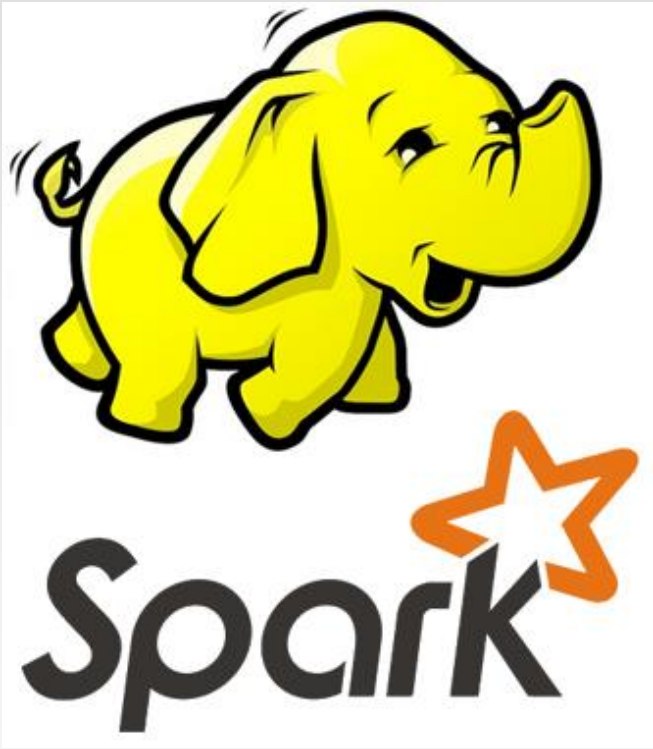
COMP9313: Big Data Management



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

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

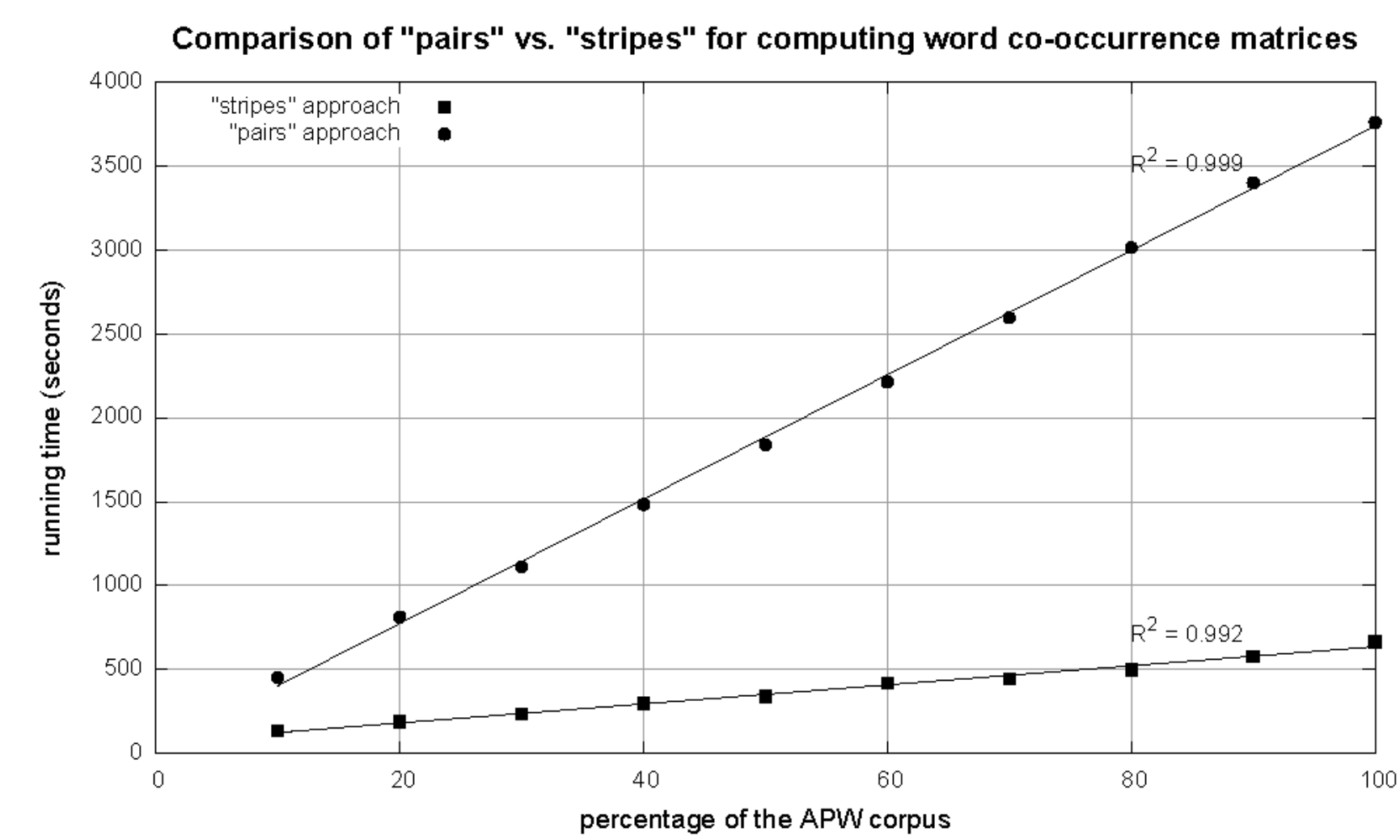
```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

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

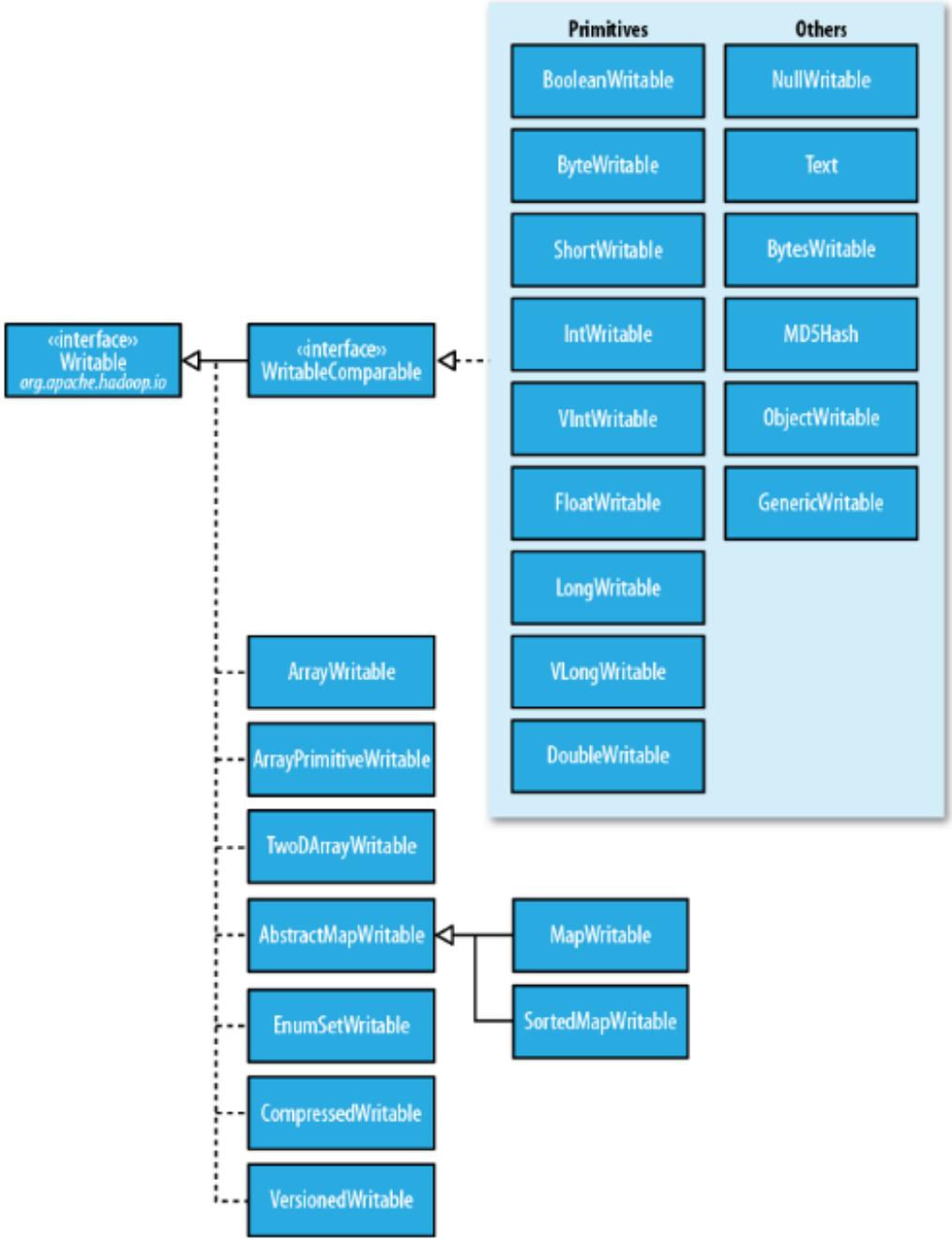
- Process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage
- Deserialization is the reverse process of serialization
- Requirements
  - Compact
    - To make efficient use of storage space
  - Fast
    - The overhead in reading and 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
ByteWritable
ShortWritable
IntWritable
VIntWritable
FloatWritable
LongWritable
VLongWritable
DoubleWritable
NullWritable
Text
BytesWritable
MD5Hash
ObjectWritable
GenericWritable
«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/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 paritioner 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

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

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

- 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

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

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

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

is now buffered in the reducer side

  - Problems?

$f(w_j|w_i)$: "Pairs"

If reducers receive pairs not sorted

$((a, b_1), \{1, 1, 1, \ldots\})$
$((c, d_1), \{1, 1, 1, \ldots\})$
$((a, b_2), \{1, 1, 1, \ldots\})$
… …

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"

$((a, b_1), \{1, 1, 1, \ldots\})$ and $((a, b_2), \{1, 1, 1, \ldots\})$ 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

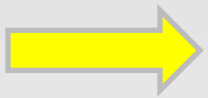
- Still suffer from the memory problem!

$f(w_j|w_i)$: "Pairs"

- Better solutions?

(a, *) → 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	
$(\text{dog}, *)$	[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]	$f(\text{zebra}\|\text{dog}) = 5/42908$
$(\text{doge}, *)$	[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 YourPatitioner 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 FirstPatitioner 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;
}
```

