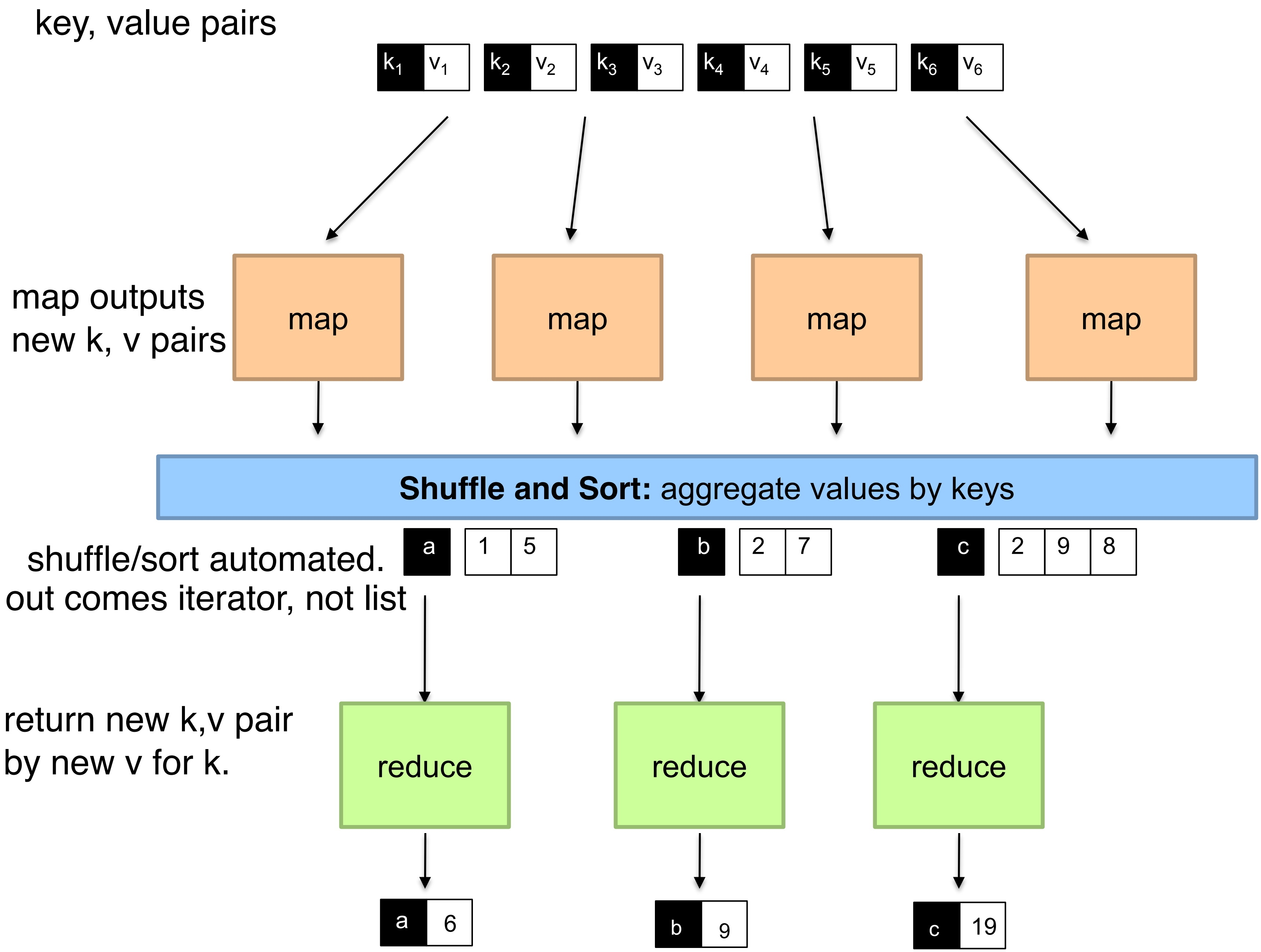
CS109 – Data Science

Joe Blitzstein, Hanspeter Pfister, Verena Kaynig-Fittkau

vkaynig@seas.harvard.edu
staff@cs109.org

Announcements

- Homework Collaboration Policy:
  - See Syllabus on CS109.org
  - The work you turn in must be your own
  - This is a data science course. It takes us 20 minutes to get a similarity ranking of all homework submissions.


