COMP9313: Big Data Management

Lecturer: Siqing Li

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

Chapter 3.1: MapReduce III

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"

e.g.

$(a, b_1),(a, b_1), (a, b_1), (a, b_1), (a, b_2),(a, b_2)$

mapper_1: $(a, b_1) \rightarrow 3$
mapper_2: $(a, b_1) \rightarrow 1$, $(a, b_2) \rightarrow 2$

shuffle: $(a, b_1) \rightarrow 3$, $(a, b_1) \rightarrow 1$, $(a, b_2) \rightarrow 2$

reducer:

$(a, b_1) \rightarrow (3 + 1) / (3 + 1 + 2)$
$(a, b_2) \rightarrow 2 / (3 + 1 + 2)$

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

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")


class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def combiner(self, word, counts):
        yield word, sum(counts)

    def reducer(self, word, counts):
        yield word, sum(counts)


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

Let's check what's happening in the MRJob.

The "reducer" function will be called twice, as two key-value pairs are given.

first call : $(a, b_1) \rightarrow 3$, $(a, b_1) \rightarrow 1$;

second call: $(a, b_2) \rightarrow 2$

How to compute the marginal?

$f(w_j|w_i)$: "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 can we 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 could 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: $\sum_{w'} N(\text{dog}, w') = 42908$
$(\text{dog}, \text{aardvark})$	[2,1]	$f(\text{aardvark}\|\text{dog}) = 3/42908$
$(\text{dog}, \text{aardwolf})$	[1]	$f(\text{aardwolf}\|\text{dog}) = 1/42908$
...		
$(\text{dog}, \text{zebra})$	[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?

Partitioner in Hadoop Streaming

- Hadoop has a library class, KeyFieldBasedPartitioner, that is useful for many applications. This class allows the Map/Reduce framework to partition the map outputs based on certain key fields, not the whole keys.

```
mapred streaming \
  -D stream.map.output.field.separator=. \
  -D stream.num.map.output.key.fields=4 \
  -D map.output.key.field.separator=. \
  -D mapreduce.partition.keypartitioner.options=-k1,2 \
  -D mapreduce.job.reduces=12 \
  -input myInputDirs \
  -output myOutputDir \
  -mapper /bin/cat \
  -reducer /bin/cat \
  -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

```
11.12.1.2
11.14.2.3
11.11.4.1
11.12.1.1
11.14.2.2
```

  - "-D stream.map.output.field.separator=." specifies "." as the field separator for the map outputs. By default, the separator is '\t'
  - "-D stream.num.map.output.key.fields=4" means the prefix up to the fourth "." in a line will be the key and the rest of the line (excluding the fourth ".") will be the value.

Partitioner in Hadoop Streaming

```
mapred streaming \
  -D stream.map.output.field.separator=. \
  -D stream.num.map.output.key.fields=4 \
  -D map.output.key.field.separator=. \
  -D mapreduce.partition.keypartitioner.options=-k1,2 \
  -D mapreduce.job.reduces=12 \
  -input myInputDirs \
  -output myOutputDir \
  -mapper /bin/cat \
  -reducer /bin/cat \
  -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

```
11.12.1.2
11.14.2.3
11.11.4.1
11.12.1.1
11.14.2.2
```

```
11.11.4.1
-----------
11.12.1.2
11.12.1.1
-----------
11.14.2.3
11.14.2.2
```

- "-D map.output.key.field.separator=." means the separator for the key is also "."
- "-D mapreduce.partition.keypartitioner.options=-k1,2" means MapReduce will partition the map outputs by the first two fields of the keys
- This guarantees that all the key/value pairs with the same first two fields in the keys will be partitioned into the same reducer.

Partitioner in Hadoop Streaming

```
mapred streaming \
  -D stream.map.output.field.separator=. \
  -D stream.num.map.output.key.fields=4 \
  -D map.output.key.field.separator=. \
  -D mapreduce.partition.keypartitioner.options=-k1,2 \
  -D mapreduce.job.reduces=12 \
  -input myInputDirs \
  -output myOutputDir \
  -mapper /bin/cat \
  -reducer /bin/cat \
  -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

```
11.12.1.2
11.14.2.3
11.11.4.1
11.12.1.1
11.14.2.2
```

```
11.11.4.1
-----------
11.12.1.2
11.12.1.1
-----------
11.14.2.3
11.14.2.2
```

- Formally, "-D mapreduce.partition.keypartitioner.options=-km,n" means MapReduce will partition the map outputs by the fields from m to n

.

Partitioner in Hadoop Streaming

- For the relative frequency computation task, you can do like:

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
        -D stream.map.output.field.separator=\\t \
        -D stream.num.map.output.key.fields=1 \
        -D map.output.key.field.separator=, \
        -D mapreduce.partition.keypartitioner.options=-k1,1 \
        -D mapreduce.job.reduces=2 \
        -input input \
        -output output \
        -mapper mapper.py \
        -reducer reducer.py \
        -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \
        -file mapper.py \
        -file reducer.py
```

Partitioner in MRJob

- In your class, configure JOBCONF, like:

```
JOBCONF = {
        'mapreduce.map.output.key.field.separator': ',',
        'mapreduce.job.reduces':2,
        'mapreduce.partition.keypartitioner.options':'-k1,1',
        'partitioner':'org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner'
}
```

  - You also need to add one line “SORT_VALUES = True” into your code.
  - Assume each key is a pair of strings separated by “,” like “term1,term2”. Hadoop performs the partitioning based on the whole key. However, the above configure would let Hadoop know that the partitioning is only based the first field of the key (i.e., “term1”).