Design Pattern 4: Value-to-key Conversion

Secondary Sort

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered

- What if want to sort value as well?
  - E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$
  - Google's MapReduce implementation provides built-in functionality
  - Unfortunately, Hadoop does not support

- Secondary Sort: sorting values associated with a key in the reduce phase, also called “value-to-key conversion”

Secondary Sort

- Sensor data from a scientific experiment: there are m sensors each taking readings on continuous basis

  (t1, m1, r_{80521})

  (t1, m2, r_{14209})

  (t1, m3, r_{76742})

  …

  (t2, m1, r_{21823})

  (t2, m2, r_{66508})

  (t2, m3, r_{98347})

- We wish to reconstruct the activity at each individual sensor over time
- In a MapReduce program, a mapper may emit the following pair as the intermediate result

  m_1 -> (t_1, r_{80521})

  - We need to sort the value according to the timestamp

Secondary Sort

- 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, (m_1, t_1)
    - The mapper emits (m_1, t_1) -> r_{80521}
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?
    - Sensor readings are split across multiple keys. Reducers need to know when all readings of a sensor have been processed
    - All pairs associated with the same sensor are shuffled to the same reducer (use partitioner)

How to Implement Secondary Sort in MapReduce?

Secondary Sort: Another Example

- Consider the temperature data from a scientific experiment. Columns are year, month, day, and daily temperature, respectively:

```
2012, 01, 01, 5
2012, 01, 02, 45
2012, 01, 03, 35
2012, 01, 04, 10
...
2001, 11, 01, 46
2001, 11, 02, 47
2001, 11, 03, 48
2001, 11, 04, 40
...
2005, 08, 20, 50
2005, 08, 21, 52
2005, 08, 22, 38
2005, 08, 23, 70
```

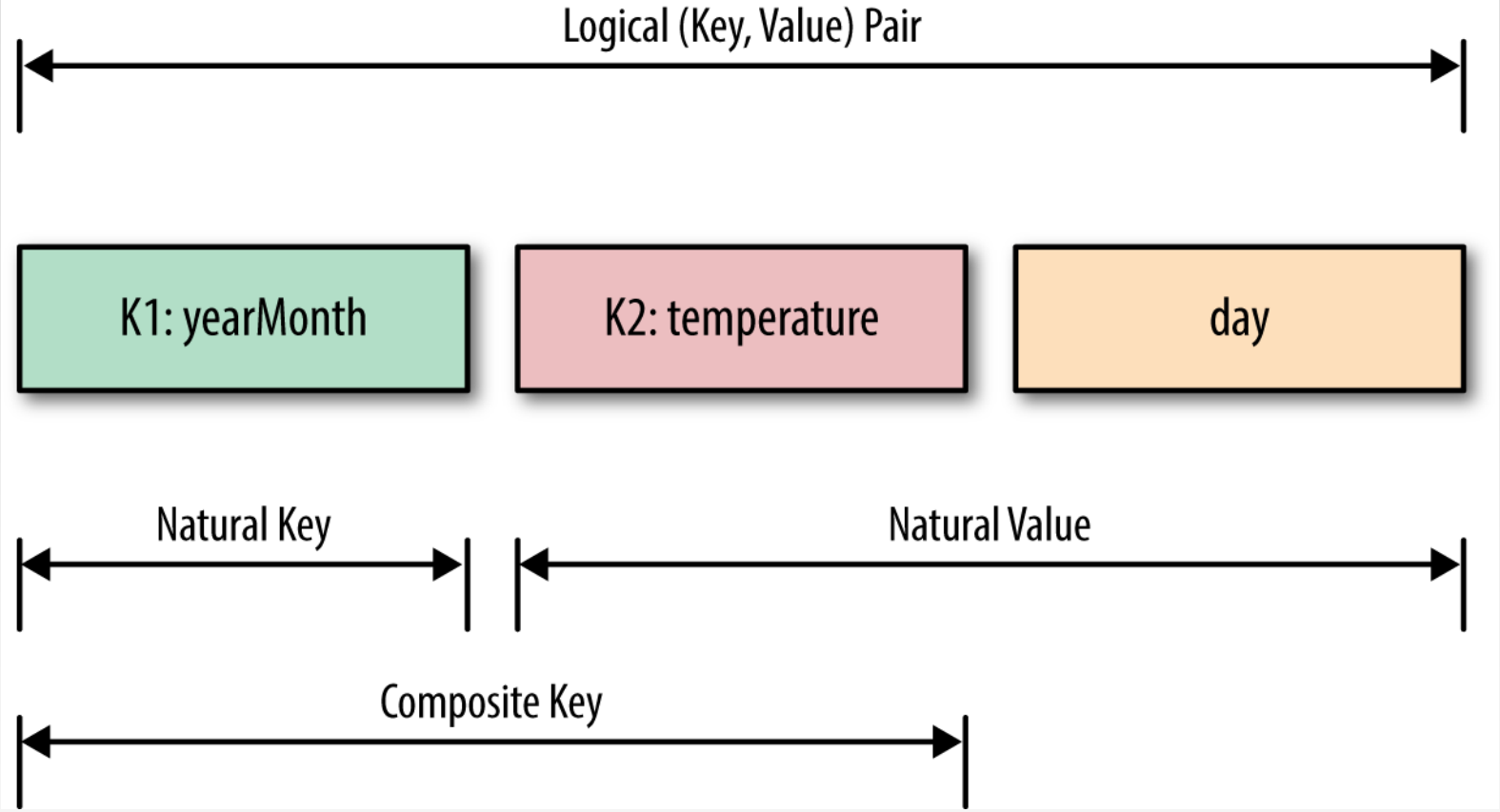
```
2012-01:  5, 10, 35, 45, ...
2001-11: 40, 46, 47, 48, ...
2005-08: 38, 50, 52, 70, ...
```

- We want to output the temperature for every year-month with the values sorted in ascending order.

