Module 3: Live Session Slides a brief recap of key ideas

James G. Shanahan

- Hadoop streaming
  - https://hadoop.apache.org/docs/r1.2.1/streaming.html
  - https://hadoop.apache.org/docs/r1.2.1/streaming.html#:~:text=Hadoop%20streaming%20is%20a%20utility,mapper%20and%2For%20the%20reducer.

sort

Back to top | Back to Section I

(Source: http://www.theunixschool.com/2012/08/linux-sort-command-examples.html)

```
%%writefile unix-sort-example.txt
Unix,30
Solaris,10
Linux,25
Linux,20
HPUX,100
AIX,25
```

```
Overwriting unix-sort-example.txt
```

sort -t"," -k1,1 sort -t"," -k2,2nr sort -t"," -k1,1 -k2,2nr

Sort by field 1 (default alphabetically), deliminator ","

Input

Output

```
cat unix-sort-example.txt
```

```
Unix,30
Solaris,10
Linux,25
Linux,20
HPUX,100
AIX,25
```

```
sort -t"," -k1,1 unix-sort-example.txt
```

```
AIX,25
HPUX,100
Linux,20
Linux,25
Solaris,10
Unix,30
```

Sort by field 2 numerically reverse, deliminator ","

```
cat unix-sort-example.txt
```

```
Unix,30
Solaris,10
Linux,25
Linux,20
HPUX,100
AIX,25
```

```
sort -t"," -k2,2nr  unix-sort-example.txt
```

```
HPUX,100
Unix,30
AIX,25
Linux,25
Linux,20
Solaris,10
```

Sort by field 1 alphabetically first, then by field 2 numeric reverse

```
cat unix-sort-example.txt
```

```
Unix,30
Solaris,10
Linux,25
Linux,20
HPUX,100
AIX,25
```

```
sort -t"," -k1,1 -k2,2nr  unix-sort-example.txt
```

```
AIX,25
HPUX,100
Linux,25
Linux,20
Solaris,10
Unix,30
```

See: HelpfulResources/TotalSortGuide/total-sort-guide-hadoop-streaming.ipynb

Order inversion pattern

[56]:
```
# part b - make sure scripts are executable (RUN THIS CELL AS IS)
!chmod a+x Frequencies/mapper.py
!chmod a+x Frequencies/combiner.py
!chmod a+x Frequencies/reducer.py
```

[57]:
```
# part b - unit test mapper script
!echo "foo foo quux labs foo bar quux" | Frequencies/mapper.py
```

```
foo     1
!total  1
foo     1
!total  1
quux    1
!total  1
labs    1
!total  1
foo     1
!total  1
bar     1
!total  1
quux    1
!total  1
```

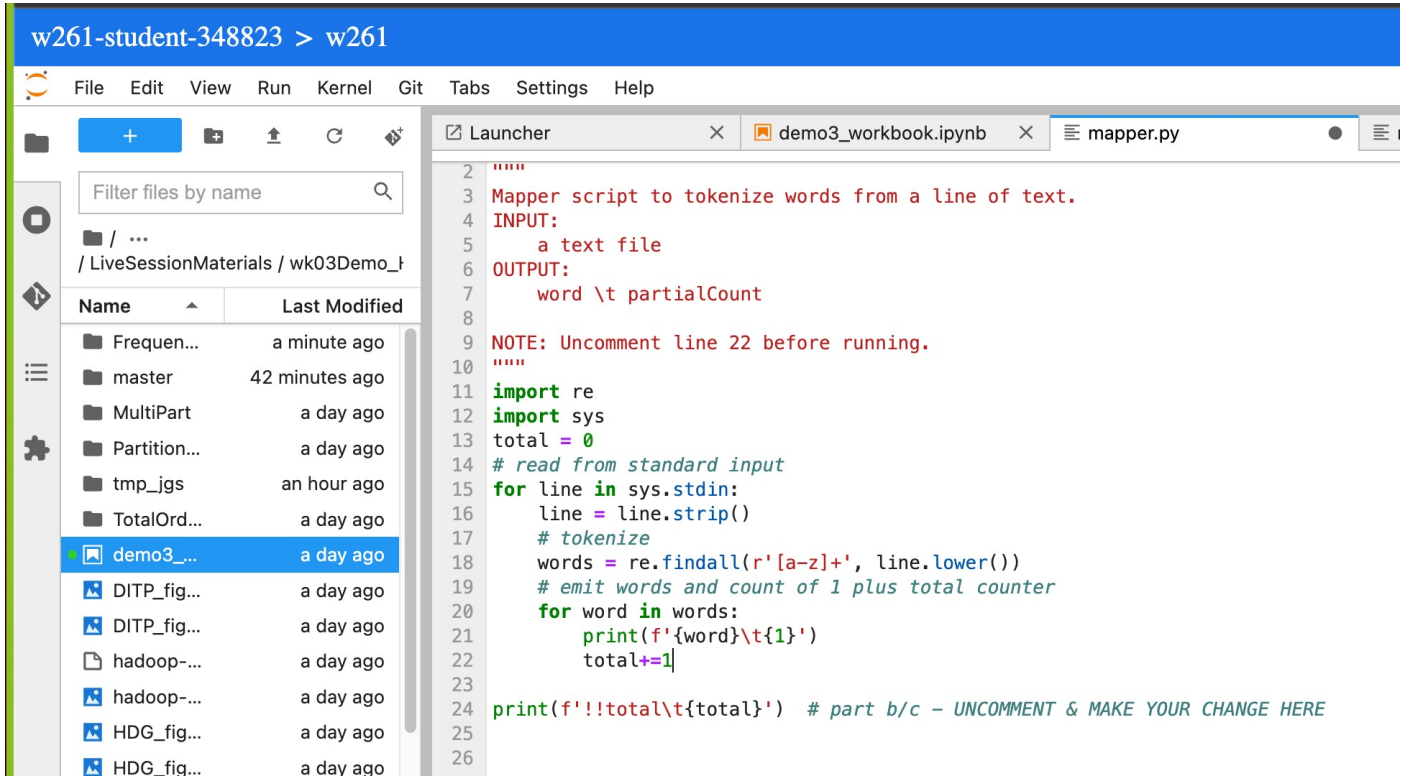
[58]:
```
# part b - unit test map-combine (sort mimics shuffle) (RUN THIS CELL AS IS)
!echo "foo foo quux labs foo bar quux" | Frequencies/mapper.py | sort -k1,1 | Frequencies/combiner.py
```

```
!total  7
bar     1
foo     3
labs    1
quux    2
```

