A Comprehensive Guide to the Total Order Sort With Hadoop Streaming

---

by Kyle Hamilton
Reservoir sampling implemenation by **Rowan Cassius** - w261 Summer 2020
University of California, Berkeley

Modified from the original by Kyle Hamilton, James G. Shanahan, Yiran Sheng
Email: kylehamilton@ischool.berkeley.edu, jimi@ischool.berkeley.edu, yiran@ischool.berkeley.edu

In [168…
```
from IPython.display import Image, HTML
```

In [169…
```
with open("code-tab.css",'r') as style:
    code_tab_style = "<style>{}</style>".format(style.read())

HTML(code_tab_style)
```

Out[169…

Table of Contents

Abstract

Back to top

Sorting refers to arranging data in a particular format. A sorting algorithm specifies the way to arrange data in a particular order. Most common orders are numerical or lexicographical order. One important application of sorting lies in the fact that information search can be optimized to a very high level if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Some of the examples of sorting in real life scenarios are the following.

- Telephone Directory – A telephone directory keeps telephone numbers of people sorted on their names so that names can be searched.
- Dictionary – A dictionary keeps words in alphabetical order so that searching of any word becomes easy.

Sorting at scale can be accomplished via MapReduce frameworks such as Hadoop Streaming which operates on data records consisting of key-value pairs. This framework, when run in default mode (by specifying either a mapper, a reducer, or both depending on what needs to be accomplished), will generate a partial order sort, whereby the reducer output will be a lot of (partition) files each of which contains key-value records that are sorted within each partition file based on the key. But there is no sort order between records in different partition files (see Figure 3). This is the default behavior for MapReduce frameworks. Sometimes it is desirable to have all data records sorted by a specified key (i.e., different to a partial order sort), across partition boundaries (see Figure 2). This can be accomplished relatively easily using Hadoop Streaming. But to do so one needs to be familiar with some core concepts and design patterns. This paper introduces the building blocks for total sorting in Hadoop Streaming. Numerous complete working examples are also provided to demonstrate the key design patterns in total order sorts. Python is used as the language for driving all three MapReduce frameworks.

Keywords: total order sort at scale, sorting in Hadoop Streaming, MapReduce

INTRODUCTION

Back to top

In this notebook we are going to demonstrate how to achieve Total Order Sort in Hadoop Streaming. Hadoop Streaming borrows heavily in terms of syntax and semantics from the Unix sort and cut commands, whereby they treat the output of the mapper as series of records, where each record can be interpreted as a collection of fields/tokens that are tab delimited by default. In this way, fields can be specified for the purposes of partitioning (routing), sometimes referred to as the primary key. The primary key is used for partitioning, and the combination of the primary and secondary keys (also specified by the programmer) is used for sorting.

We'll start by describing the Linux/Unix sort command (syntax and semantics) and build on that understanding to explain Total Order Sort in Hadoop Streaming. Partitioning is not just matter of specifying the fields to be used for routing the records to the reducers. We also need to consider how best to partition the data that has skewed distributions. To that end, we'll demonstrate how to partition the data via sampling and assigning custom keys.

At each step we are going to build on the previous steps, so it's important to view this notebook in order. For example, we'll cover key points for sorting with a single reducer in the Hadoop Streaming implementation, and these concepts will apply to the subsequent sections.

Anatomy of a MapReduce Job

Back to top

When tackling large scale data problems with modern computing frameworks such as MapReduce (Hadoop), one generally uses a divide-and-conquer strategy to divide these large problems into chunks of key-value records that can be processed by individual compute nodes (via mapper/reducer procedures). Upon importing the data into the Hadoop Distributed File System (HDFS), it is chunked in the following way: typically, there is a chunk for each input file. If the input file is too big (bigger than the HDFS block size) then we have two or more map chunks/splits associated to the same input file. (Please refer to the method getSplits() of the FileInputFormat class in Hadoop for more details.)

Once the data has been uploaded to HDFS, MapReduce jobs can be run on the data. First we'll focus on a basic MapReduce/Hadoop job. The programmer provides map and reduce functions and, subsequently, the Application Master will launch one MapTask for each map split/chunk.

map(key1, value1) → list(key2, value2)

reduce(key2, list(value2)) → list(key3, value3)

First, the map() function receives a key-value pair input, (key1, value1). Then it outputs any number of key-value pairs, (key2, value2). Next, the reduce() function receives as input another key-value pair, (key2, list(value2)), and outputs any number of (key3, value3) pairs.

