COMP9313: Big Data Management

Lecturer: Xin Cao

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

Chapter 2.2: MapReduce II

Overview of Previous Lecture

- Motivation of MapReduce
- Data Structures in MapReduce: (key, value) pairs
- Hadoop MapReduce Programming
  - Mapper
    - Output pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output pairs.
  - Reducer
    - Reducer has 3 primary phases: shuffle, sort and reduce.
  - Combiner
    - Users can optionally specify a combiner to perform local aggregation of the intermediate outputs
  - Partitioner
    - The total number of partitions is the same as the number of reduce tasks for the job. Users can control which go to which Reducer by implementing a custom Partitioner.
  - Driver: configure the job and start running

For Large Datasets

- 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.
- 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

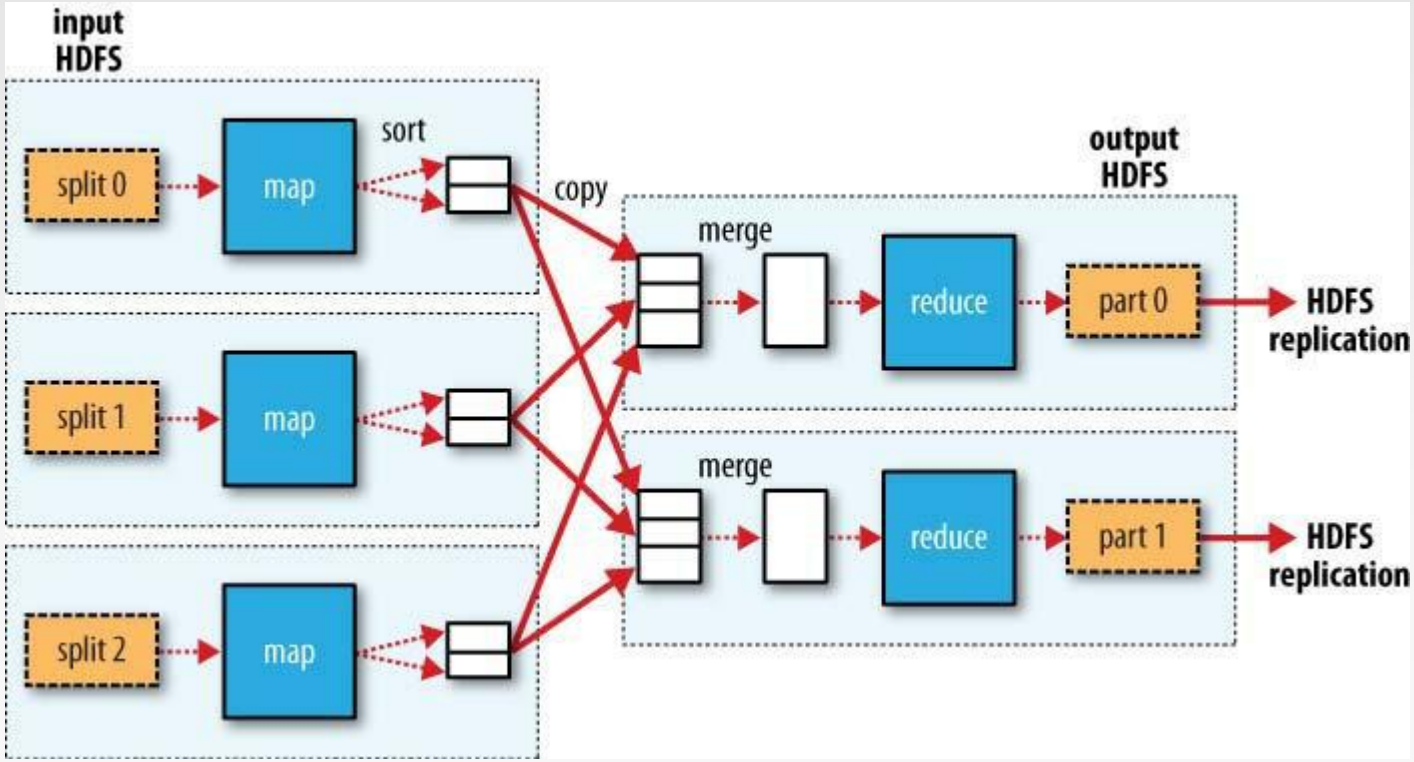
- Map tasks write their output to local disk (not to HDFS)
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - Storing it in HDFS with replication, would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on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

