



UNIVERSITY OF  
**WATERLOO**

# Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2016)

Week 2: MapReduce Algorithm Design (2/2)

January 14, 2016

Jimmy Lin

David R. Cheriton School of Computer Science  
University of Waterloo

These slides are available at <http://lintool.github.io/bigdata-2016w/>

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States  
See <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> for details





Source: Wikipedia (Japanese rock garden)

# MapReduce Algorithm Design

- How do you express everything in terms of  $m$ ,  $r$ ,  $c$ ,  $p$ ?
- Toward “design patterns”

# MapReduce

A wide-angle photograph of a massive server room, likely a Google data center. The room is filled with floor-to-ceiling server racks, their front panels glowing with various colors (blue, yellow, green) from integrated LED lights. A complex network of grey metal walkways and support structures spans the entire space, with stairs leading up to different levels. The ceiling is a dark, multi-layered steel truss structure with recessed lighting. In the center of the image, the words "MapReduce" are overlaid in a large, bold, white sans-serif font.

# MapReduce: Recap

- Programmers must specify:

**map** ( $k, v$ )  $\rightarrow \langle k', v' \rangle^*$

**reduce** ( $k', v'$ )  $\rightarrow \langle k', v' \rangle^*$

- All values with the same key are reduced together

- Optionally, also:

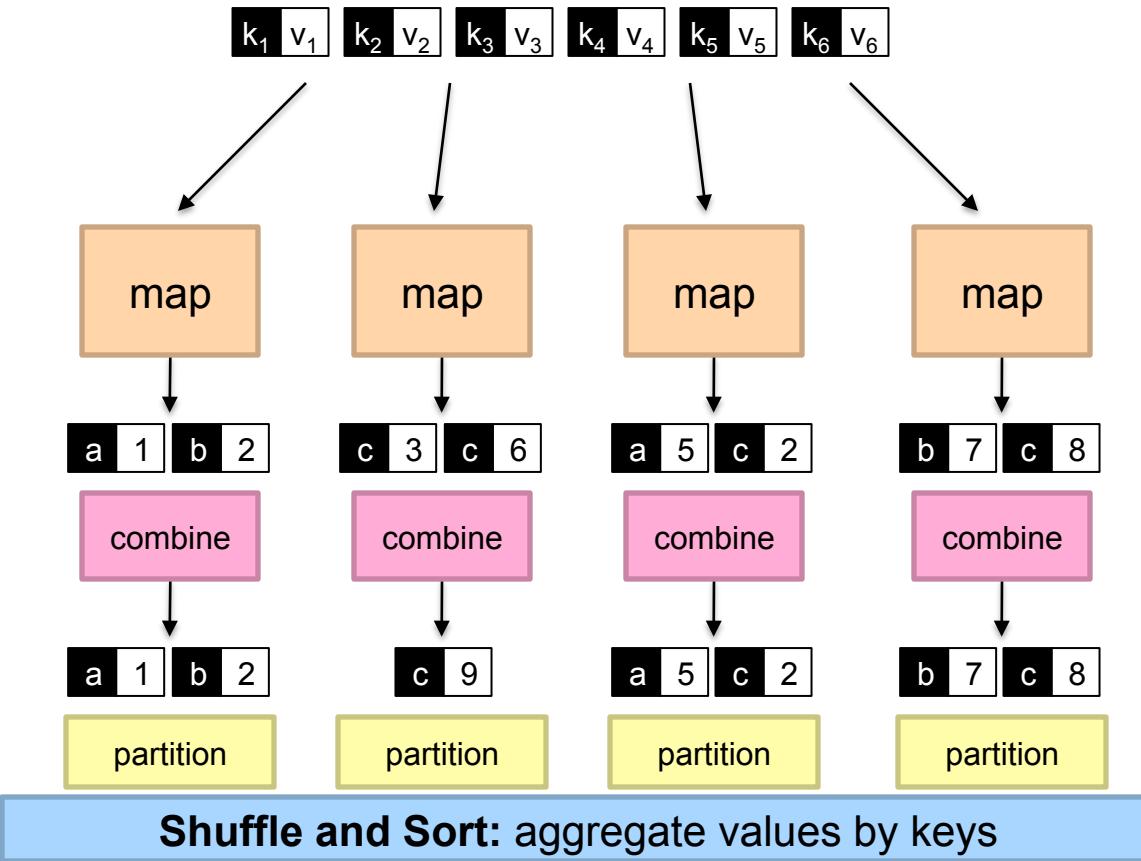
**partition** ( $k'$ , number of partitions)  $\rightarrow$  partition for  $k'$

- Often a simple hash of the key, e.g.,  $\text{hash}(k') \bmod n$
- Divides up key space for parallel reduce operations