key, value pairs
k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map outputs new k, v pairs
map
map
map
map
Shuffle and Sort: aggregate values by keys
shuffle/sort automated. out comes iterator, not list
a 1 5
b 2 7
c 2 9 8
return new k,v pair by new v for k.
reduce
reduce
reduce
a 6
b 9
c 19

Example Input File

```
I am Sam

I am Sam
Sam I am

That Sam I am
That Sam I am
I do not like
that Sam I am

Do you like
green eggs and ham

I do not like them
Sam I am
I do not like
green eggs and ham
```

```python
from mrjob.job import MRJob

class mrWordCount(MRJob):

    def mapper(self,key,line):
        for word in line.split(' '):
            yield word.lower(),1
```

yield returns iterator

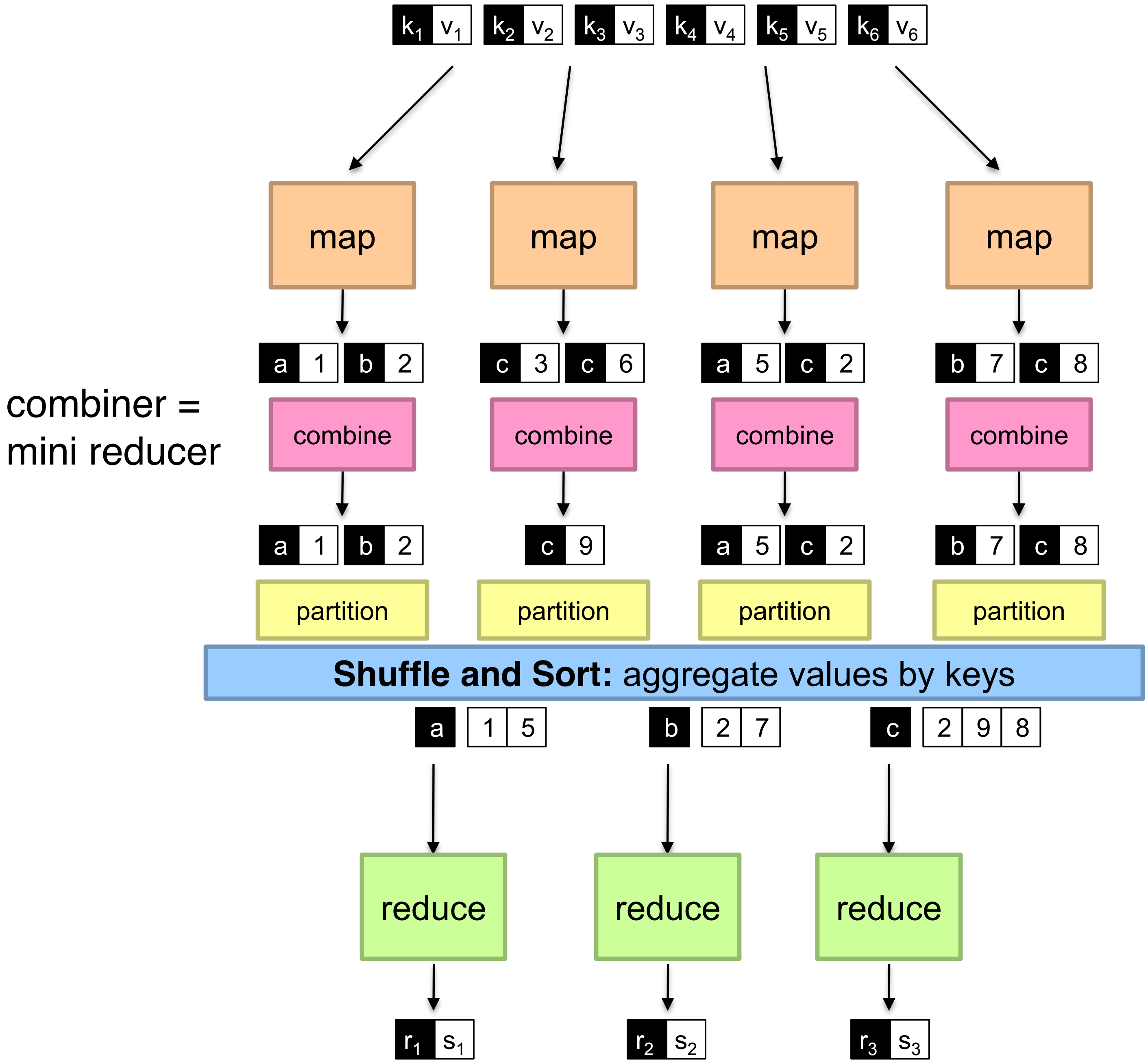
```python
    def reducer(self, word, occurrences):
        yield word, sum(occurrences)

if __name__ == '__main__':
    mrWordCount.run()
```