For Large Datasets

- 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

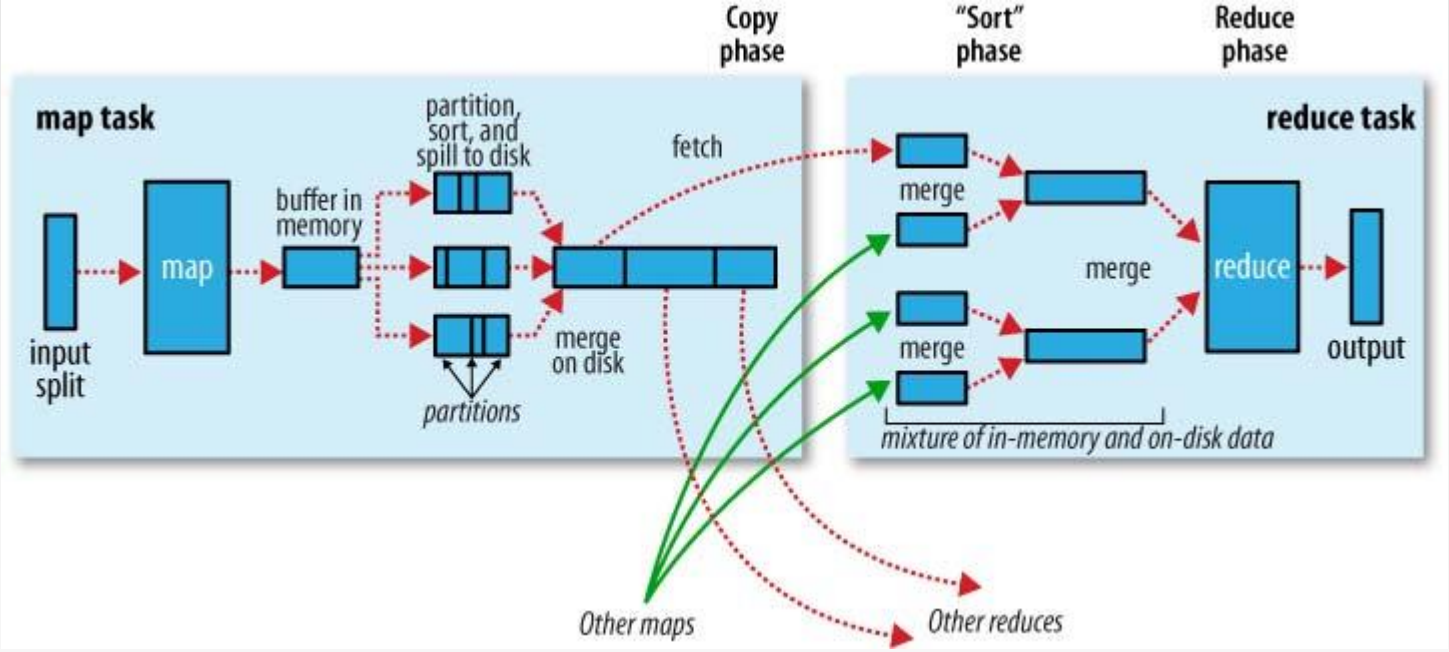
More Detailed MapReduce Dataflow

- 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



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: 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 ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← 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 ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             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

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         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

```
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 [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] 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 [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         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

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))

1: class Combiner
2:     method Combine(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))

1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, pair (r_avg, cnt))
```

Fixed?