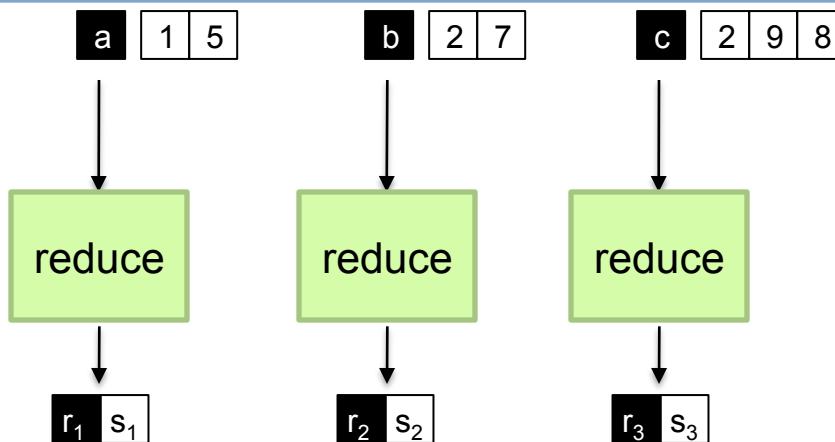
**combine** ( $k', v'$ )  $\rightarrow \langle k', v' \rangle^*$

- Mini-reducers that run in memory after the map phase
- Used as an optimization to reduce network traffic

- The execution framework handles everything else...



### Shuffle and Sort: aggregate values by keys



# “Everything Else”

- The execution framework handles everything else...
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
  - All algorithms must expressed in m, r, c, p
- You don't know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

# Tools for Synchronization

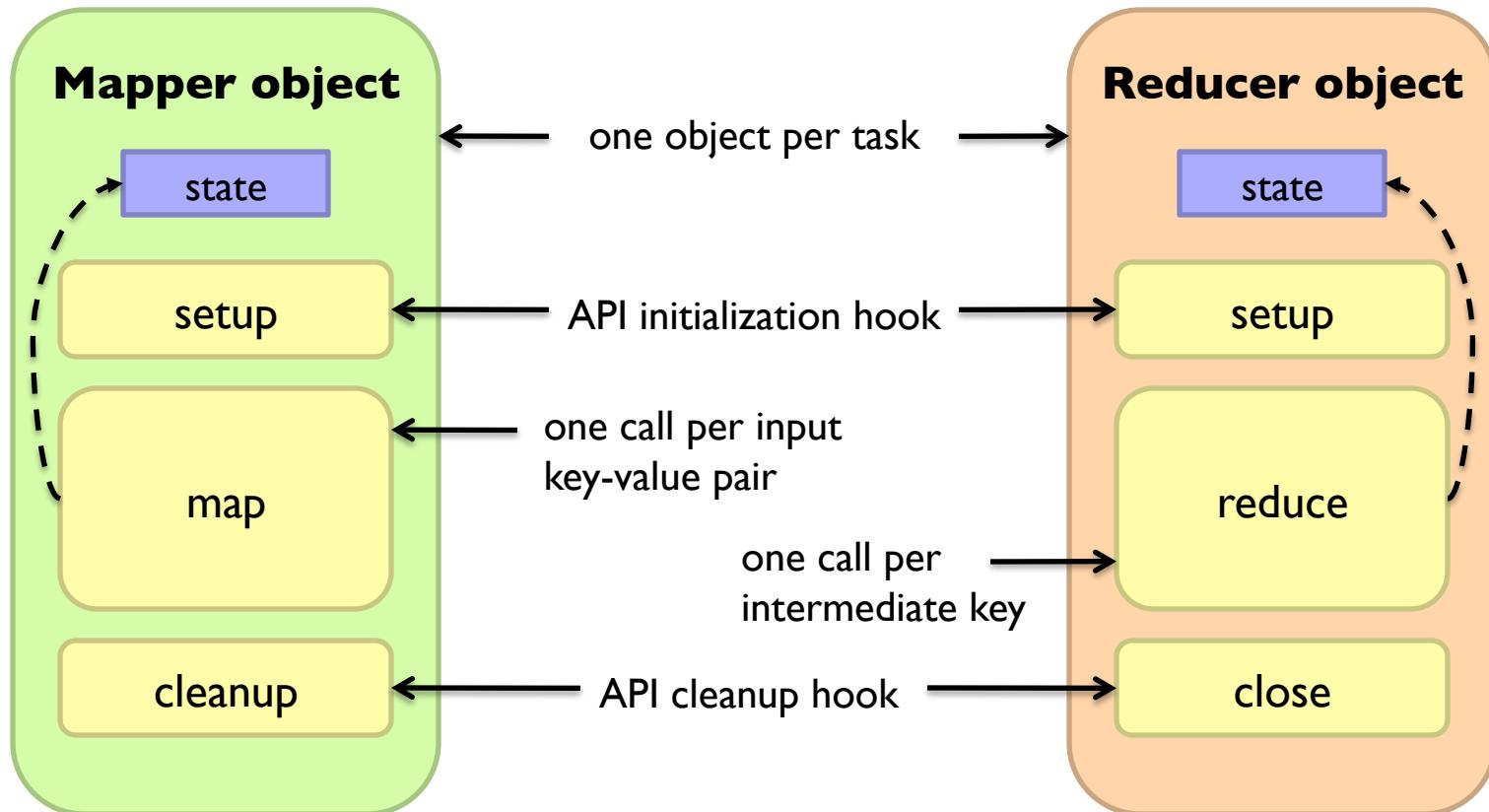
- Cleverly-constructed data structures
  - Bring partial results together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple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