more efficient: avoid communication overhead!
generate as few key value pairs as possible

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
  – Two possibilities:
    - Combiners
    - In-mapper combining

reduce num key value pair w/ combining.



k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combiner = mini reducer
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Combiner

- “mini-reducers”
- Takes mapper output before shuffle and sort
- Can significantly reduce network traffic
- No access to other mappers
- Not guaranteed to get all values for a key
- Not guaranteed to run at all! ie, if mapper slow, then it may skip combine step because it’s optimizing time in real-time
- Key and value output must match mapper

Why does the key and value output have to match the mapper output? combiner may not run at all: need to keep types consistent.

Word Count with Combiner

```
from mrjob.job import MRJob

class mrWordCount(MRJob):

    def mapper(self,key,line):
        for word in line.split(' '):
            yield word.lower(),1

    def combiner(self,word,occurrences):
        yield word, sum(occurrences)

    def reducer(self, word, occurrences):
        yield word, sum(occurrences)

if __name__ == '__main__':
    mrWordCount.run()
```

stuff into combiner: same as into reducer.

stuff out of combiner same as out of mapper.

keys still word and an integer.

Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Remember: combiners 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 [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)
```

Why can't we use reducer as combiner?

mean of means not same as mean of numbers.

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})

Why doesn't this work? now combiner output is different than mapper

Computing the Mean: Version 3

class MAPPER
  method MAP(string t, integer r)
    EMIT(string t, pair $(r, 1)$)

now mapper outputs a pair

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? What if combiner does not run?

the cnt is 1

In-Mapper Combining

- "Fold the functionality of the combiner into the mapper by preserving state across multiple map calls

class MAPPER
  method INITIALIZE
    $S \leftarrow$ new ASSOCIATIVEARRAY
    $C \leftarrow$ new ASSOCIATIVEARRAY
  method MAP(string t, integer r)
    $S\{t\} \leftarrow S\{t\} + r$
    $C\{t\} \leftarrow C\{t\} + 1$
  method CLOSE
    for all term $t \in S$ **do**
      EMIT(term t, pair $(S\{t\}, C\{t\})$)

In-Mapper Combining

- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

Word Count with In-Mapper-Comb.

```
from collections import defaultdict
from mrjob.job import MRJob

class mrWordCount(MRJob):
    def __init__(self, *args, **kwargs):
        super(mrWordCount, self).__init__(*args, **kwargs)
        self.localWordCount = defaultdict(int)

    def mapper(self,key,line):
        if False:
            yield
        for word in line.split(' '):
            self.localWordCount[word.lower()]+=1

    def mapper_final(self):
        for (word, count) in self.localWordCount.iteritems():
            yield word, count

    def reducer(self, word, occurrences):
        yield word, sum(occurrences)

