

COMP9313: Big Data Management



Lecturer: Xin Cao

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

Chapter 3: MapReduce II

Overview of Previous Lecture

- Motivation of MapReduce
- Data Structures in MapReduce: (key, value) pairs
- Map and Reduce Functions
- Hadoop MapReduce Programming
 - Mapper
 - Reducer
 - Combiner
 - Partitioner
 - Driver

Combiner Function

- To minimize the data transferred between map and reduce tasks
- Combiner function is run on the map output
- Both input and output data types must be consistent with the output of mapper (or input of reducer)
- But 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: $\max(\max(a, b), c) = \max(a, \max(b, c))$

MapReduce Algorithm Design Patterns

Design Pattern 1: 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

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

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

What's the impact of combiners?

Word Count: Version 1

```
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: preserve state across
input key-value pairs!

▷ Tally counts *across* documents

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 all integers associated with the same key

Computing the Mean: Version 1

```
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 [r1, r2, ...])
7:         sum ← 0
8:         cnt ← 0
9:         for all integer r ∈ integers [r1, r2, ...] do
10:             sum ← sum + r
11:             cnt ← cnt + 1
12:             ravg ← sum/cnt
13:             EMIT(string t, integer ravg)
```

Why can't we use reducer as combiner?

$$\text{Mean}(1, 2, 3, 4, 5) \neq \text{Mean}(\text{Mean}(1, 2), \text{Mean}(3, 4, 5))$$

Computing the Mean: Version 2

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, integer r)

1: class COMBINER
2:     method COMBINE(string t, integers [r1, r2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r1, r2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         EMIT(string t, pair (sum, cnt))           ▷ Separate sum and count

1: class REDUCER
2:     method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         ravg ← sum/cnt
9:         EMIT(string t, integer ravg)
```

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

```
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?

Check the correctness by removing the combiner

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 ( $S\{t\}$ ,  $C\{t\}$ ))
```

How to Implement In-mapper Combiner in MapReduce?

Lifecycle of Mapper/Reducer

- 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 cleanup() to emit the final key-value pairs