# MapReduce API\*

- **Mapper<K<sub>in</sub>, V<sub>in</sub>, K<sub>out</sub>, V<sub>out</sub>>**
  - void setup(Mapper.Context context)  
Called once at the beginning of the task
  - void map(K<sub>in</sub> key, V<sub>in</sub> value, Mapper.Context context)  
Called once for each key/value pair in the input split
  - void cleanup(Mapper.Context context)  
Called once at the end of the task
- **Reducer<K<sub>in</sub>, V<sub>in</sub>, K<sub>out</sub>, V<sub>out</sub>>/Combiner<K<sub>in</sub>, V<sub>in</sub>, K<sub>out</sub>, V<sub>out</sub>>**
  - void setup(Reducer.Context context)  
Called once at the start of the task
  - void reduce(K<sub>in</sub> key, Iterable<V<sub>in</sub>> values, Reducer.Context context)  
Called once for each key
  - void cleanup(Reducer.Context context)  
Called once at the end of the task

\*Note that there are two versions of the API!

# Preserving State



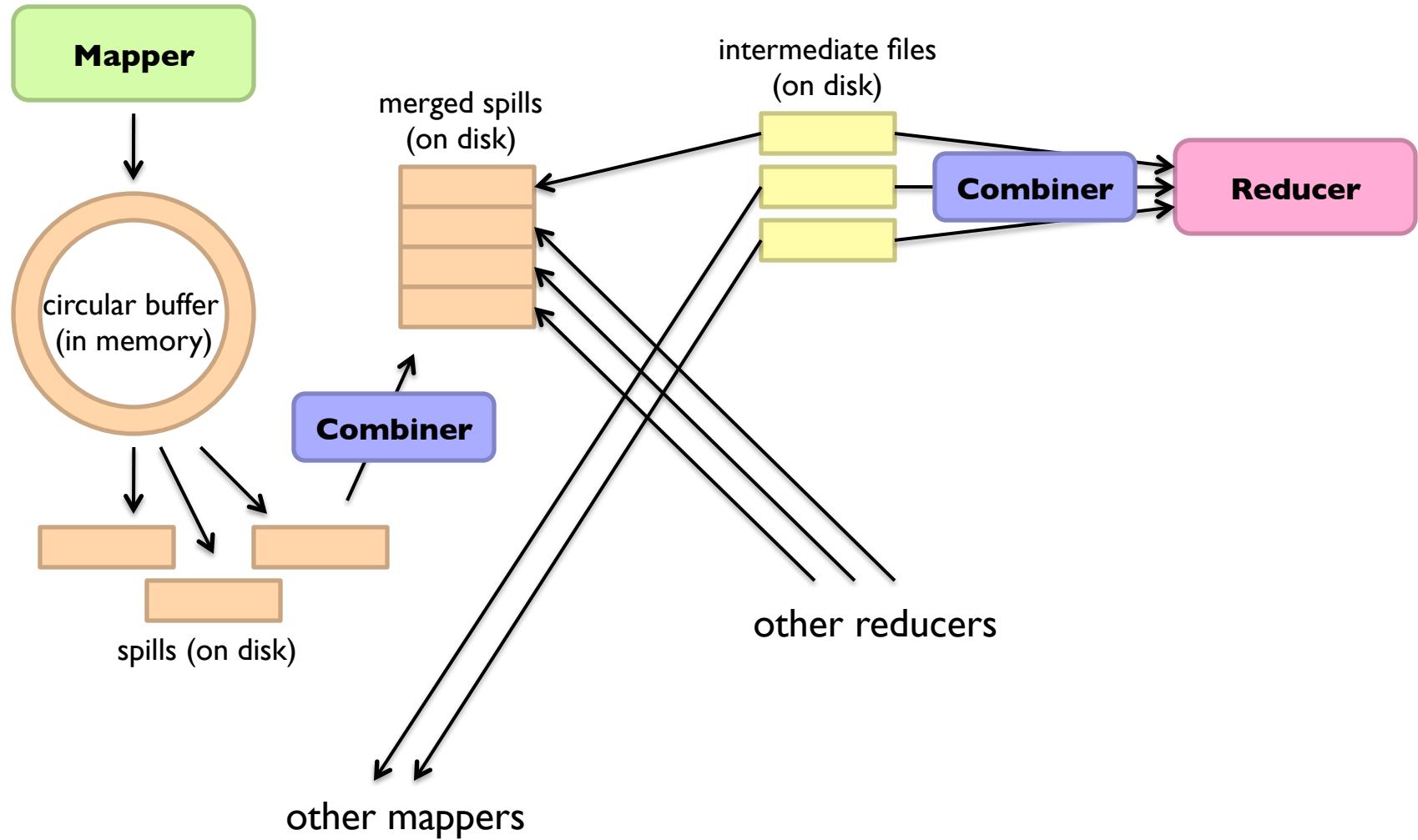
# Scalable Hadoop Algorithms: Themes

- Avoid object creation
  - (Relatively) costly operation
  - Garbage collection
- Avoid buffering
  - Limited heap size
  - Works for small datasets, but won't scale!

# Importance of Local Aggregation

- Ideal scaling characteristics:
  - Twice the data, twice the running time
  - Twice the resources, half the running time
- Why can't we achieve this?
  - Synchronization requires communication
  - Communication kills performance
- Thus... avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help

# Shuffle and Sort



# Word Count: Baseline

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         for all term t ∈ doc d do
4:             EMIT(term t, count 1)

1: class REDUCER
2:     method REDUCE(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         EMIT(term t, count s)
```

**What's the impact of combiners?**

# Word Count: Version I

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

**Are combiners still needed?**

# Word Count: Version 2

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

Key idea: preserve state across  
input key-value pairs!

▷ Tally counts *across* documents

Are combiners still needed?

# Design Pattern for Local Aggregation

- “In-mapper combining”
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

# Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not...
- Remember: combiner are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of integers associated with the same key