Check the correctness by removing the combiner

Computing the Mean: Version 4

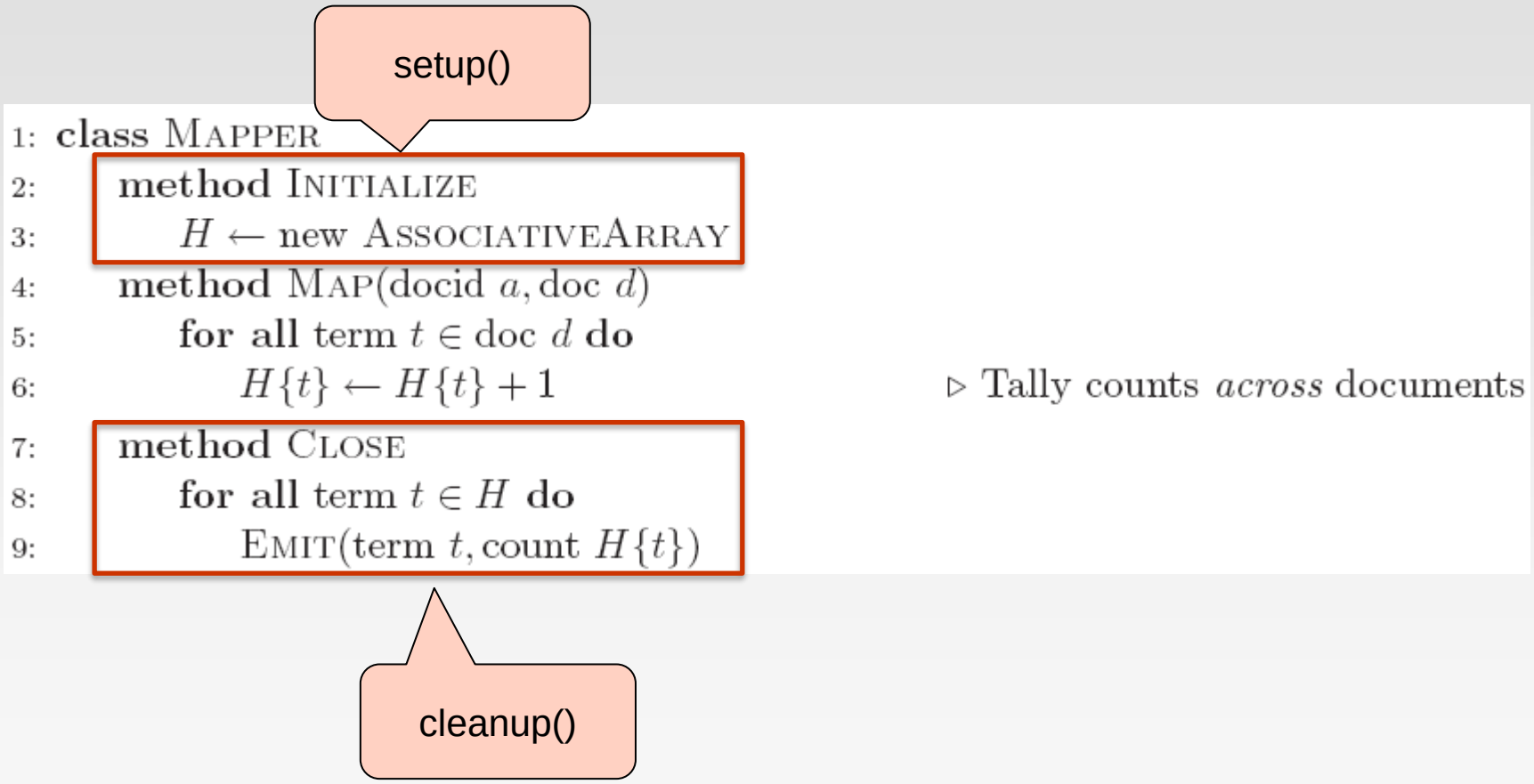
```
1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S{t} ← S{t} + r
7:         C{t} ← C{t} + 1
8:     method Close
9:         for all term t ∈ S do
10:            Emit(term t, pair (S{t}, 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

Word Count: Version 2



Implementation in MRJob

- A 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()

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!

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ 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 [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← 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) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2

a → { 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

