# MapReduce: Algorithm Design

Juliana Freire

Some slides borrowed from Jimmy Lin, Jeff Ullman, Jerome Simeon, and Jure Leskovec

# MapReduce: Recap

- Sequentially read a *lot* of data
- Map: extract something we care about

  **map** (k, v) → <k', v'>*

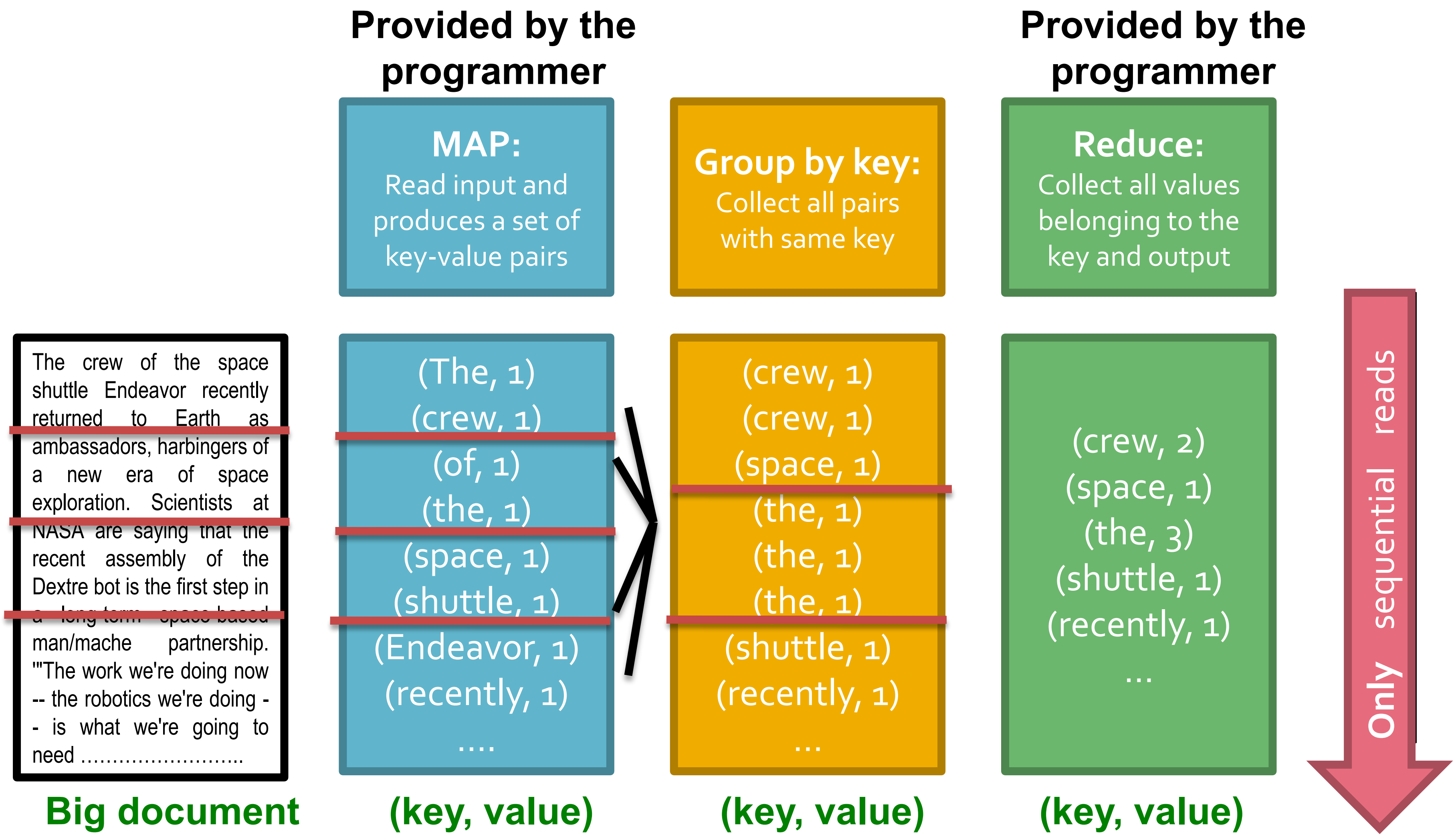
- Group by key: Sort and Shuffle

- Reduce: aggregate, summarize, filter, or transform

  **reduce** (k', v') → <k', v'>*
- Write the result

*Structure remains the same, Map and Reduce change to fit the problem*

# Mapreduce: WordCount



# MapReduce: Recap

- Combiners combines the values of all keys of a single mapper
  - Much less data needs to be shuffled
- Partition function: controls how keys get partitioned
  - Default: hash(key) mod R
  - Sometimes it is useful to override the default, e.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

# MapReduce: Environment

Takes care of:

- Partitioning the input data
- Scheduling the execution of the program on multiple machines
- Performing the *group by key* step
- Handling machine failures
- Managing inter-machine communication

# MapReduce: Data Flow

- Input and final output are stored on the *distributed file system* (DFS)
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data

  You can specify a directory where your input files reside using *MultipleInputs.addInputPath*

- Intermediate results are stored on *local FS* of Map and Reduce workers

- Output is often input to another MapReduce task

# Coordination: Master Node

- The master node handles coordination:
  - Task status: (idle, in-progress, completed)
  - *Idle tasks* get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its *R* intermediate files, one for each reducer
  - Master pushes this info to reducers

