MapReduce programming

DAT470/DIT065 Computational techniques for large-scale data

2024-04-15

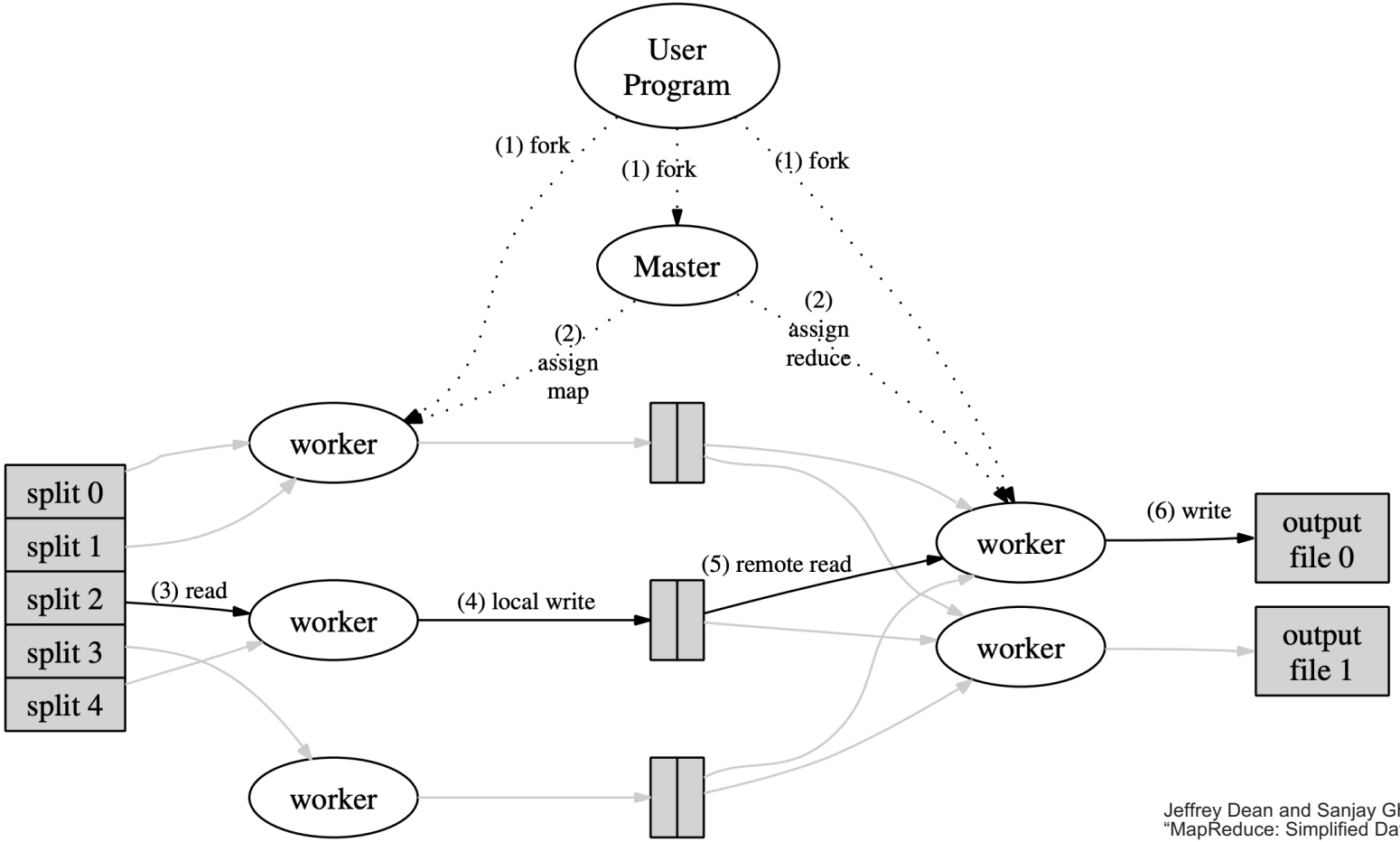
Matti Karppa

Motivation

- Computing on a large number of computers has challenges
  - Data has to be stored in a distributed fashion: no individual node can hold all data
  - Computing is **error-prone**: computers fail
  - **I/O bottleneck** presents a challenge: moving data from one compute node to another is very expensive
  - Coordinating computation is very difficult and laborious
- **MapReduce** is a solution that attempts to solve these some of these problems by assuming a (somewhat rigid) model of computation
  - Computation is **brought to data**: reduces the amount of I/O
  - Computation is modelled operations that adhere to a strict pattern, so they are easy to schedule
  - Likewise, if we detect a fault, we can simply replay those operations that failed