A Complete Example

Inputs:

$(a_1, b_1), (a_1, b_1), (a_1, b_2), (a_1, b_2), (a_1, b_2)$

$(a_2, b_1), (a_2, b_2), (a_1, b_2), (a_1, b_2), (a_1, b_2)$

mapper1:

$(a_1, b_1) \rightarrow 2$

$(a_1, b_2) \rightarrow 3$

$(a_1, *) \rightarrow 5$

mapper2:

$(a_2, b_1) \rightarrow 1$

$(a_2, b_2) \rightarrow 1$

$(a_1, b_2) \rightarrow 3$

$(a_1, *) \rightarrow 5$

$(a_2, *) \rightarrow 2$

shuffle:

$(a_1, b_1) \rightarrow 2$

$(a_1, b_2) \rightarrow 3$

$(a_1, *) \rightarrow 5$

$(a_2, b_1) \rightarrow 1$

$(a_2, b_2) \rightarrow 1$

$(a_1, b_2) \rightarrow 3$

$(a_1, *) \rightarrow 5$

$(a_2, *) \rightarrow 2$

$\rightarrow$

$(a_1, *) \rightarrow 5$

$(a_1, *) \rightarrow 5$

$(a_1, b_1) \rightarrow 2$

$(a_1, b_2) \rightarrow 3$

$(a_1, b_2) \rightarrow 3$

$(a_2, *) \rightarrow 2$

$(a_2, b_1) \rightarrow 1$

$(a_2, b_2) \rightarrow 1$

Tips for MRJob

- In MRJob, the “reducer” function receives key-value pairs according to keys, not partitions. For example, MRJob will call “reducer” 6 times because 6 unique keys are detected.

	key	values
1th call:	$(a_1, *)$	[5,5]
2th call	(a_1, b_1)	[2]
3th call	(a_1, b_2)	[3,3]
4th call	$(a_2, *)$	[2]
5th call	(a_2, b_1)	[1]
6th call	(a_2, b_2)	[1]

- Received values are stored in a generator, not a list
- Use “reducer_init()” and “reducer_final()” to initialize and finalize your reducer.

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.