Now consider the following key-value pair, (key2, list(value2)), as an input for a reducer:

list(value2) = (V1, V2, ..., Vn)

where there is no ordering between reducer values (V1, V2, ..., Vn).

The MapReduce framework performs the important tasks of routing, collating records with the same key, and transporting these records from the mapper to the reducer. These tasks are commonly referred to as the shuffle phase of a MapReduce job and are typically run in default mode (whereby the programmer does not customize the phase).

So, for completeness, a typical MapReduce job consists of three phases (that are similar in style for all MapReduce frameworks):

def MapReduceJob(Mapper, Shuffler, Reducer):

- Mapper (programmer writes)
    - Mapper Init (setups a state for the mapper if required)
    - map(key1, value1) → list(key2, value2)
    - Mapper Final phase (tears down the map and outputs the mapper state to the stream if required)
- Shuffle Phase(Partitioner=Key, Sort=(Key, alphanumeric increasing), Combiner=None)
    - Where the default behaviors for the shuffle are as follows:
        - Partitioner=Key,

    - Sort=(Key, alphanumeric increasing),
    - Combiner=None
  - Partitioner: specify key (or subset of fields) used for routing each record to the reducer
  - Sort specification: set of fields that make up the keys and the order required
  - Combiner: DEFAULT: no combiner
- Write the Reducer (programmer writes)
  - Reducer Init phase (setups a state for the reducer if required)
  - reduce(key2, list(value2)) → list(key3, value3)
  - Reducer Final phase (tears down the reduce task and outputs the reduce state to HDFS if required)

To perform a total order sort, one needs to override the default shuffle behavior by providing a custom partitioner, a custom sort specification (e.g., numeric/alphanumeric increasing/decreasing), and a combiner. Note that a combiner is not required for a total order sort but makes the MapReduce job more efficient.