Map & Reduce

- In MapReduce programming, all data is treated as **key-value** pairs
- Computations are composed of two kinds of operations: **map** and **reduce** operations
- Map: $(K_1, V_1) \rightarrow \text{List}((K_2, V_2))$
  - Each key-value pair is mapped to a one or more key-value pairs
  - Maps are executed in parallel on different nodes (wherever the data lies)
  - Cf. Python's `map`
- After the map, the data is **shuffled** by hashing the keys
  - This (pseudorandomly) distributes the output to different nodes
  - Each key-value pair sharing the same key is guaranteed to end on the same node
- Reduce: $(K_2, \text{List}(V_2)) \rightarrow \text{List}((K_3, V_3))$
  - All values that were mapped to the same key are given as a list
  - **Typically**, reduce would then compute a single value from that, but can in practice return multiple key-value pairs
  - The function should be **commmutative** and **associative** because the order of values cannot be guaranteed
  - Cf. Python's `reduce`



Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters" (2004)

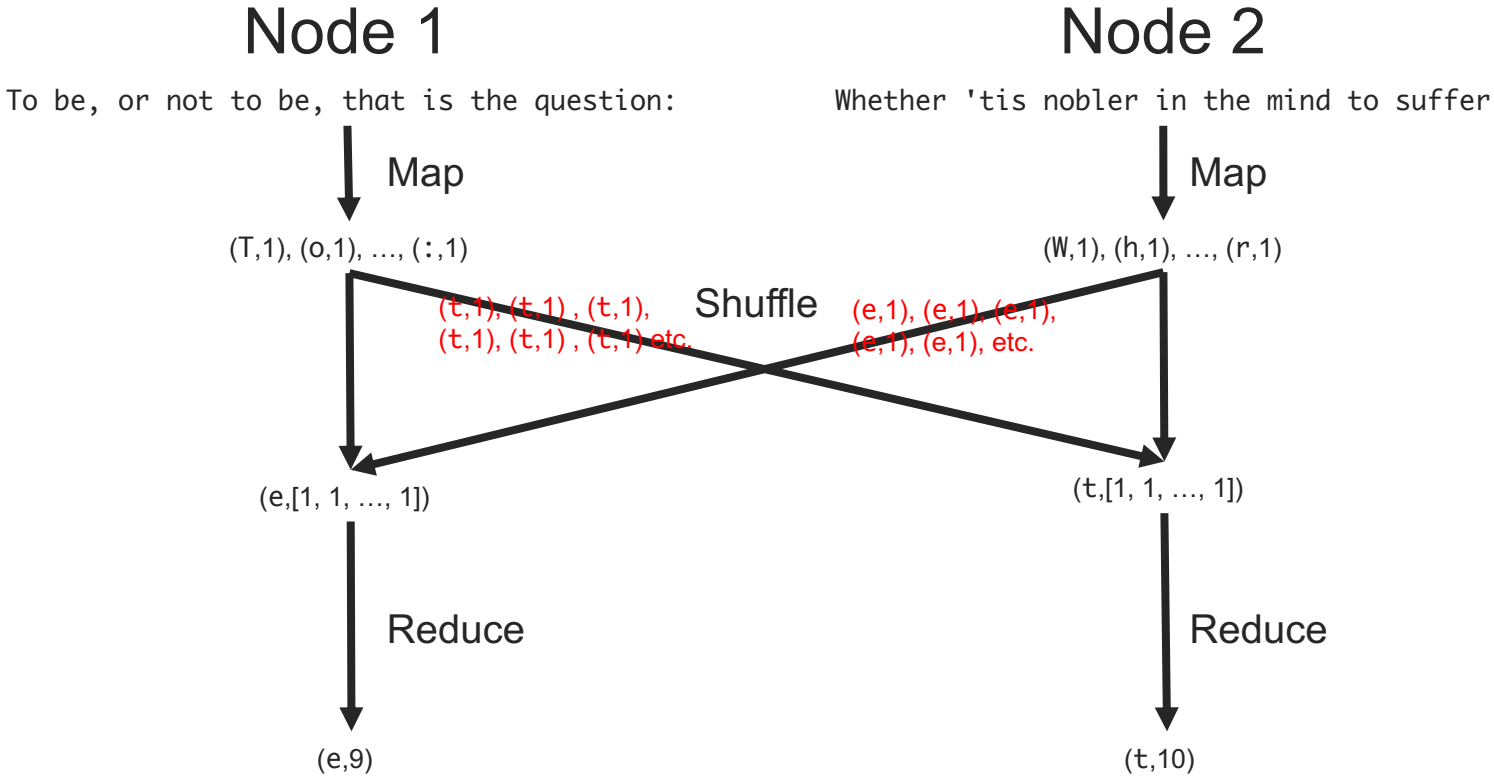
Example

- Let us count the frequency of each letter as in the previous lecture
- Data will be split by the lines; the input key is not significant (it can be a dummy key or line number or something of the sort)
- Map:
  - Key: *ignored*, value: line of text
  - For each character c in the line, emit (c,1)
  - "We saw c once"
- Data will be shuffled to different nodes; all 1s of the same character will end up on the same node
- Reduce:
  - Key: c, value: list $v_1, v_2, \ldots$ of 1s, corresponding to the count
  - Emit (c, $\sum v_i$)

Execution example

- Suppose node 1 holds the line `To be, or not to be, that is the question:` and node 2 holds the line `Whether 'tis nobler in the mind to suffer`
- The **mapper** of node 1 emits (T,1), (o,1), …, (:,1)
- The mapper of node 2 emits (W,1), (h,1), …, (r,1)
- Data is then shuffled
  - Suppose e ends up on node 1 and t on node 2
  - Five pairs (e,1) are communicated from node 2 to node 1
  - Six pairs (t,1) are communicated from node 1 to node 2
- For the key e, the reducer on node 1 takes in (e,[1,1,…,]), a list of 9 ones
- Reducer on node 1 emits (e,9)
- For the key t, the reducer on node 2 takes in (t,[1,1,…,]), a list of 10 ones
- Reducer on node 2 emits (t,10)