- Master pings workers periodically to detect failures

# Failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce task is aborted and client is notified

# How many Map and Reduce tasks?

- *M* map tasks, *R* reduce tasks
- **Rule of thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up

  recovery from worker failures

  Note: You can increase the number of Map task by modifying JobConf's *conf.setNumMapTasks(int num)*
- ***Usually R is smaller than M***
  - Output is spread across *R* files

# Some Constraints and Unknowns

- 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

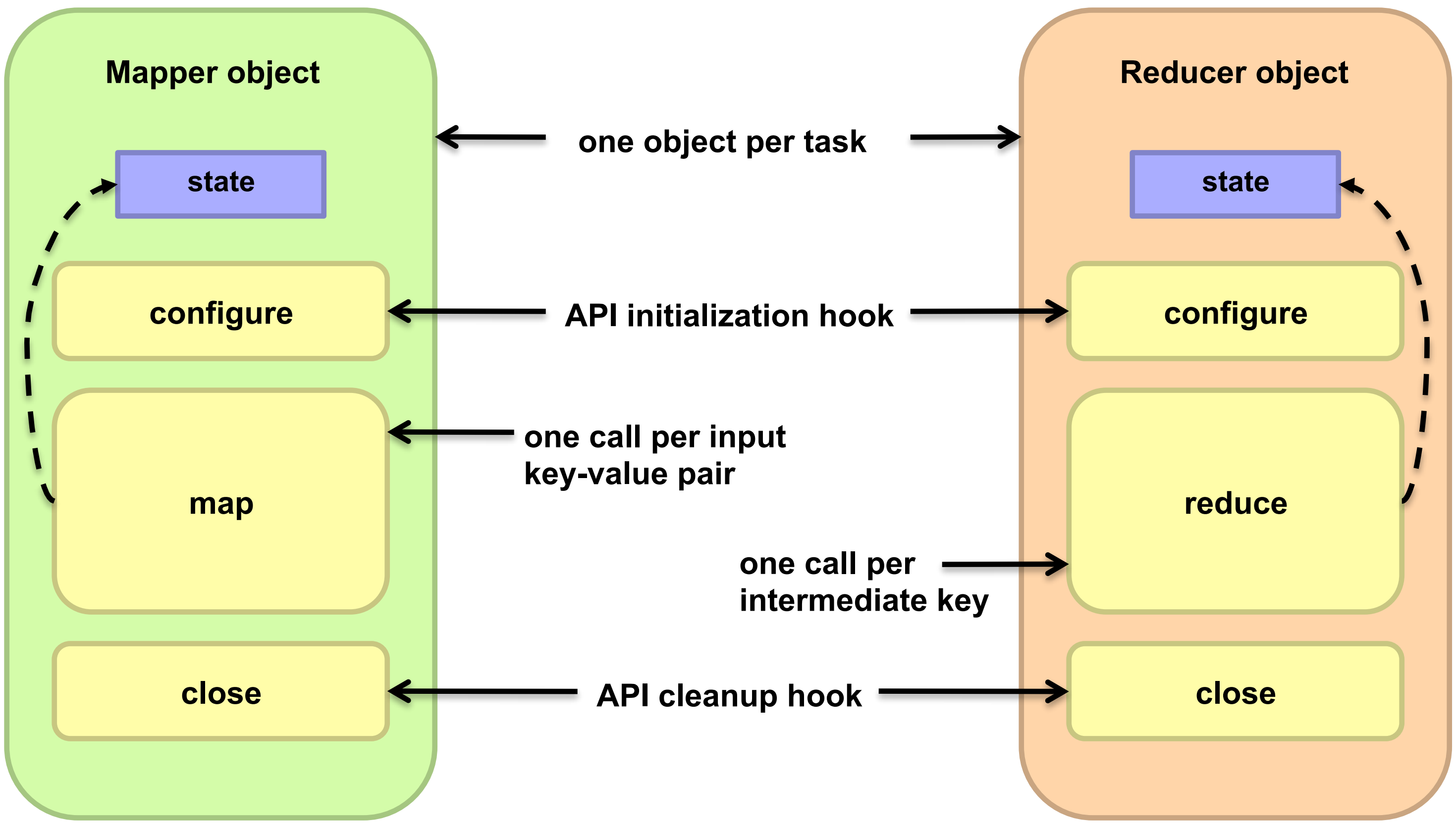
# Designing Algorithms for MapReduce

- Need to adapt to a restricted model of computation
- Goals
  - Scalability: adding machines will make the algo run faster
  - Efficiency: resources will not be wasted
- The translation some algorithms into MapReduce isn't always obvious
- But there are useful design patterns that can help
- We will cover some and use examples to illustrate how they can be applied

