MapReduce:
Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
jeff@google.com, sanjay@google.com

Google, Inc.

Professor: Manar Alsaid
Location: East Texas A&M University

Agenda

2. Programming Model Overview

1

Main Idea /Central Theme

Breaking computations into map and reduce functions simplifies writing distributed programs.

2

Problem Addressed

The dilemma of decomposing complex data processing tasks into simple, scalable operations and avoid getting bogged down in system details.

3

Methodology Used

Leverages functional programming which turns large-scale data operations into simply:

map(transformation)

reduce (aggregation)

4

Key Findings /Results

Even conceptually simple operations can run in parallel across many machines effectively bypassing the need for low-level programming.

5

Insights /Takeaways

Emphasizes clarity and simplicity in code design which allows developers to focus on computation and to let the system handle distribution.

3. Implementation Overview

1

Main Idea /Central Theme

The runtime system focuses on automatic task distribution and fault tolerance.

2

Problem Addressed

The challenges of running tasks on commodity hardware where failures are frequent and performance may vary.

3

Methodology Used

Data partitioning, task scheduling across nodes, and automatic re-assignment in the event of node failures.

4

Key Findings /Results

The system achieves high scalability and resilience with impressive performance on real-world data loads.

5

Insights /Takeaways

Encapsulating system complexities allows developers to write less error-prone code and design robust distributed applications.

2.3 More Examples (execute presentation with varying input/output)

	1	2	3	4	5
	2.3.1 Distributed Grep	2.3.3 Reverse Web-Link Graph	2.3.4 Term-Vector per Host	2.3.5 Inverted Index	2.3.6 Distributed Sort
k1	record identifier	source page URL	document identifier	document identifier	record identifier
v1	text line	list of out-links	document contents	document contents	record content
k2	file name/const key	out-link URL	host name parsed from the URL	word token	sort key (often record content)
v2	matching line	source page URL	partial term frequency pair (tuple)	document identifier or occurrence info	full record or placeholder

Mock MapReduce

Note: The following Jupyter notebook presents MapReduce in the Python language with English commentary being quoted from the article and embedded as comments.

Read the comments as you read and execute the presentation.

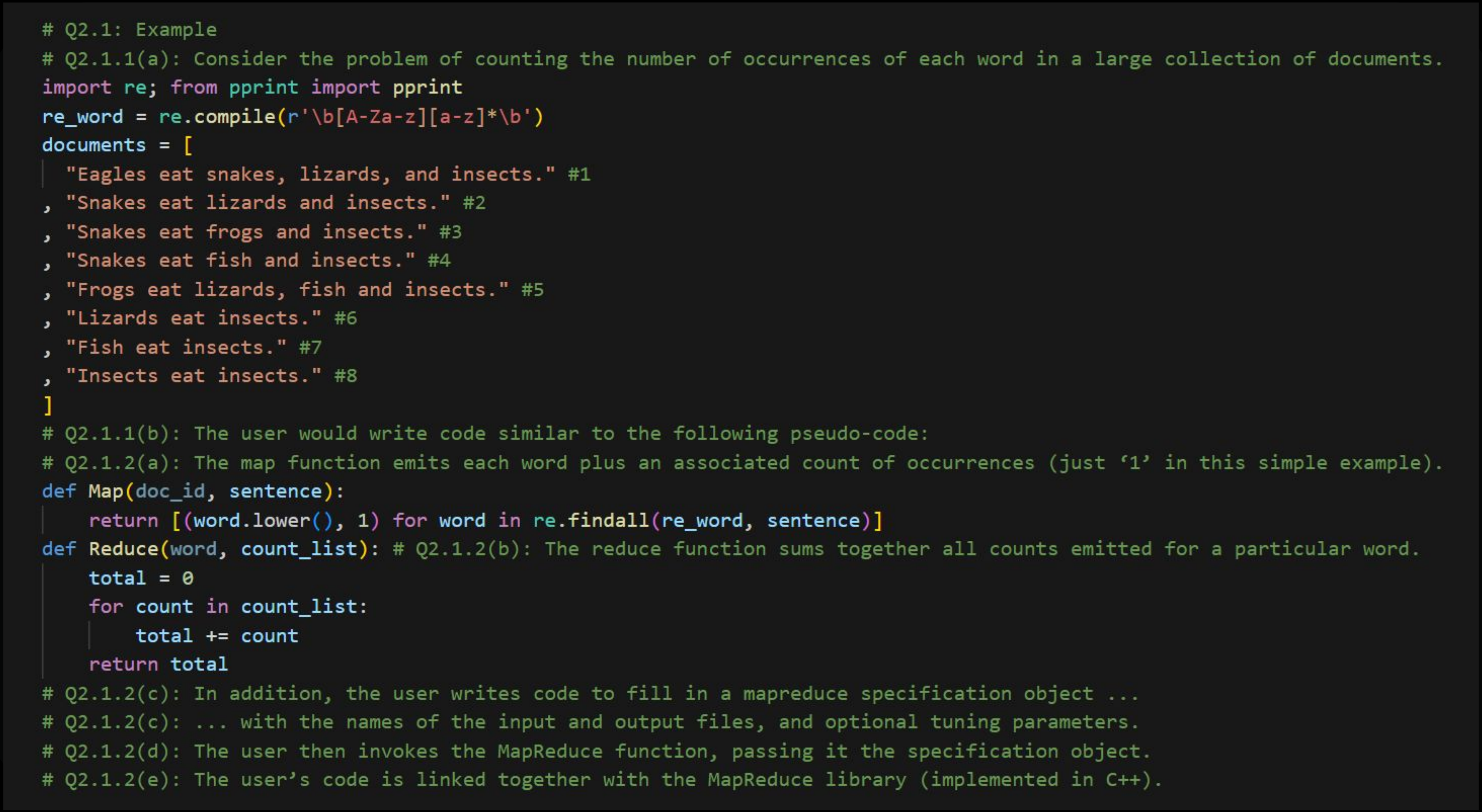
MapReduce: Simplified Data Processing on Large Clusters by Jeffrey Dean and Sanjay Ghemawat at Google, Inc.