Execution example



UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- **Apache Hadoop** is a free software ecosystem for big data processing
- Hadoop MapReduce is the standard implementation of MapReduce (originally introduced by Google)
- The ecosystem contains other supporting components, such as Hadoop Distributed Filesystem (HDFS), a fault-tolerant distributed filesystem that we'll discuss on the next lecture

MRJob

- MRJob is a Python library for implementing MapReduce jobs
- Works like this: the user implements a subclass of job class and implements the required operations
- Jobs can be run locally (with the “local runner”, this is the approach we’ll adopt in our assignments), or through the Hadoop infrastructure
- Specialty: The Python file implementing the class **must not contain** anything else
  - This is because the job is sent to Hadoop for running and this sets limitations on, e.g., how the Python code may interact with its environment

Generator functions

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- The functions in MRJob are written as **generator functions**
- These functions are syntactic sugar for implementing **lazy functions** that behave as if **iterators**
- In Python, an object is **iterable** if it implements `__iter__` and `__next__`, that is, it allows one to create an **iterator** and (typically) go through the data structure one element at a time by advancing the iterator
  - All basic data structures in the Python standard library are iterable
- The functions `__iter__` and `__next__` are usually called via the convenience functions `iter` and `next`, respectively
  - Suppose `X` is some kind of collection
  - `it = iter(X)` returns an iterator to the collection
  - `next(it)` returns the next element in the collection and advances the iterator; if all elements have been returned, it raises the `StopIteration` exception instead
- The Python for `x in X` loop is essentially syntactic sugar for the following construct:
  - Create an iterator with `it = iter(X)`
  - Get the next element with `x = next(it)`
  - Catch the `StopIteration` exception and break from loop

Example

- These two are functionally the same

```
for x in X:
    print(x)
```

```
it = iter(X)
while True:
    try:
        x = next(it)
        print(x)
    except StopIteration:
        break
```