```
    a → { b: 1,         d: 5, e: 3 }
+   a → { b: 1, c: 2, d: 2,        f: 2 }
    a → { b: 2, c: 2, d: 7, e: 3, f: 2 }
```

Key: cleverly-constructed data structure brings together partial results

Stripes: Pseudo-Code

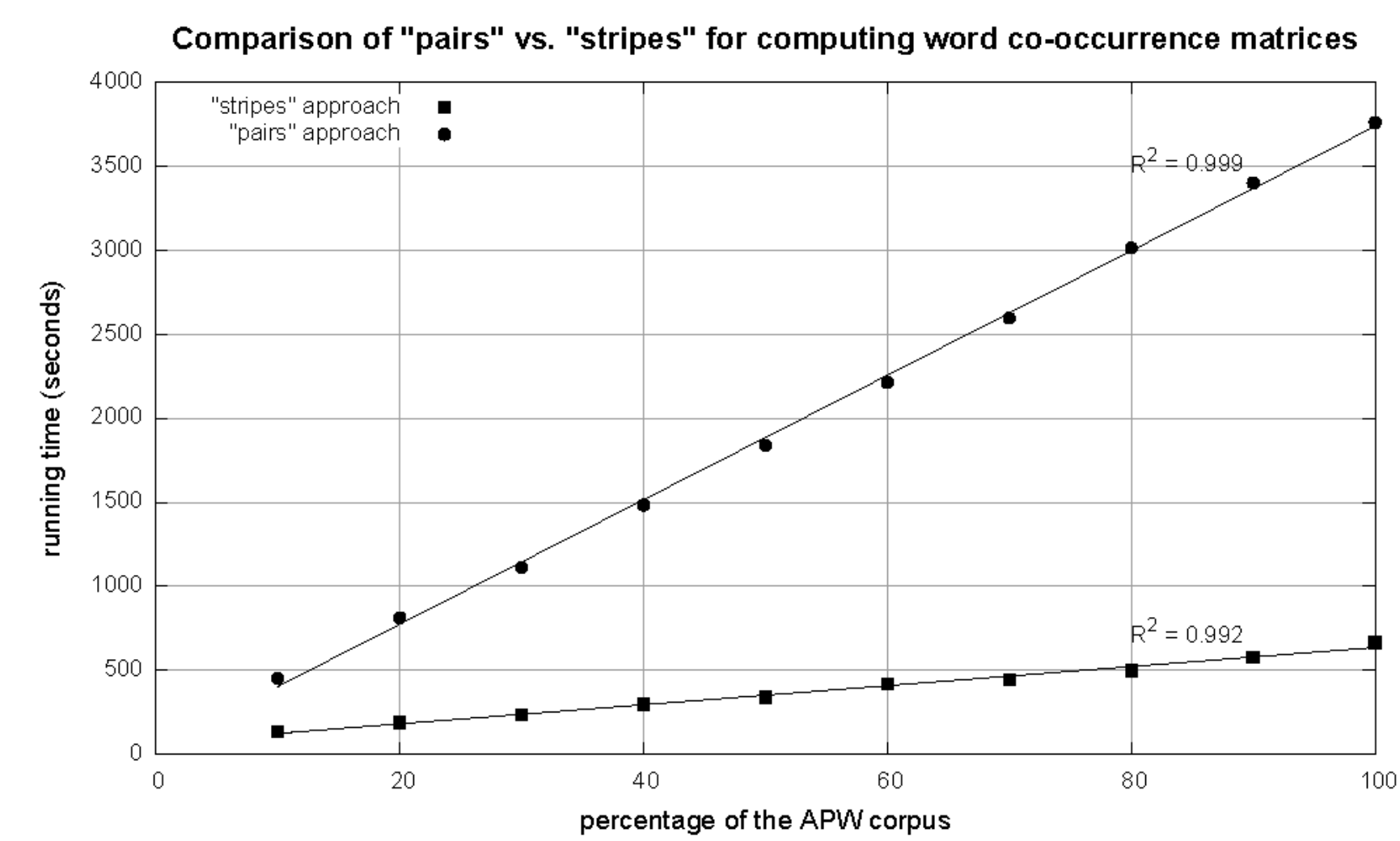
```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H_1, H_2, H_3, ...])
3:         H_f ← new AssociativeArray
4:         for all stripe H ∈ stripes [H_1, H_2, H_3, ...] do
5:             Sum(H_f, H)                        ▷ Element-wise sum
6:         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 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?

