MapReduce:
Algorithm Design Patterns

Juliana Freire & Cláudio Silva

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

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

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

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

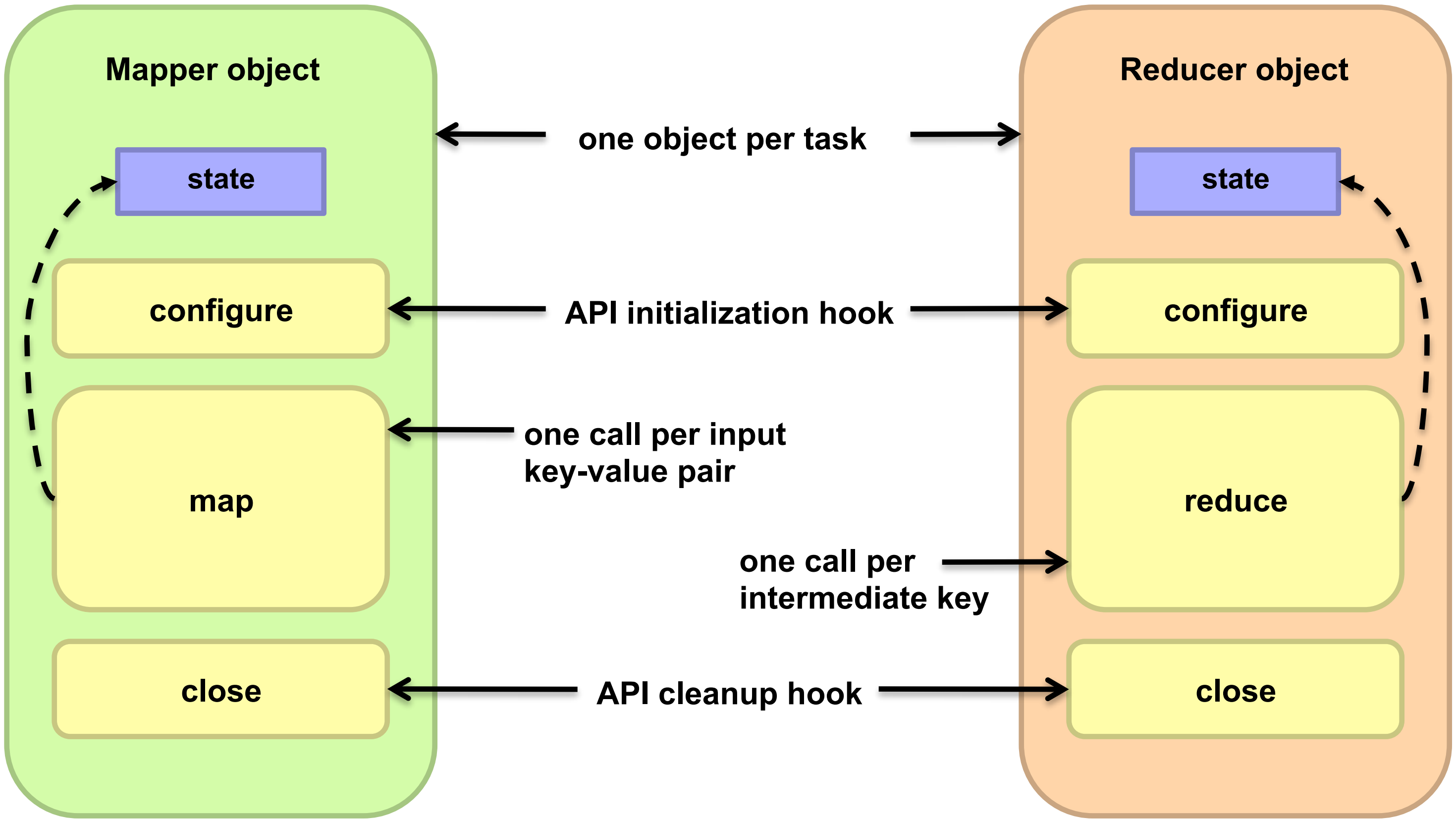
Towards Scalable Hadoop Algorithms

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

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
- Execute initialization and termination code before and after map/reduce tasks

Preserving State



DESIGN PATTERNS

Pattern 1: 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 for this mapreduce job?

What are the communication costs if we add a combiner?

