

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%20For%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), delimiter ","

Sort by field 2 numerically reverse, delimiter ","

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

Input

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

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

Output

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

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

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

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



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

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

```
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
```

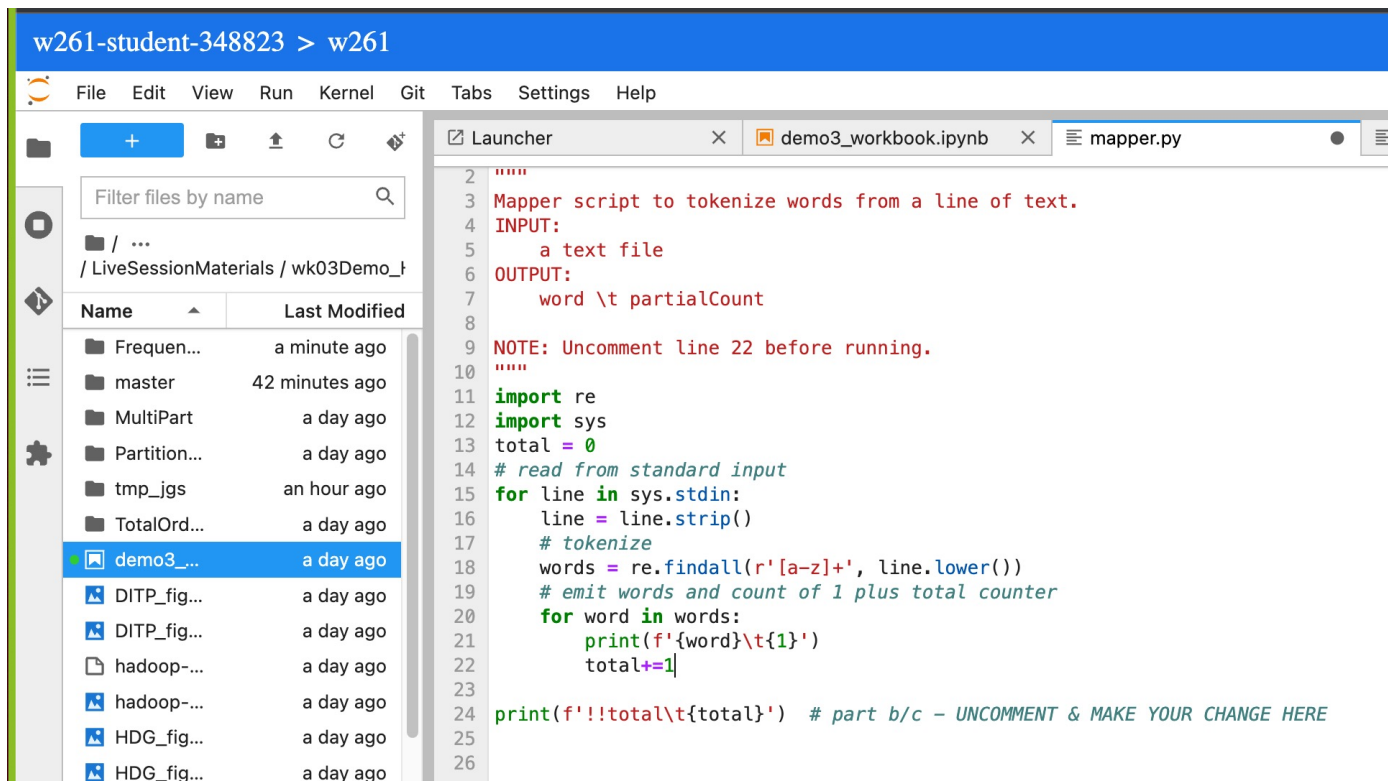
```
[58]: # part b - unit test map-combine (sort mimics shuffle) (RUN THIS CELL AS IS)
!echo "foo foo quux labs foo bar quux" | Frequencies/mapper.py | sort -k1,1 | Frequencies/combiner.py
```

```
!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



```
w261-student-348823 > w261

File Edit View Run Kernel Git Tabs Settings Help

+ [Icons]

Filter files by name

/ ...
/ LiveSessionMaterials / wk03Demo_...