# Computing the Mean: Version I

```
1: class MAPPER
2:     method MAP(string  $t$ , integer  $r$ )
3:         EMIT(string  $t$ , integer  $r$ )
4:
5: class REDUCER
6:     method REDUCE(string  $t$ , integers  $[r_1, r_2, \dots]$ )
7:         sum  $\leftarrow 0$ 
8:         cnt  $\leftarrow 0$ 
9:         for all integer  $r \in \text{integers } [r_1, r_2, \dots]$  do
10:             sum  $\leftarrow sum + r$ 
11:             cnt  $\leftarrow cnt + 1$ 
12:              $r_{avg} \leftarrow sum / cnt$ 
13:             EMIT(string  $t$ , integer  $r_{avg}$ )
```

**Why can't we use reducer as combiner?**

# Computing the Mean: Version 2

```
1: class MAPPER
2:     method MAP(string  $t$ , integer  $r$ )
3:         EMIT(string  $t$ , integer  $r$ )
4:
5: class COMBINER
6:     method COMBINE(string  $t$ , integers  $[r_1, r_2, \dots]$ )
7:          $sum \leftarrow 0$ 
8:          $cnt \leftarrow 0$ 
9:         for all integer  $r \in$  integers  $[r_1, r_2, \dots]$  do
10:              $sum \leftarrow sum + r$ 
11:              $cnt \leftarrow cnt + 1$ 
12:         EMIT(string  $t$ , pair ( $sum, cnt$ ))           ▷ Separate sum and count
13:
14: class REDUCER
15:     method REDUCE(string  $t$ , pairs  $[(s_1, c_1), (s_2, c_2) \dots]$ )
16:          $sum \leftarrow 0$ 
17:          $cnt \leftarrow 0$ 
18:         for all pair  $(s, c) \in$  pairs  $[(s_1, c_1), (s_2, c_2) \dots]$  do
19:              $sum \leftarrow sum + s$ 
20:              $cnt \leftarrow cnt + c$ 
21:          $r_{avg} \leftarrow sum / cnt$ 
22:         EMIT(string  $t$ , integer  $r_{avg}$ )
```

**Why doesn't this work?**

# Computing the Mean: Version 3

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, pair (r, 1))
4:
5: class COMBINER
6:     method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
7:         sum ← 0
8:         cnt ← 0
9:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
10:             sum ← sum + s
11:             cnt ← cnt + c
12:         EMIT(string t, pair (sum, cnt))
13:
14: class REDUCER
15:     method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
16:         sum ← 0
17:         cnt ← 0
18:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
19:             sum ← sum + s
20:             cnt ← cnt + c
21:         ravg ← sum / cnt
22:         EMIT(string t, pair (ravg, cnt))
```

Fixed?