if __name__ == '__main__':
    mrWordCount.run()
```

Which is better?

- For large dictionaries?
  - Combiner has no memory problems
- For skewed word distributions (“the”)?
  - In-mapper reduces load on reducer

Pairs and Stripes:

- 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
  - Context can be a sentence, sequence of m words, etc.
  - In this case co-occurrence matrix is symmetric

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

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 [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

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

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: count_b, c: count_c, d: count_d … }
- Reducers perform element-wise sum of associative arrays

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

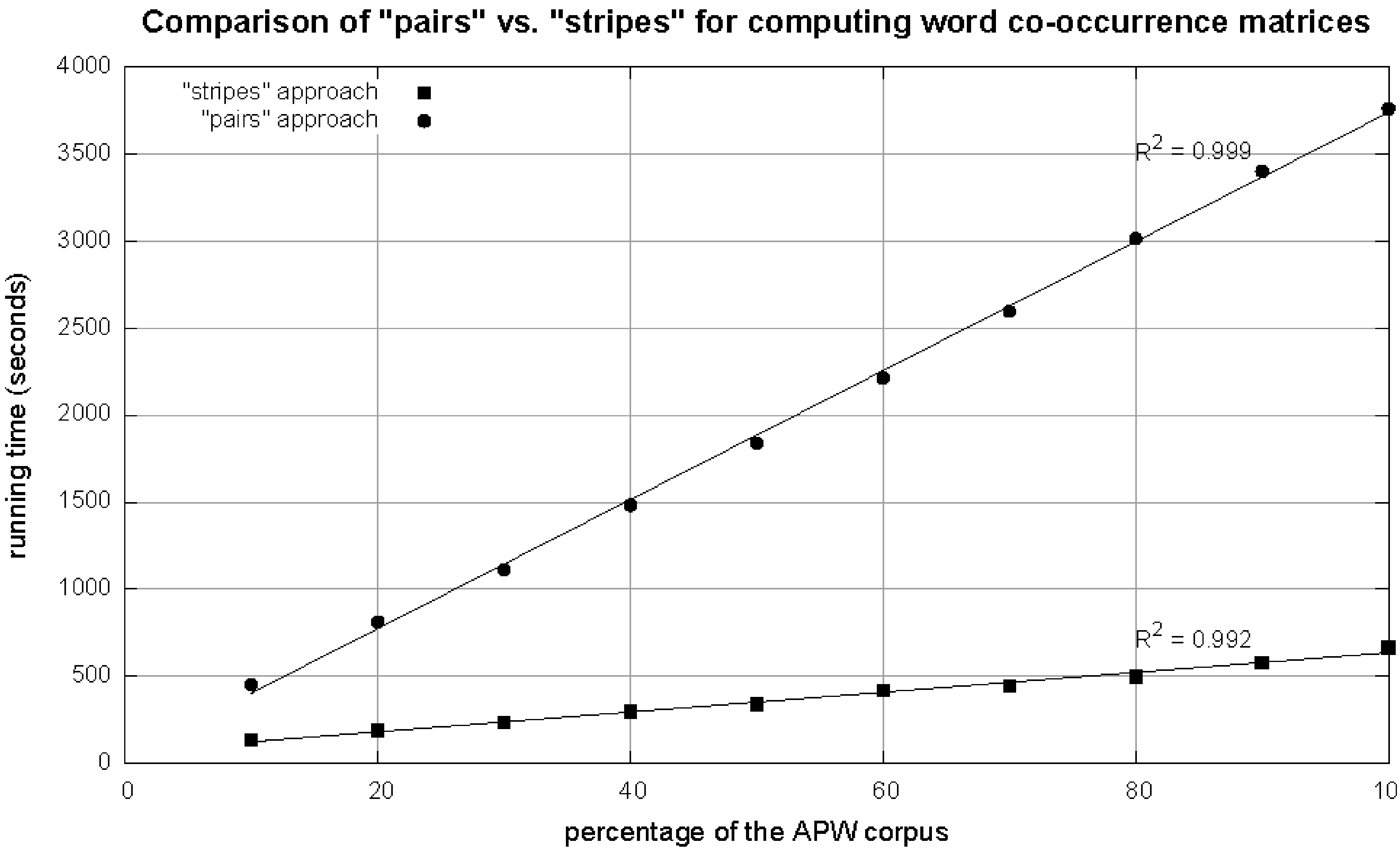
Key: cleverly-constructed data structure brings together partial results

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)
```

“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Keys are less unique than in pairs approach
  - Can make better use of combiners
- Disadvantages
  - More difficult to implement
  - Underlying object more heavyweight
  - Fundamental limitation in terms of size of event space



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)

Map Reduce for Machine Learning

- Random Forest?
- SVM?