Name Last Modified
Frequen... a minute ago
master 42 minutes ago
MultiPart a day ago
Partition... a day ago
tmp_jgs an hour ago
TotalOrd... a day ago
demo3_... a day ago
DITP_fig... a day ago
DITP_fig... a day ago
hadoop-... a day ago
hadoop-... a day ago
HDG_fig... a day ago
HDG_fig... a day ago

2 """
3 Mapper script to tokenize words from a line of text.
4 INPUT:
5     a text file
6 OUTPUT:
7     word \t partialCount
8
9 NOTE: Uncomment line 22 before running.
10 """
11 import re
12 import sys
13 total = 0
14 # read from standard input
15 for line in sys.stdin:
16     line = line.strip()
17     # tokenize
18     words = re.findall(r'[a-z]+', line.lower())
19     # emit words and count of 1 plus total counter
20     for word in words:
21         print(f'{word}\t{1}')
22         total+=1
23
24 print(f'!!total\t{total}') # part b/c - UNCOMMENT & MAKE YOUR CHANGE HERE
25
26
```

Prefix a partition key in the mapper

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

```
3 Mapper partitions based on first letter in word.
4 INPUT:
5     word \t count
6 OUTPUT:
7     partitionKey \t word \t count
8     ""
9 import re
10 import sys
11
12 def getPartitionKey(word, count):
13     """
14     Helper function to assign partition key ('A', 'B', or 'C').
15     Args: word (str) ; count (int)
16     """
17     ##### YOUR CODE HERE #####
18     if count > 8:                # <--- SOLUTION --->
19         return 'B'                # <--- SOLUTION --->
20     elif count > 4:              # <--- SOLUTION --->
21         return 'C'                # <--- SOLUTION --->
22     else:                        # <--- SOLUTION --->
23         return 'A'                # <--- SOLUTION --->
24
25 # provided implementation: (run this first, then make your changes in part e)
26 if word[0] < 'h':
27     return 'A'
28 elif word[0] < 'p':
29     return 'B'
30 else:
31     return 'C'
32 ##### (END) YOUR CODE #####
33
34 # read from standard input
35 for line in sys.stdin:
36     word, count = line.strip().split()
37     count = int(count)
38     partitionKey = getPartitionKey(word, count)
39     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		file: part-00001		file: part-00002	
27	driver	30	do	26	descent
27	creating	28	dataset	26	def
27	experiements	15	computing	25	compute
19	consists	15	document	24	done
19	evaluate	15	computational	24	code
17	drivers	14	center	23	descent
10	clustering	5	distributed	22	corresponding
9	during	4	develop	13	efficient
9	change	3	different	1	cell
7	contour	2	cluster	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		file: part-00001		file: part-00002	
19	consists	10	clustering	30	do
19	evaluate	9	during	28	dataset
17	drivers	9	change	27	driver
15	computing	7	contour	27	creating
15	document	5	distributed	26	descent
15	computational	4	develop	26	def
14	center	3	different	25	compute
13	efficient	2	cluster	24	done
		1	cell	24	code
		0	current	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		file: part-00001		file: part-00002	
30	do	19	evaluate	9	during
28	dataset	19	consists	9	change
27	creating	17	drivers	7	contour
27	driver	15	document	5	distributed
27	experiements	15	computing	4	develop
26	def	15	computational	3	different
26	descent	14	center	2	cluster
		13	efficient	1	cell
		10	clustering	0	current
23	descent				
22	corresponding				

See: [HelpfulResources/TotalSortGuide/total-sort-guide-hadoop-streaming.ipynb](https://helpfulresources.github.io/TotalSortGuide/total-sort-guide-hadoop-streaming.ipynb)

Live session 3: part total order sort notebook

w261-student-348823 > w261

Toggle Markdown Text Calls

File Edit View Run Kernel Git Tabs Settings Help

TOTAL-SORT-GUIDE-HADOOP-STREAMING.IPYNB

Prepare Dataset

Section I - Understanding Unix Sort

Importance of Unix Sort

Unix Sort Overview

sort examples

Section II - Hadoop Streaming

II.A. Hadoop's Default Sorting Behavior

Key points:

II.B. Hadoop Streaming parameters

Configure Hadoop Streaming: Prerequisites

Configure Hadoop Streaming: Step 1

Configure Hadoop Streaming: Step 2

Configure Hadoop Streaming: Step 3

Summary of Common Practices for Sorting Related Configuration

Side-by-side Examples: Unix sort vs. Hadoop Streaming

II.C. Hadoop Streaming implementation

Start Hadoop

II.C.1. Hadoop Streaming Implementation - single reducer

Key points:

Steps

Setup:

total-sort-X demo3_wcX TOS_mapperX TOS_mapperX demo2_wcX demo3_wcX Untitled.ipX mapper.pyX mapper.pyX

```
12 s = np.random.uniform(0,1)
13 if s < .01:
14     print(line.strip())
```

Overwriting RandomSample.py

```
[20]: 1 !hdfs dfs -rm -r {HDFS_DIR}/sort/sampleData
2 !hadoop jar {JAR_FILE} \
3     -files RandomSample.py \
4     -mapper RandomSample.py \
5     -input {HDFS_DIR}/sort/random_numbers \
6     -output {HDFS_DIR}/sort/sampleData \
7     -numReduceTasks 0 \
8     -cmdenv PATH={PATH}
```

...

```
[21]: 1 !hdfs dfs -ls {HDFS_DIR}/sort/sampleData/
```

Found 3 items

-rw-r--r--	1	root	supergroup	0	2020-05-21 10:57	/user/root/TOS/sort/sampleData/_SUCCESS
-rw-r--r--	1	root	supergroup	111430	2020-05-21 10:57	/user/root/TOS/sort/sampleData/part-00000
-rw-r--r--	1	root	supergroup	111896	2020-05-21 10:57	/user/root/TOS/sort/sampleData/part-00001

```
[22]: 1 !hdfs dfs -cat {HDFS_DIR}/sort/sampleData/part-* > sampleData.txt
```

```
[23]: 1 !cat sampleData.txt | head
```

```
6.219041831485558447
6.686344258768484039
6.769544208921161932
7.639744831367128342
2.139872044496024195
3.798373714588547667
2.678375024342094513
6.667164453774735655
0.708364758734653099
1.880770912957316909
cat: write error: Broken pipe
```

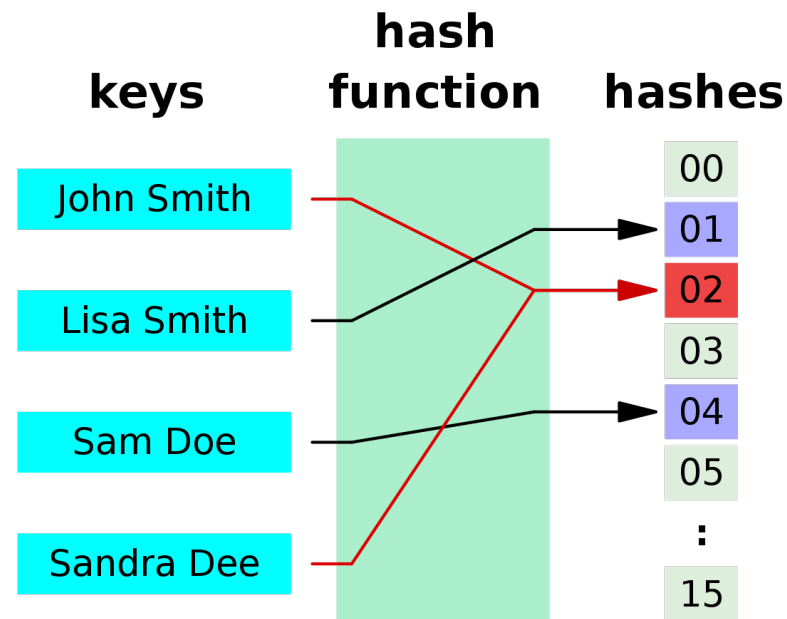
```
[24]: 1 !cat sampleData.txt | tail
```

```
28.039338755785088608
```