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  $H\{t\}$ )
```

setup()

▷ Tally counts *across documents*

cleanup()

Design Pattern 2: Pairs vs Stripes

Term Co-occurrence Computation

- Term co-occurrence matrix for a text collection
 - $M = N \times N$ 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 $(a, b) \rightarrow \text{count}$
- Reducers sum up counts associated with these pairs
- Use combiners!

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

1: class REDUCER
2:   method REDUCE(pair  $p$ , counts  $[c_1, c_2, \dots]$ )
3:      $s \leftarrow 0$ 
4:     for all count  $c \in \text{counts } [c_1, c_2, \dots]$  do
5:        $s \leftarrow s + c$                       ▷ 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

$(a, b) \rightarrow 1$

$(a, c) \rightarrow 2$

$(a, d) \rightarrow 5$

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

$(a, e) \rightarrow 3$

$(a, f) \rightarrow 2$

- 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: 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  $w \in$  doc  $d$  do
4:        $H \leftarrow$  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:      $H_f \leftarrow$  new ASSOCIATIVEARRAY
12:     for all stripe  $H \in$  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

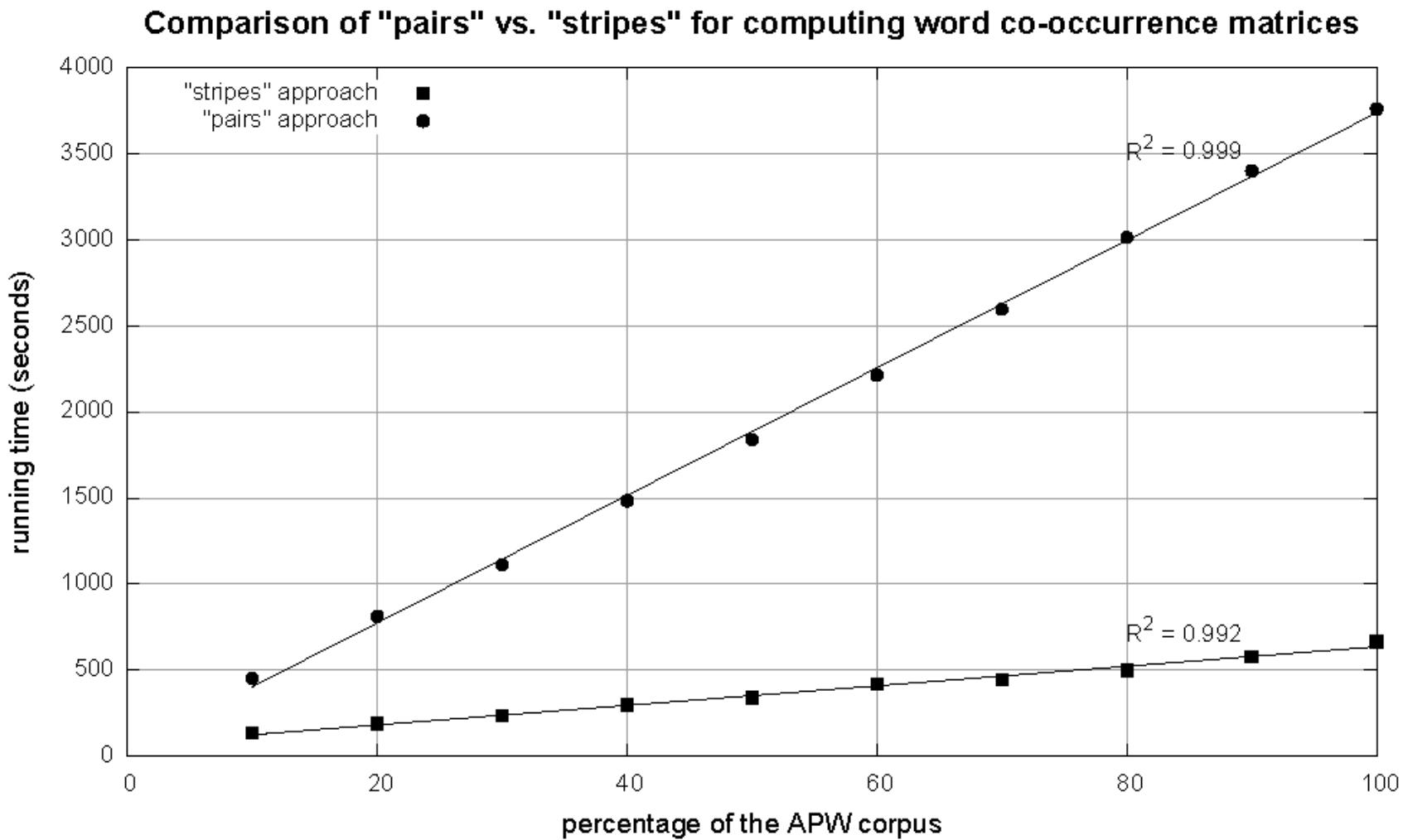
■ Advantages

- 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

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 shorter 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?

Serialization

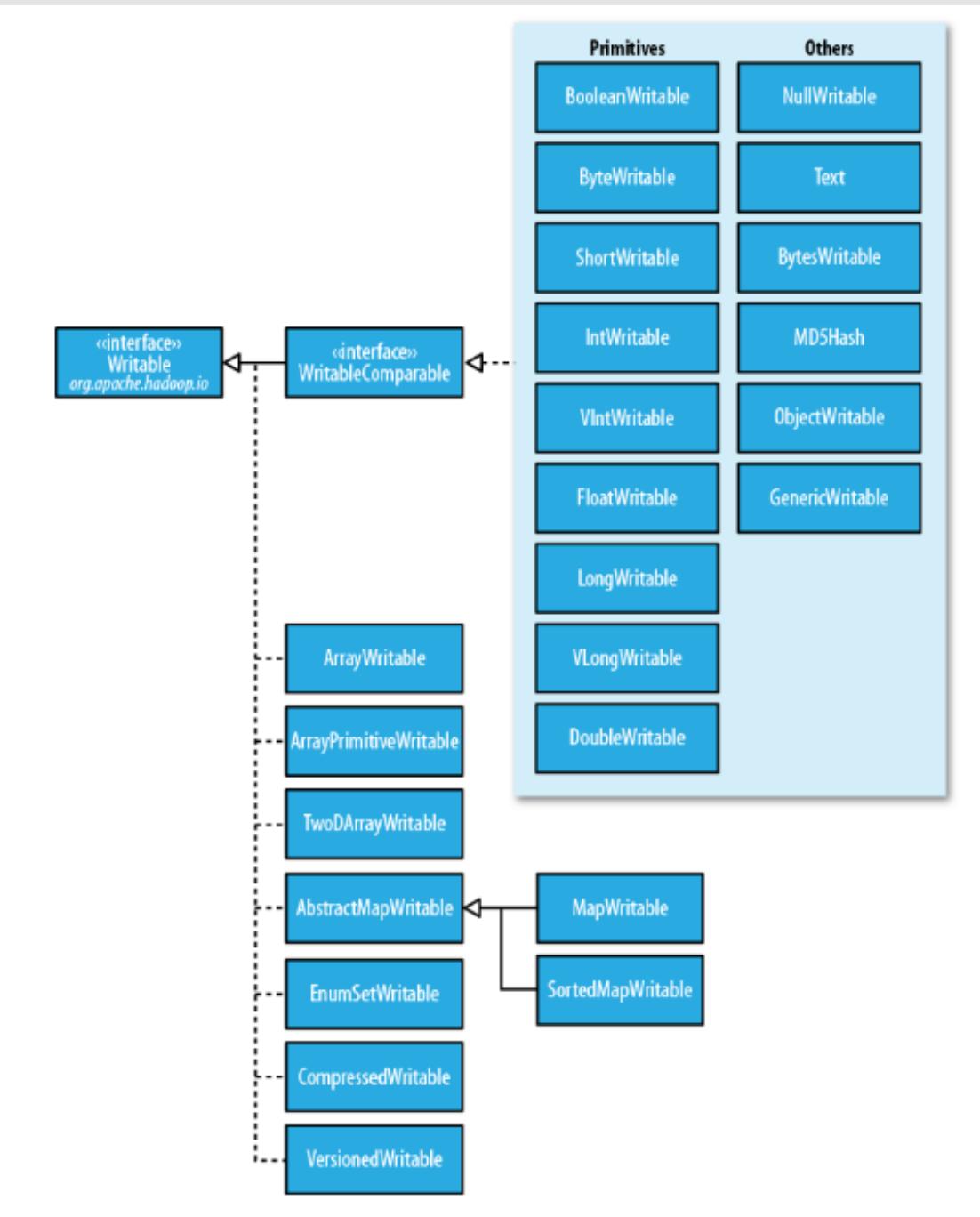
- Process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage
- Deserialization is the reverse process of serialization
- Requirements
 - Compact
 - ▶ To make efficient use of storage space
 - Fast
 - ▶ The overhead in reading and writing of data is minimal
 - Extensible
 - ▶ We can transparently read data written in an older format
 - Interoperable
 - ▶ We can read or write persistent data using different language

Writable Interface

- Hadoop defines its own “box” classes for strings (Text), integers (IntWritable), etc.
- Writable is a serializable object which implements a simple, efficient, serialization protocol