[59]:
```
# part b - unit test map-combine-reduce (sort mimics shuffle) (RUN THIS CELL AS IS)
!echo "foo foo quux labs foo bar quux" | Frequencies/mapper.py | sort -k1,1 | Frequencies/combiner.py | Frequencies/reducer.py
```

```
!total  1.0
bar     0.14285714285714285
foo     0.42857142857142855
labs    0.14285714285714285
quux    0.2857142857142857
```

Make mapper more efficient with a small state



Prefix a partition key in the mapper

- Use Hadoop partition key option to partition the data
- Sort using all components of the compound key

```
Mapper partitions based on first letter in word.
INPUT:
    word \t count
OUTPUT:
    partitionKey \t word \t count
"""
import re
import sys

def getPartitionKey(word,count):
    """
    Helper function to assign partition key ('A', 'B', or 'C').
    Args:  word (str) ; count (int)
    """
    ############ YOUR CODE HERE ##########
    if count > 8:                           # <--- SOLUTION --->
        return  'B'                         # <--- SOLUTION --->
    elif count > 4:                         # <--- SOLUTION --->
        return  'C'                         # <--- SOLUTION --->
    else:                                   # <--- SOLUTION --->
        return 'A'                          # <--- SOLUTION --->

    # provided implementation: (run this first, then make your changes in part e)
     if word[0] < 'h':
        return  'A'
    elif word[0] < 'p':
        return  'B'
    else:
        return 'C'
    ############ (END) YOUR CODE #########

# read from standard input
for line in sys.stdin:
    word, count = line.strip().split()
    count = int(count)
    partitionKey = getPartitionKey(word, count)
    print(f"{partitionKey}\t{word}\t{count}")
```

```
[40]:
# <--- SOLUTION --->
# part a - Hadoop streaming command
!hdfs dfs -rm -r {HDFS_DIR}/psort-output
!hadoop jar {JAR_FILE} \
  -D stream.num.map.output.key.fields=3 \
  -D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapred.lib.KeyFieldBasedComparator \
  -D mapreduce.partition.keycomparator.options="-k3,3nr" \
  -D mapreduce.partition.keypartitioner.options="-k1,1"  \
  -files PartitionSort/mapper.py \
  -mapper mapper.py \
  -reducer /bin/cat \
  -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \
  -input {HDFS_DIR}/sample.txt \
  -output {HDFS_DIR}/psort-output \
  -cmdenv PATH={PATH}\
  -numReduceTasks 3
```

- partitionID, word, frequency

Sort and partitions

3.3.1. Partial Sort

file: part-00000

27	driver
27	creating
27	experiements
19	consists
19	evaluate
17	drivers
10	clustering
9	during
9	change
7	contour

file: part-00001

30	do
28	dataset
15	computing
15	document
15	computational
14	center
5	distributed
4	develop
3	different
2	cluster

file: part-00002

26	descent
26	def
25	compute
24	done
24	code
23	descent
22	corresponding
13	efficient
1	cell
0	current

Keys are assigned to buckets without any ordering. Keys are sorted within each bucket (the key is the the number in the first column rendered in red).