# 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

# Preserving State



# Towards Scalable Hadoop Algorithms

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

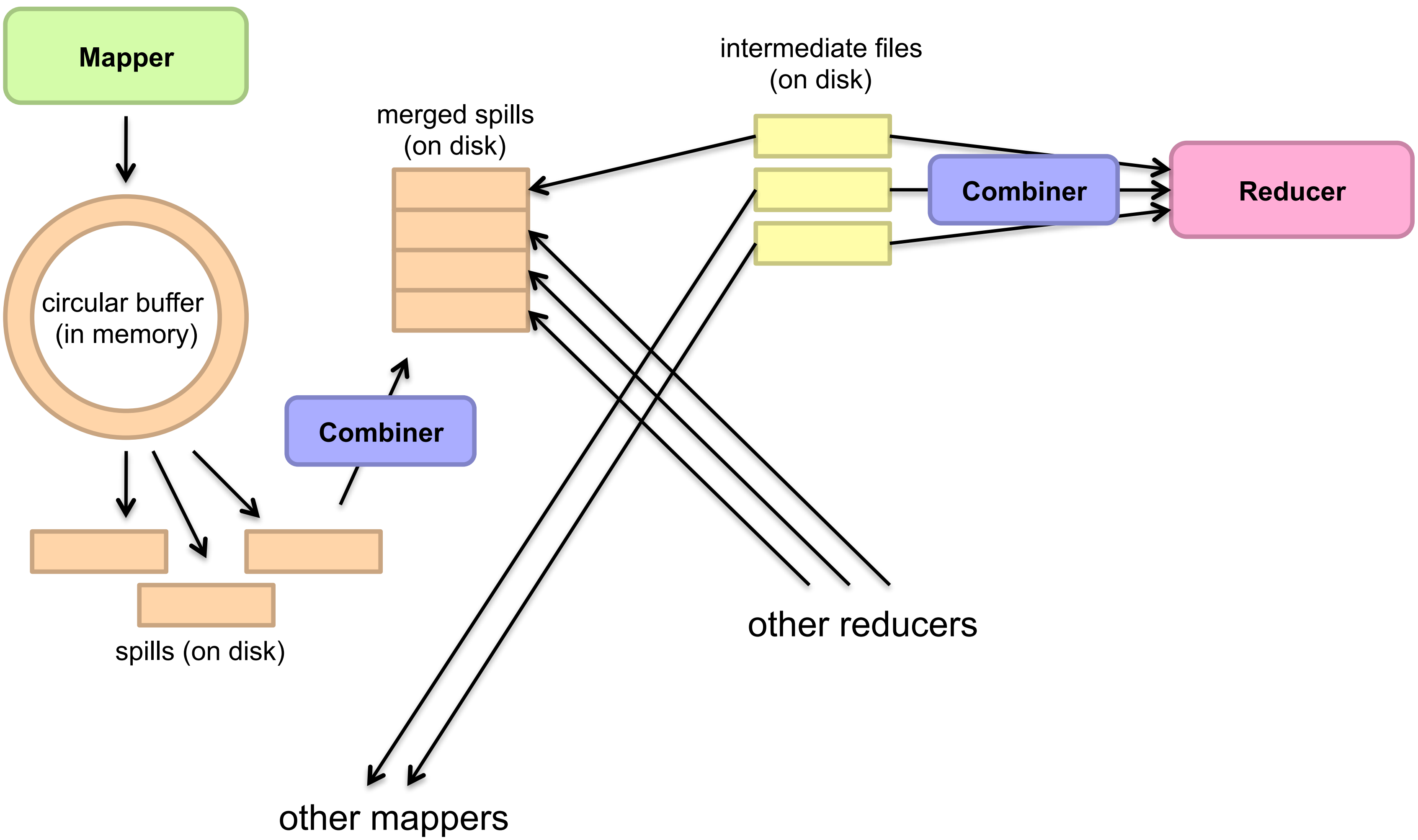
# Towards Scalable Hadoop Algorithms

- 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 in Hadoop

- Probably the most complex aspect of MapReduce!
- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are “spilled” to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs here
- Reduce side
  - First, *map outputs are copied over to reducer machine*
  - *“Sort” is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs here*
  - Final merge pass goes directly into reducer

# Shuffle and Sort



# DESIGN PATTERNS

# Strategy: Local Aggregation

- Use combiners
- Do aggregation inside mappers

# 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)
```

Suppose the collection has a total of *n* terms and *d* distinct terms.
What are the communication costs without a combiner?

What are the communication costs with a combiner?

# Word Count: Aggregate in Mapper

```
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})
```

Are combiners still needed?

# Word Count: Aggregate in Mapper (v 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})
```

Key: preserve state across input key-value pairs!

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
  – Explicit control aggregation
  – Speed

*Why is this faster than actual combiners?*

# 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
  - Explicit control aggregation
  - Speed
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

# Limiting Memory Usage

- To limit memory usage when using the in-mapper combining technique, block input key-value pairs and flush in-memory data structures periodically
  - E.g., counter variable that keeps track of the number of input key-value pairs that have been processed
- Memory usage threshold needs to be determined empirically: with too large a value, the mapper may run out of memory, but with too small a value, opportunities for local aggregation may be lost
- Note: Hadoop physical memory is split between multiple tasks that may be running on a node concurrently – difficult to coordinate resource consumption

# Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners

*When is this the case?*

# Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners

    *When is this the case?*
  - Often, not…works only when reducer is commutative and associative
- Remember: combiner are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of all integers associated with the same key

# 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 [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
```

Can we use the reducer as a combiner?

# Computing the Mean: Version 1

Mean(**1**; **2**; **3**; **4**; **5**) **?=?**

Mean(Mean(**1**; **2**);Mean(**3**; **4**; **5**))

# 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 [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        r_avg ← sum/cnt
        EMIT(string t, integer r_avg)
```

How would you fix this?

