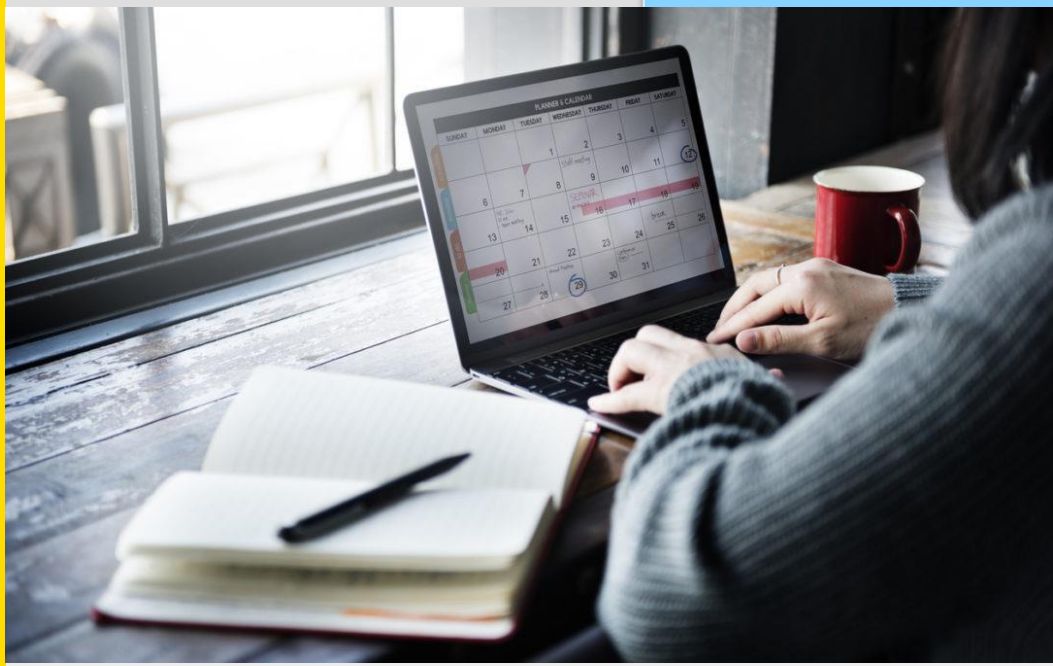
Welcome to ZZEN9313 Big Data Management



- We'll be starting at 8:30 pm
- In preparation for this webinar please check that your microphone is muted.
- We'll be taking questions by chat

This webinar will be recorded and made available for your ongoing reference.

UNSW SYDNEY

Week 4's Tutor: Linhan Zhang and Siqing Li

Week 4's QA Sessions:

Wednesday 7—8 pm

Thursday 7—8 pm

Friday 7—8 pm

Saturday 7—8 pm

Part 1: MapReduce Data Flow

Overview of MapReduce

- 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

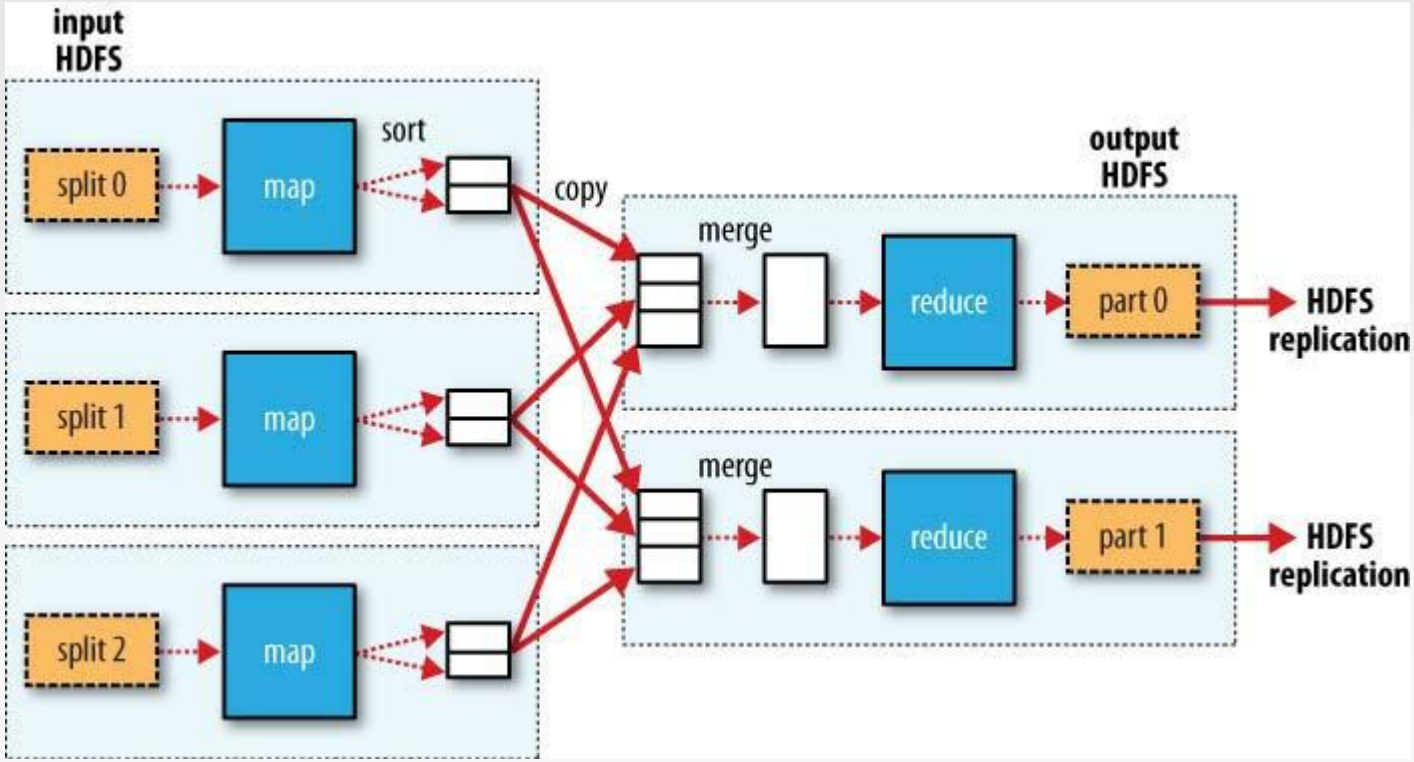
- Map tasks write their output to local disk (not to HDFS)
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - Storing it in HDFS with replication, would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on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