Simple 0 5 Python 3 | Idle Saving completed Mode: Command Ln 1, Col 1 total-sort-guide-hadoc

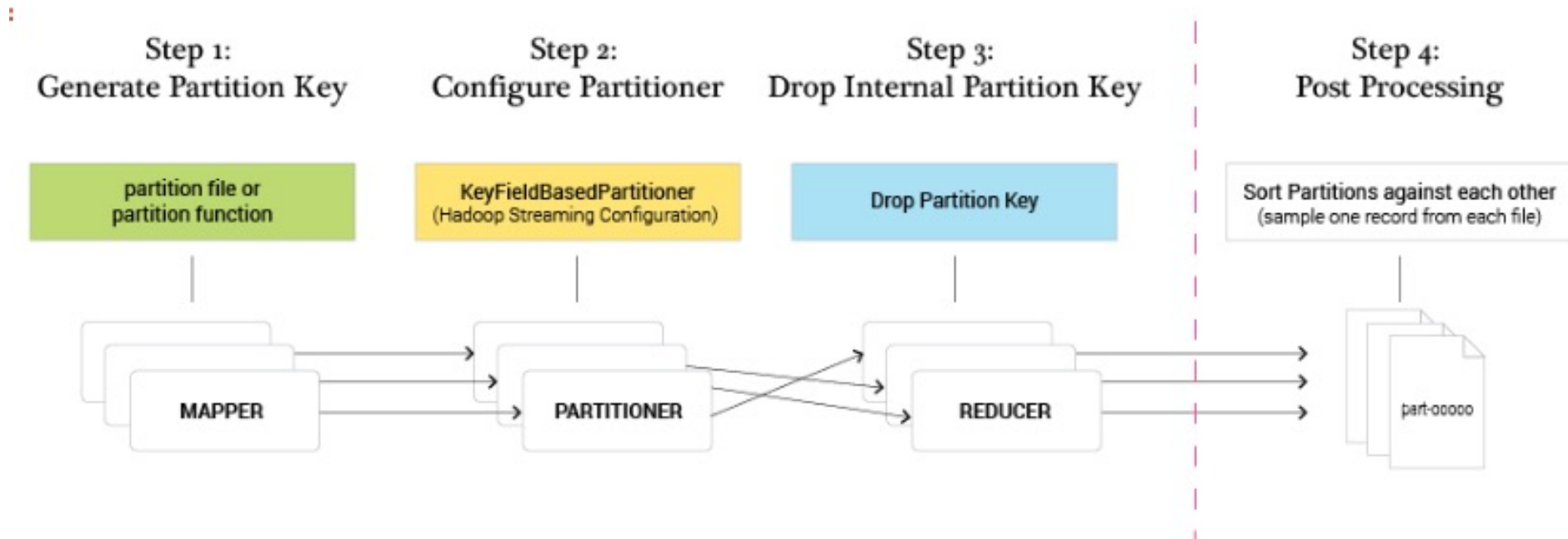
hash_function(key) → hash

- 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:

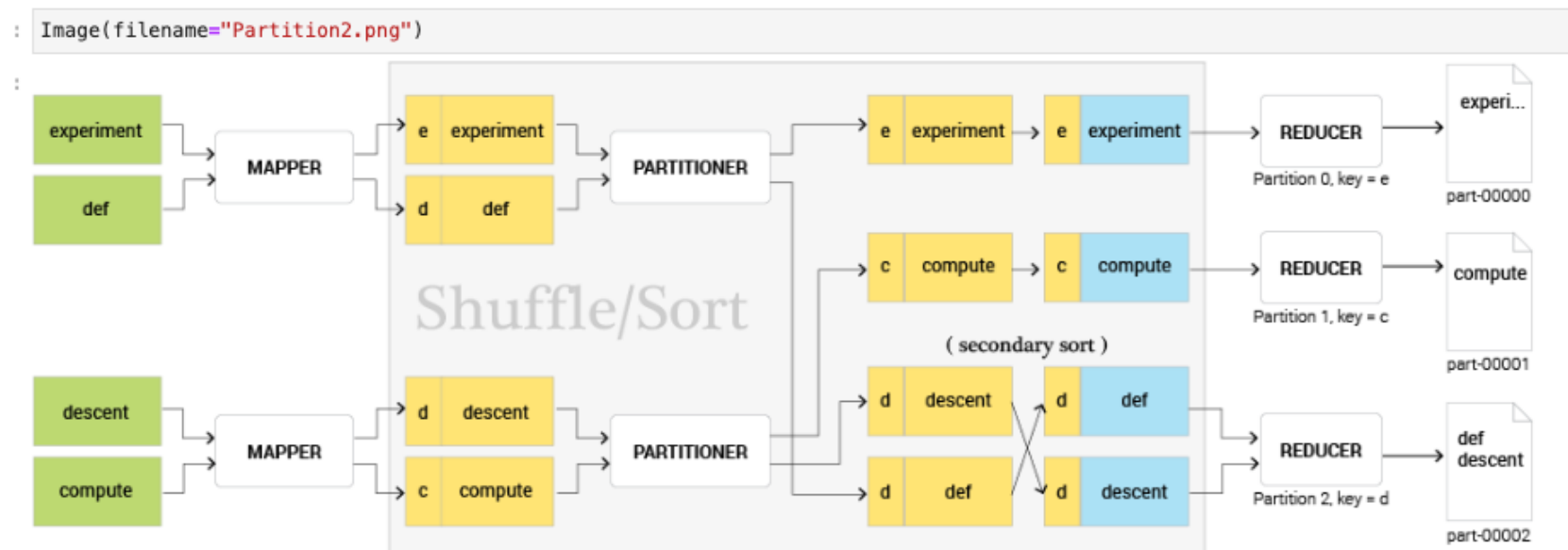


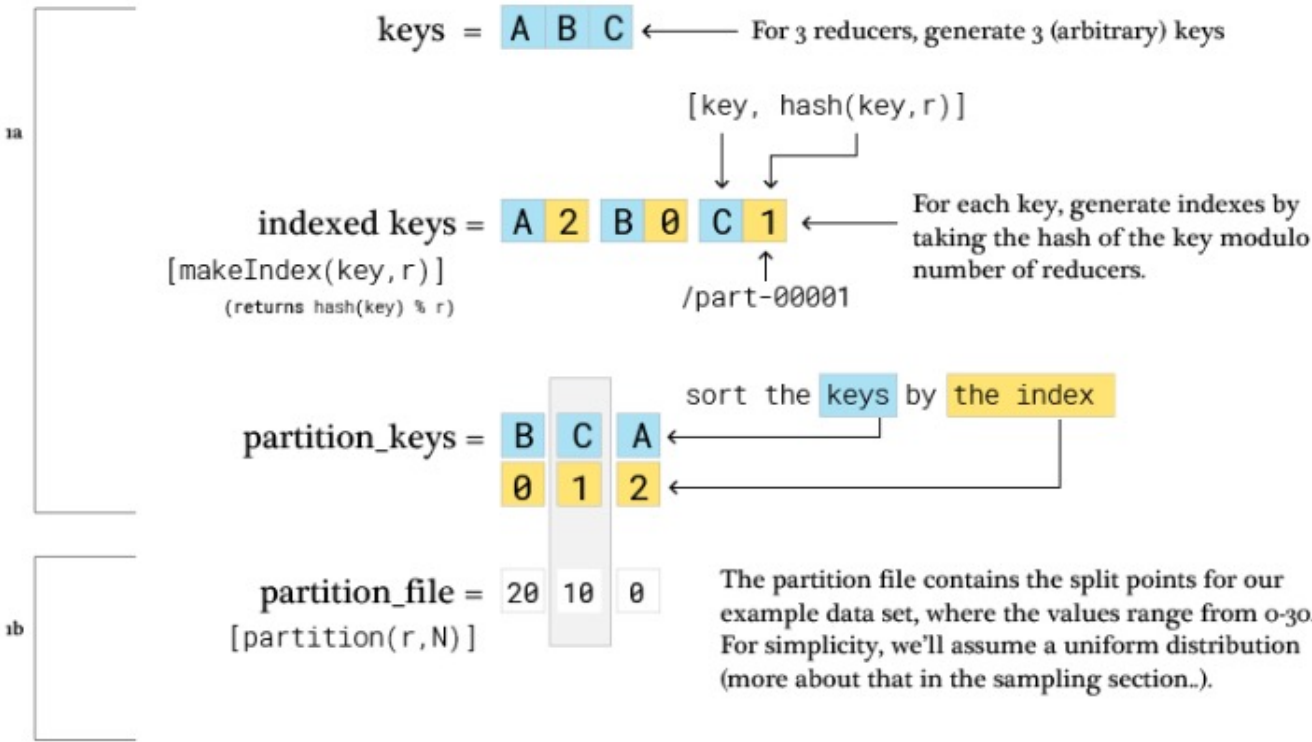
Figure 3. Partial Order Sort