# 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 this work?

# 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 [(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
        Emit(string t, pair (sum, cnt))
```

```
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, pair (r_avg, cnt))
```

Fixed? Can you make this more efficient?

# 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}))
```

# Algorithm Design: Running Example

- Term co-occurrence matrix for a text collection
  - M = N x N matrix (N = 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) → count
- Reducers sum up counts associated with these pairs
- Use combiners!

# 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 [c_1, c_2, ...])
        s ← 0
        for all count c ∈ counts [c_1, c_2, ...] do
            s ← s + c                             ▷ Sum co-occurrence counts
        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

$$
\begin{array}{l}
(a, b) \rightarrow 1 \\
(a, c) \rightarrow 2 \\
(a, d) \rightarrow 5 \\
(a, e) \rightarrow 3 \\
(a, f) \rightarrow 2
\end{array}
\qquad\qquad
a \rightarrow \{\ b{:}\ 1,\ c{:}\ 2,\ d{:}\ 5,\ e{:}\ 3,\ f{:}\ 2\ \}
$$

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: $\text{count}_b$, c: $\text{count}_c$, d: $\text{count}_d$ … }
- Reducers perform element-wise sum of associative arrays

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

Key: cleverly-constructed data structure brings together partial results

# 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)
```

What are the advantages of stripes?