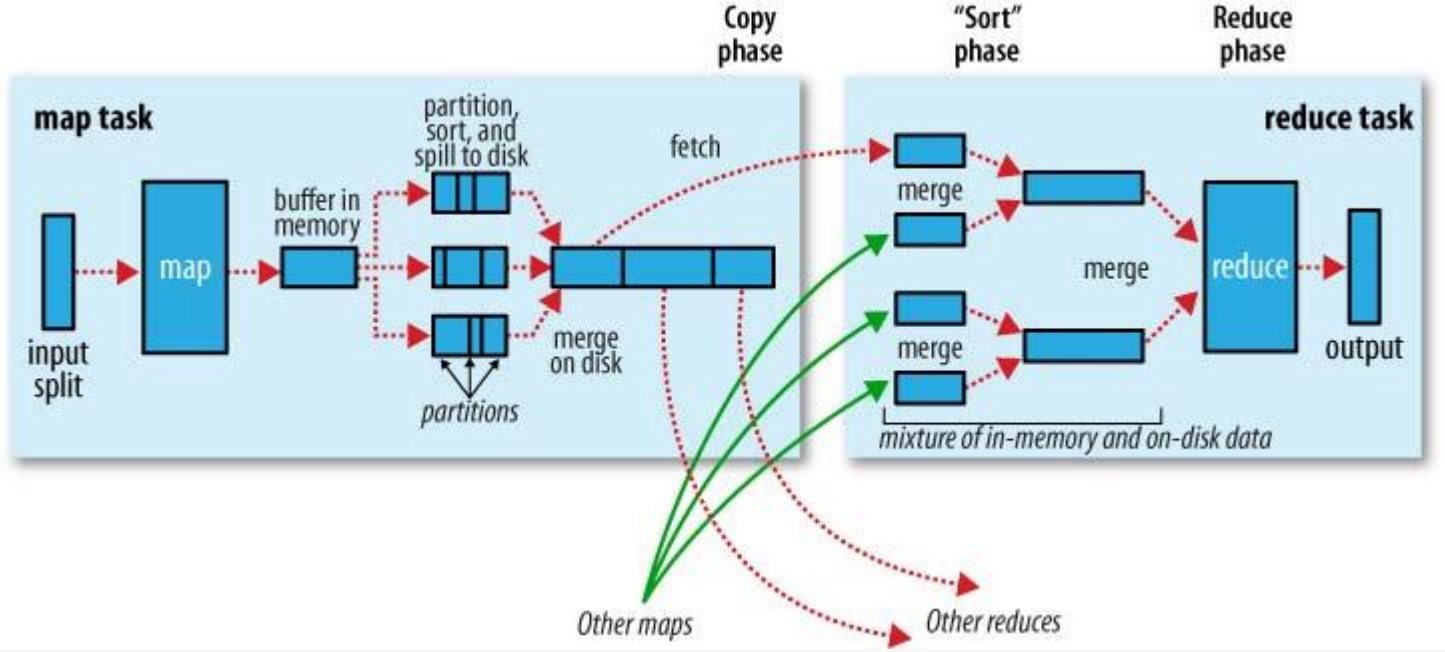
More Detailed MapReduce Dataflow

- 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



Partitioner

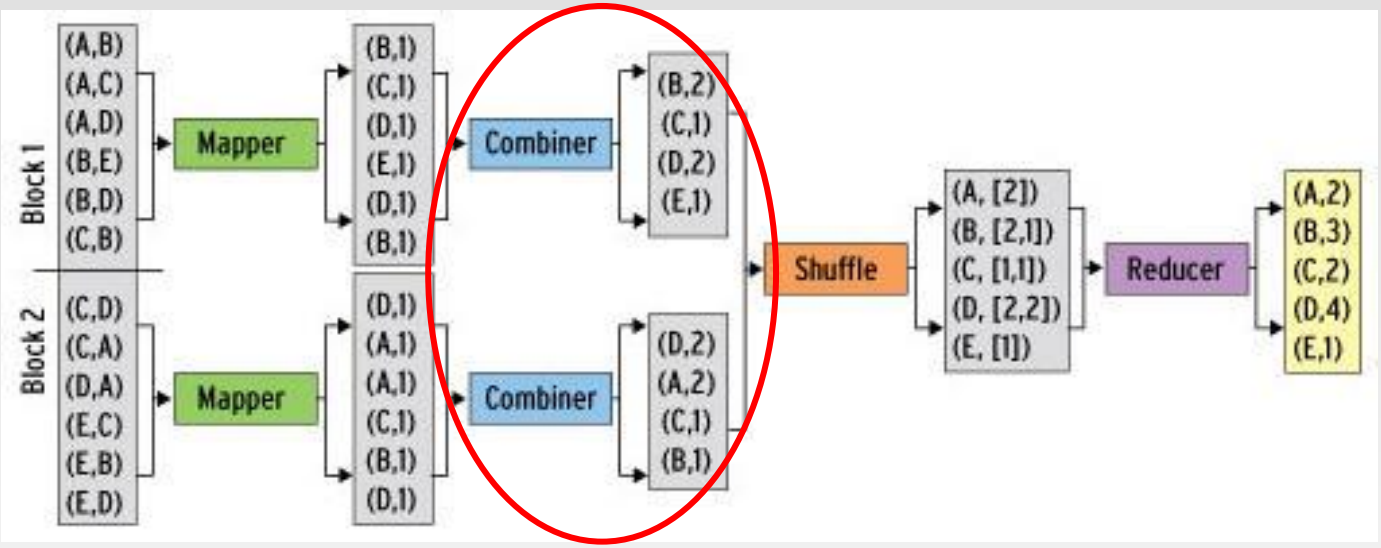
- Partitioner controls the partitioning of the keys of the intermediate map-outputs.
  - The key (or a subset of the key) is used to derive the partition, typically by a *hash function*.
  - The total number of partitions is the same as the number of reduce tasks for the job.
    - This controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.
- System uses HashPartitioner by default:
  - hash(key) mod R
- Sometimes useful to override the hash function:
  - E.g., ***hash(hostname(URL)) mod R*** ensures URLs from a host end up in the same output file
    - https://www.unsw.edu.au/faculties and https://www.unsw.edu.au/about-us will be stored in one file
- Job sets Partitioner implementation (in Main)