In [171…
```
Image('anatomy-of-MR.png')
```

Out[171…

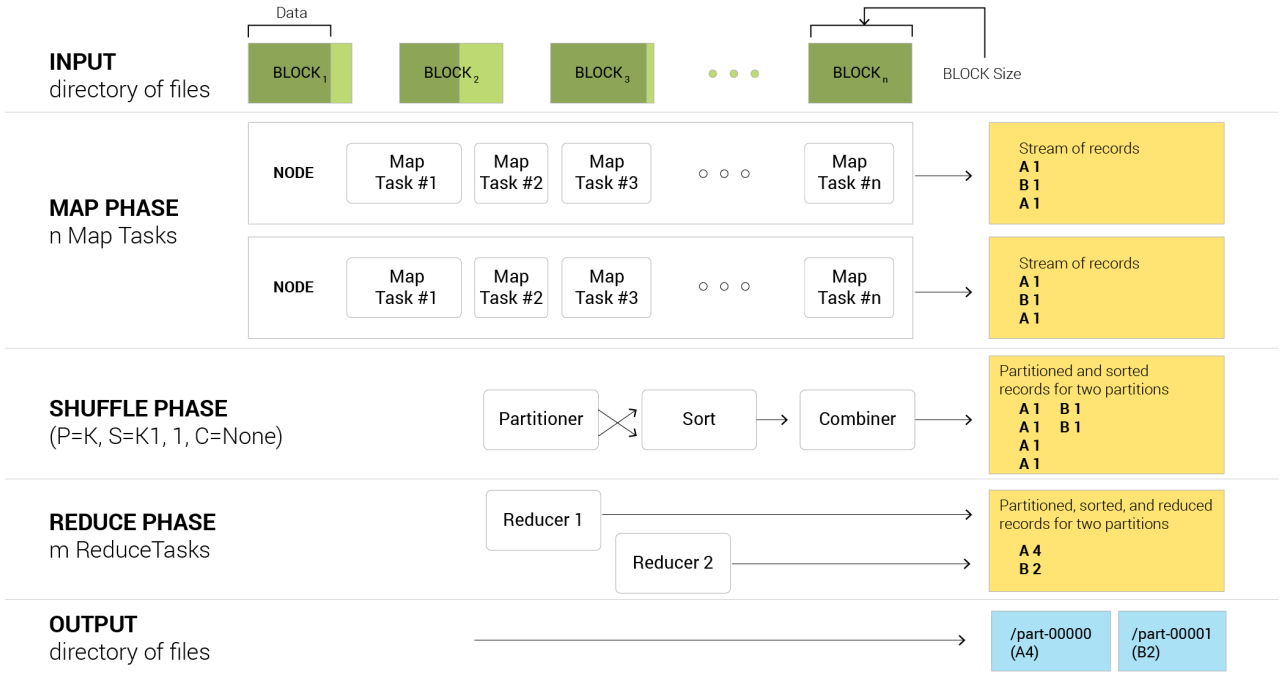


Figure 1: Anatomy of a MapReduce Job from input data to output data via map, shuffle, and reduce steps.

Terminology

Back to top

Apache Hadoop is a framework for running applications on large clusters built of commodity hardware. The Hadoop framework transparently provides applications both reliability and data motion. Hadoop implements a computational paradigm named Map/Reduce, where the

application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. In addition, it provides a distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both MapReduce and the Hadoop Distributed File System are designed so that node failures are automatically handled by the Hadoop framework. http://wiki.apache.org/hadoop/

Partial Sort - The reducer output will be lot of (partition) files, each of which contains key-value records that are sorted within each partition file based on the key. This is the default behavior for MapReduce frameworks such as Hadoop.

Total Sort (Unordered partitions) - Total sort refers to an ordering of all key-value pairs based upon a specified key. This total ordering will run across all output partition files unlike the partial sort described above. One caveat here is that partition files will need to be re-stacked to generate a total ordering (a small post-processing step that is required after the map-reduce job finishes).

Total Sort (Ordered partitions) - Total sort where the partition file names are also assigned in order.

Secondary Sort - Secondary sorting refers to controlling the ordering of records based on the key and also using the values (or part of the value). That is, sorting can be done on two or more field values.

Hadoop Streaming

Back to top

[Hadoop Streaming] (http://hadoop.apache.org/docs/stable1/streaming.html#Hadoop+Streaming) is a utility that comes with the Hadoop distribution. The utility allows a user to create and run Map-Reduce jobs with any executable or script as the mapper and/or the reducer.

In [1]:
```
$HADOOP_HOME/bin/hadoop  jar $HADOOP_HOME/hadoop-streaming.jar \
    -input myInputDirs \
    -output myOutputDir \
    -mapper /bin/cat \
    -reducer /bin/wc
```

In the above example, both the mapper and the reducer are executables that read the input from stdin (line by line) and emit the output to stdout. The utility will create a MapReduce job, submit the job to an appropriate cluster, and monitor the progress of the job until it completes. When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized.

As the mapper task runs, it converts its inputs into lines and feeds the lines to the stdin of the process. In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper. By default, the prefix of a line up to the first tab character is the key and the rest of the

line (excluding the tab character) is the value. If there is no tab character in the line, then the entire line is considered the key and the value is null. However, this can be customized, as discussed later.

When an executable is specified for reducers, each reducer task launches the executable as a separate process, and then the reducer is initialized. As the reducer task runs, it converts its input key/values pairs into lines and feeds the lines to the stdin of the process. In the meantime, the reducer collects the line-oriented outputs from the stdout of the process, converts each line into a key/value pair, which is collected as the output of the reducer. By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) is the value. However, this can be customized, as discussed later.

This is the basis for the communication protocol between the MapReduce framework and the streaming mapper/reducer.

Examples of Different Sort Types (in context of Hadoop and HDFS)

Back to top

Demonstrated below is an example dataset in text format on HDFS. It includes three partitions in an HDFS directory. Each partition stores records in the format of `{Integer} [TAB] {English Word}`.

```
files in hdfs directory
2016-07-20 22:04:56          0 _SUCCESS
2016-07-20 22:04:45    2392650 part-00000
2016-07-20 22:04:44    2368850 part-00001
2016-07-20 22:04:45    2304038 part-00002
```

Partial Sort | **Total Sort (Unorderd Partitions)** | **Total Sort (Ordered Partitions)**

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).