```
public interface Writable {  
    void write(DataOutput out) throws IOException;  
    void readFields(DataInput in) throws IOException;  
}
```

- All values must implement interface Writable
- All keys must implement interface WritableComparable
- context.write(WritableComparable, Writable)
 - You cannot use java primitives here!!



Writable Wrappers for Java Primitives

- There are **Writable** wrappers for all the Java primitive types except short and char (both of which can be stored in an **IntWritable**)
- **get()** for retrieving and **set()** for storing the wrapped value
- Variable-length formats
 - If a value is between -122 and 127, use only a single byte
 - Otherwise, use first byte to indicate whether the value is positive or negative and how many bytes follow

Java Primitive	Writable Implementation	Serialized Size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
int	IntWritable	4
	VIntWritable	1~5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1~9
double	DoubleWritable	8

Writable Examples

■ Text

- Writable for UTF-8 sequences
- Can be thought of as the Writable equivalent of `java.lang.String`
- Maximum size is 2GB
- Use standard UTF-8
- Text is mutable (like all Writable implementations, except `NullWritable`)
 - ▶ Different from `java.lang.String`
 - ▶ You can reuse a Text instance by calling one of the `set()` method

■ NullWritable

- Zero-length serialization
- Used as a placeholder
- A key or a value can be declared as a **NullWritable** when you don't need to use that position

Stripes Implementation

- A stripe key-value pair $a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$:
 - Key: the term a
 - Value: the stripe $\{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$
 - ▶ In Java, easy, use map (hashmap)
 - ▶ How to represent this stripe in MapReduce?
- MapWritable: the wrapper of Java map in MapReduce
 - put(Writable key, Writable value)
 - get(Object key)
 - containsKey(Object key)
 - containsValue(Object value)
 - entrySet(), returns Set<Map.Entry<Writable, Writable>>, used for iteration
- More details please refer to

<https://hadoop.apache.org/docs/r2.7.2/api/org/apache/hadoop/io/MapWritable.html>

Pairs Implementation

- Key-value pair (a, b) → count
 - Value: count
 - Key: (a, b)
 - ▶ In Java, easy, implement a pair class
 - ▶ *How to store the key in MapReduce?*
- You must customize your own key, which must implement interface WritableComparable!
- First start from a easier task: when the value is a pair, which must implement interface Writable

Multiple Output Values

- If we are to output multiple values for each key
 - E.g., a pair of String objects, or a pair of int
- How do we do that?
- WordCount output a single number as the value
- Remember, our object containing the values needs to implement the Writable interface
- We could use Text
 - Value is a string of comma separated values
 - Have to convert the values to strings, build the full string
 - Have to parse the string on input (not hard) to get the values

Implement a Custom Writable

- Suppose we wanted to implement a custom class containing a pair of integers. Call it IntPair.
- How would we implement this class?
 - Needs to implement the Writable interface
 - Instance variables to hold the values
 - Construct functions
 - A method to set the values (two integers)
 - A method to get the values (two integers)
 - write() method: serialize the member variables (two integers) objects in turn to the output stream
 - readFields() method: deserialize the member variables (two integers) in turn from the input stream
 - As in Java: hashCode(), equals(), toString()

Implement a Custom Writable

■ Implement the Writable interface