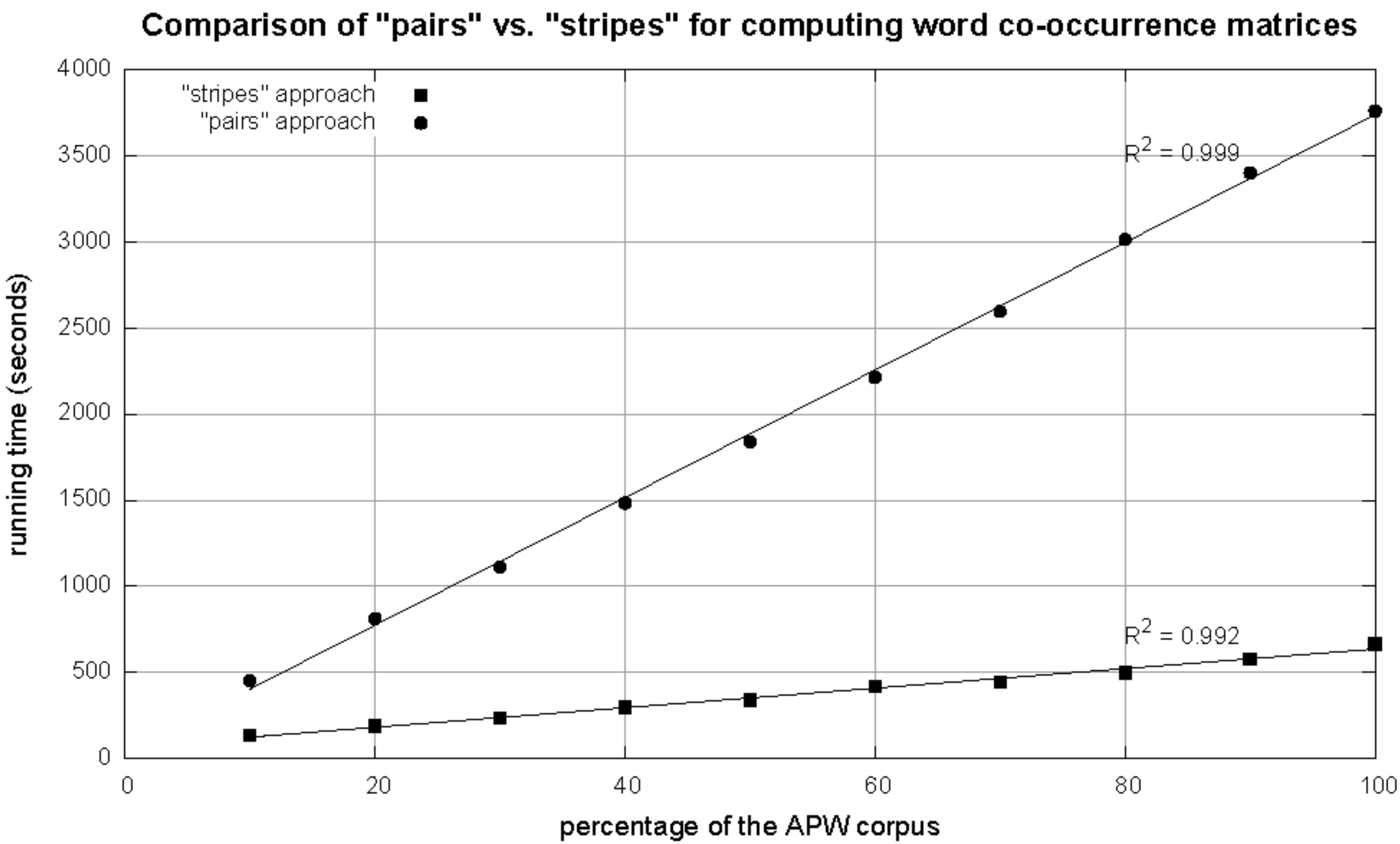
# “Stripes” Analysis

- 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

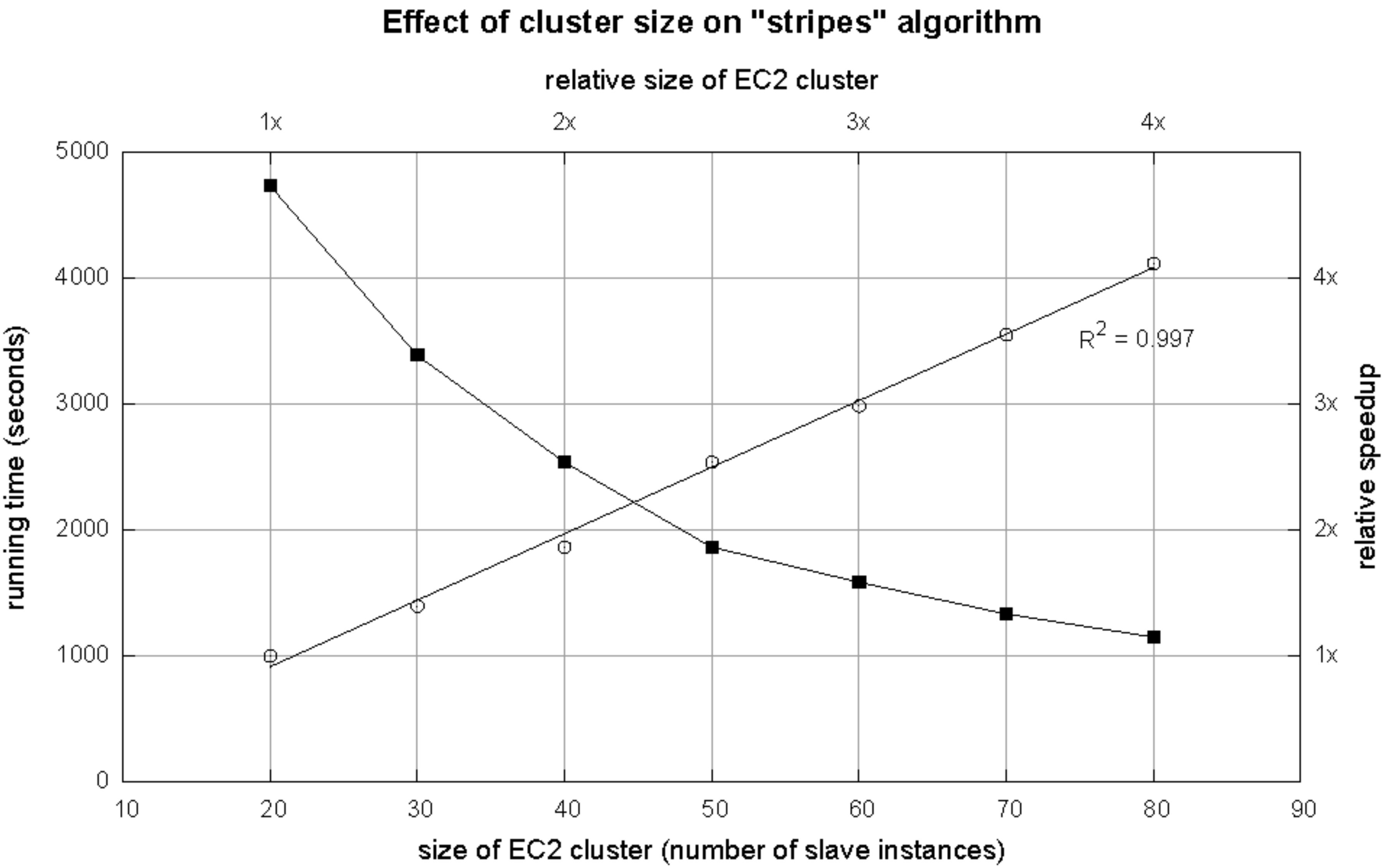
# What about combiners?

- Both algorithms can benefit from the use of combiners, since the respective operations in their reducers (addition and element-wise sum of associative arrays) are both commutative and associative.

- Are combiners equally effective in both pairs and 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)



Effect of cluster size on "stripes" algorithm
relative size of EC2 cluster
1x
2x
3x
4x
5000
4000
3000
2000
1000
0
running time (seconds)
$R^2 = 0.997$
relative speedup
4x
3x
2x
1x
10
20
30
40
50
60
70
80
90
size of EC2 cluster (number of slave instances)

# Relative Frequencies

- How do we estimate relative frequencies from counts?

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

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

# f(B|A): "Stripes"

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

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

# f(B|A): “Pairs”

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

$(a, b_1) \to 3$
$(a, b_2) \to 12$
$(a, b_3) \to 7$
$(a, b_4) \to 1$
…

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

- 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
  - 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
- Optimizations
  - Apply in-memory combining pattern to accumulate marginal counts
  - Should we apply combiners?