# Computing the Mean: Version 4

```
1: class MAPPER
2:   method INITIALIZE
3:      $S \leftarrow$  new ASSOCIATIVEARRAY
4:      $C \leftarrow$  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 ( $S\{t\}$ ,  $C\{t\}$ ))
```

**Are combiners still needed?**

# **MapReduce API**

`Mapper<Kin,Vin,Kout,Vout>`

`Combiner<Kin,Vin,Kout,Vout>`

`Reducer<Kin,Vin,Kout,Vout>`

# Algorithm Design: Running Example

- Term co-occurrence matrix for a text collection
  - $M = N \times N$  matrix ( $N = \text{vocabulary size}$ )
  - $M_{ij}$ : number of times  $i$  and  $j$  co-occur in some context  
(for concreteness, let's say context = sentence)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

# MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection
  - = specific instance of a large counting problem
    - A large event space (number of terms)
    - A large number of observations (the collection itself)
    - Goal: keep track of interesting statistics about the events
- Basic approach
  - Mappers generate partial counts
  - Reducers aggregate partial counts

**How do we aggregate partial counts efficiently?**

# First Try: “Pairs”

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit  $(a, b) \rightarrow \text{count}$
- Reducers sum up counts associated with these pairs
- Use combiners!

# Pairs: Pseudo-Code

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

1: class REDUCER
2:   method REDUCE(pair p, counts [c1, c2, ...])
3:     s  $\leftarrow$  0
4:     for all count c  $\in$  counts [c1, c2, ...] do
5:       s  $\leftarrow$  s + c                       $\triangleright$  Sum co-occurrence counts
6:     EMIT(pair p, count s)
```

# “Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)
  - Not many opportunities for combiners to work

# Another Try: “Stripes”

- Idea: group together pairs into an associative array

$(a, b) \rightarrow 1$

$(a, c) \rightarrow 2$

$(a, d) \rightarrow 5$

$(a, e) \rightarrow 3$

$(a, f) \rightarrow 2$

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

- Each mapper takes a sentence:

- Generate all co-occurring term pairs
- For each term, emit  $a \rightarrow \{ b: \text{count}_b, c: \text{count}_c, d: \text{count}_d \dots \}$

- Reducers perform element-wise sum of associative arrays

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

Key idea: cleverly-constructed data structure  
brings together partial results

# 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:          $H\{u\} \leftarrow H\{u\} + 1$                                  $\triangleright$  Tally words co-occurring with  $w$ 
7:       EMIT(Term  $w$ , Stripe  $H$ )
8:
9: class REDUCER
10:   method REDUCE(term  $w$ , stripes  $[H_1, H_2, H_3, \dots]$ )
11:      $H_f \leftarrow \text{new ASSOCIATIVEARRAY}$ 
12:     for all stripe  $H \in \text{stripes } [H_1, H_2, H_3, \dots]$  do
13:       SUM( $H_f, H$ )                                          $\triangleright$  Element-wise sum
14:     EMIT(term  $w$ , stripe  $H_f$ )
```