Total Order Sort with ordered partitions - illustrated

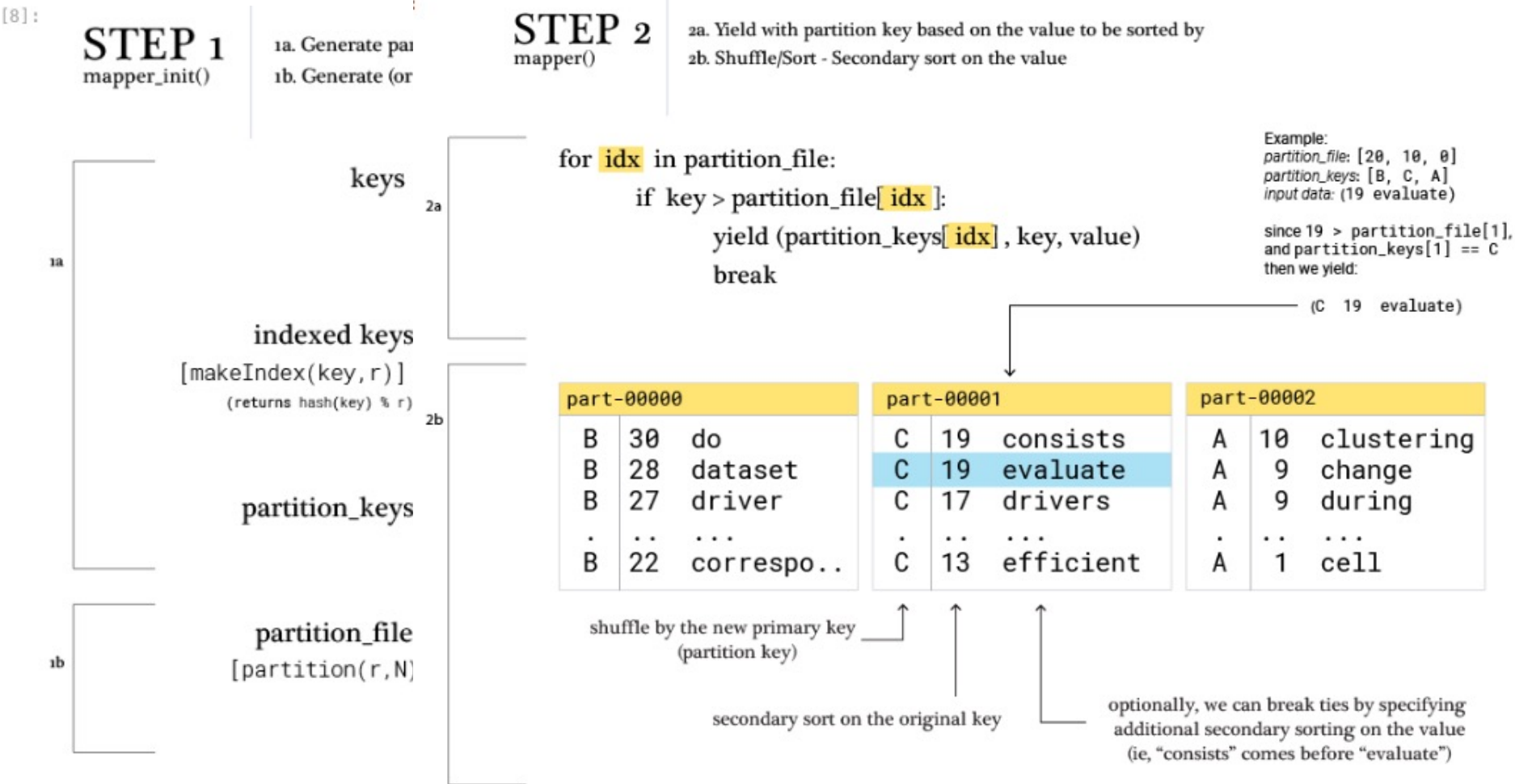
[8]:

STEP 1 mapper_init()

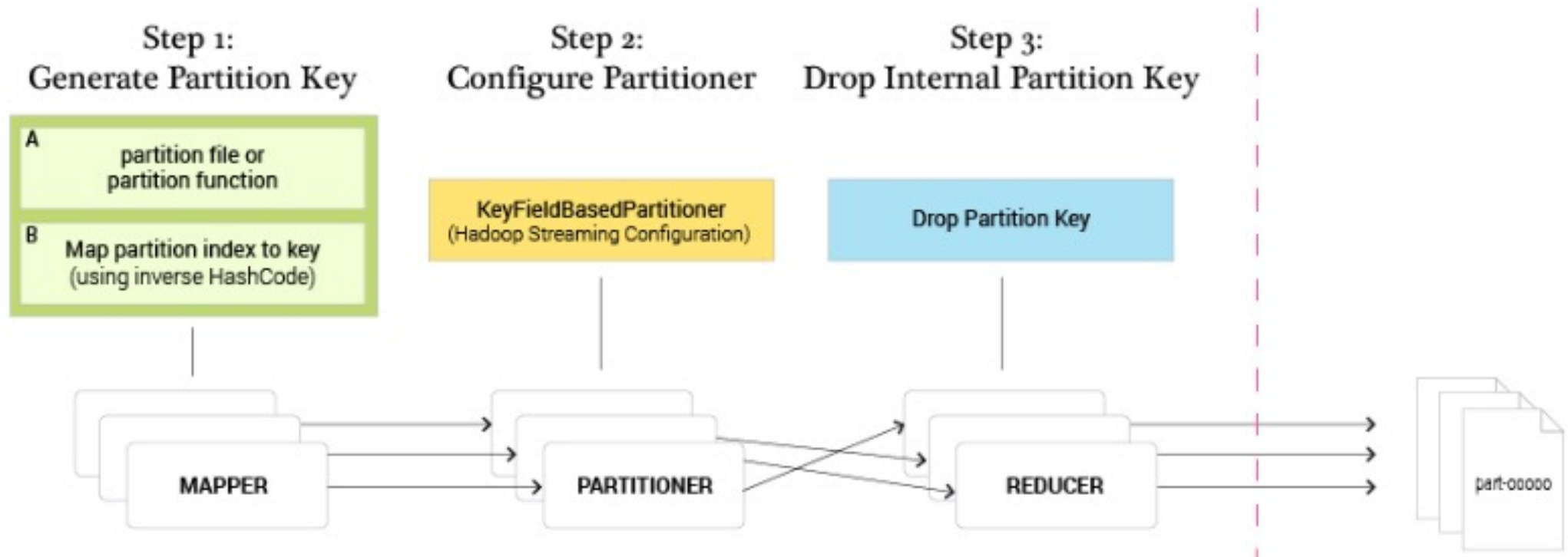
- 1a. Generate partition keys based on the number of reducers, r.
- 1b. Generate (or read in) partition file based on number of reducers r, and size of data N