# 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., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$

# 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

- 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

# REASONING ABOUT COST

# Cost Measures for Algorithms

1. *Communication cost* = total I/O of all processes.
2. *Elapsed communication cost* = max of I/O along any path.
3. (*Elapsed* ) *computation costs* analogous, but count only running time of processes.

# Example: Cost Measures

- For a map-reduce algorithm:
  - Communication cost = input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - Elapsed communication cost is the sum of the largest input + output for any map process, plus the same for any reduce process.

# What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates.
  – Ignore one or the other.
- Total costs tell what you pay in rent from your friendly neighborhood cloud.
- Elapsed costs are wall-clock time using parallelism.

# JOINS IN MAPREDUCE

# Join By Map-Reduce

- Our first example of an algorithm in this framework is a map-reduce example.
- Compute the natural join $R(A,B) \bowtie S(B,C)$.
- R and S each are stored in files.
- Tuples are pairs (a,b) or (b,c).

# Map-Reduce Join – (2)

- Use a hash function *h* from B-values to 1..*k*.
- A Map process turns input tuple R(a,b) into key-value pair (b,(a,R)) and each input tuple S(b,c) into (b,(c,S)).

# Map-Reduce Join – (3)

- Map processes send each key-value pair with key *b* to Reduce process h(b).
  - Hadoop does this automatically; just tell it what *k* is.