# “Stripes” Analysis

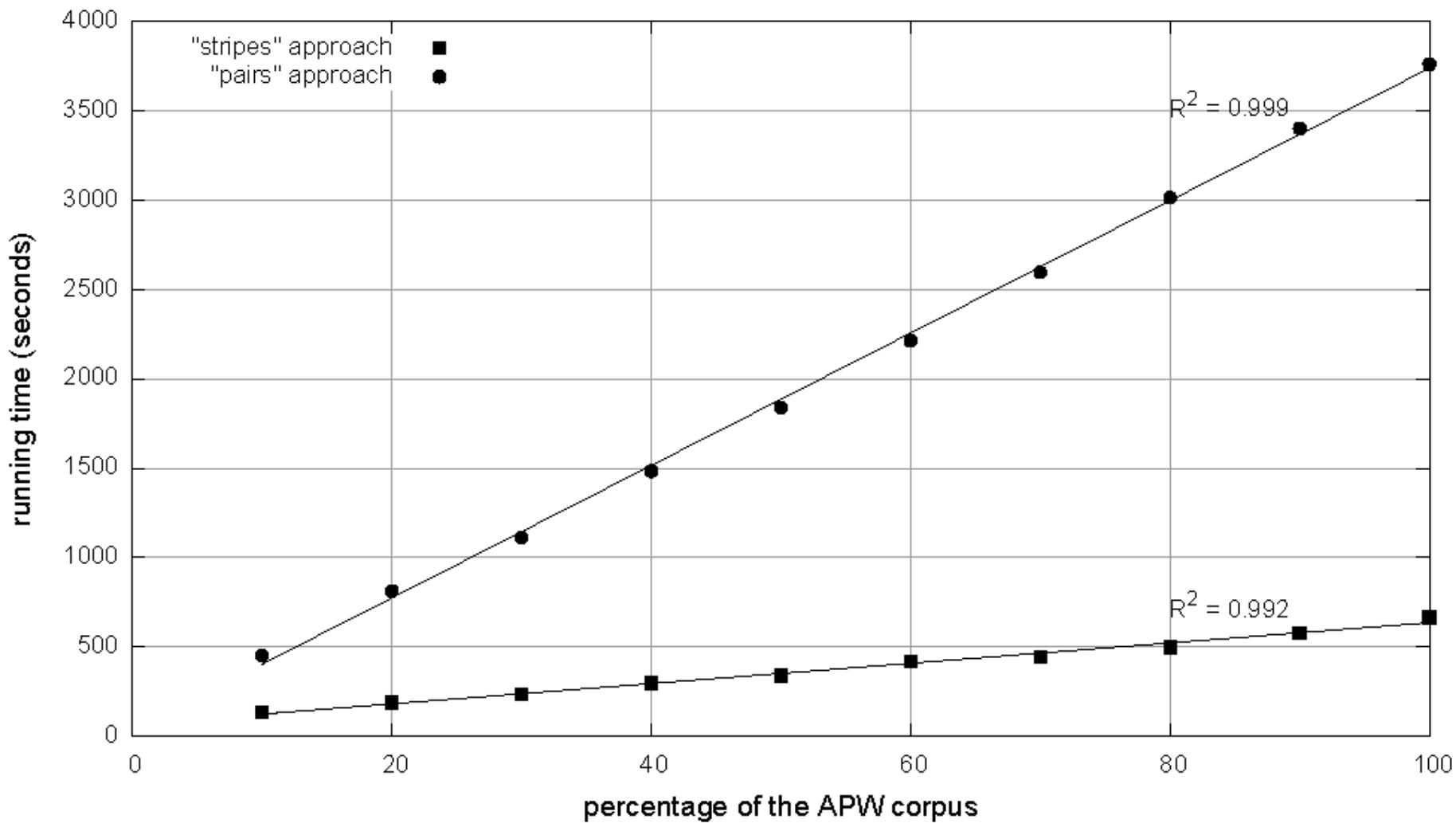
- Advantages

- Far less sorting and shuffling of key-value pairs
- Can make better use of combiners

- Disadvantages

- More difficult to implement
- Underlying object more heavyweight
- Fundamental limitation in terms of size of event space

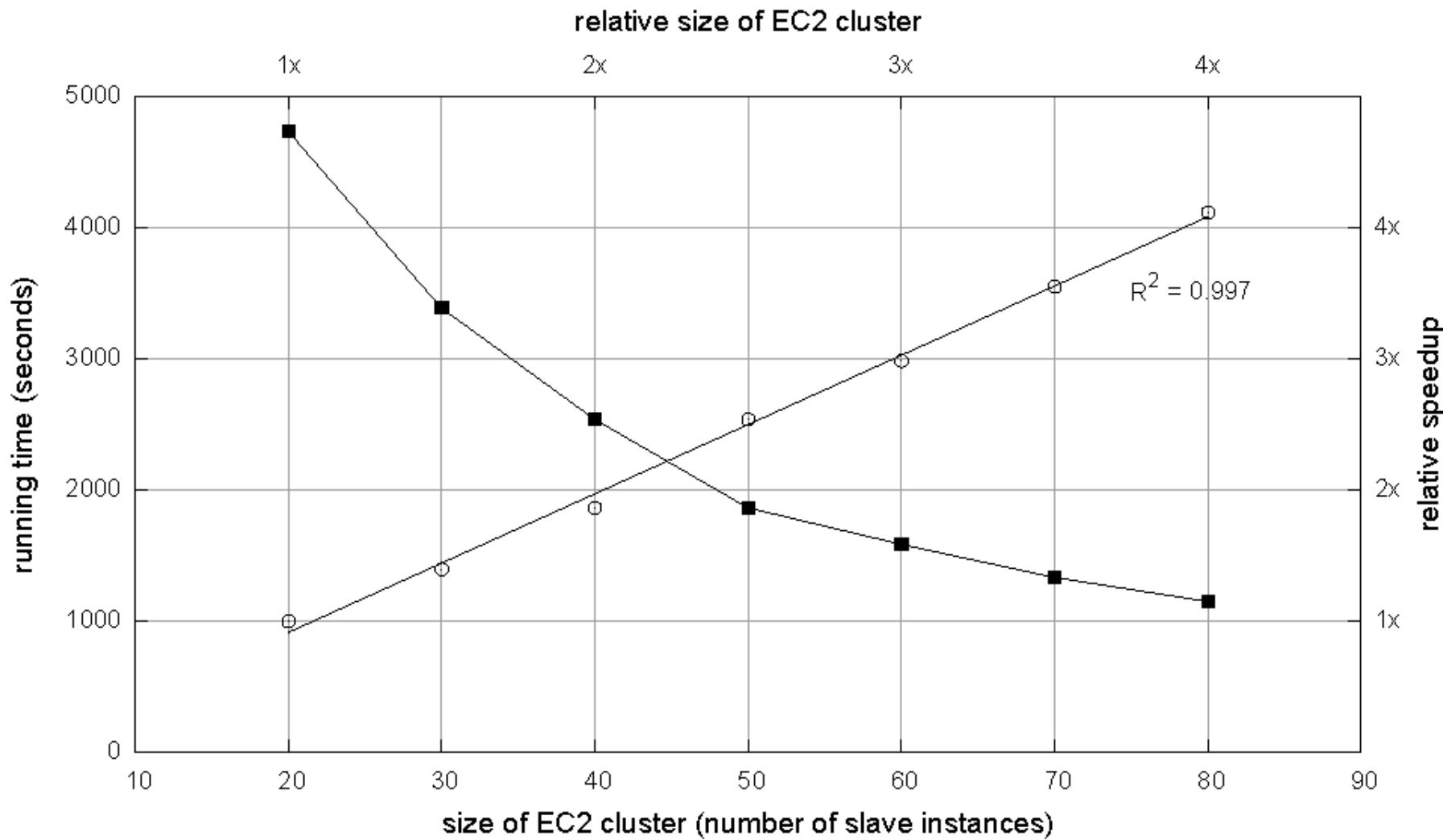
## Comparison of "pairs" vs. "stripes" for computing word co-occurrence matrices