Solutions to the Secondary Sort Problem

- Use the *Value-to-Key Conversion* design pattern:
  - form a composite intermediate key, (K, V), where V is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V) into a reducer key, simply create a composite key
    - K: year-month
    - V: temperature data
- Let the MapReduce execution framework do the sorting (rather than sorting in memory, let the framework sort by using the cluster nodes).
- Preserve state across multiple key-value pairs to handle processing. Write your own partitioner: partition the mapper's output by the natural key (year-month).

Secondary Sorting Keys



Customize The Composite Key

```
public class DateTemperaturePair
      implements Writable, WritableComparable<DateTemperaturePair> {
            private Text yearMonth = new Text(); // natural key
            private IntWritable temperature = new IntWritable(); // secondary key
            … …
            @Override
            /**
            * This comparator controls the sort order of the keys.
            */
            public int compareTo(DateTemperaturePair pair) {
                  int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
                  if (compareValue == 0) {
                        compareValue = temperature.compareTo(pair.getTemperature());
                  }
                  return compareValue; // sort ascending
            }
            … …
}
```

Customize The Partitioner

```
public class DateTemperaturePartitioner
      extends Partitioner<DateTemperaturePair, Text> {
            @Override
            public int getPartition(DateTemperaturePair pair, Text text, int numberOfPartitions) {
            // make sure that partitions are non-negative
                  return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
            }
}
```

Utilize the natural key only for partitioning

Grouping Comparator

- Controls which keys are grouped together for a single call to Reducer.reduce() function.

```
public class DateTemperatureGroupingComparator extends WritableComparator {
        … …
        protected DateTemperatureGroupingComparator(){
                super(DateTemperaturePair.class, true);
        }
        @Override
        /* This comparator controls which keys are grouped together into a single call to the
reduce() method */
        public int compare(WritableComparable wc1, WritableComparable wc2) {
                DateTemperaturePair pair = (DateTemperaturePair) wc1;
                DateTemperaturePair pair2 = (DateTemperaturePair) wc2;
                return pair.getYearMonth().compareTo(pair2.getYearMonth());
        }
}
```

Consider the natural key only for grouping

- Configure the grouping comparator using Job object:

```
job.setGroupingComparatorClass(DateTemperatureGroupingComparator.class);
```

MapReduce Algorithm Design

- Aspects that are not under the control of the designer
  - Where a mapper or reducer will run
  - When a mapper or reducer begins or finishes
  - Which input key-value pairs are processed by a specific mapper
  - Which intermediate key-value paris are processed by a specific reducer
- Aspects that can be controlled
  - Construct data structures as keys and values
  - Execute user-specified initialization and termination code for mappers and reducers (pre-process and post-process)
  - Preserve state across multiple input and intermediate keys in mappers and reducers (in-mapper combining)
  - Control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys (order inversion)
  - Control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer (partitioner)

Application: Building Inverted Index

MapReduce in Real World: Search Engine

- Information retrieval (IR)
  - Focus on textual information (= text/document retrieval)
  - Other possibilities include image, video, music, …
- Boolean Text retrieval
  - Each document or query is treated as a "bag" of words or terms. Word sequence is not considered
  - Query terms are combined logically using the Boolean operators AND, OR, and NOT.
    - E.g., ((data AND mining) AND (NOT text))
  - Retrieval
    - Given a Boolean query, the system retrieves every document that makes the query logically true.
    - Called exact match
  - The retrieval results are usually quite poor because term frequency is not considered and results are not ranked

Boolean Text Retrieval: Inverted Index

- The inverted index of a document collection is basically a data structure that
  - attaches each distinctive term with a list of all documents that contains the term.
  - The documents containing a term are sorted in the list

- Thus, in retrieval, it takes constant time to
  - find the documents that contains a query term.
  - multiple query terms are also easy handle as we will see soon.