Word Count: Aggregate in Mapper

H(dog) += 1
H(cat) += 1
H(dog) += 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})
```

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
  - Explicitly 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
  - Explicitly control aggregation
  - Speed

Why is this faster than actual combiners?

No need to write all intermediate key-value pairs to disk!

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
  - Explicitly control aggregation
  - Speed

How/when can local aggregation help with reduce stragglers?

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
  - Explicitly control aggregation
  - Speed

How/when can local aggregation help with stragglers?

When value distribution is skewed

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 – if associative array grows too big, it will not fit in memory!
  - Preserving state across multiple key-value pairs may lead to potential for order-dependent bugs
    - Not a problem for word count…

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
- Combiners are an optional optimization
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of all integers associated with the same key
  - Access logs: (user_id, session_id, session_length)

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})

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 [r_1, r_2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r_1, r_2, ...] 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 $[r_1, r_2, \ldots]$)
    $sum \leftarrow 0$
    $cnt \leftarrow 0$
    for all integer $r \in$ integers $[r_1, r_2, \ldots]$ **do**
      $sum \leftarrow sum + r$
      $cnt \leftarrow cnt + 1$
    EMIT(string t, pair (sum, cnt)) ▷ Separate sum and count

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, 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 $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        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))

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}))
```

Pattern 2: Pairs and Stripes

- Keep track of joint events across a large number of observations
  - Common in natural language processing
  - Point-of-sale analysis to identify correlated product purchases
    - E.g., if customer buys milk she also buys bread
    - Assist in inventory management and product placement on store shelves
- 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)

MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection is a specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of observations (the collection itself)
  - Space requirement: n^2
  - 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?

- Real-world English corpora can be hundreds of thousands of words, or even billions of words in web-scale collections

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

Note the use of a complex key

```
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)
```

“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around
  - 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 $a \rightarrow \{ b: count_b, c: count_c, d: count_d \ldots \}$
- Reducers perform element-wise sum of associative arrays