3.3.2. Total Sort (Unorderd Partitions)

file: part-00000

19	consists
19	evaluate
17	drivers
15	computing
15	document
15	computational
14	center
13	efficient

file: part-00001

10	clustering
9	during
9	change
7	contour
5	distributed
4	develop
3	different
2	cluster
1	cell
0	current

file: part-00002

30	do
28	dataset
27	driver
27	creating
27	experiements
26	descent
26	def
25	compute
24	done
24	code
23	descent
22	corresponding

Keys are assigned to buckets according to their numeric value. The result is that all keys between 20-30 end up in one bucket, keys between 10-20 end up in another bucket, and keys 0-10 end up in another bucket. Keys are sorted within each bucket. Partitions are not assigned in sorted order.

3.3.3. Total Sort (Ordered Partitions)

file: part-00000

30	do
28	dataset
27	creating
27	driver
27	experiements
26	def
26	descent
23	descent
22	corresponding

file: part-00001

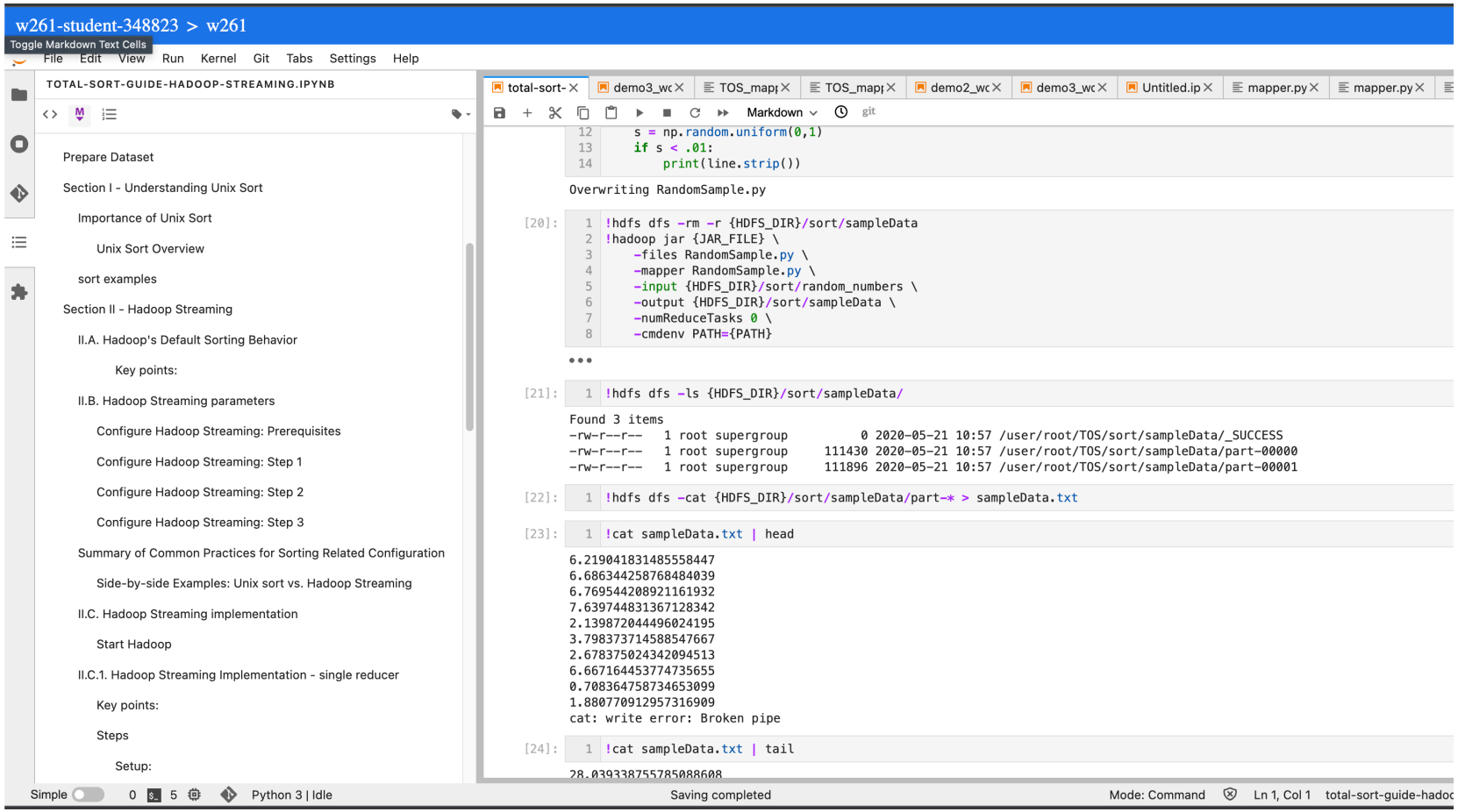
19	evaluate
19	consists
17	drivers
15	document
15	computing
15	computational
14	center
13	efficient
10	clustering

file: part-00002

9	during
9	change
7	contour
5	distributed
4	develop
3	different
2	cluster
1	cell
0	current