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)

## Effect of cluster size on "stripes" algorithm



# Stripes >> Pairs?

- Tradeoff: Developer code vs. framework
- Tradeoff: CPU vs. RAM vs. disk vs. network
- Number of key-value pairs
  - Sorting and shuffling data 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
- Watch out for load imbalance

# Tradeoffs

- Pairs:
  - Generates *a lot* more key-value pairs
  - Less combining opportunities
  - More sorting and shuffling
  - Simple aggregation at reduce
- Stripes:
  - Generates fewer key-value pairs
  - More opportunities for combining
  - Less sorting and shuffling
  - More complex (slower) aggregation at reduce

**Where's the potential for load imbalance?**

# Relative Frequencies

- How do we estimate relative frequencies from counts?

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

- Why do we want to do this?
- How do we do this with MapReduce?

# **f(B|A): “Stripes”**

$a \rightarrow \{b_1:3, b_2 :12, b_3 :7, b_4 :1, \dots\}$

- Easy!
  - One pass to compute  $(a, *)$
  - Another pass to directly compute  $f(B|A)$

# $f(B|A)$ : “Pairs”

- What’s the issue?
  - Computing relative frequencies requires marginal counts
  - But the marginal cannot be computed until you see all counts
  - Buffering is a bad idea!
- Solution:
  - What if we could get the marginal count to arrive at the reducer first?

# $f(B|A)$ : “Pairs”

$(a, *) \rightarrow 32$

Reducer holds this value in memory

$(a, b_1) \rightarrow 3$

$(a, b_2) \rightarrow 12$

$(a, b_3) \rightarrow 7$

$(a, b_4) \rightarrow 1$

...



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

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

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

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

...

- For this to work:

- Must emit extra  $(a, *)$  for every  $b_n$  in mapper
- Must make sure all  $a$ 's get sent to same reducer (use partitioner)
- Must make sure  $(a, *)$  comes first (define sort order)
- Must hold state in reducer across different key-value pairs

# “Order Inversion”

- Common design pattern:
  - Take advantage of sorted key order at reducer to sequence computations
  - Get the marginal counts to arrive at the reducer before the joint counts
- Optimization:
  - Apply in-memory combining pattern to accumulate marginal counts

# Synchronization: Pairs vs. Stripes

- Approach 1: turn synchronization into an ordering problem
  - Sort keys into correct order of computation
  - Partition key space so that each reducer gets the appropriate set of partial results
  - Hold state in reducer across multiple key-value pairs to perform computation
  - Illustrated by the “pairs” approach
- Approach 2: construct data structures that bring partial results together
  - Each reducer receives all the data it needs to complete the computation
  - Illustrated by the “stripes” approach

# Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
- What if want to sort value also?
  - E.g.,  $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r) \dots$

# Secondary Sorting: Solutions

- Solution 1:
  - Buffer values in memory, then sort
  - Why is this a bad idea?
- Solution 2:
  - “Value-to-key conversion” design pattern: form composite intermediate key,  $(k, v_1)$
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?