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

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 [H1, H2, H3, ...])
        Hf ← new AssociativeArray
        for all stripe H ∈ stripes [H1, H2, H3, ...] do
            Sum(Hf, H)                         ▷ Element-wise sum
        Emit(term w, stripe Hf)
```

What are the advantages of stripes?

“Stripes” Analysis

- Advantage: Far less sorting and shuffling of key-value pairs
- Disadvantages
  - More difficult to implement
  - Underlying object more heavyweight – higher serialization and de-serialization overhead
  - Fundamental limitation in terms of size of event space

What about combiners?

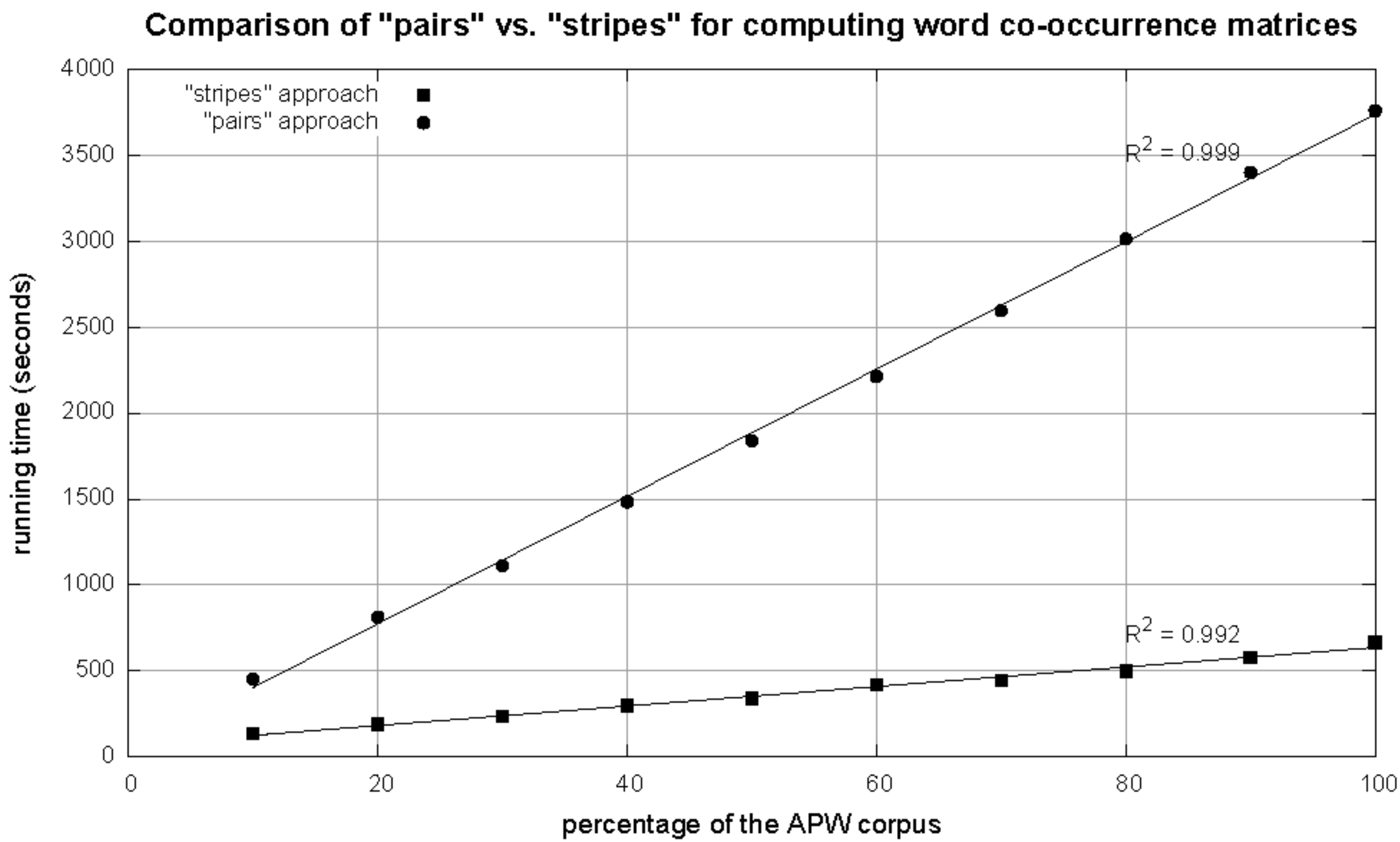
- Both algorithms can benefit from the use of combiners, since the respective operations in their reducers (addition and element-wise sum of associative arrays) are both commutative and associative.

- Are combiners equally effective in both pairs and stripes?

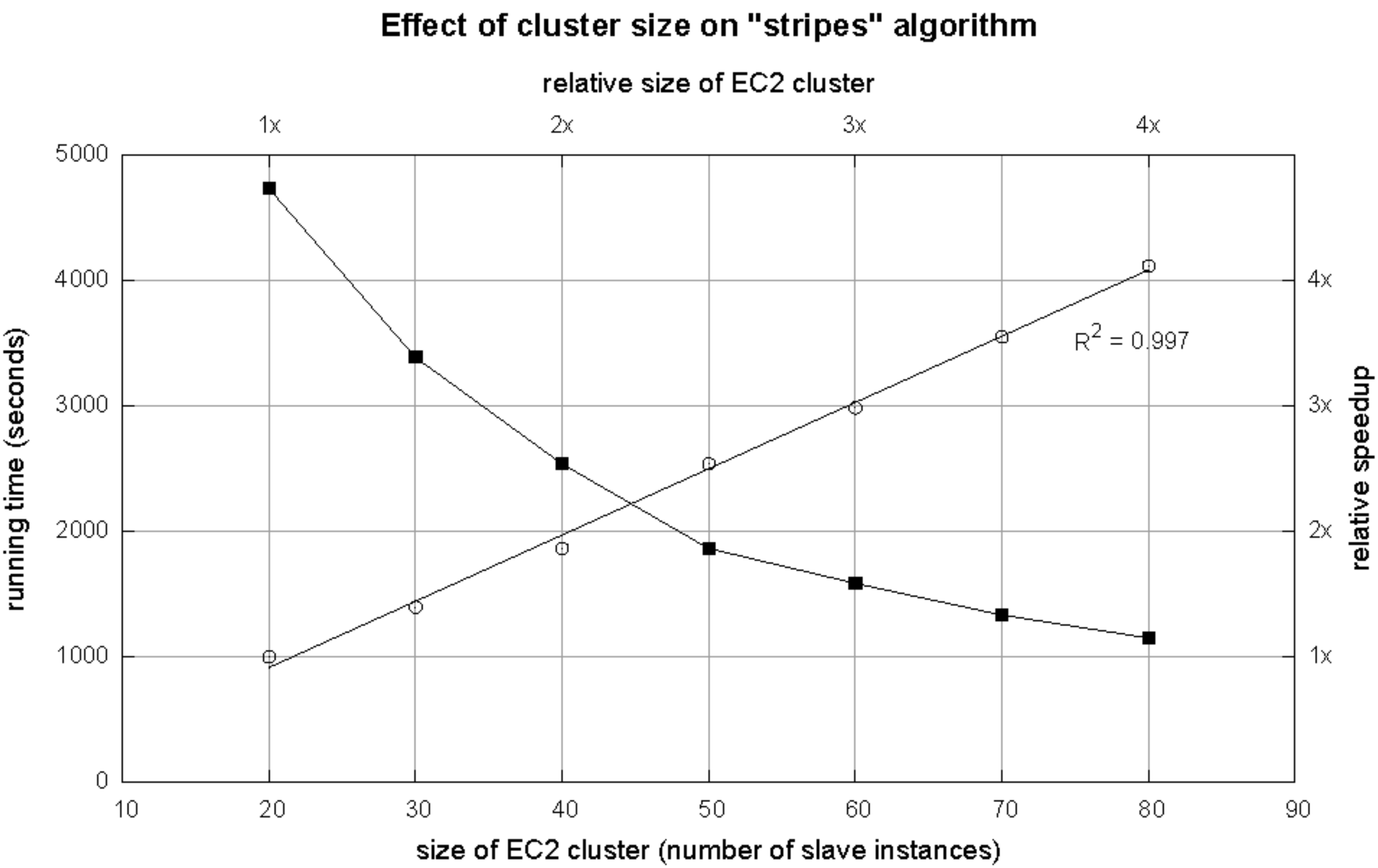
Pairs vs. Stripes

	Pairs	Stripes
Total time	62 min	11 min
Intermediate keys	2.6 billion (31.2GB)	653 million (48.1GB)
Intermediate keys +combiners	1.1 billion	28.8 million
Reducer pairs	142 million (number of non-zero cells)	1.69 miilion (number of rows)