Generator functions

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- If a Python function contains a `yield` keyword, it is implicitly wrapped into an iterator object!
  - The next value returned by `__next__` is determined by `yield`
  - When the function ends executing, it raises `StopIteration`
  - This means that we can say `x in f()`
- Advantages:
  - From the caller's point of view, the function behaves like a collection even though the values do not exist at the moment of call
  - This enables us to lazily construct the values that the caller wants only upon need
  - This can save **a lot of** space and time if we do not need access to all values at one time, or if we abort early
- Disadvantages:
  - This introduces tacitly new stateful objects which we need to be aware of
  - This can be inefficient if used inconsiderately

Example: return the *n* first Fibonacci numbers

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

```
def fibonacci(n):
    x = 0
    y = 0
    z = 1
    for _ in range(n):
        x = y
        y = z
        z = x+y
        yield y

if __name__ == '__main__':
    for x in fibonacci(10):
        print(x)
```

Example: Letter count with MRJob

```
from mrjob.job import MRJob

class MRLettercount(MRJob):
    def mapper(self, _, line):
        for c in line:
            yield (c,1)

    def reducer(self, c, ones):
        yield (c,sum(ones))

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

This class defines a MapReduce job; we extend it

The map function is defined by overriding the `mapper` function

We ignore the key

Values are returned lazily; the function is actually a generator

The reducer works just the same

The default runner reads data from a file or stdin and provides standard command-line arguments; there must be **nothing** else in the .py file

Example: Letter count with MRJob