```
public class IntPair implements Writable {
```

■ Instance variables to hold the values

```
    private int first, second;
```

■ Construct functions

```
    public IntPair() {  
    }
```

```
    public IntPair(int first, int second) {  
        set(first, second);  
    }
```

■ set() method

```
    public void set(int left, int right) {  
        first = left;  
        second = right;  
    }
```

Implement a Custom Writable

■ get() method

```
public int getFirst() {  
    return first;  
}  
public int getSecond() {  
    return second;  
}
```

■ write() method

```
public void write(DataOutput out) throws IOException {  
    out.writeInt(first);  
    out.writeInt(second);  
}
```

- Write the two integers to the output stream in turn

■ readFields() method

```
public void readFields(DataInput in) throws IOException {  
    first = in.readInt();  
    second = in.readInt();  
}
```

- Read the two integers from the input stream in turn

Complex Key

- If the key is not a single value
 - E.g., a pair of String objects, or a pair of int
- How do we do that?
- The co-occurrence matrix problem, a pair of terms as the key
- Our object containing the values needs to implement the WritableComparable interface
 - Why Writable is not competent?
- We could use Text again
 - Value is a string of comma separated values
 - Have to convert the values to strings, build the full string
 - Have to parse the string on input (not hard) to get the values
 - **Objects are compared according to the full string!!**

Implement a Custom WritableComparable

- Suppose we wanted to implement a custom class containing a pair of String objects. Call it StringPair.
- How would we implement this class?
 - Needs to implement the WritableComparable interface
 - Instance variables to hold the values
 - Construct functions
 - A method to set the values (two String objects)
 - A method to get the values (two String objects)
 - write() method: serialize the member variables (i.e., two String) objects in turn to the output stream
 - readFields() method: deserialize the member variables (i.e., two String) in turn from the input stream
 - As in Java: hashCode(), equals(), toString()
 - compareTo() method: specify how to compare two objects of the self-defind class

Implement a Custom WritableComparable

- implement the Writable interface

```
public class StringPair implements WritableComparable<StringPair> {
```

- Instance variables to hold the values

```
    private String first, second;
```

- Construct functions

```
    public StringPair() {  
    }
```

```
    public StringPair(String first, String second) {  
        set(first, second);  
    }
```

- set() method