Interactive Article - Mock simulation of MapReduce for interactive demonstration

- Section 2.1 Example
- Section 3.1 Execution Overview.

Future challenges for the student:

- Section 2.2: More Examples: use varying input/output, i.e. k1, v1, k2, and v2 types.
- Section 3.2: Master Data Structures: implement a basic scheduling component



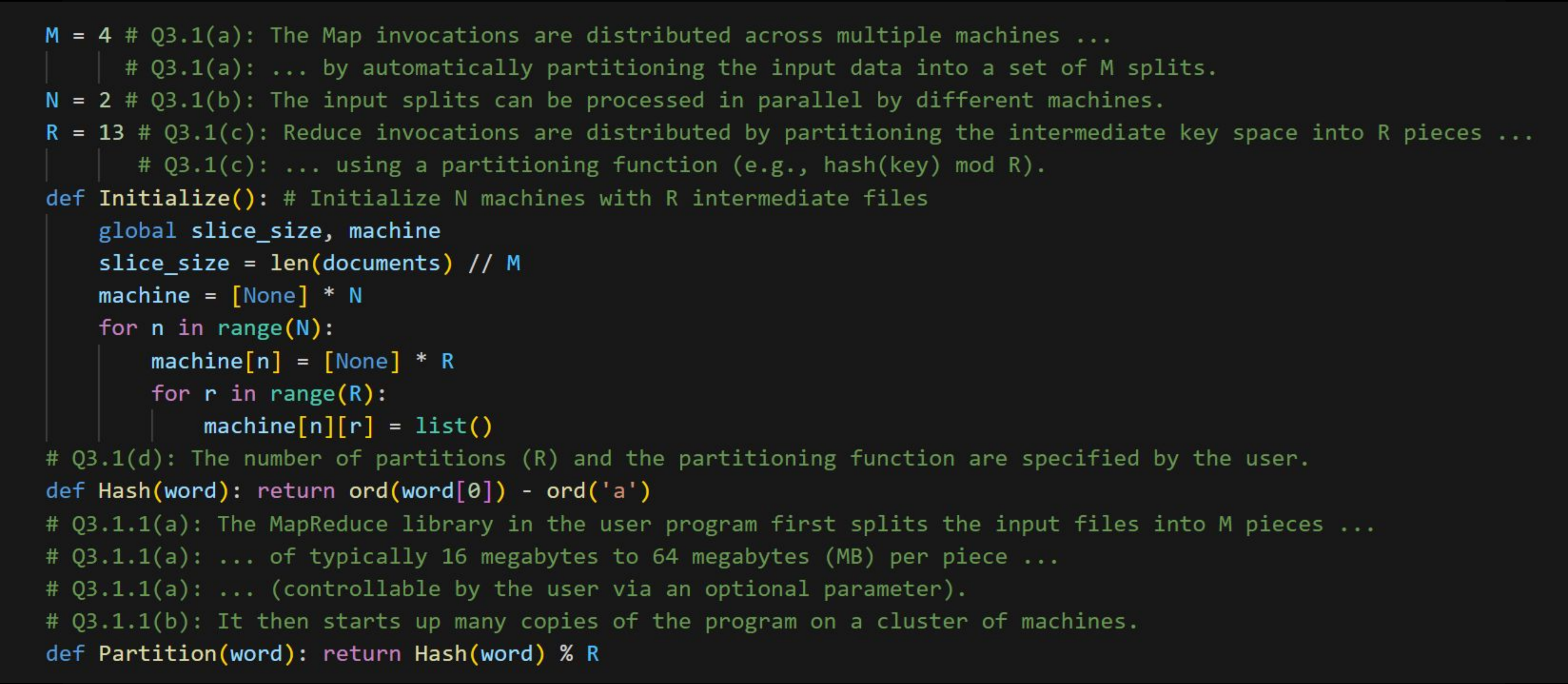
```python
# Q2.1: Example
# Q2.1.1(a): Consider the problem of counting the number of occurrences of each word in a large collection of documents.
import re; from pprint import pprint
re_word = re.compile(r'\b[A-Za-z][a-z]*\b')
documents = [
  "Eagles eat snakes, lizards, and insects." #1
, "Snakes eat lizards and insects." #2
, "Snakes eat frogs and insects." #3
, "Snakes eat fish and insects." #4
, "Frogs eat lizards, fish and insects." #5
, "Lizards eat insects." #6
, "Fish eat insects." #7
, "Insects eat insects." #8
]
# Q2.1.1(b): The user would write code similar to the following pseudo-code:
# Q2.1.2(a): The map function emits each word plus an associated count of occurrences (just ‘1’ in this simple example).
def Map(doc_id, sentence):
    return [(word.lower(), 1) for word in re.findall(re_word, sentence)]
def Reduce(word, count_list): # Q2.1.2(b): The reduce function sums together all counts emitted for a particular word.
    total = 0
    for count in count_list:
        total += count
    return total
# Q2.1.2(c): In addition, the user writes code to fill in a mapreduce specification object ...
# Q2.1.2(c): ... with the names of the input and output files, and optional tuning parameters.
# Q2.1.2(d): The user then invokes the MapReduce function, passing it the specification object.
# Q2.1.2(e): The user’s code is linked together with the MapReduce library (implemented in C++).
```

Guidelines for readability:

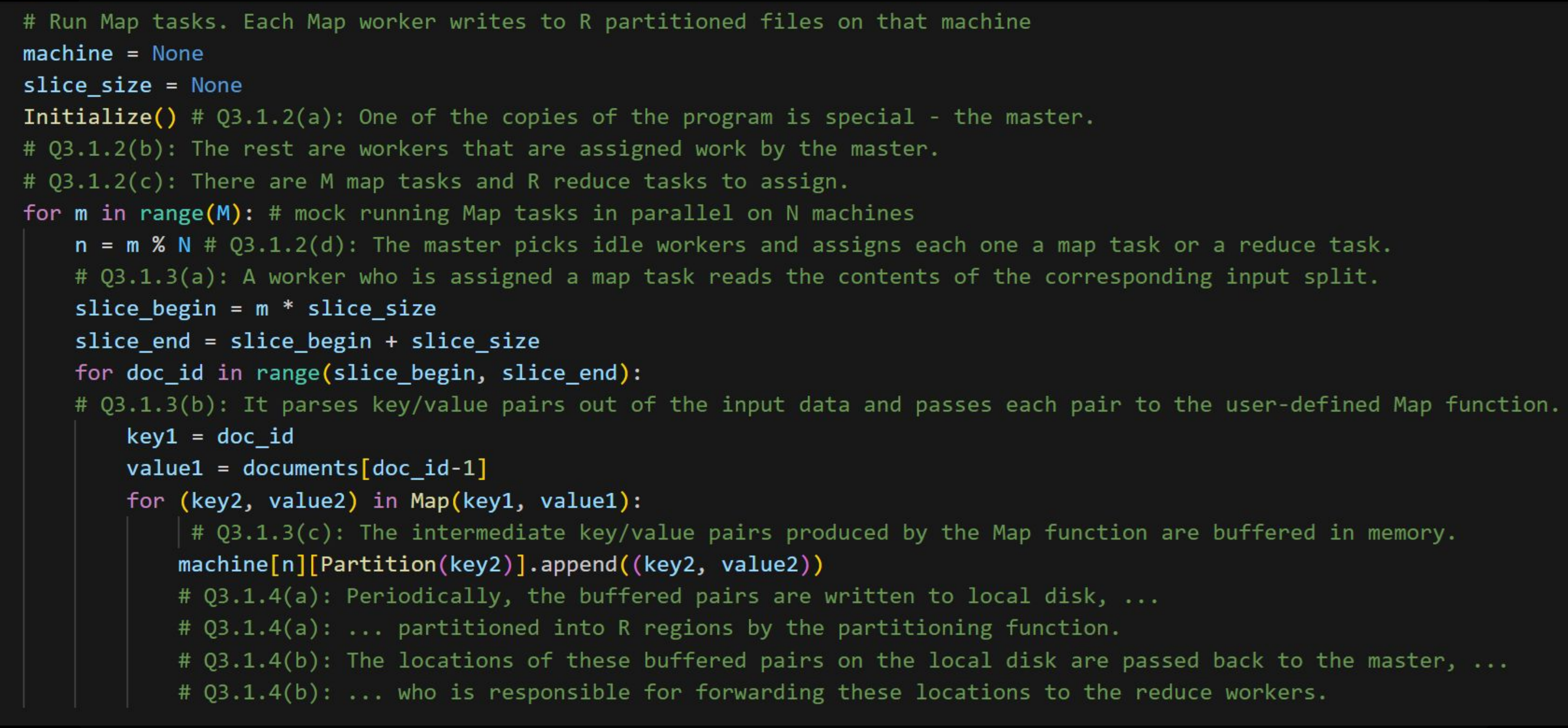
- **N, n**, **N**umber of machines
- **M, m**, **M**ap tasks
- **R, r**, **R**educe tasks
- **key1**, **value1** versus **k1, v1** in the article for the input domain
- **key2**, **value2** versus **k2, v2** in the article for the intermediate/output domain

Components provided by the user

- map(k1, v1) → list(k2, v2)
- reduce(k2, list(v2)) → aggregate(v2)
- partition(k2) → hash(k2) mod R



```python
M = 4 # Q3.1(a): The Map invocations are distributed across multiple machines ...
      # Q3.1(a): ... by automatically partitioning the input data into a set of M splits.
N = 2 # Q3.1(b): The input splits can be processed in parallel by different machines.
R = 13 # Q3.1(c): Reduce invocations are distributed by partitioning the intermediate key space into R pieces ...
       # Q3.1(c): ... using a partitioning function (e.g., hash(key) mod R).
def Initialize(): # Initialize N machines with R intermediate files
    global slice_size, machine
    slice_size = len(documents) // M
    machine = [None] * N
    for n in range(N):
        machine[n] = [None] * R
        for r in range(R):
            machine[n][r] = list()
# Q3.1(d): The number of partitions (R) and the partitioning function are specified by the user.
def Hash(word): return ord(word[0]) - ord('a')
# Q3.1.1(a): The MapReduce library in the user program first splits the input files into M pieces ...
# Q3.1.1(a): ... of typically 16 megabytes to 64 megabytes (MB) per piece ...
# Q3.1.1(a): ... (controllable by the user via an optional parameter).
# Q3.1.1(b): It then starts up many copies of the program on a cluster of machines.
def Partition(word): return Hash(word) % R
```