```
(mrjob) karppa@CM-DWQ77WNX67 lecture6 % cat lines.txt | tee /dev/tty | python3 mrjob_lettercount1.py
-r local --num-cores 4 2> /dev/null
To be, or not to be, that is the question:
Whether 'tis nobler in the mind to suffer
"q"     1
"r"     4
"s"     4
"t"     10
"e"     9
"f"     2
"h"     5
"i"     5
"l"     1
"m"     1
"n"     5
"o"     7
"'"     1
","     2
":"     1
"T"     1
"W"     1
"a"     1
"b"     3
"d"     1
"u"     2
" "     16
```

Protip: print the intermediate status on screen by writing to the terminal file

Use “local” runner (default is debug runner which does not scale)

Set the number of cores (workers)

Protip: MRJob writes a lot of stuff into stderr; this suppresses it for clearer output for demonstration purposes

Looks about right

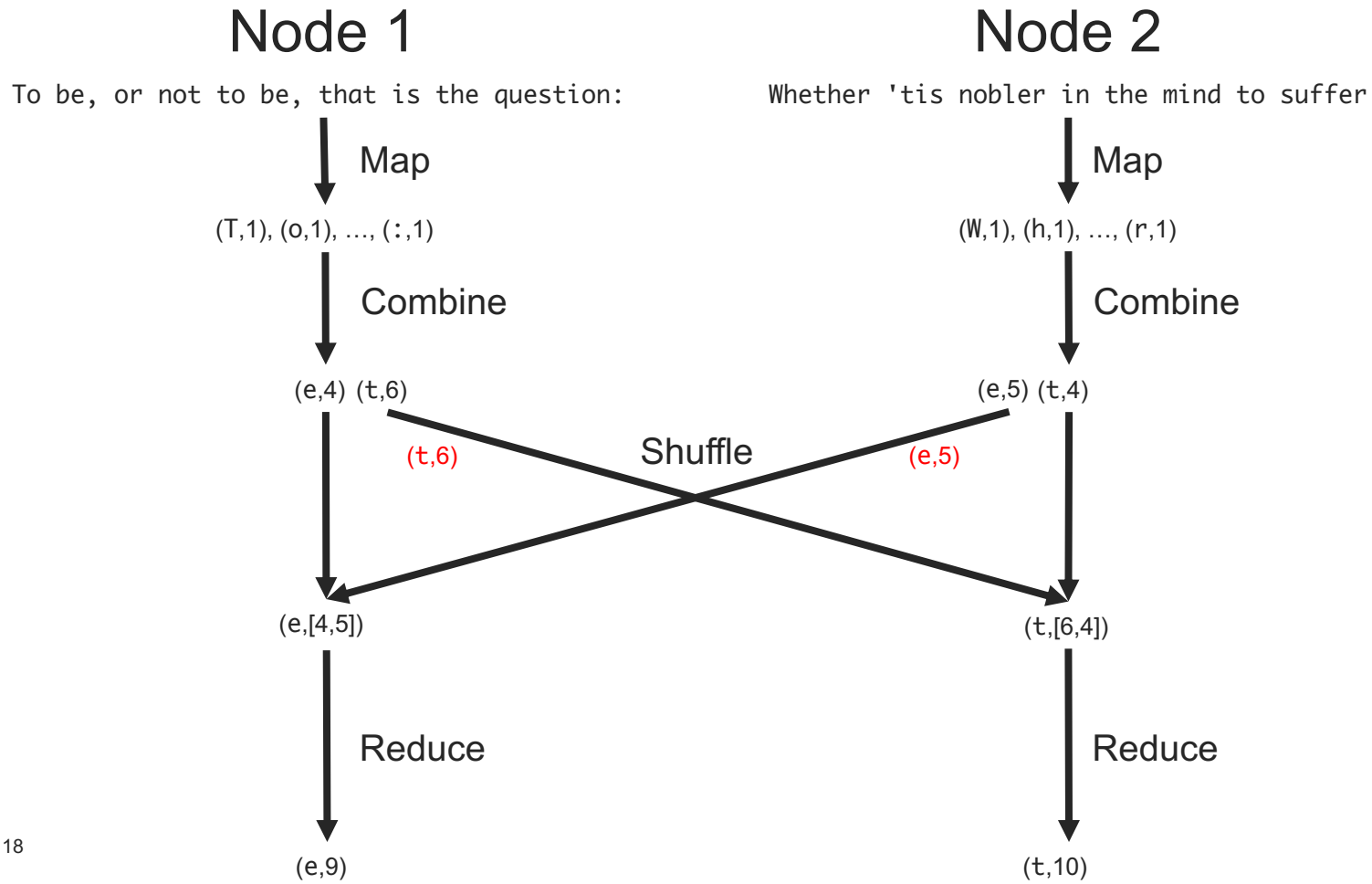
Performance considerations

- MapReduce only makes sense if the data is sufficiently large such that it cannot neatly fit in RAM
  - Then the framework solves the problem of coordinating work (although in a somewhat rigid way)
  - Computation is brought to the data, as compute is faster than I/O
  - If the data fits in RAM, then MapReduce only introduces unnecessary overhead
- The key to a performant MapReduce algorithm is to limit the amount of I/O
  - In practice: limit the amount of data that is communicated in the shuffling stage
- Storage of data matters
  - If the data is stored evenly distributed, we're good, but skewed distribution (e.g., all relevant pieces on one node) hurts parallelization

Combiners

- A problem with our example is that it can generate **a lot of** communication
  - Each occurrence of a letter is potentially communicated separately to another node
  - However, we only care about the **sum**: would it not be more efficient to just communicate the sum of counts on a node?
- **Combiners** solve this problem
- Combiners are essentially “mini-reducers” that are executed **locally** after the map operation but before shuffling
- Combiners perform a reduce operation locally, and only the result is then communicated
  - E.g., in this case, the combiner could simply count the sum of the occurrences locally

Execution example



Example

```
from mrjob.job import MRJob

class MRLettercount(MRJob):
    def mapper(self, _, line):
        for c in line:
            yield (c,1)

    def combiner(self, c, ones):
        yield (c,sum(ones))