```
    public void set(String left, String right) {  
        first = left;  
        second = right;  
    }
```

Implement a Custom WritableComparable

■ get() method

```
public String getFirst() {  
    return first;  
}  
public String getSecond() {  
    return second;  
}
```

■ write() method

```
public void write(DataOutput out) throws IOException {  
    String[] strings = new String[] { first, second };  
    WritableUtils.writeStringArray(out, strings);  
}
```

- Utilize WritableUtils.

■ readFields() method

```
public void readFields(DataInput in) throws IOException {  
    String[] strings = WritableUtils.readStringArray(in);  
    first = strings[0];  
    second = strings[1];  
}
```

Implement a Custom WritableComparable

■ compareTo() method:

```
public int compareTo(StringPair o) {  
    int cmp = compare(first, o.getFirst());  
    if(cmp != 0){  
        return cmp;  
    }  
    return compare(second, o.getSecond());  
}  
  
private int compare(String s1, String s2){  
    if (s1 == null && s2 != null) {  
        return -1;  
    } else if (s1 != null && s2 == null) {  
        return 1;  
    } else if (s1 == null && s2 == null) {  
        return 0;  
    } else {  
        return s1.compareTo(s2);  
    }  
}
```

Implement a Custom WritableComparable

- You can also make the member variables as Writable objects
- Instance variables to hold the values

```
private Text first, second;
```

- Construct functions

```
public StringPair() {  
    set(new Text(), new Text());  
}  
  
public StringPair(Text first, Text second) {  
    set(first, second);  
}
```

- set() method

```
public void set(Text left, Text right) {  
    first = left;  
    second = right;  
}
```

Implement a Custom WritableComparable

■ get() method

```
public Text getFirst() {  
    return first;  
}  
public Text getSecond() {  
    return second;  
}
```

■ write() method

```
public void write(DataOutput out) throws IOException {  
    first.write(out);  
    second.write(out);  
}
```

- Delegated to Text

■ readFields() method

```
public void readFields(DataInput in) throws IOException {  
    first.readFields(in);  
    second.readFields(in);  
}
```

- Delegated to Text

Implement a Custom WritableComparable

- In some cases such as secondary sort, we also need to override the hashCode() method.
 - Because we need to make sure that all key-value pairs associated with the first part of the key are sent to the same reducer!