Combiners

- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key k
  - E.g., popular words in the word count example
- Combiners are a general mechanism to reduce the amount of intermediate data, thus saving network time
  - They could be thought of as “mini-reducers”
- Warning!
  - The use of combiners must be thought carefully
    - Optional in Hadoop: the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
    - A combiner operates on each map output key. It must have the same output key-value types as the Mapper.
    - A combiner can produce summary information from a large dataset because it replaces the original Map output

Combiners in WordCount

- Combiner combines the values of all keys of a single mapper node (single machine):



- Much less data needs to be copied and shuffled!
- If combiners take advantage of all opportunities for local aggregation we have at most m × V intermediate key-value pairs
  - m: number of mappers
  - V: number of unique terms in the collection
- Note: not all mappers will see all terms

Combiner and Reducer

- In general, reducer and combiner are not interchangeable
- Reducer and combiner are same only if the reduce function is commutative and associative
  - Commutative: a + b = b + a
  - Associative: a + b + c = (a + b) + c = a + (b + c)
  - Similarly, max, min, etc.
  - Therefore, in word count the reducer and the combiner has the same code
- How about the average computation?

key	value
panda	0
pink	3
pirate	3
panda	1
pink	4

key	value
panda	0.5
pink	3.5
pirate	3

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

Part 2: Introduction to MRJob

MRJob

- MRJob is the easiest route to writing Python programs that run on Hadoop. If you just need to run local MapReduce jobs, you even do not need to install Hadoop.
  - You can test your code locally without installing Hadoop
  - You can run it on a cluster of your choice.
  - MRJob has extensive integration with AWS EMR and Google Dataproc. Once you're set up, it's as easy to run your job in the cloud as it is to run it on your laptop.
- MRJob has a number of features that make writing MapReduce jobs easier. In MRJob, you can:
  - Keep all MapReduce code for one job in a single class.
  - Easily upload and install code and data dependencies at runtime.
  - Switch input and output formats with a single line of code.
  - Automatically download and parse error logs for Python tracebacks.
  - Put command line filters before or after your Python code.

MRJob WordCount

- Open a file called mr_word_count.py and type this into it:

```
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()
```

- Run the code locally: python mr_word_count.py inputText

How MRJob Works

- A job is defined by a class that inherits from MRJob. This class contains methods that define the steps of your job.
- A step consists of a mapper, a combiner and a reducer. All of these are optional, though you must have at least one. So you could have a step that's just a mapper, or just a combiner and a reducer.
- When you only have one step, all you have to do is write methods called mapper(), combiner() and reducer().
- The mapper() method takes a key and a value as args and yields as many key-value pairs as it likes.
- The reduce() method takes a key and an iterator of values, and also yields as many key-value pairs as it likes.
- The final required component of a job file is to include the following two lines at the end of the file, every time:

```
if __name__ == '__main__':
        MRWordCounter.run() # where MRWordCounter is your job class
```

  - These lines pass control over the command line arguments and execution to mrjob. Without them, your job will not work.