Serialization

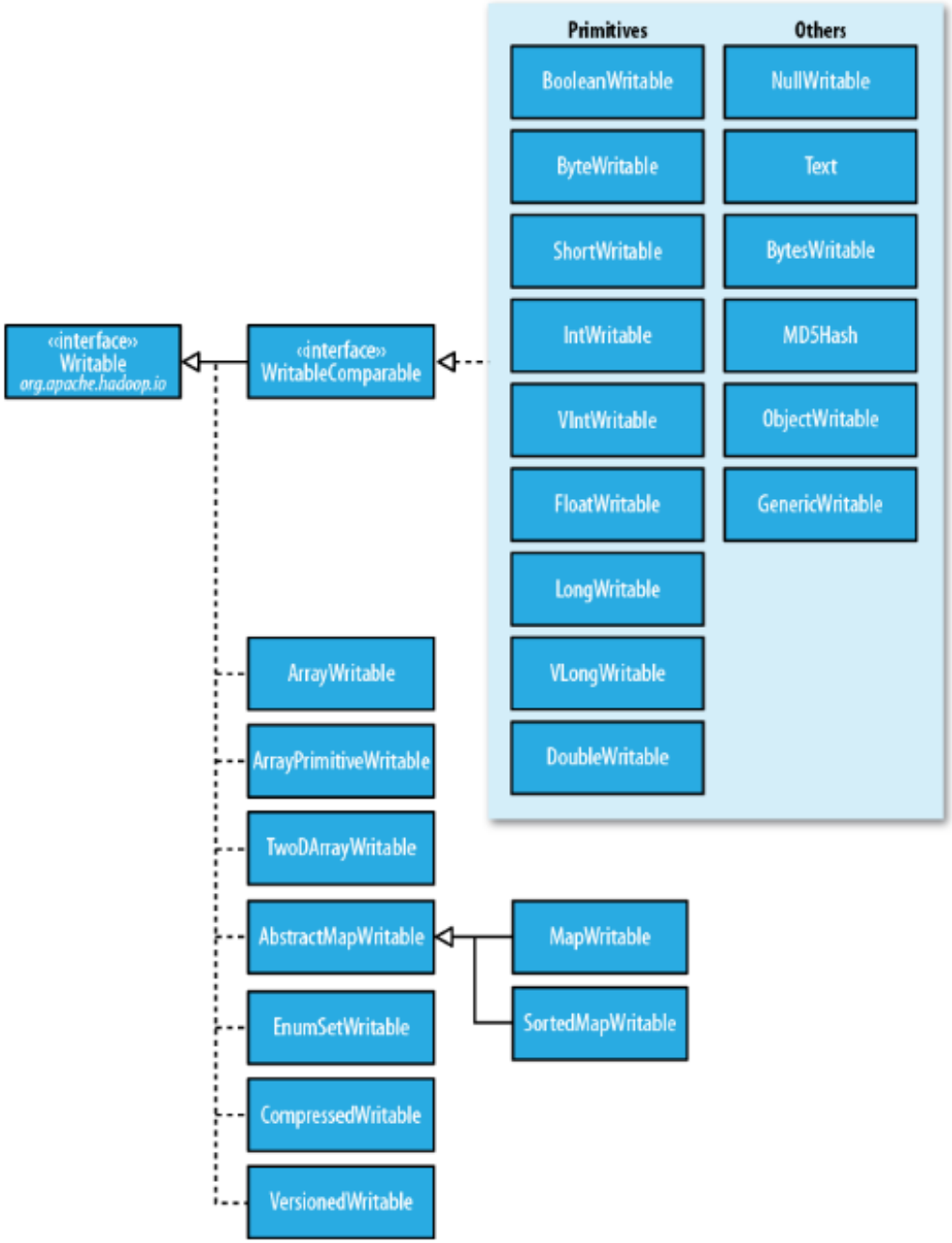
- Process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage
- Deserialization is the reverse process of serialization
- Requirements
  - Compact
    - To make efficient use of storage space
  - Fast
    - The overhead in reading and 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!!


Primitives
Others
BooleanWritable
NullWritable
ByteWritable
Text
ShortWritable
BytesWritable
IntWritable
MD5Hash
VIntWritable
ObjectWritable
FloatWritable
GenericWritable
LongWritable
VLongWritable
DoubleWritable
«interface»
Writable
org.apache.hadoop.io
«interface»
WritableComparable
ArrayWritable
ArrayPrimitiveWritable
TwoDArrayWritable
AbstractMapWritable
MapWritable
SortedMapWritable
EnumSetWritable
CompressedWritable
VersionedWritable

Writable Wrappers for Java Primitives

- There are **Writable** wrappers for all the Java primitive types except shot 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* → *{ 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/r3.3.1/api/org/apache/hadoop/io/MapWritable.html
- How to implement the combiner?

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 → { 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 → { 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

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 an 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 paritioner to do this job

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…

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