8:      $r_{avg} \leftarrow sum / cnt$ 
9:     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.,  $\text{Mean}(1,2,3,4,5)$   
 $\neq \text{Mean}(\text{Mean}(1,2), \text{Mean}(3,4,5))$
- It's not a problem for WordCount, but it's a problem here.

# Computing the Mean: Version 2

```
1: class MAPPER
2:   method MAP(string  $t$ , integer  $r$ )
3:     EMIT(string  $t$ , integer  $r$ )

1: class COMBINER
2:   method COMBINE(string  $t$ , integers  $[r_1, r_2, \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all integer  $r \in$  integers  $[r_1, r_2, \dots]$  do
6:        $sum \leftarrow sum + r$ 
7:        $cnt \leftarrow cnt + 1$ 
8:     EMIT(string  $t$ , pair ( $sum, cnt$ )) ▷ Separate sum and count

1: class REDUCER
2:   method REDUCE(string  $t$ , pairs  $[(s_1, c_1), (s_2, c_2) \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all pair  $(s, c) \in$  pairs  $[(s_1, c_1), (s_2, c_2) \dots]$  do
6:        $sum \leftarrow sum + s$ 
7:        $cnt \leftarrow cnt + c$ 
8:      $r_{avg} \leftarrow sum / cnt$ 
9:     EMIT(string  $t$ , integer  $r_{avg}$ )
```

➤ Does it work? Why?

# Computing the Mean: Version 3

```
1: class MAPPER
2:   method MAP(string  $t$ , integer  $r$ )
3:     EMIT(string  $t$ , pair ( $r$ , 1))

1: class COMBINER
2:   method COMBINE(string  $t$ , pairs  $[(s_1, c_1), (s_2, c_2) \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all pair  $(s, c) \in$  pairs  $[(s_1, c_1), (s_2, c_2) \dots]$  do
6:        $sum \leftarrow sum + s$ 
7:        $cnt \leftarrow cnt + c$ 
8:     EMIT(string  $t$ , pair ( $sum$ ,  $cnt$ ))

1: class REDUCER
2:   method REDUCE(string  $t$ , pairs  $[(s_1, c_1), (s_2, c_2) \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all pair  $(s, c) \in$  pairs  $[(s_1, c_1), (s_2, c_2) \dots]$  do
6:        $sum \leftarrow sum + s$ 
7:        $cnt \leftarrow cnt + c$ 
8:      $r_{avg} \leftarrow sum / cnt$ 
9:     EMIT(string  $t$ , pair ( $r_{avg}$ ,  $cnt$ ))
```

➤ Does it work? Why?

# Computing the Mean: Version 4

```
1: class MAPPER
2:   method INITIALIZE
3:      $S \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:      $C \leftarrow \text{new ASSOCIATIVEARRAY}$ 
5:   method MAP(string  $t$ , integer  $r$ )
6:      $S\{t\} \leftarrow S\{t\} + r$ 
7:      $C\{t\} \leftarrow C\{t\} + 1$ 
8:   method CLOSE
9:     for all term  $t \in S$  do
10:       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
  - $M = n \times n$  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:

```
1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:     for all term  $w \in \text{doc } d$  do
4:       for all term  $u \in \text{NEIGHBORS}(w)$  do
5:         EMIT(pair ( $w, u$ ), count 1)      ▷ Emit count for each co-occurrence

1: class REDUCER
2:   method REDUCE(pair  $p$ , counts [ $c_1, c_2, \dots$ ])
3:      $s \leftarrow 0$ 
4:     for all count  $c \in \text{counts } [c_1, c_2, \dots]$  do
5:        $s \leftarrow s + c$                   ▷ Sum co-occurrence counts
6:     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:
- $$\begin{aligned}(a, b) &\rightarrow 1 \\(a, c) &\rightarrow 2 \\(a, d) &\rightarrow 5 \\(a, e) &\rightarrow 3 \\(a, f) &\rightarrow 2\end{aligned}$$
- $$a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$$
- Each mapper emits  $a \rightarrow \{ b: \text{count}_b, c: \text{count}_c, d: \text{count}_d \dots \}$
- Reducers perform element-wise sum of associative arrays

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

# Stripes

## ➤ Pseudo-code:

```
1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:     for all term  $w \in \text{doc } d$  do
4:        $H \leftarrow \text{new ASSOCIATIVEARRAY}$ 
5:       for all term  $u \in \text{NEIGHBORS}(w)$  do
6:          $H\{u\} \leftarrow H\{u\} + 1$  ▷ Tally words co-occurring with  $w$ 
7:       EMIT(Term  $w$ , Stripe  $H$ )

1: class REDUCER
2:   method REDUCE(term  $w$ , stripes  $[H_1, H_2, H_3, \dots]$ )
3:      $H_f \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:     for all stripe  $H \in \text{stripes } [H_1, H_2, H_3, \dots]$  do
5:       SUM( $H_f, H$ ) ▷ Element-wise sum
6:     EMIT(term  $w$ , stripe  $H_f$ )
```