Run in Different Ways

- The most basic way to run your job is on the command line, using:
  - python my_job.py input.txt
  - By default, the output will be written to stdout.
- By default, MRJob will run your job in a single Python process. This provides the friendliest debugging experience, but it's not exactly distributed computing!
- You change the way the job is run with the -r/--runner option. You can use -r inline (the default), -r local, -r hadoop or -r emr.
  - To run your job in multiple subprocesses with a few Hadoop features simulated, use -r local (by default)
  - To run it on your Hadoop cluster, use -r hadoop
    - python my_job.py –r hadoop hdfs:///my_home/my_inputfile
    - python my_job.py –r hadoop hdfs:///my_home/my_inputfile –o hdfs:///my_home/my_outputfolder
  - If you have EMR/Dataproc configured, you can run it there with -r emr/dataproc.

Chained MapReduce Job (MRStep)

- To define multiple steps, override steps() to return a list of MRSteps:

```
class MRMostUsedWord(MRJob):

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)
        ]
```

Assessment 3B

- **Task: Find the average salary of the employees, to the nearest dollar.**
- It is not a good solution to create a fixed key and put everything into a single reducer.
- A better solution?
  - Round 1: use multiple reducers to compute the number of people and the total salary for each group.
    - If you have 10 reducers, it is like you divide the employees into 10 groups.
  - Round 2: Take the output of the first round as input, and use a single reducer to compute the final average.
  - Note that after you learned the combiner, you can also implement a combiner with 1 round map/reduce

Test and Debug MRJob on Hadoop

- Run you code locally, you can observe that the mapper output is not sorted. Since there is no sorting and shuffling and parititioning phases in this simulated environment.

- Use MRStep to define a step with only mapper to test your mapper first, and then include the reducer and run on Hadoop.

- Use sys.stderr to log the necessary information of your program

Check the Logs of Your Job

- Check the logs to see the error:
  - In Ed, after running a job, check the logs at /tmp/hadoop/logs/userlogs
  - In this directory, you can see a folder containing all the information about your job

```
[user@sahara /tmp/hadoop/logs/userlogs]$ ls -l
total 4
drwx--x--- 6 user user 4096 Jul 25 19:10 application_1658739349317_0001
```

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

```
[user@sahara /tmp/hadoop/logs/userlogs/application_1658740859997_0001]$ ls
container_1658740859997_0001_01_000001  container_1658740859997_0001_01_000003
container_1658740859997_0001_01_000002  container_1658740859997_0001_01_000004
```

Partitioner in MRJob (Optional)

- In your class, configure JOBCONF, like:

```
JOBCONF = {
            'map.output.key.field.separator': ',',
            'mapred.reduce.tasks':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 sorting 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”).

Sort Order of Keys in MRJob (Optional)

- Hadoop has a library class, KeyFieldBasedComparator, which can be used to specify the order of keys

```
JOBCONF = {
        'map.output.key.field.separator': ',',
        'mapred.reduce.tasks':2,
        'mapreduce.partition.keypartitioner.options':'-k1,1',
        'partitioner':'org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner'
        'mapreduce.job.output.key.comparator.class':'org.apache.hadoop.mapreduc
e.lib.partition.KeyFieldBasedComparator',
        'mapreduce.partition.keycomparator.options':'-k1,1 –k2,2nr'
}
```

  - You also need to add one line “SORT_VALUES = True” into your code.
  - Assume each key is a pair of a string and an integer separated by “,” like “term,102”. The above configure would let Hadoop know that the partitioning is only based the first field of the key (i.e., “term”), and sort the pair keys by the string first and then by the integer (the option “n” in “-k2,2nr”) in descending order (the option “r” in “-k2,2nr”) .

Assessment 4

- This assignment is due Monday 5pm in week 5.
- Part A: no combiner required, but only one mapper and one reducer are enough
- Part B: you need to implement a combiner, following the hints given today

End of Week 4