COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 2.2: MapReduce II

Another Example: Analysis of Weather Dataset

- Data from NCDC(National Climatic Data Center)
  - A large volume of log data collected by weather sensors: e.g. temperature
- Data format
  - Line-oriented ASCII format
  - Each record has many elements
  - We focus on the temperature element
  - Data files are organized by date and weather station
  - There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year
- Query
  - What's the highest recorded global temperature for each year in the dataset?

Year | Temperature

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

Contents of data files

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

List of data files

Analyzing the Data with Unix Tools

- To provide a performance baseline
- Use *awk* for processing line-oriented data
- Complete run for the century took **42 minutes** on a single EC2 High-CPU Extra Large Instance

```
#!/usr/bin/env bash
for year in all/*
do
  echo -ne `basename $year .gz`"\t"
  gunzip -c $year | \
    awk '{ temp = substr($0, 88, 5) + 0;
           q = substr($0, 93, 1);
           if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
         END { print max }'
done
```

```
% ./max_temperature.sh
1901    317
1902    244
1903    289
1904    256
1905    283
...
```

How Can We Parallelize This Work?

- To speed up the processing, we need to run parts of the program in **parallel**
- Challenges?
  - Divide the work into even distribution is not easy
    - File size for different years varies
  - Combining the results is complicated
    - Get the result from the maximum temperature for each chunk
  - We are still limited by the processing capacity of a single machine
    - Some datasets grow beyond the capacity of a single machine
- To use **multiple machines**, we need to consider a variety of complex problems
  - Coordination: Who runs the overall job?
  - Reliability: How do we deal with failed processes?
- **Hadoop** can take care of these issues

MapReduce Design

- We need to answer these questions:
  - What are the map input key and value types?
  - What does the mapper do?
  - What are the map output key and value types?
  - Can we use a combiner?
  - Is a partitioner required?
  - What does the reducer do?
  - What are the reduce output key and value types?

- And: What are the file formats?
  - For now we are using text files
  - We may use binary files

MapReduce Types

- General form

$$\overbrace{}^{\neq}$$

map: (K1, V1) → list(K2, V2)

=

reduce: (K2, list(V2)) → list(K3, V3)

- Combine function

```
map: (K1, V1) → list(K2, V2)
combine: (K2, list(V2)) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

  - The same form as the reduce function, except its output types
  - Output type is the same as Map
  - The combine and reduce functions may be the same

- Partition function

```
partition: (K2, V2) → integer
```

  - Input intermediate key and value types
  - Returns the partition index

What does the Mapper Do?

- Pull out the year and the temperature
  - Indeed in this example, the map phase is simply data preparation phase
  - Drop bad records(filtering)

Input File

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

Output of Map Function (key, value)

Input of Map Function (key, value)

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

Map →

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

What does the Mapper Do?

- The output from the map function is processed by MapReduce framework
  - Sorts and groups the key-value pairs by key

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

Sort and Group By

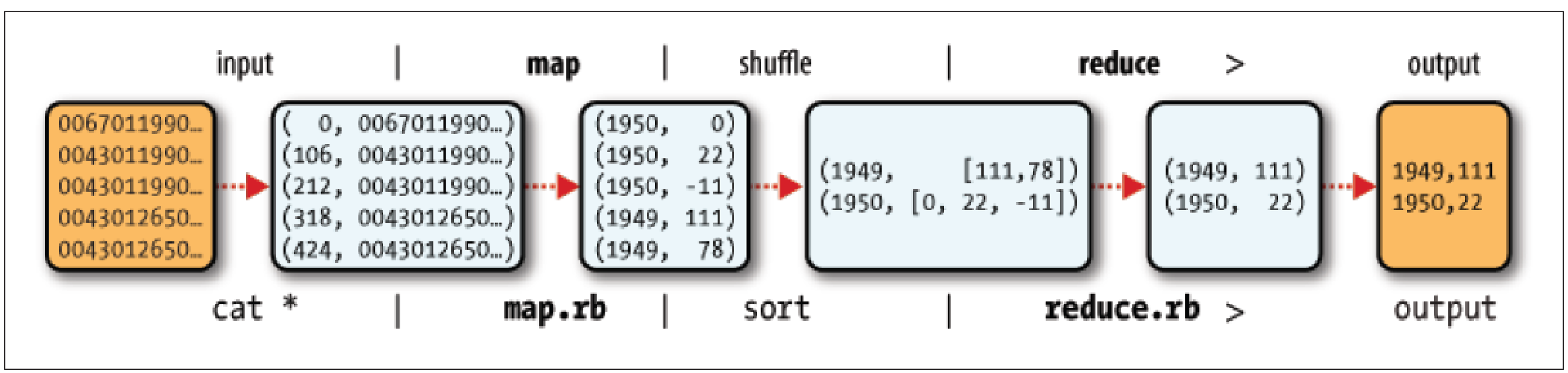
```
(1949, [111, 78])
(1950, [0, 22, -11])
```

- Reduce function iterates through the list and pick up the maximum value