- Each Reduce process matches all the pairs (b,(a,R)) with all (b,(c,S)) and outputs (a,b,c).

# Cost of Map-Reduce Join

- Total communication cost = $O(|R|+|S|+|R \bowtie S|)$.
- Elapsed communication cost = $O(s)$.
  - We're going to pick $k$ and the number of Map processes so I/O limit $s$ is respected.
  - We put a limit $\boldsymbol{s}$ on the amount of input or output that any one process can have. $\boldsymbol{s}$ **could be:**
    - What fits in main memory
    - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size.
  - So computation costs are like communication costs.

# Three-Way Join

- We shall consider a simple join of three relations, the natural join $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$.
- One way: cascade of two 2-way joins, each implemented by map-reduce.
- Fine, unless the 2-way joins produce large intermediate relations.

# Example: Large Intermediate Relations

- A = "good pages"; B, C = "all pages"; D = "spam pages."
- R, S, and T each represent links.
- 3-way join = "path of length 3 from good page to spam page.
- R $\bowtie$ S = paths of length 2 from good page to any; S $\bowtie$ T = paths of length 2 from any page to spam page.

# Another 3-Way Join

- Reduce processes use hash values of entire S(B,C) tuples as key.
- Choose a hash function *h* that maps B- and C-values to *k* buckets.
- There are $k^2$ Reduce processes, one for each (B-bucket, C-bucket) pair.

# Mapping for 3-Way Join

- We map each tuple S(b,c) to

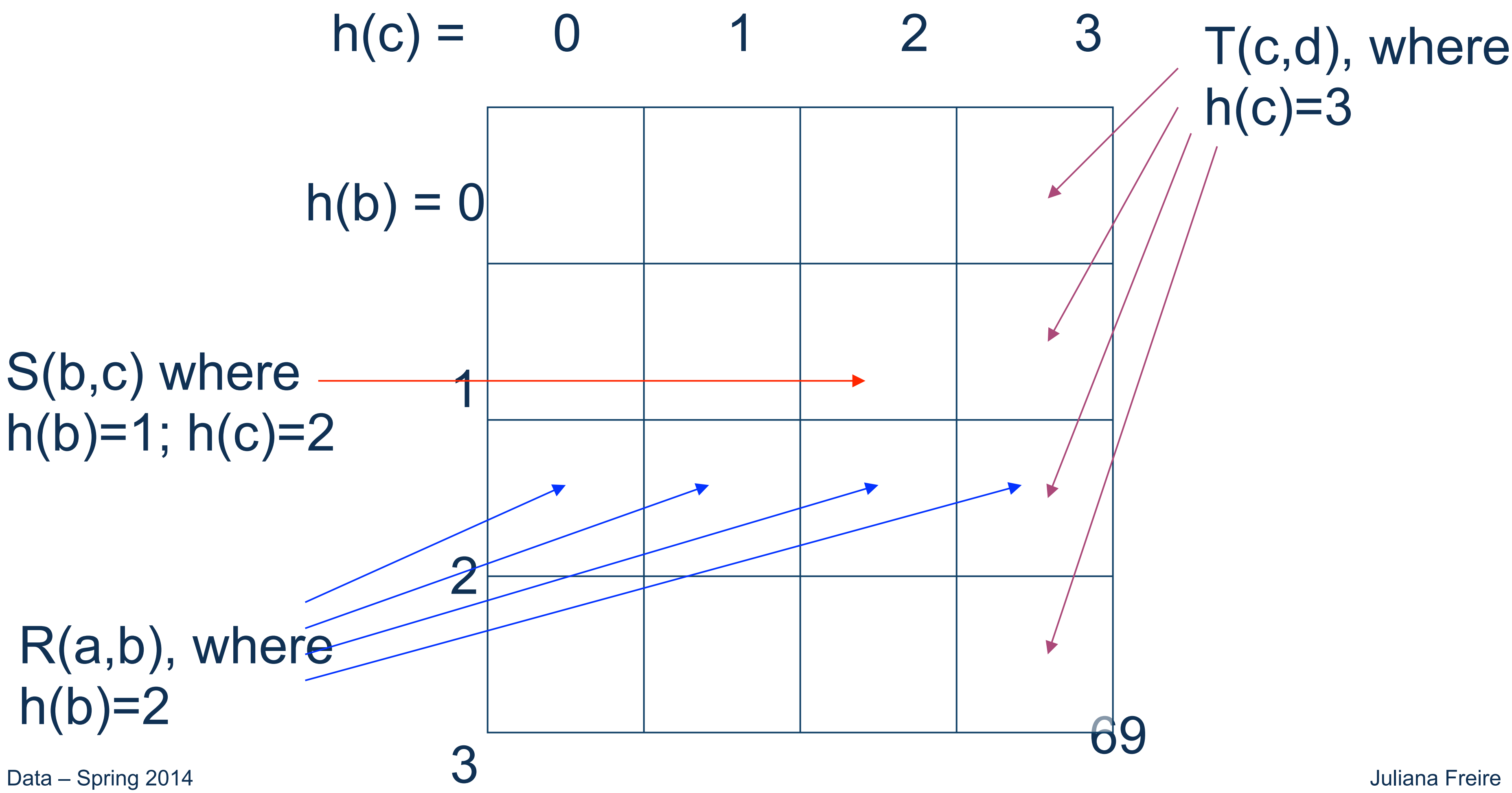
((h(b), h(c)), (S, b, c)).

- We map each R(a,b) tuple to
((h(b), y), (R, a, b)) for all $y = 1, 2,\ldots,k$.