Implementation of both algorithms in Hadoop
Corpus of 2.27 million documents from the Associated Press Worldstream (APW) totaling 5.7 GB.8
(see textbook -- Lin and Dyer -- for details)



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)
$R^2 = 0.997$
4x
3x
2x
1x
relative speedup
10
20
30
40
50
60
70
80
90
size of EC2 cluster (number of slave instances)

Relative Frequencies

- Absolute counts does not take into account the fact that some words appear more frequently than others, e.g., “the”
- How do we estimate relative frequencies from counts? What proportion of time does B appear in the context of A?

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

- How do we 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, *): 3+12+7+1+… = count(A)
  - Another pass to directly compute f(B|A):

    $count(b_1|a) = 3/count(A)$

f(B|A): "Pairs"

- How would you implement this?
  - Given (a,b) → count, how to compute f(b|a)?
- To compute the marginal, keep state in the reducer
  - Build associative array in reducer
- Sort for *a* then *b* to detect if all pairs associated with *a* have been encountered
- Also need to guarantee that all *a* go to the same reducer: Define custom partitioner based on *a*
  - (dog, aardvark) and (dog, zebra) do not necessarily go to the same reducer!
- Same drawback as stripes – can run out of memory

Can we improve on this?

f(B|A) "Pairs": An Improvement

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

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

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

How do we ensure tuples arrive at the reducer in the right order?

Pattern 3: "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: get the marginal counts to arrive at the reducer before the joint counts

key	**values**	
$(\text{dog}, *)$	$[6327, 8514, \ldots]$	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
$(\text{dog}, \text{aardvark})$	$[2,1]$	$f(\text{aardvark}\|\text{dog}) = 3/42908$
$(\text{dog}, \text{aardwolf})$	$[1]$	$f(\text{aardwolf}\|\text{dog}) = 1/42908$
$\ldots$		
$(\text{dog}, \text{zebra})$	$[2,1,1,1]$	$f(\text{zebra}\|\text{dog}) = 5/42908$
$(\text{doge}, *)$	$[682, \ldots]$	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$

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

Pattern 4: Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
- What if want to sort values 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?

