COMP9313: Big Data Management

Lecturer: Xin Cao

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

Ch ce ll

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 opti

  - The number of calling (even zer ect 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))

MapRedu https://eduassistpro.github.io/ ign Patterns

Design Pa r 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
  - Communica

- Thus… avoid communication!
  - Reduce intermediate data via lo n
  - Combiners can help

WordCount Baseline

What’s the impact of combiners?

Word Count: Version 1

Are combiners still needed?

Word Count: Version 2

rve state across
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 f
- 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
  - May be run
- Example: find average of all integer ith the same key

Computing the Mean: Version 1

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

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

Fixed?

Check the correctness by removing the combiner

Computing the Mean: Version 4

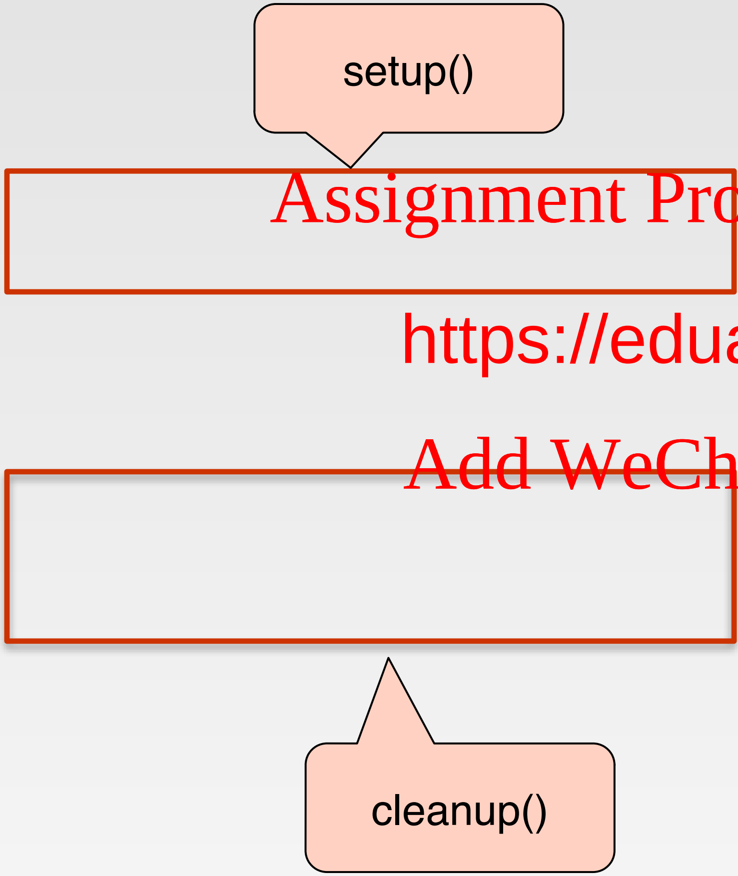
How to Implement In-mapper Combiner

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 Com
  - Use setup() data structure
  - Use clearnup() to emit the final

Word Count: Version 2



Design s 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 e
    - A large n ection itself)
    - Goal: keep track of interesti out 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!

“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand

- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)
  - Not many o rk

Another Try: "Stripes"

- Idea: group together pairs into an associative array

$(a, b) \rightarrow 1$
$(a, c) \rightarrow 2$
$(a, d) \rightarrow 5$
$(a, e) \rightarrow 3$
$(a, f) \rightarrow 2$

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

- Each mapper ta

  - Generate all co-occurring term

  - For each term, emit $a \rightarrow \{ b: count_b, \; c: count_c, \; d: count_d \ldots \}$
- Reducers perform element-wise sum of associative arrays

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

Key: cleverly-constructed data structure brings together partial results

Stripes: Pseudo-Code

“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners

- Disadvantages
  - More difficul
  - Underlying
  - Fundamental limitation in terms t 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 appro
  - Keep track o e same term
  - Generates fewer and shorted in s
  - The framework has less sorting
  - 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"

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 sto
  - 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

ception;

```
public interface Writable {
    void wri
    void rea                                         IOException;
}
```


- All values must implement interface

- All keys must implement interface WritableComparable

- context.write(WritableComparable, Writable)
  - You cannot use java primitives here!!

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

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, u r the value is positive or negative

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 muta tions, except NullWritable
    - Different from java.lang.Stri
    - You can reuse a Text instan ne 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: th educe
  - put
  - get(Object key)
  - containsKey(Object ke
  - 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 custo [illegible] 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 interfac

- We could use T

  - Value is a string of comma sep
  - Have to convert the values to st ful 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 fu
  - A method to
  - A method to get the values (two
  - write() method : serialize the les (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(D
        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 WritableCompar
  - Why Writabl
- We could use Text again
  - Value is a string of comma sep
  - 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 fu
  - A method to cts)
  - A method to get the values (two)
  - write() method : serialize the les (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(D
        String[] strings = new String[] { first, sec
        WritableUtils.writeStringArray(out, strin
}
```


  - 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(Stri
    if (s1 == null && s2
        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(D
        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 onl Code of the first part.
  - You can also write a paritioner to do this job

Design Inversion

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

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 ive frequencies $f(w_j|w_i)$
    - What prop 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 t and we're done

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

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

- 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,

No way to compute $f(b_1|a)$ becau a) 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

$a \rightarrow \{b_1:3, b_2:12, b_3:$

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, …})
((c, d_1), {1, 1, 1, …})
((a, b_2), {1, 1, 1, …})
…

W rginal?

- We must define the sort order of th
  - 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
  - The default value of the intermediate key, modulo the n ers
  - For a complex key, the raw byt n 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**
$(a, b_1) \rightarrow 3$	$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12$	$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7$	$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow$	$) \rightarrow 1 / 32$
…	

- The key is to properly sequence dat reducers
  - If it were possible to compute th he 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

  - Using combi nts will be aggregated before being sent to the reduce

- 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:

  - 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

  - Emit a special key-value pair to arginal
  - Control the sort order of the inte 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 cou

- Optimizations
  - Apply in-memory combining pat late 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 structur rtial results together

  - Each reducer receives all the d complete the computation
  - Illustrated by the "stripes" approach

How to Implement Order Inversion

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 releva em


```
public static class Fi air, IntWritable>{
```


- It overrides the *getPartition* function ee parameters


```
    public int getPartition(WritableComparabl ue, 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.