# Pairs vs. Stripes

## ➤ Pairs:

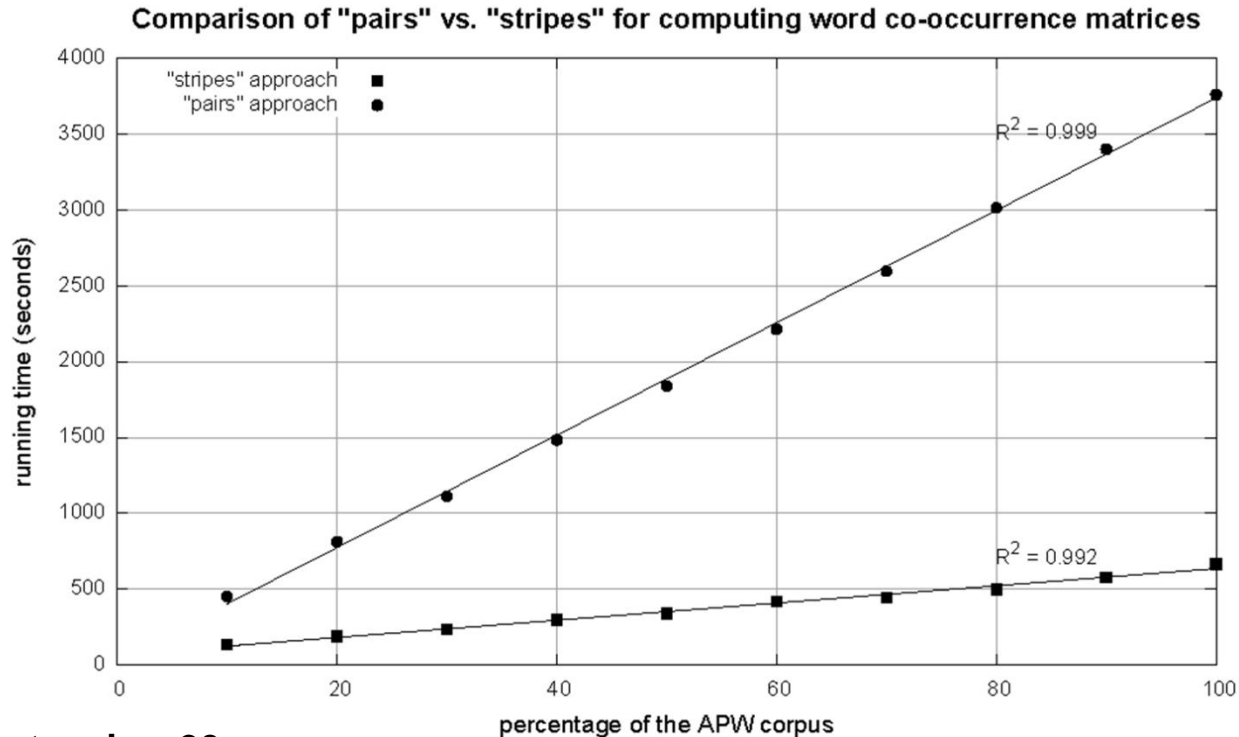
- Pro: Easy to understand and implement
- Con: Generate many key-value pairs

## ➤ 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:

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

- 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

$(a, b_1) \rightarrow 3$

$(a, b_2) \rightarrow 12$

$(a, b_3) \rightarrow 7$

$(a, b_4) \rightarrow 1$

...



$(a, b_1) \rightarrow 3 / 32$

$(a, b_2) \rightarrow 12 / 32$

$(a, b_3) \rightarrow 7 / 32$

$(a, b_4) \rightarrow 1 / 32$

...

Correct approach:

- 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
- 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 all pairs is a bad idea!
- Trick: getting the marginal counts to arrive at the reducer before the joint counts
  - For every pair, the mapper adds the count to the marginal count
  - When all pairs are processed, the mapper also emits the marginal count
  - Using order inversion, the marginal count arrives before the pairs at the reducer
  - The reducer remembers the marginal count, and computes the relative frequency for each pair being observed

## ➤ 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
- Main program → `job.setSortComparatorClass()`

# 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:
  - $\text{id} \rightarrow (t, v)$
- 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:
    - $(id, t) \rightarrow v$
  - 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