Boolean Text Retrieval: Inverted Index

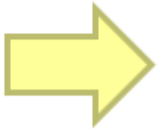
Doc 1 one fish, two fish

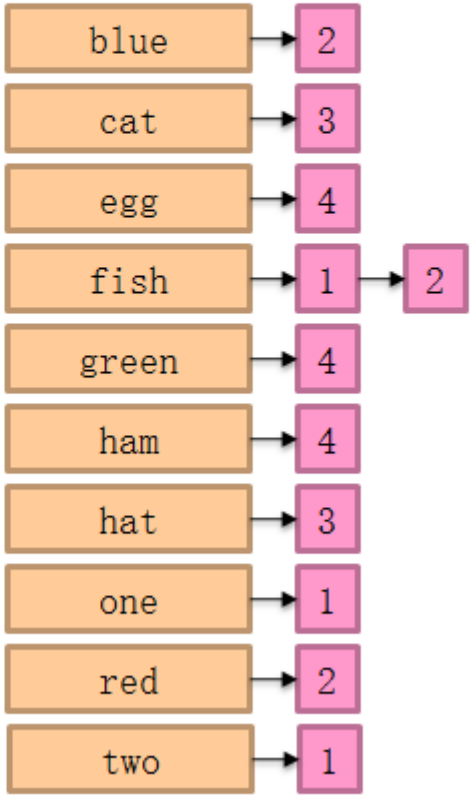
Doc 2 red fish, blue fish

Doc 3 cat in the hat

Doc 4 green eggs and ham

	1	2	3	4
blue		1		
cat			1	
egg				1
fish	1	1		
green				1
ham				1
hat			1	
one	1			
red		1		
two	1			



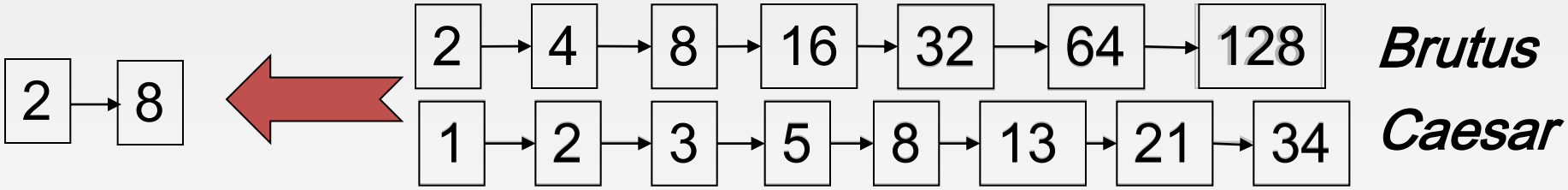


Search Using Inverted Index

- Given a query **q**, search has the following steps:
  - Step 1 (vocabulary search): find each term/word in q in the inverted index.
  - Step 2 (results merging): Merge results to find documents that contain all or some of the words/terms in q.
  - Step 3 (Rank score computation): To rank the resulting documents/pages, using:
    - content-based ranking
    - link-based ranking
    - Not used in Boolean retrieval

Boolean Query Processing: AND

- Consider processing the query: ***Brutus*** *AND* ***Caesar***
  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate *Caesar* in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings:
    - Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are x and y, the merge takes O(x+y) operations.

Crucial: postings sorted by docID.

MapReduce it?

- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important
  - For the web, crawling is a challenge in itself
- The retrieval problem
  - Must have sub-second response time
  - For the web, only need relatively few results

Perfect for MapReduce!

Uh… not so good…

MapReduce: Index Construction

- Input: documents: (docid, doc), ..
- Output: (term, [docid, docid, …])
  - E.g., (long, [1, 23, 49, 127, …])
    - The docid are sorted !! (used in query phase)
  - docid is an internal document id, e.g., a unique integer. Not an external document id such as a URL
- How to do it in MapReduce?

MapReduce: Index Construction

- A simple approach:
  - Each Map task is a document parser
    - Input: A stream of documents
      - (1, long ago …), (2, once upon …)
    - Output: A stream of (term, docid) tuples
      - (long, 1) (ago, 1) … (once, 2) (upon, 2) …
  - Reducers convert streams of keys into streams of inverted lists
    - Input: (long, [1, 127, 49, 23, …])
    - The reducer sorts the values for a key and builds an inverted list
      - Longest inverted list must fit in memory
    - Output: (long, [1, 23, 49, 127, …])
- Problems?
  - Inefficient
  - docids are sorted in reducers

References

- Chapters 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.

End of Chapter3