Total Order Sort with ordered partitions - illustrated



Inverse hashCode Function



```

1  %writefile multipleReducerTotalOrderSort_mapper.py
2  #!/usr/bin/env python
3  """
4  INPUT:
5      count \t word
6  OUTPUT:
7      partitionKey \t count \t word
8  """
9
10 import os
11 import re
12 import sys
13 import numpy as np
14 from operator import itemgetter
15
16
17 N = int(os.getenv('mapreduce_job_reduces', default=1))
18
19 def makeIndex(key, num_reducers = N):
20     """
21     Mimic the Hadoop string-hash function.
22
23     key            the key that will be used for partitioning
24     num_reducers   the number of reducers that will be configured
25     """
26     byteof = lambda char: int(format(ord(char), 'b'), 2)
27     current_hash = 0
28     for c in key:
29         current_hash = (current_hash * 31 + byteof(c))
30     return current_hash % num_reducers
31
32 def makeKeyFile(num_reducers = N):
33     KEYS = list(map(chr, range(ord('A'), ord('Z')+1)))[:num_reducers]
34     partition_keys = sorted(KEYS, key=lambda k: makeIndex(k,num_reducers))
35
36     return partition_keys
37
38
39 # call your helper function to get partition keys
40 pKeys = makeKeyFile()
41
42 def makePartitionFile():
43     # returns a list of split points
44     # For the sake of simplicity this is hardcoded.
45     # See the sampling section below for more information.
46     return [20,10,0]
47
48 pFile = makePartitionFile()
49
50 ### Mapper starts on each input record
51 ###
52 for line in sys.stdin:
53     line = line.strip()
54     key,value = line.split('\t')
55
56     for idx in range(N):
57         if float(key) > pFile[idx]:
58             print(str(pKeys[idx])+"\t"+key+"\t"+value)
59             break

```


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

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](#)

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:

```
[ ]: 1 def makeIndex(key, num_reducers):
2     bytearray = bytearray(char: int(format(ord(char), 'b'), 2)
3     current_hash = 0
4     for c in key:
5         current_hash = (current_hash * 31 + bytearray(c))
6     return current_hash % num_reducers
7
8 # partition indexes for keys: A,B,C; with 3 partitions
9 [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 in 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".

```
14 # helper functions
15 def makeKeyHash(key, num_reducers):
16     """
17     Mimic the Hadoop string-hash function.
18
19     key            the key that will be used for partitioning
20     num_reducers   the number of reducers that will be configured
21     """
22     bytearray = lambda char: int(format(ord(char), 'b'), 2)
23     current_hash = 0
24     for c in key:
25         current_hash = (current_hash * 31 + bytearray(c))
26     return current_hash % num_reducers
27
28 # helper function
29 def getPartitionsFromFile(fpath='partitions.txt'):
30     """
31     Args:    partition file path
32     Returns: partition_keys (sorted list of strings)
33             partition_values (descending list of floats)
34
35     NOTE 1: make sure the partition file gets passed into Hadoop
36     """
37     # load in the partition values from file
38     assert os.path.isfile(fpath), 'ERROR with partition file'
39     with open(fpath, 'r') as f:
40         vals = f.read()
41     partition_cuts = sorted([float(v) for v in vals.split(',')], reverse=True)
42
43     # use the first N uppercase letters as custom partition keys
44     N = len(partition_cuts)
45     KEYS = list(map(chr, range(ord('A'), ord('Z')+1)))[0:N]
46     partition_keys = sorted(KEYS, key=lambda k: makeKeyHash(k, N))
47
48     return partition_keys, partition_cuts
49
50
51 # call your helper function to get partition keys & cutpoints
52 pKeys, pCuts = getPartitionsFromFile()
53
```

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]: 1 def makeIndex(key, num_reducers):
      2     byteof = lambda char: int(format(ord(char), 'b'), 2)
      3     current_hash = 0
      4     for c in key:
      5         current_hash = (current_hash * 31 + byteof(c))
      6     return current_hash % num_reducers
      7
      8 # partition indexes for keys: A,B,C; with 3 partitions
      9 [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