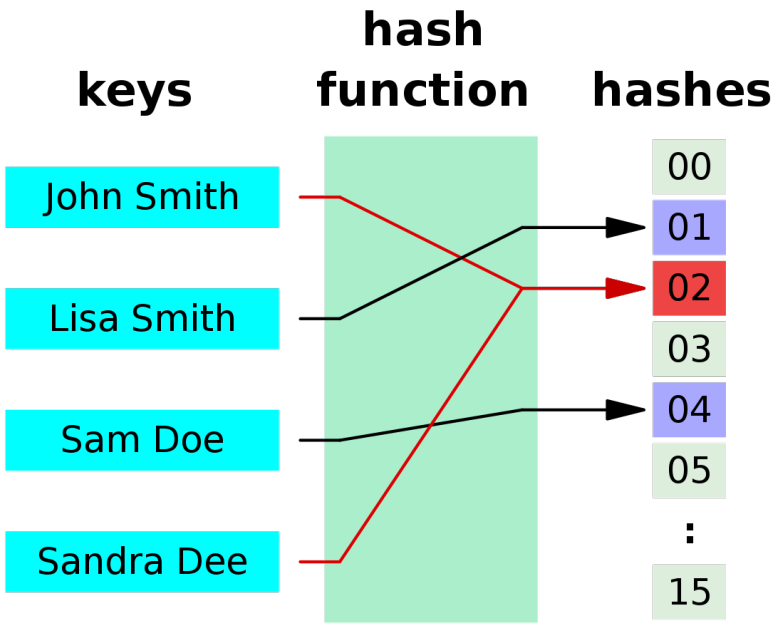
See: HelpfulResources/TotalSortGuide/total-sort-guide-hadoop-streaming.ipynb

Live session 3: part total order sort notebook



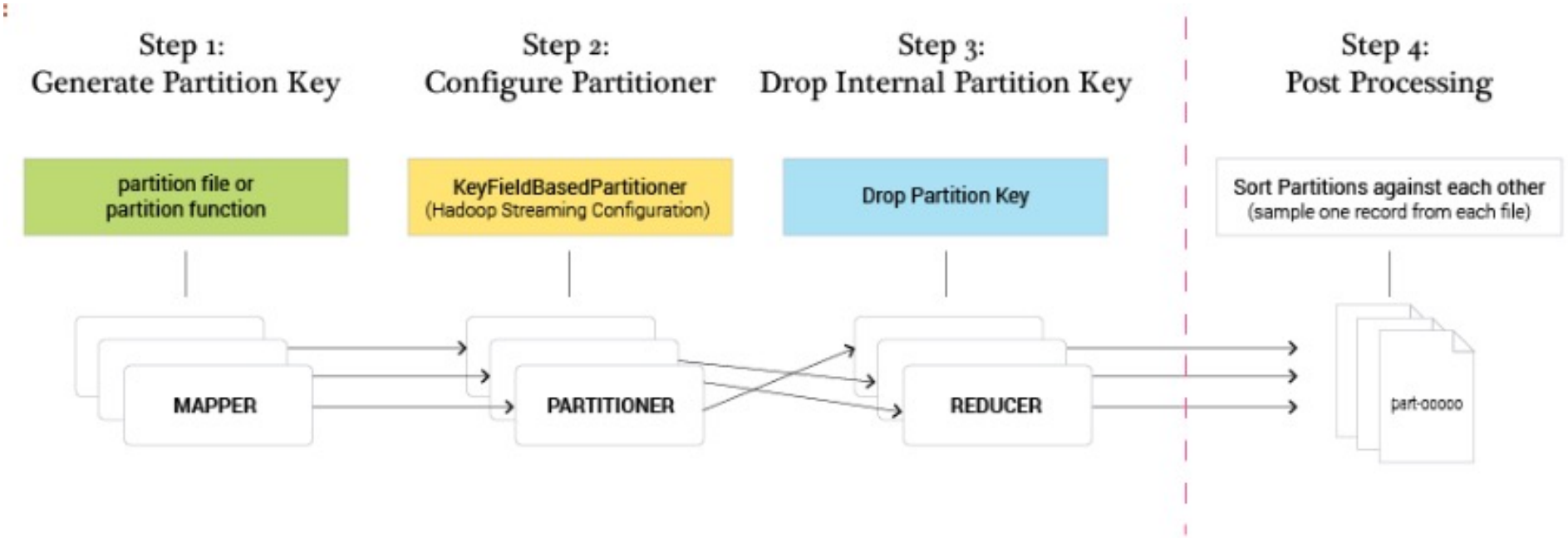
hash_function(key) → hash

- A **hash function** is any function that can be used to map data of arbitrary size to fixed-size values.
- The values returned by a hash function are called *hash values*, *hash codes*, *digests*, or simply *hashes*. The values are usually used to index a fixed-size table called a *hash table*. Use of a hash function to index a hash table is called *hashing* or *scatter storage addressing*.