```
public int hashCode()
    return first.hashCode();
}
```

- By doing this, partitioner will only use the hashCode of the first part.
- You can also write a partitioner to do this job

Design Pattern 3: Order Inversion

Computing Relative Frequencies

- “Relative” Co-occurrence matrix construction
 - Similar problem as before, same matrix
 - Instead of absolute counts, we take into consideration the fact that some words appear more frequently than others
 - ▶ Word w_i may co-occur frequently with word w_j simply because one of the two is very common
 - We need to convert absolute counts to relative frequencies $f(w_j|w_i)$
 - ▶ What proportion of the time does w_j appear in the context of w_i ?

- Formally, we compute:

$$f(w_j|w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

- $N(\cdot, \cdot)$ is the number of times a co-occurring word pair is observed
- The denominator is called the marginal

$f(w_j|w_i)$: “Stripes”

- In the reducer, the counts of all words that co-occur with the conditioning variable (w_i) are available in the associative array
- Hence, the sum of all those counts gives the marginal
- Then we divide the joint counts by the marginal and we’re done

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

$$f(b_1|a) = 3 / (3 + 12 + 7 + 1 + \dots)$$

■ Problems?

- Memory

$f(w_j|w_i)$: “Pairs”

- The reducer receives the pair (w_i, w_j) and the count
- From this information alone it is not possible to compute $f(w_j|w_i)$
 - Computing relative frequencies requires marginal counts
 - But the marginal cannot be computed until you see all counts

$((a, b_1), \{1, 1, 1, \dots\})$

No way to compute $f(b_1|a)$ because the marginal is unknown

$f(w_j|w_i)$: “Pairs”

- Solution 1: Fortunately, as for the mapper, also the reducer can preserve state across multiple keys
 - We can buffer in memory all the words that co-occur with w_i and their counts
 - This is basically building the associative array in the stripes method

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

is now buffered in the reducer side

- Problems?

$f(w_j|w_i)$: “Pairs”

If reducers receive pairs not sorted

$((a, b_1), \{1, 1, 1, \dots\})$

$((c, d_1), \{1, 1, 1, \dots\})$

$((a, b_2), \{1, 1, 1, \dots\})$

.... . . .

When we can compute the marginal?

■ We must define the sort order of the pair !!

- In this way, the keys are first sorted by the left word, and then by the right word (in the pair)
- Hence, we can detect if all pairs associated with the word we are conditioning on (w_i) have been seen
- At this point, we can use the in-memory buffer, compute the relative frequencies and emit

$f(w_j|w_i)$: “Pairs”

$((a, b_1), \{1, 1, 1, \dots\})$ and $((a, b_2), \{1, 1, 1, \dots\})$ may be assigned to different reducers!

Default partitioner computed based on the whole key.

- We must define an appropriate partitioner
 - The default partitioner is based on the hash value of the intermediate key, modulo the number of reducers
 - For a complex key, the raw byte representation is used to compute the hash value
 - ▶ Hence, there is no guarantee that the pair (dog, aardvark) and (dog, zebra) are sent to the same reducer
 - What we want is that all pairs with the same left word are sent to the same reducer
- Still suffer from the memory problem!

$f(w_j | w_i)$: “Pairs”

■ Better solutions?

$(a, *) \rightarrow 32$

Reducer holds this value in memory, rather than the stripe

$(a, b_1) \rightarrow 3$

$(a, b_2) \rightarrow 12$

$(a, b_3) \rightarrow 7$

$(a, b_4) \rightarrow 1$

...



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

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

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

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

...

■ The key is to properly sequence data presented to reducers

- If it were possible to compute the marginal in the reducer before processing the join counts, the reducer could simply divide the joint counts received from mappers by the marginal
- The notion of “before” and “after” can be captured in the **ordering of key-value pairs**
- The programmer can define the sort order of keys so that data needed earlier is presented to the reducer before data that is needed later

$f(w_j|w_i)$: “Pairs” – Order Inversion

- A better solution based on order inversion
- The mapper:
 - additionally emits a “special” key of the form ($w_i, *$)
 - The value associated to the special key is one, that represents the contribution of the word pair to the marginal
 - Using combiners, these partial marginal counts will be aggregated before being sent to the reducers
- The reducer:
 - We must make sure that the special key-value pairs are processed before any other key-value pairs where the left word is w_i (**define sort order**)
 - We also need to guarantee that all pairs associated with the same word are sent to the same reducer (**use partitioner**)

$f(w_j|w_i)$: “Pairs” – Order Inversion

■ Example:

- The reducer finally receives:

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

- The pairs come in order, and thus we can compute the relative frequency immediately.

$f(w_j|w_i)$: “Pairs” – Order Inversion

■ Memory requirements:

- Minimal, because only the marginal (an integer) needs to be stored
- No buffering of individual co-occurring word
- No scalability bottleneck

■ Key ingredients for order inversion

- Emit a special key-value pair to capture the marginal
- Control the sort order of the intermediate key, so that the special key-value pair is processed first
- Define a custom partitioner for routing intermediate 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

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

How to Implement Order Inversion in MapReduce?

Implement a Custom Partitioner

- You need to implement a “pair” class first as the key data type
- A customized partitioner extends the *Partitioner* class

```
public static class YourPartitioner extends Partitioner<Key, Value>{
```

- The *key* and *value* are the intermediate key and value produced by the map function
- In the relevant frequencies computing problem

```
public static class FirstPartitioner extends Partitioner<StringPair, IntWritable>{
```

- It overrides the *getPartition* function, which has three parameters

```
public int getPartition(WritableComparable key, Writable value, int numPartitions)
```

- The *numPartitions* is the number of reducers used in the MapReduce program and it is specified in the driver program (by default 1)
- In the relevant frequencies computing problem

```
public int getPartition(StringPair key, IntWritable value, int numPartitions){  
    return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;  
}
```

References

- Chapters 3.3, 3.4, 4.2, 4.3, and 4.4. Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.
- Chapter 5 Hadoop I/O. Hadoop The Definitive Guide.

End of Chapter3