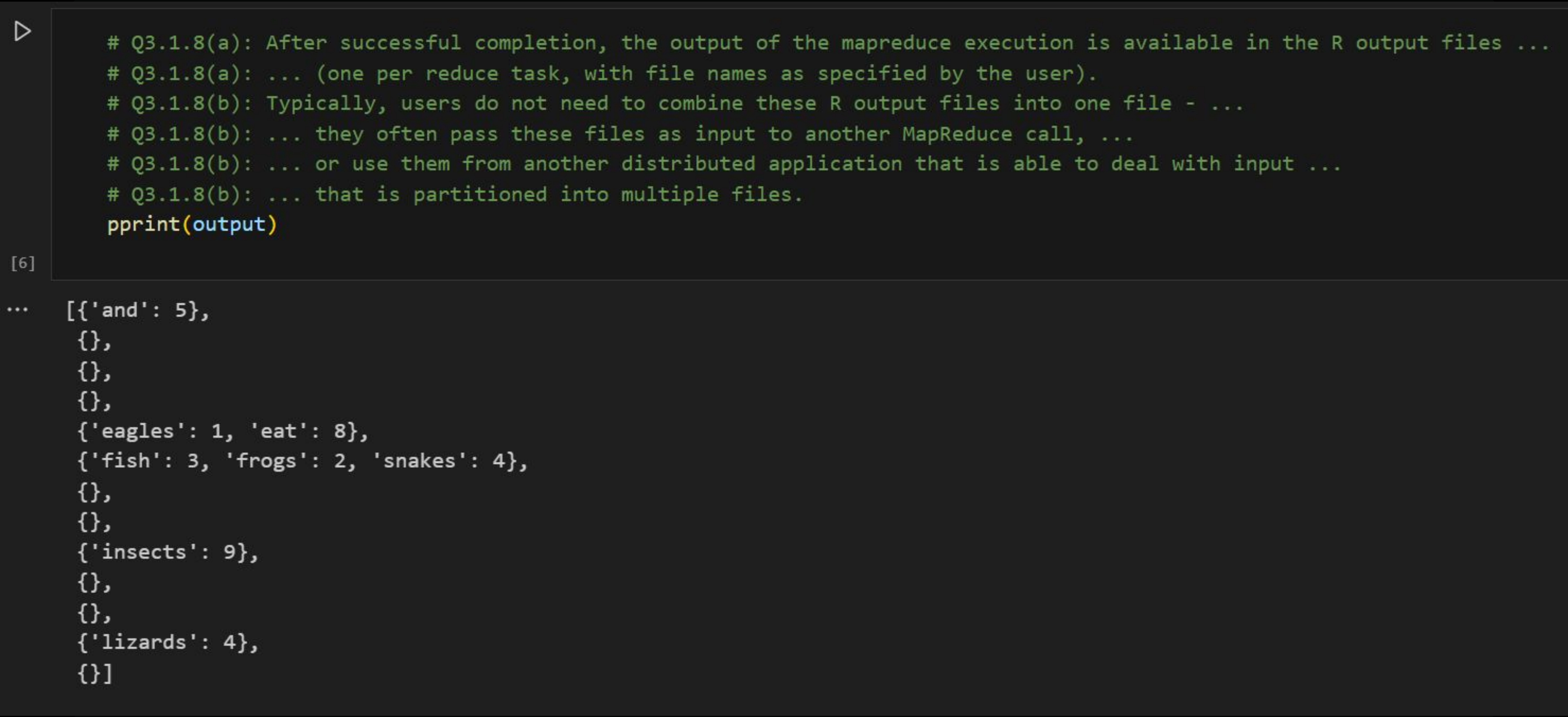


```
# Run Map tasks. Each Map worker writes to R partitioned files on that machine
machine = None
slice_size = None
Initialize() # Q3.1.2(a): One of the copies of the program is special - the master.
# Q3.1.2(b): The rest are workers that are assigned work by the master.
# Q3.1.2(c): There are M map tasks and R reduce tasks to assign.
for m in range(M): # mock running Map tasks in parallel on N machines
    n = m % N # Q3.1.2(d): The master picks idle workers and assigns each one a map task or a reduce task.
    # Q3.1.3(a): A worker who is assigned a map task reads the contents of the corresponding input split.
    slice_begin = m * slice_size
    slice_end = slice_begin + slice_size
    for doc_id in range(slice_begin, slice_end):
    # Q3.1.3(b): It parses key/value pairs out of the input data and passes each pair to the user-defined Map function.
        key1 = doc_id
        value1 = documents[doc_id-1]
        for (key2, value2) in Map(key1, value1):
            # Q3.1.3(c): The intermediate key/value pairs produced by the Map function are buffered in memory.
            machine[n][Partition(key2)].append((key2, value2))
            # Q3.1.4(a): Periodically, the buffered pairs are written to local disk, ...
            # Q3.1.4(a): ... partitioned into R regions by the partitioning function.
            # Q3.1.4(b): The locations of these buffered pairs on the local disk are passed back to the master, ...
            # Q3.1.4(b): ... who is responsible for forwarding these locations to the reduce workers.
```

```python
# Show intermediate file contents of the two machines
pprint(machine)
```

```
[[[('and', 1), ('and', 1), ('and', 1)],
  [],
  [],
  [],
  [('eat', 1), ('eagles', 1), ('eat', 1), ('eat', 1), ('eat', 1)],
  [('snakes', 1), ('snakes', 1), ('fish', 1), ('frogs', 1), ('fish', 1)],
  [],
  [],
  [('insects', 1),
   ('insects', 1),
   ('insects', 1),
   ('insects', 1),
   ('insects', 1)],
  [],
  [],
  [('lizards', 1), ('lizards', 1)],
  []],
 [[('and', 1), ('and', 1)],
  [],
  [],
  [],
  [('eat', 1), ('eat', 1), ('eat', 1), ('eat', 1)],
  [('snakes', 1), ('snakes', 1), ('frogs', 1), ('fish', 1)],
  [],
  [],
  [('insects', 1), ('insects', 1), ('insects', 1), ('insects', 1)],
  [],
  [],
  [('lizards', 1), ('lizards', 1)],
  []]]
```

```python
# Run Reduce tasks.
output = [dict() for r in range(R)]
for r in range(R): # mock executing Reduce tasks in parallel on N machines
    memory = list() # mock gathering data in local memory
    this_machine = r % N # mock scheduling task on selected machine
    for n in range(N): # mock for all reported ready partitions