Total order sort in Hadoop

See: **HelpfulResources/TotalSortGuide/total-sort-guide-hadoop-streaming.ipynb**



```
def partition_function(word):
    assert len(word) > 0
    return word[0]
```

It is important to note that a partition function must preserve sort order, i.e. all partitions need to be sorted against each other. For instance, the following pa

```
def partition_function(word):
    assert len(word) > 0
    return word[-1]
```

The mapper output or the four words with this partition scheme is:

```
e    experiements
d    def
d    descent
c    compute
```

The following diagram outlines the flow of data with this example:

```
Image(filename="Partition2.png")
```

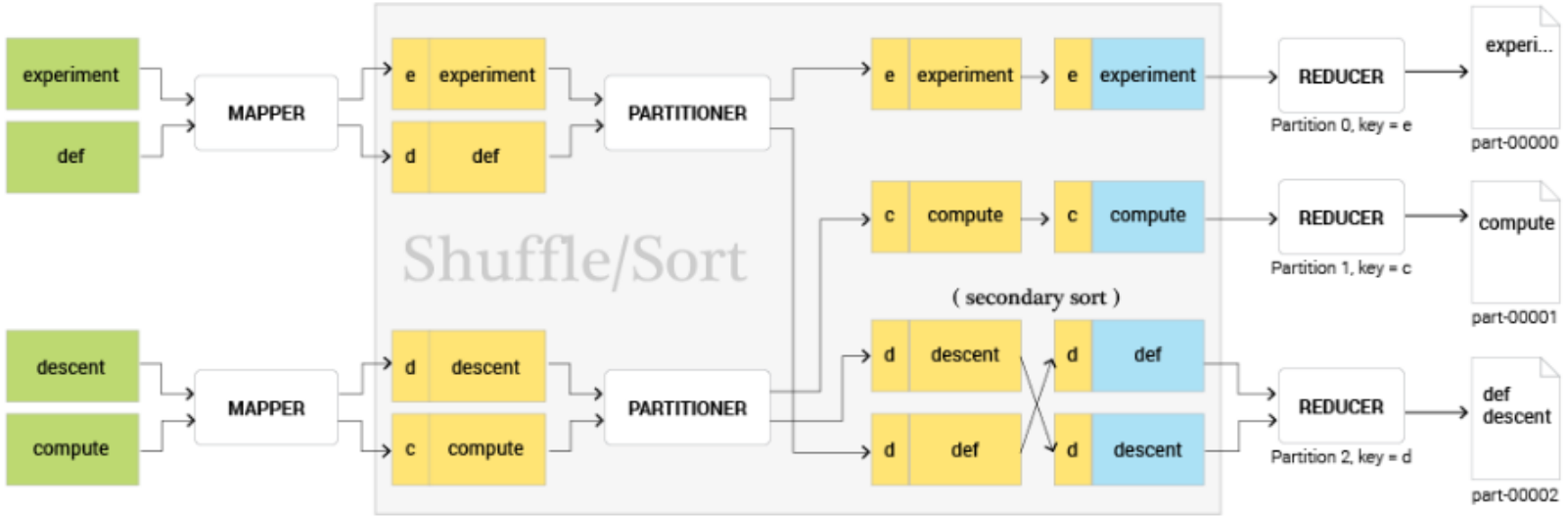
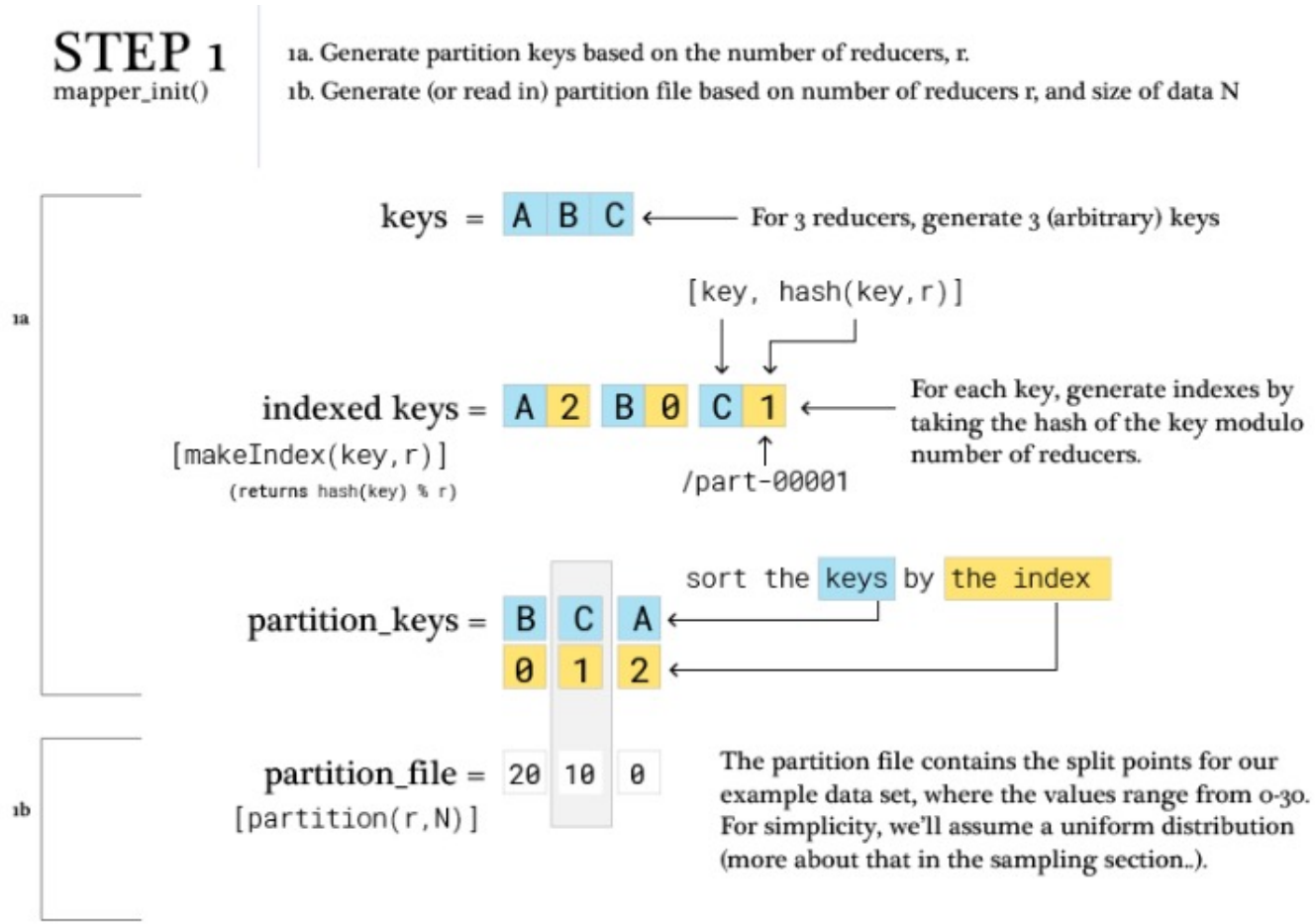