```
(1949, [111, 78])
(1950, [0, 22, -11])
```

Reduce

```
(1949, 111)
(1950, 22)
```



MRJob Implementation of the Example

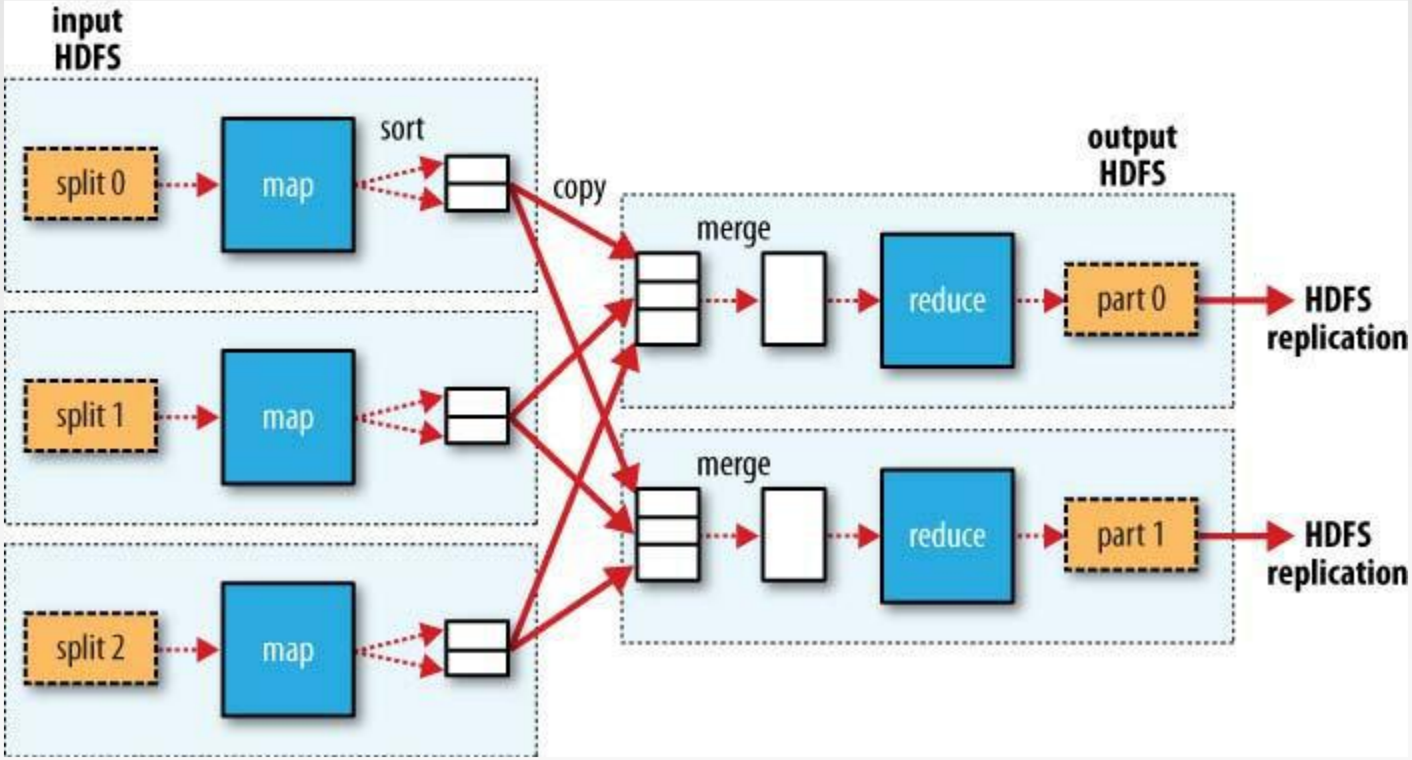
```
#!/usr/bin/env python
from mrjob.job import MRJob