        # Q3.1.5(a): When a reduce worker is notified by the master about these locations, ...
        # Q3.1.5(a): ... it uses remote procedure calls to read the buffered data ...
        # Q3.1.5(a): ... from the local disks of the map workers.
        if n != this_machine:
            partition = (globals()['machine'][n])[r] # mock remote machine access
        else: partition = machine[n][r] # mock local machine access
        for (key2, value2) in partition:
            memory.append((key2, value2))
    # Q3.1.5(b): When a reduce worker has read all intermediate data, ...
    # Q3.1.5(b): ... it sorts it by the intermediate keys ...
    # Q3.1.5(b): ... so that all occurrences of the same key are grouped together.
    sorted_memory = dict() # mock sorting local memory
    for (key2, value2) in memory:
        if key2 not in sorted_memory:
            sorted_memory[key2] = [value2]
        else: sorted_memory[key2].append(value2)
    # Q3.1.6(a): The reduce worker iterates over the sorted intermediate data ...
    # Q3.1.6(a): ... and for each unique intermediate key encountered, ...
    # Q3.1.6(a): ... it passes the key and the corresponding set of intermediate values to the user’s Reduce function.
    # Q3.1.6(b): The output of the Reduce function is appended to a final output file for this reduce partition.
    for (key2, list_value2) in sorted_memory.items():
        output[r][key2] = Reduce(key2, list_value2) # notice the list passed to Reduce has only values (no keys)
    # Q3.1.5(c): The sorting is needed because typically many different keys map to the same reduce task.
    # Q3.1.5(d): If the amount of intermediate data is too large to fit in memory, an external sort is used.
# Q3.1.7(a): When all map tasks and reduce tasks have been completed, the master wakes up the user program.
# Q3.1.7(b): At this point, the MapReduce call in the user program returns back to the user code.
```



```
# Q3.1.8(a): After successful completion, the output of the mapreduce execution is available in the R output files ...
# Q3.1.8(a): ... (one per reduce task, with file names as specified by the user).
# Q3.1.8(b): Typically, users do not need to combine these R output files into one file - ...
# Q3.1.8(b): ... they often pass these files as input to another MapReduce call, ...
# Q3.1.8(b): ... or use them from another distributed application that is able to deal with input ...
# Q3.1.8(b): ... that is partitioned into multiple files.
pprint(output)
```

[6]

```
[{'and': 5},
 {},
 {},
 {},
 {'eagles': 1, 'eat': 8},
 {'fish': 3, 'frogs': 2, 'snakes': 4},
 {},
 {},
 {'insects': 9},
 {},
 {},
 {'lizards': 4},
 {}]
```

The End

Questions?
Please reach out to
lcherryh@yahoo.com

Thank you!