- We map each T(c,d) tuple to
((x, h(c)), (T, c, d)) for all $x = 1, 2,\ldots,k$.

Keys

Values

# Assigning Tuples to Reducers

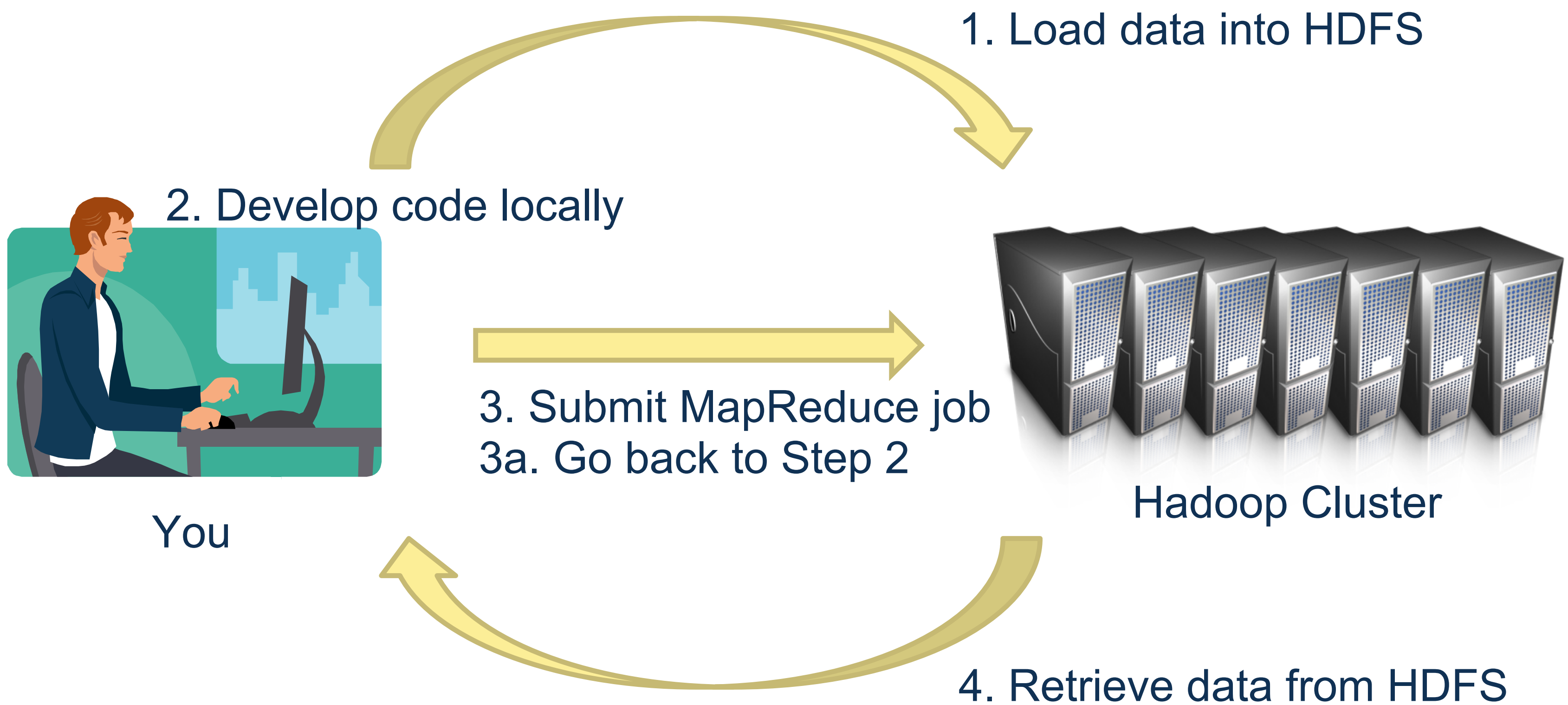


# Job of the Reducers

- Each reducer gets, for certain B-values *b* and C-values *c* :
  1. All tuples from R with B = *b*,
  2. All tuples from T with C = *c*, and
  3. The tuple S(b,c) if it exists.
- Thus it can create every tuple of the form (a, b, c, d) in the join.

# RUNNING MAPREDUCE JOBS

# Hadoop Workflow



# On Amazon: With EC2

0. Allocate Hadoop cluster

1. Load data into HDFS

EC2

2. Develop code locally

You

3. Submit MapReduce job

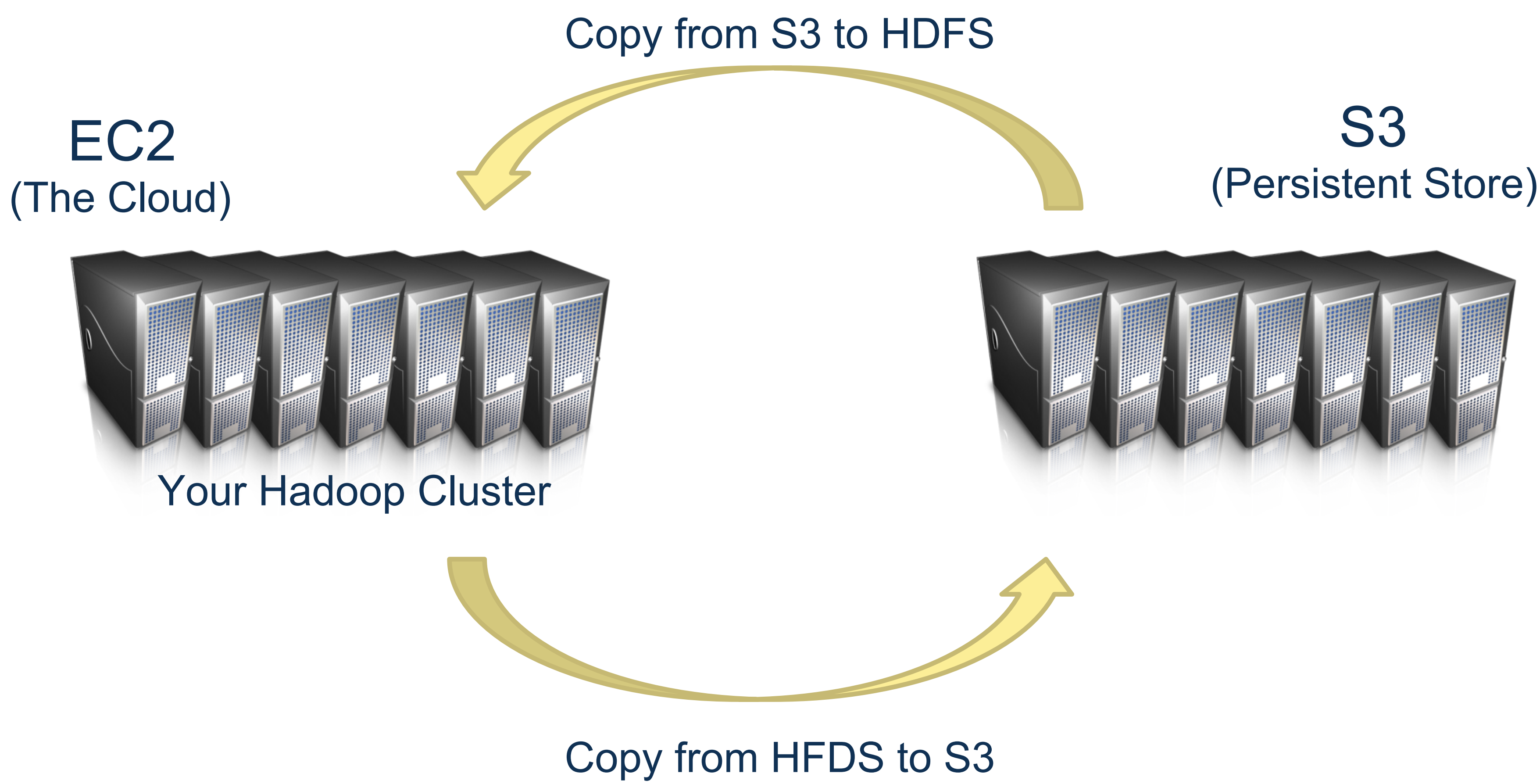
3a. Go back to Step 2


4. Retrieve data from HDFS

5. Clean up!

Uh oh.  Where did the data go?

# On Amazon: EC2 and S3



# Debugging Hadoop

- First, take a deep breath
- Start small, start locally
- Strategies
  - Learn to use the webapp
  - Where does println go?
  - Don’t use println, use logging
  - Throw RuntimeExceptions

# Discussion About Assignment 2

- Constraints are your friends, in their absence the semantics can be confusing
- In your assignment, the tables did not have keys or foreign keys
  - Join on station name was problematic
- Real data is messy, noisy
- When you get data from different sources, you need to be careful about combining them
  - Set of stations in one table was different from the other

# References

- Data Intensive Text Processing with MapReduce, Lin and Dyer (Chapter 3)
- Mining of Massive Data Sets, Rajaraman et al. (Chapter 2)