class Weather(MRJob):
        def mapper(self, _, line):
                val = line.strip()
                (year, temp) = (val[15:19], val[87:92])
                if (temp != "+9999"):
                        yield year, int(temp)
        def reducer(self, key, values):
                yield key, max(values)
if __name__ == '__main__':
        Weather.run()
```

- How to implement the combiner?

MapReduce Dataflow

- When there are multiple reducers, the map tasks partition their output:
  - One partition for each reduce task
  - The records for every key are all in a single partition
  - Partitioning can be controlled by a user-defined partitioning function



For Large Datasets (Mapper)

- Data stored in HDFS (organized as blocks)
- Hadoop MapReduce Divides input into fixed-size pieces***, input splits***
  - Hadoop creates one map task for each split
  - Map task runs the user-defined map function for each *record* in the split
  - Size of a split is normally the size of a HDFS block (e.g., 64Mb)
  - The number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.

For Large Datasets (Mapper)

- Data locality optimization
  - Run the map task on a node where the input data resides in HDFS
  - This is the reason why the split size is the same as the block size
    - The largest size of the input that can be guaranteed to be stored on a single node
    - If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks

For Large Datasets (Mapper)

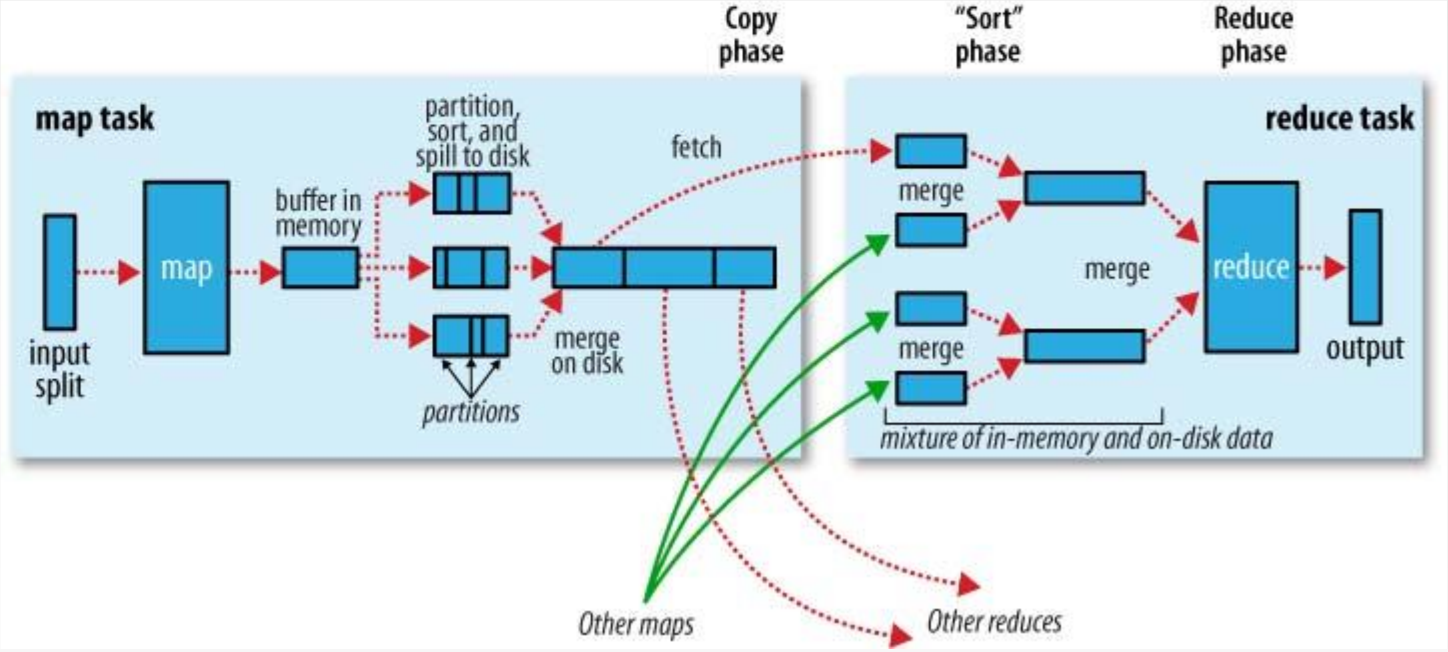
- Map tasks write their output to local disk (not to HDFS)
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - Storing it in HDFS with replication, would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on another node

For Large Datasets (Reducer)

- Reduce tasks don’t have the advantage of data locality
  - Input to a single reduce task is normally the output from all mappers
  - Output of the reduce is stored in HDFS for reliability
  - The number of reduce tasks is not governed by the size of the input, but is specified independently
  - The right number of reduces seems to be 0.95 or 1.75 multiplied by (<no. of nodes> * <no. of maximum containers per node>)
    - With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing

More Detailed MapReduce Dataflow

- When there are multiple reducers, the map tasks partition their output:
  - One partition for each reduce task
  - The records for every key are all in a single partition
  - Partitioning can be controlled by a user-defined partitioning function



MapReduce Algorithm Design Patterns

Design Pattern 1: Combiner/In-mapper Combining

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?
  - Data synchronization requires communication
  - Communication kills performance
- Thus… avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help

WordCount 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)
```