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

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

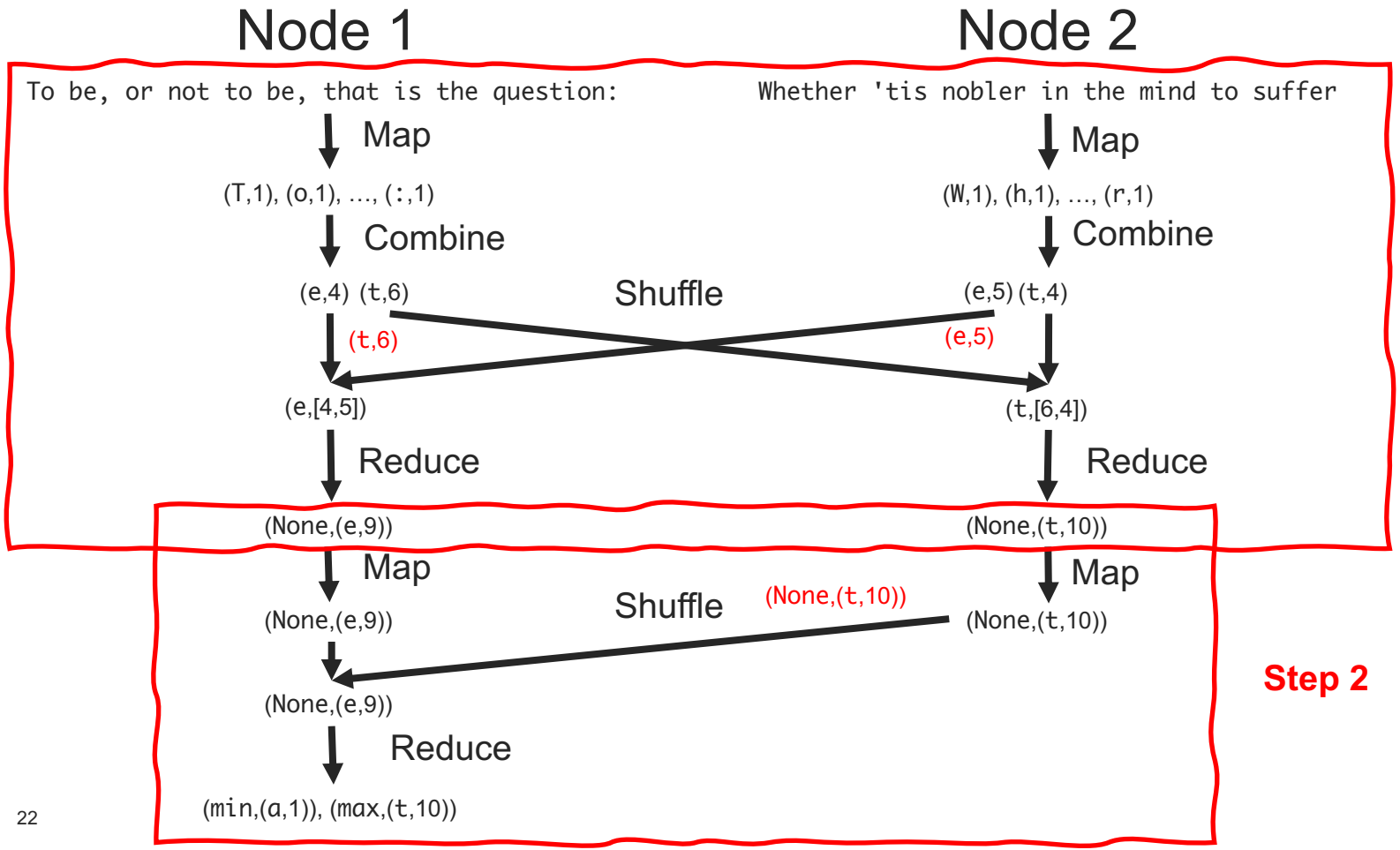
```
(mrjob) karppa@CM-DWQ77WNX67 lecture6 % cat lines.txt | tee /dev/tty | python3 mrjob_lettercount2.py
-r local --num-cores 4 2> /dev/null
To be, or not to be, that is the question:
Whether 'tis nobler in the mind to suffer
"o"     7
"q"     1
"r"     4
"b"     3
"d"     1
"e"     9
"f"     2
"h"     5
"i"     5
"l"     1
"m"     1
"n"     5
":"     1
"T"     1
"W"     1
"a"     1
"s"     4
"t"     10
"u"     2
" "     16
"'"     1
","     2
```

Multiple steps

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- What if we wanted to figure out the minimum, maximum, and average counts of characters?
- What if we wanted to figure out **which** characters have the minima and maxima counts?
- We can solve this by creating **multiple steps**
  - Each step is a MapReduce operation that is applied to the key-value pairs output by the previous step
- We change the first step such that we output the count of characters as **values**: (character,count)-tuples; we use a dummy key
- We don't really need a mapper in the second step as we only do another reduce step (mapper can be thought to be as the identity):
  - We traverse through the lists and identify the minima and maxima
- We return multiple key-value pairs to identify the identities and the values requested

Execution example



Steps in MRJob

- Multiple steps can be created by
  1. Implement all operations required as distinct member functions
  2. Implement the `steps()` function that returns a list of MRStep objects (each object defines the mapper, combiner, and reducer for the step)
  3. That's it!