# Recap: Tools for Synchronization

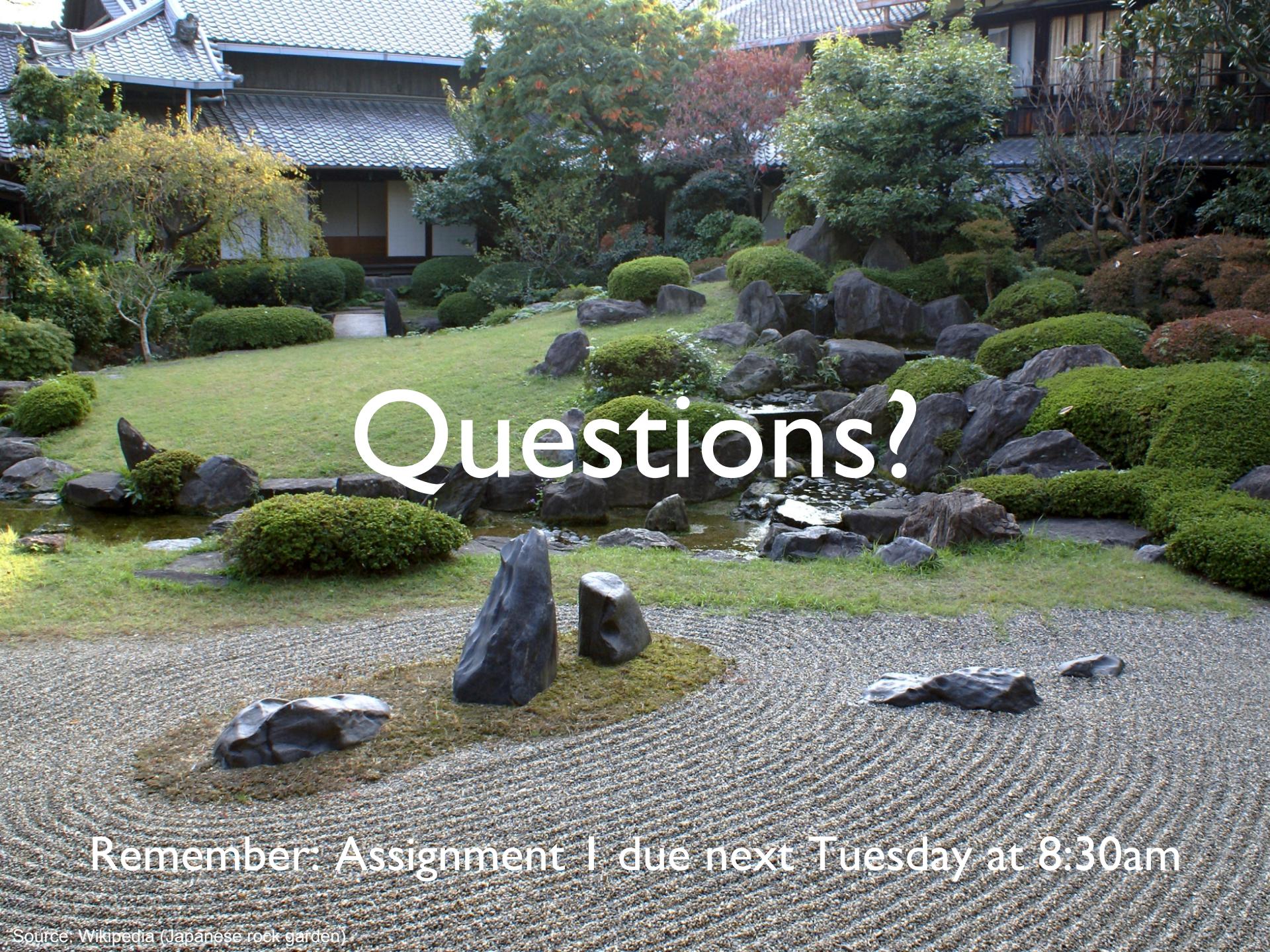
- Cleverly-constructed data structures
  - Bring data together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values

# Issues and Tradeoffs

- Tradeoff: Developer code vs. framework
- Tradeoff: CPU vs. RAM vs. disk vs. network
- Number of key-value pairs
  - Sorting and shuffling data 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
- Watch out for load imbalance

# 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!
  - There's no such thing as "consistent data"
  - Watch out for corner cases
  - Isolate unexpected behavior, bring local

A photograph of a traditional Japanese rock garden. The foreground features a gravel path with raked patterns. Several large, dark, irregular stones are scattered across the garden. In the middle ground, there is a small pond with rocks and a waterfall. The background consists of a building with a tiled roof and various trees and shrubs, some with autumn-colored leaves.

# Questions?

Remember: Assignment I due next Tuesday at 8:30am