What's the impact of combiners?

Word Count: Version 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: Version 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!

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

- Both input and output data types must be consistent with the output of mapper (or input of reducer)
- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Hadoop do not guarantee how many times it will call combiner function for a particular map output record
  - It is just optimization
  - The number of calling (even zero) does not affect the output of Reducers

max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25

- Applicable on problems that are commutative and associative
  - Commutative: max(a, b) = max(b, a)
  - Associative: max (max(a, b), c) = max(a, max(b, c))

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(1, 2, 3, 4, 5) != Mean(Mean(1, 2), Mean(3, 4, 5))

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)
```

Why doesn't this work?

Combiners must have the same input and output type, consistent with the input of reducers (output of mappers)

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?

Check the correctness by removing the combiner

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}))
```

How to Implement In-mapper Combiner in MapReduce?

Lifecycle of Mapper/Reducer (Java)

- Lifecycle: setup -> map -> cleanup
  - setup(): called once at the beginning of the task
  - map(): do the map
  - cleanup(): called once at the end of the task.
  - We do not invoke these functions
- In-mapper Combining:
  - Use setup() to initialize the state preserving data structure
  - Use clearnup() to emit the final key-value pairs

Implementation in MRJob

- One step consists of a mapper, a combiner and a reducer.
- In addition, there are more methods you can override to write a one-step job
  - mapper_init()
  - combiner_init()
  - reducer_init()
  - mapper_final()
  - combiner_final()
  - reducer_final()
- For im-mapper combing
  - Initialize the “AssociativeArray” in mapper_init(),
  - Update the “AssociativeArray” in mapper()
  - Yield the results in mapper_final()