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class MRLettercount(MRJob):
    def mapper(self, _, line):
        for c in line:
            yield (c,1)

    def combiner(self, c, ones):
        yield (c,sum(ones))

    def reducer(self, c, counts):
        yield (None, (c,sum(counts)))

    def reducer2(self, _, charcounts):
        minimum = (None, float('inf'))
        maximum = (None, -1)
        numchars = 0
        total = 0
        for (c,count) in charcounts:
            numchars += 1
            total += count
            if count < minimum[1]:
                minimum = (c,count)
            if count > maximum[1]:
                maximum = (c,count)
        yield ('min', minimum)
        yield ('max', maximum)
        yield ('avg', total / numchars)

    def steps(self):
        return [MRStep(mapper=self.mapper, combiner=self.combiner,
                       reducer=self.reducer),
                MRStep(reducer=self.reducer2)]
```

Example

```
(mrjob) karppa@CM-DWQ77WNX67 lecture6 % cat lines.txt | tee /dev/tty | python3 mrjob_lettercount3.py
-r local --num-cores 4 2>/dev/null
To be, or not to be, that is the question:
Whether 'tis nobler in the mind to suffer
"min"   ["q",1]
"max"   [" ",16]
"avg"   3.772727272727273
```

Command line parameters

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- MRJob provides a basic `argparse`-compatible command-line interface
- Suppose we want to make it optional to only include ASCII characters in our word count, how would we do this?
- We implement the member function `configure_args()` and add a **passthrough** argument
- The argument is then available through `self.options`

Example

```
class MRLettercount(MRJob):
    def configure_args(self):
        super(MRLettercount, self).configure_args()
        self.add_passthru_arg(
            '--ascii-only', default=False, action='store_true',
            help='Only apply to ASCII characters')

    def mapper(self, _, line):
        if self.options.ascii_only:
            for c in line:
                if ord(c) < 128:
                    yield (c,1)
        else:
            for c in line:
                yield (c,1)
```

Required so that ordinary options are added as usual (call superclass version)

This adds the actual argument; note that this is exactly the same as using `argparse`

Args dictionary is available through self.options; it is exactly like the output of `parse_args`

Running MRJob programmatically

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- It was already mentioned that MRJob class files must not contain any other code
- This is related to the way they are handled by Hadoop
- If you want to run MRJob jobs manually (e.g., to instrumentate the code to be able to measure runtimes!), you will need to create a custom runner
- Do as follows:
  1. Construct your object, use the constructor argument `args` to pass the command-line parameters as usual (this can include your custom parameters)
  2. Construct the runner object with the member function `make_runner()` (idiom: use the with construction to gracefully close the object when it's finished)
  3. Use the `run()` member function of the runner to actually run the task
  4. Recover output using the `parse_output()` and `cat_output()` functions

Example

Use local runner (do this on bayes as well)

Construct the job

Set the number of cores

```
from mrjob_lettercount4 import MRLettercount
import time
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-w', '--num-workers', type=int, default=1)
    parser.add_argument('filename')
    args = parser.parse_args()
    mr_job = MRLettercount(args=['-r', 'local', '--num-cores',
                                 str(args.num_workers), '--ascii-only',
                                 args.filename])
    start = time.time()
    with mr_job.make_runner() as runner:
        runner.run()
        for key, value in mr_job.parse_output(runner.cat_output()):
            print(key,value)
    end = time.time()
    print(f'Number of workers: {args.num_workers}')
    print(f'Time elapsed: {end-start} s')
```

Create runner, it will be closed graceully when it goes out of scope from the with block

Run the job

Custom arguments

Raw text output

Parse output into key-value pairs

Example

```
(mrjob) karppa@CM-DWQ77WNX67 lecture6 % python3 mrjob_lettercount_runner.py -w 4 ../lecture4/shakespe
are.txt
No configs specified for local runner
min ['\\', 1]
max [' ', 949869]
avg 61111.96511627907
Number of workers: 4
Time elapsed: 4.8429319858551025 s
```