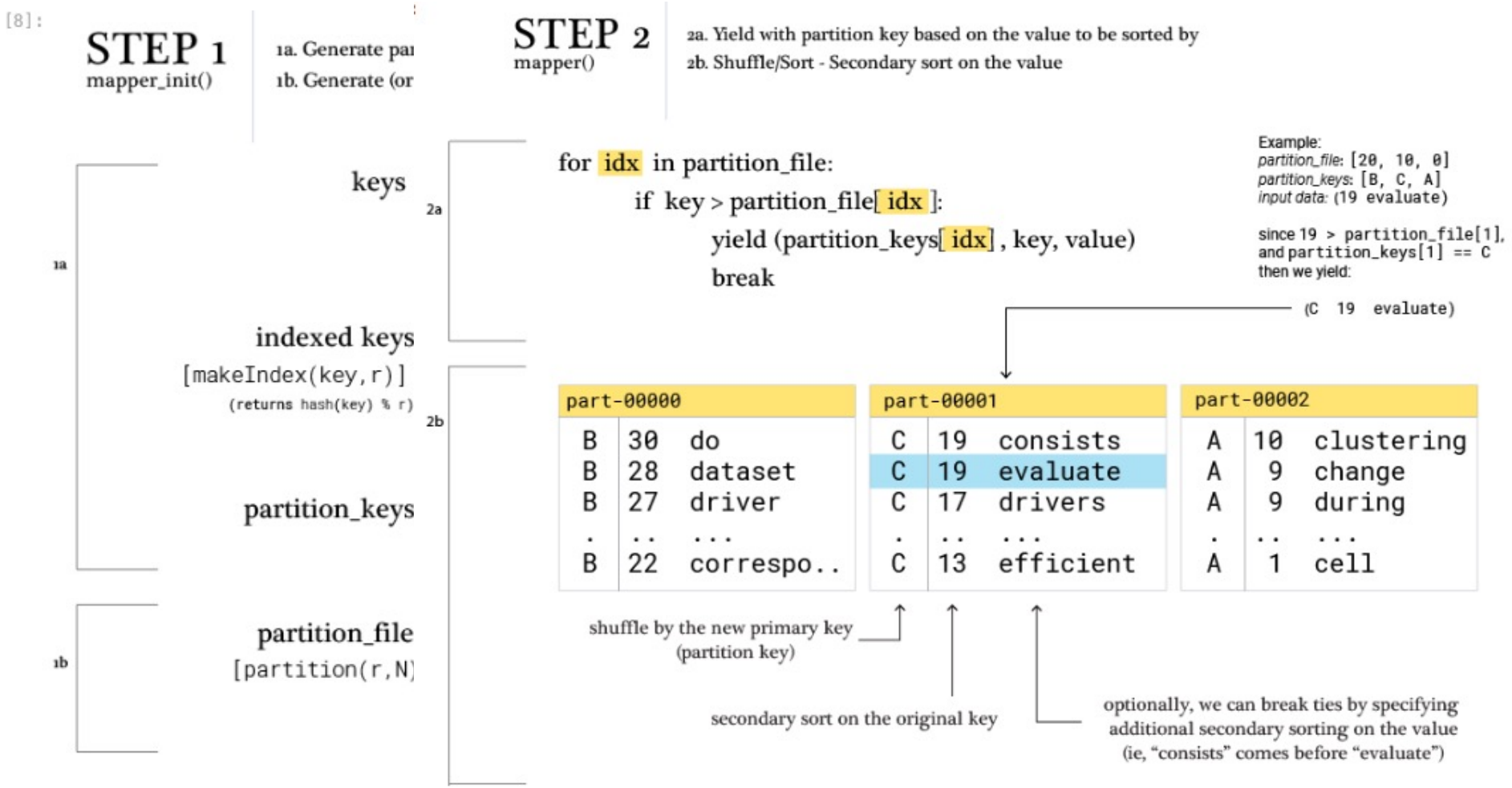
Figure 3. Partial Order Sort

Total Order Sort with ordered partitions - illustrated

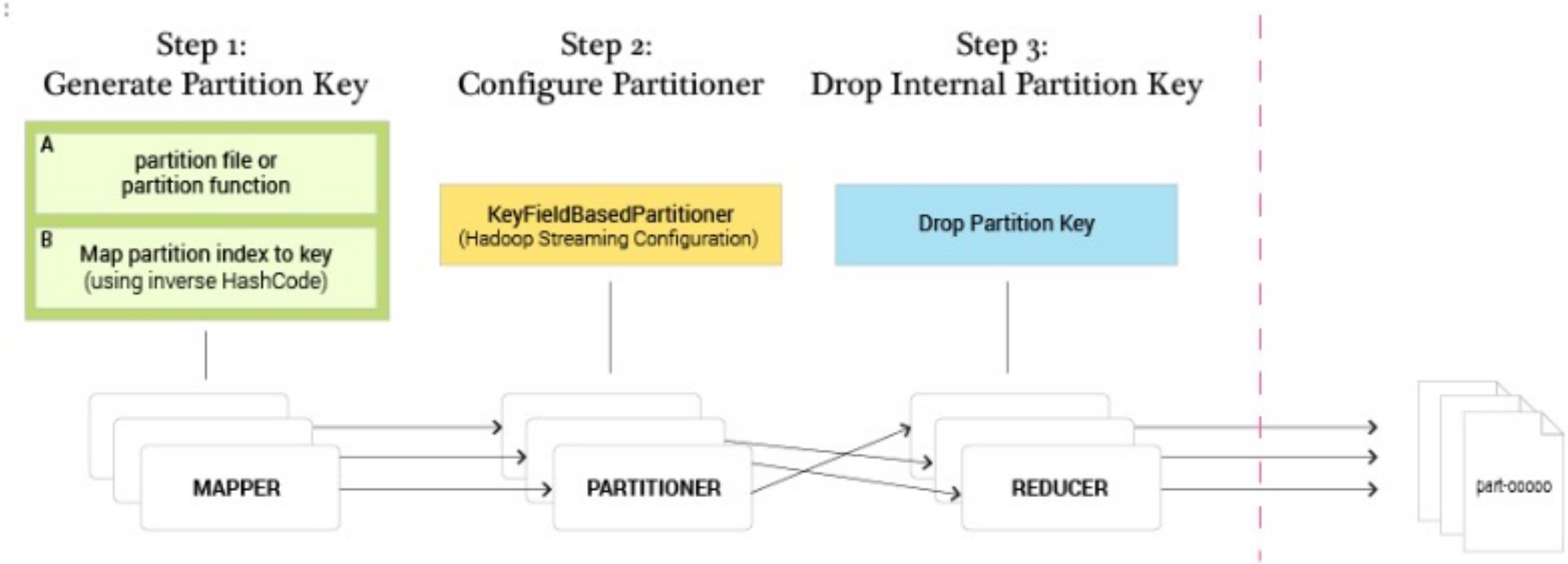
[8]:



Total Order Sort with ordered partitions - illustrated



Inverse HashCode Function



```
%%writefile multipleReducerTotalOrderSort_mapper.py
#!/usr/bin/env python
"""
INPUT:
    count \t word
OUTPUT:
    partitionKey \t count \t word
"""

import os
import re
import sys
import numpy as np
from operator import itemgetter


N = int(os.getenv('mapreduce_job_reduces', default=1))

def makeIndex(key, num_reducers = N):
    """
    Mimic the Hadoop string-hash function.

    key             the key that will be used for partitioning
    num_reducers    the number of reducers that will be configured
    """
    byteof = lambda char: int(format(ord(char), 'b'), 2)
    current_hash = 0
    for c in key:
        current_hash = (current_hash * 31 + byteof(c))
    return current_hash % num_reducers

def makeKeyFile(num_reducers = N):
    KEYS = list(map(chr, range(ord('A'), ord('Z')+1)))[:num_reducers]
    partition_keys = sorted(KEYS, key=lambda k: makeIndex(k,num_reducers))

    return partition_keys


# call your helper function to get partition keys
pKeys = makeKeyFile()

def makePartitionFile():
    # returns a list of split points
    # For the sake of simplicity this is hardcoded.
    # See the sampling section below for more information.
    return [20,10,0]

pFile = makePartitionFile()

### Mapper starts on each input record
###
for line in sys.stdin:
    line = line.strip()
    key,value = line.split('\t')

    for idx in range(N):
        if float(key) > pFile[idx]:
            print(str(pKeys[idx])+"\t"+key+"\t"+value)
            break
```

"inverse hashCode function" for power laws (compound keys)

hash_function(key) → hash

```
# helper functions
def makeKeyHash(key, num_reducers):
    """
    Mimic the Hadoop string-hash function.

    key             the key that will be used for partitioning
    num_reducers    the number of reducers that will be configured
    """
    byteof = lambda char: int(format(ord(char), 'b'), 2)
    current_hash = 0
    for c in key:
        current_hash = (current_hash * 31 + byteof(c))
    return current_hash % num_reducers

# helper function
def getPartitionsFromFile(fpath='partitions.txt'):
    """
    Args:   partition file path
    Returns:    partition_keys (sorted list of strings)
                partition_values (descending list of floats)

    NOTE 1: make sure the partition file gets passed into Hadoop
    """
    # load in the partition values from file
    assert os.path.isfile(fpath), 'ERROR with partition file'
    with open(fpath,'r') as f:
        vals = f.read()
    partition_cuts = sorted([float(v) for v in vals.split(',')], reverse=True)

    # use the first N uppercase letters as custom partition keys
    N = len(partition_cuts)
    KEYS = list(map(chr, range(ord('A'), ord('Z')+1)))[:N]
    partition_keys = sorted(KEYS, key=lambda k: makeKeyHash(k,N))

    return partition_keys, partition_cuts


# call your helper function to get partition keys & cutpoints
pKeys, pCuts = getPartitionsFromFile()
```

Find partition key values and sort them so they
The partition key used produces a total order sort that is aligned with the problem key space.
In the example below:

B → 0
C→ 1
A→ 2

Back to top | Back to Section 3

inverse has 1/3

In order to preserve partition key ordering, we will construct an "inverse hashCode function", which takes as input the desired partition index and total number of partitions, and returns the partition key. This key, when supplied to the Hadoop framework (KeyBasedPartitioner), will hash to the returned desired index.

First, let's implement the core of HashPartitioner in Python:

```
def makeIndex(key, num_reducers):
    byteof = lambda char: int(format(ord(char), 'b'), 2)
    current_hash = 0
    for c in key:
        current_hash = (current_hash * 31 + byteof(c))
    return current_hash % num_reducers

# partition indexes for keys: A,B,C; with 3 partitions
[makeIndex(x, 3) for x in "ABC"]
```

```
[2, 0, 1]
```

A simple strategy to implement an inverse hashCode function is to use a lookup table. For example, assuming we have 3 reducers, we can compute the partition index with makeIndex for keys "A", "B", and "C". The results are listed the the table below.

Partition Key	Partition Index
A	2
B	0
C	1

In the mapper stage, if we want to assign a record to partition 0, for example, we can simply look at the partition key that generated the partition index 0 which in this case is "B".

See `KeyBasedPartitioner` source code for the actual implementations.

Inverse HashCode Function

Back to top | Back to Section 3

In order to preserve partition key ordering, we will construct an "inverse hashCode function", which takes as input the desired partition index and total number of partitions, and returns the partition key. This key, when supplied to the Hadoop framework (KeyBasedPartitioner), will hash to the returned desired index.

First, let's implement the core of HashPartitioner in Python:

[20]:
```
def makeIndex(key, num_reducers):
    byteof = lambda char: int(format(ord(char), 'b'), 2)
    current_hash = 0
    for c in key:
        current_hash = (current_hash * 31 + byteof(c))
    return current_hash % num_reducers

# partition indexes for keys: A,B,C; with 3 partitions
[makeIndex(x, 3) for x in "ABC"]
```

[20]:
```
[2, 0, 1]
```

A simple strategy to implement an inverse hashCode function is to use a lookup table. For example, assuming we have 3 reducers, we can compute the partition index with makeIndex for keys "A", "B", and "C". The results are listed the the table below.

Partition Key	Partition Index
A	2
B	0