Word Count: Version 2

mapper_init()

```
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})
```

mapper_final()

MRJob Code

```
import re
from mrjob.job import MRJob

class WordCount(MRJob):

    def mapper_init(self):
        self.tmp = {}

    def mapper(self, _, line):
        words = re.split("[ *$&#/\t\n\f\"\'\\,.:;?!\[\](){}<>~\-_]", line.lower())
        for word in words:
            if len(word):
                self.tmp[word] = self.tmp.get(word, 0) + 1

    def mapper_final(self):
        for k, v in self.tmp.items():
            yield (k, v)


    def reducer(self, key, values):
        yield key, sum(values)


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

Design Pattern 2: Pairs vs Stripes

Term Co-occurrence Computation

- 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)
  - 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!

```
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 (upper bound?)
  - Not many opportunities for combiners to work

Another Try: "Stripes"

- Idea: group together pairs into an associative array

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

- 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 … }
- Reducers perform element-wise sum of associative arrays

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

Key: cleverly-constructed data structure brings together partial results

Stripes: Pseudo-Code

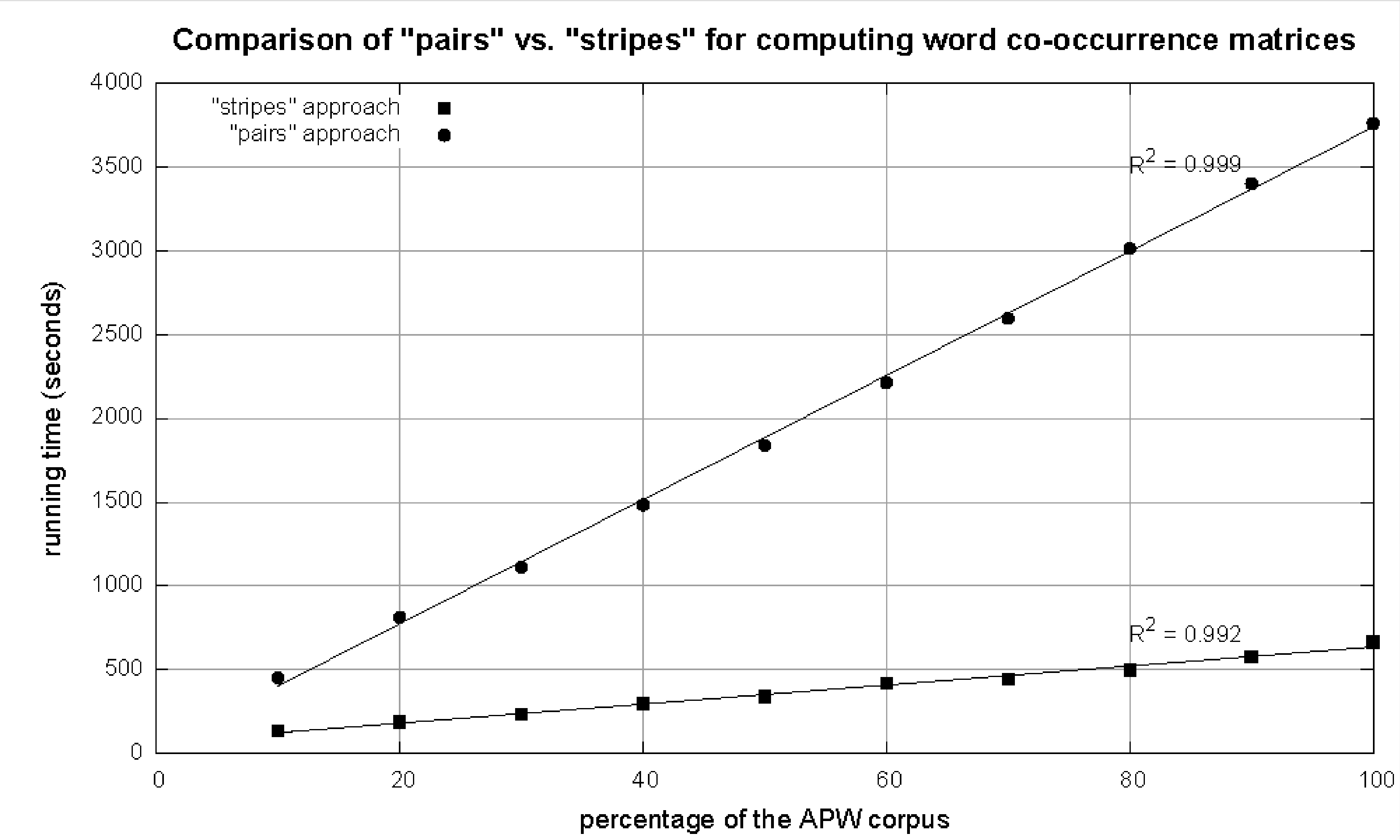
```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            H ← new AssociativeArray
            for all term u ∈ Neighbors(w) do
                H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
            Emit(Term w, Stripe H)

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)
```