- Need to specify how pairs should be sorted using the compareTo() method
- Need a custom partitioner!

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

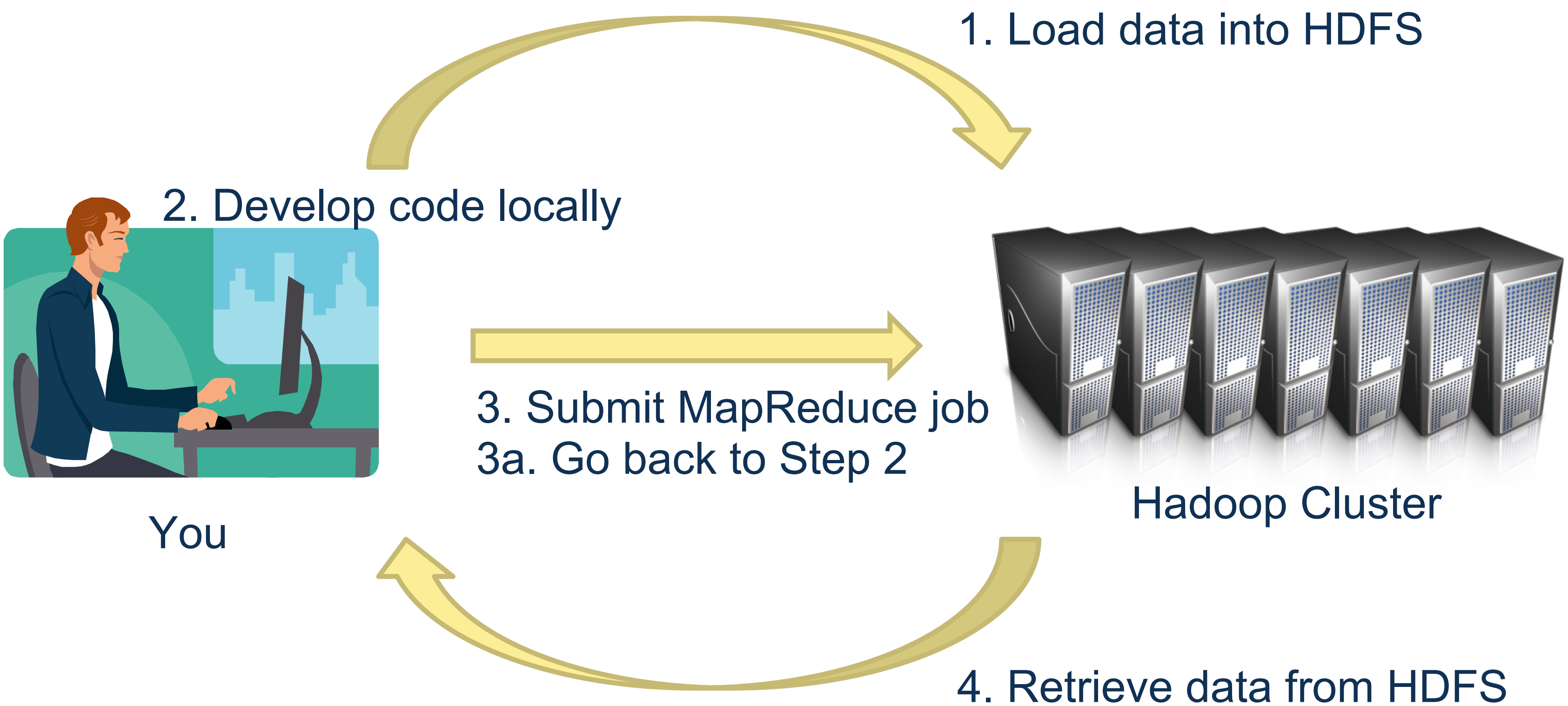
- Number of key-value pairs
  - Object creation overhead
  - Time for sorting and shuffling pairs across the network
- Size of each key-value pair
  - De/serialization overhead
- Local aggregation
  - Opportunities to perform local aggregation varies
  - Combiners make a big difference
  - Combiners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

RUNNING MAPREDUCE JOBS

Hadoop Workflow



On Amazon: With EC2

0. Allocate Hadoop cluster
1. Load data into HDFS

EC2

2. Develop code locally

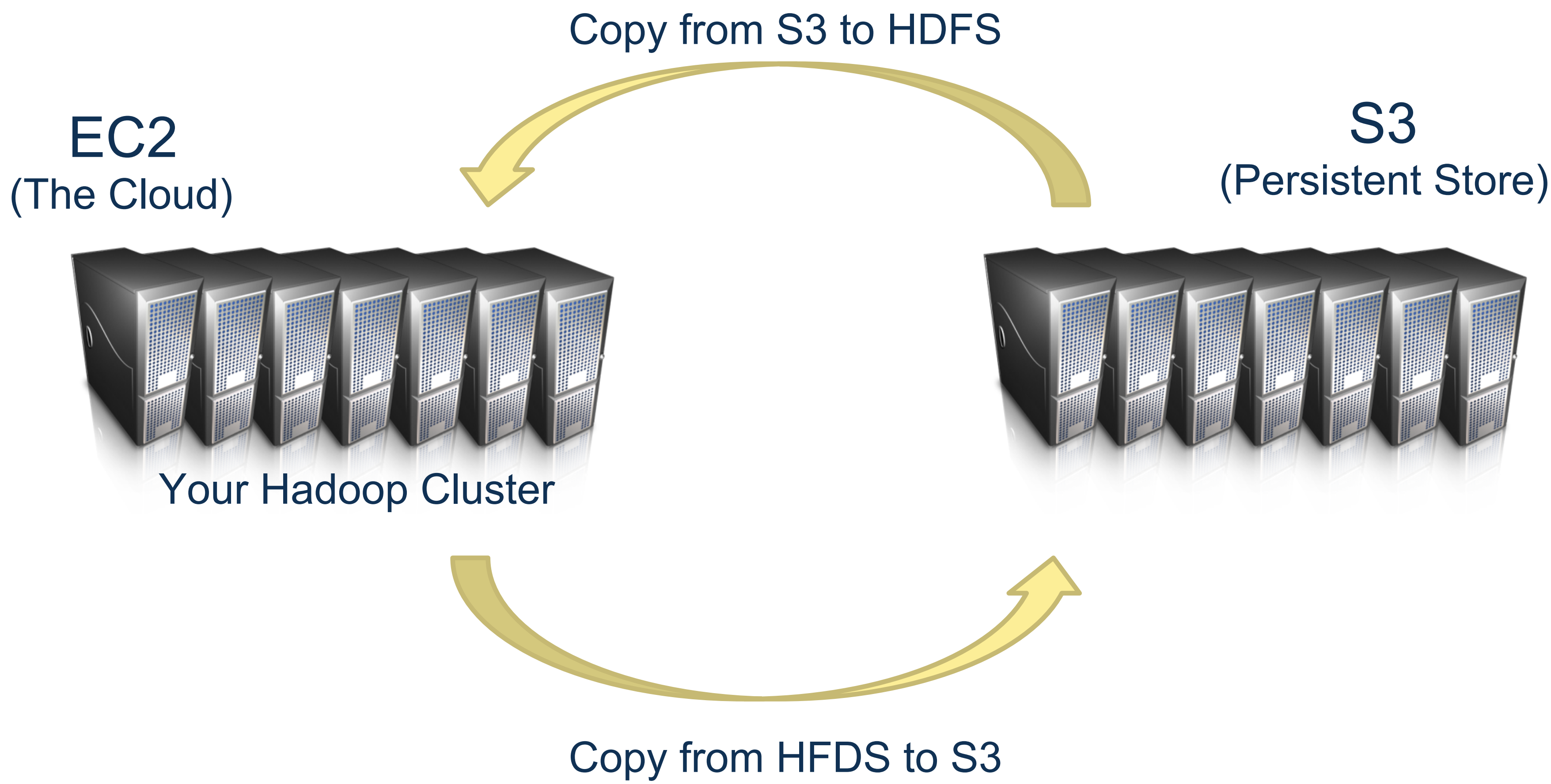
You

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

References

- Data Intensive Text Processing with MapReduce, Lin and Dyer (Chapter 3)
- Mining of Massive Data Sets, Rajaraman et al. (Chapter 2)
- Hadoop tutorial: https://developer.yahoo.com/hadoop/tutorial/module4.html
- Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters

http://labs.google.com/papers/mapreduce.html

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System

http://labs.google.com/papers/gfs.html