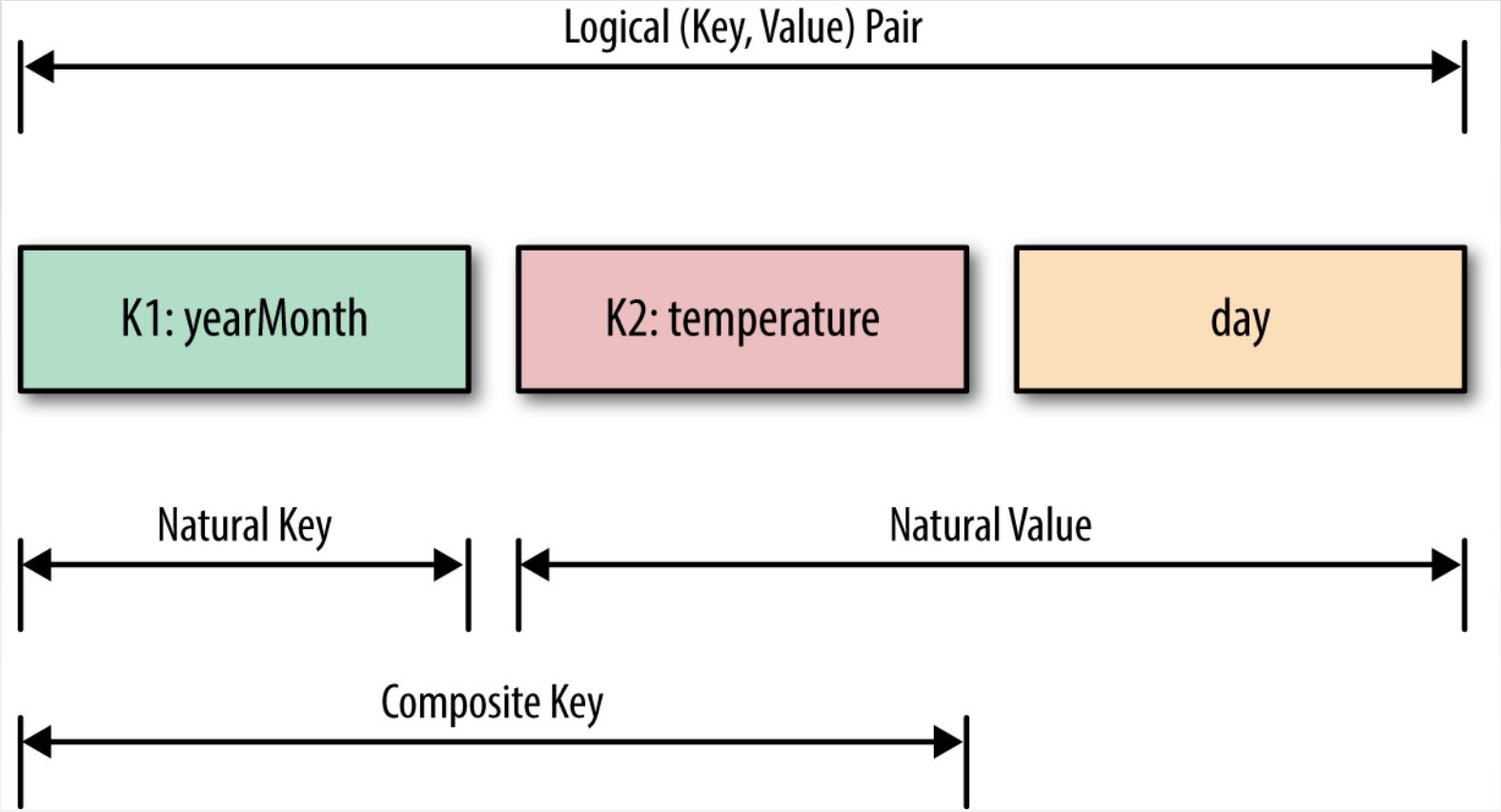
Solutions to the Secondary Sort Problem

- 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



Secondary Sort by Hadoop Streaming

- Hadoop has a library class, KeyFieldBasedComparator, that is useful for secondary sort.

```
mapred streaming \
  -D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapreduce.lib.partition.KeyFieldBasedComparator \
  -D stream.map.output.field.separator=. \
  -D stream.num.map.output.key.fields=4 \
  -D mapreduce.map.output.key.field.separator=. \
  -D mapreduce.partition.keycomparator.options=-k2,2nr \
  -D mapreduce.job.reduces=1 \
  -input myInputDirs \
  -output myOutputDir \
  -mapper /bin/cat \
  -reducer /bin/cat
```

```
11.12.1.2        11.14.2.3
11.14.2.3        11.14.2.2
11.11.4.1        11.12.1.2
11.12.1.1        11.12.1.1
11.14.2.2        11.11.4.1
```

- The map output keys of the above Map/Reduce job have four fields separated by “.”
- MapReduce will sort the outputs by the second field of the keys using the -D mapreduce.partition.keycomparator.options=-k2,2nr option.
  - -n specifies that the sorting is numerical sorting
  - -r specifies that the result should be reversed
  - -km,n means sort by the fields from m to n

Secondary Sort in MRJob

- In your class, configure JOBCONF, like:

```
JOBCONF = {
            'mapreduce.map.output.key.field.separator': ',',
            'mapreduce.job.reduces':2,
            'mapreduce.partition.keycomparator.options':'-k1,1 –k2,2r'
            'mapreduce.job.output.key.comparator.class':'org.apache.hadoop.mapreduc
e.lib.partition.KeyFieldBasedComparator'
}
```

  - You also need to add one line “SORT_VALUES = True” into your code.

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 pairs 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)

Test and Debug MRJob Locally

- To test your mapper, add the --mapper option to your run command:
  - python job.py --mapper text.txt
- You can store the results of your mapper in an output file
  - python job.py --mapper text.txt > output.txt
- Run your code locally, you can observe that the mapper output is not sorted. Since there is no sorting and shuffling and partitioning phases in this simulated environment.
- Before passing the file storing mapper output to your reducer, we need to utilize the Linux "sort" command to first sort the mapper output
  - cat output.txt | sort -k1,1 | python job.py --reducer
- You can also run your mapper and pipe the results to your reducer
  - python job.py --mapper text.txt | sort -k1,1 | python job.py --reducer

Test and Debug MRJob on Hadoop

- Use MRStep to define a step with mapper only to test your mapper on Hadoop first, and then include the reducer and run on Hadoop.
- Use sys.stderr to log the necessary information of your program
- Check the logs to see the error:
  - After running a job, check the logs at $HADOOP_HOME/logs/userlogs
  - In this directory, you can see a folder containing all the information about your job

```
comp9313@comp9313-VirtualBox:~/hadoop-3.3.2$ cd logs/userlogs
comp9313@comp9313-VirtualBox:~/hadoop-3.3.2/logs/userlogs$ ls
application_1664214774281_0001
```

  - Go into this folder and check in each container log ("stderr" in each container log folder)

```
comp9313@comp9313-VirtualBox:~/hadoop-3.3.2/logs/userlogs/application_1664214774
281_0001/container_1664214774281_0001_01_000004$ ls
directory.info      prelaunch.err  stderr  syslog
launch_container.sh prelaunch.out  stdout  syslog.shuffle
```

References

- Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.
- Hadoop The Definitive Guide. Hadoop I/O, and MapReduce Features chapters.

End of Chapter 3.1