“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

Compare "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)

Pairs vs. Stripes

- The pairs approach
  - Keep track of each team co-occurrence separately
  - Generates a large number of key-value pairs (also intermediate)
  - The benefit from combiners is limited, as it is less likely for a mapper to process multiple occurrences of a word
- The stripe approach
  - Keep track of all terms that co-occur with the same term
  - Generates fewer and shorted intermediate keys
  - The framework has less sorting to do
  - Greatly benefits from combiners, as the key space is the vocabulary
  - More efficient, but may suffer from memory problem
- These two design patterns are broadly useful and frequently observed in a variety of applications
  - Text processing, data mining, and bioinformatics

How to Implement “Pairs” and “Stripes” in MapReduce?

Pairs Implementation (Python)

- In mapper:
  - key: a pair of two terms as a string
  - value: a value 1
  - Iterate over the words in the line to generate all pairs
  - print (key+"\t1")
- In Reducer:
  - Receive the pairs one by one
  - Aggregate the 1s for the same pair to obtain the final co-occurrence (similar to word count)
  - Print the pair and the final count to stdout
- How about a combiner?

Pairs Implementation (MRJob)

- Using MRJob is even simpler than Hadoop streaming
- In mapper:
  - key: a pair of two terms as a string
  - value: a value 1
  - Iterate over the words in the line to generate all pairs
  - yield(key, 1)
- In Reducer:
  - Receive the list of pairs
  - Aggregate the 1s to obtain the final co-occurrence (similar to word count)
  - Yield the pair of the term and the final co-occurrence
- A combiner, but not too much helpful…

Stripes Implementation (Python)

- In Hadoop streaming, mapper/reducer reads input from stdin and outputs results to stdout, and thus using Python is quite different
- A stripe key-value pair $a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$
- In mapper:
  - key: the term itself as a string
  - value: a dictionary object
  - Iterate over the words in the line to generate the stripes
  - print (key+"\t"+str(value))
- In Reducer:
  - Receive the stripes one by one, and convert each to a dictionary object
  - Aggregate the stripes for the same key to obtain the final stripe
  - Print the term and the final stripe to stdout
- What does the combiner look like?

Stripes Implementation (MRJob)

- Using MRJob is even simpler than Hadoop streaming
- A stripe key-value pair $a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$
- In mapper:
  - key: the term itself as a string
  - value: a dictionary object
  - Iterate over the words in the line to generate the stripes
  - yield (key, str(value))
- In Reducer:
  - Receive the list of stripes, and convert each to a dictionary object
  - Aggregate the stripes for the same key to obtain the final stripe
  - yield the term and the final stripe as the result
- A combiner

References

- MapReduce Chapter of <<Hadoop The Definitive Guide>>
- Chapters 3.1, 3.2. Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.

End of Chapter 2.2