Prepare Dataset

Back to top

Here we generate the data which we will use throughout the rest of this notebook. This is a toy dataset with 30 records, and consists of two fields in each record, separated by a tab character. The first field contains random integers between 1 and 30 (a hypothetical word count), and the second field contains English words. **The goal is to sort the data by word count from highest to lowest.**

In [296…
```
%%writefile generate_numbers.py
#!/usr/bin/python
words = ["cell","center","change","cluster","clustering","code","computational",
         "contour","corresponding","creating","current","dataset","def","descent
         "distributed","do","document","done","driver","drivers","during","effic
import random
N = 30
for n in range(N):
    print random.randint(0,N),"\t",words[n]
```

```
Overwriting generate_numbers.py
```

In [297…
```
# give the python file exacutable permissions, write the file, and inspect numbe
!chmod +x generate_numbers.py;
!./generate_numbers.py > generate_numbers.output
!wc -l generate_numbers.output
```

```
30 generate_numbers.output
```

In [298…
```
# view the raw dataset
!cat generate_numbers.output
```

```
11      cell
5       center
14      change
24      cluster
12      clustering
27      code
19      computational
18      compute
21      computing
20      consists
29      contour
10      corresponding
0       creating
12      current
14      dataset
9       def
2       descent
12      descent
26      develop
8       different
```

```
6       distributed
3       do
23      document
6       done
21      driver
23      drivers
11      during
3       efficient
28      evaluate
1       experiements
```

Section I - Understanding Unix Sort

Back to top

Importance of Unix Sort

Back to top | Back to Section I

Sort is a simple and very useful command found in Unix systems. It rearranges lines of text numerically and/or alphabetically. Hadoop Streaming's KeyBasedComparator is modeled after Unix sort, and utilizes command line options which are the same as Unix sort command line options.

Unix Sort Overview

Back to top | Back to Section I

```
# sort syntax
sort [OPTION]... [FILE]...
```

Sort treats a single line of text as a single datum to be sorted. It operates on fields (by default, the whole line is considered a field). It uses tabs as the delimiter by default (which can be configured with -t option), and splits a (non-empty) line into one or more parts, whereby each part is considered a field. Each field is identified by its index (POS).

The most important configuration option is perhaps -k or --key. Start a key at POS1 (origin 1), end it at POS2 (default end of line):

```
-k, --key=POS1[,POS2]
```

For example, -k1 (without ending POS), produces a key that is the whole line, and -k1,1 produces a key that is the first field. Multiple -k options can be supplied, and applied left to right. Sort keys can be tricky sometimes, and should be treated with care. For example:

```
sort -k1 -k2,2n
```

Will not work properly, as -k1 uses the whole line as key, and trumps -k2,2n.

Another example:

```
sort -k2 -k3
```

This is redundant: it's equivalent to sort -k2.

A good practice for supplying multiple sort keys is to make sure they are non-overlapping.

Other commonly used flags/options are: -n, which sorts the keys numerically, and -r which reverses the sort order.

`sort` examples

Back to top | Back to Section I

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

In [299…

```
%%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 ","

```
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
```

Section II - Hadoop Streaming

Back to top

II.A. Hadoop's Default Sorting Behavior

Back to top | Back to Section II

Key points:

- By default, Hadoop performs a partial sort on mapper output keys, i.e. within each partition keys are sorted.
- By default, keys are sorted as strings.
  - When processing a mapper output record, first the partitioner decides which partition the record should be sent to.
  - In shuffle and sort stage, keys within a partition are sorted.
- If there is only one partition, mapper output keys will be sorted in total order
- The partition index of a given key from mapper outputs is determined by the partitioner, the default partitioner is HashPartitioner which relies on Java's hashCode function to compute an integer hash for the key. The partition index is derived next by hash modulo number of reducers.

II.B. Hadoop Streaming parameters

Back to top | Back to Section II

Hadoop streaming can be further fine-grain controlled through the command line options below. Through these, we can fine-tune the Hadoop framework to better understand line-oriented record structure, and achieve the versatility of single-machine Unix sort, but in a distributed and efficient manner.

```
stream.num.map.output.key.fields
stream.map.output.field.separator
mapreduce.partition.keypartitioner.options
KeyFieldBasedComparator
keycomparator.options
partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

In a sorting task, Hadoop Streaming provides the same interface as Unix sort. Both consume a stream of lines of text, and produce a permutation of input records, based on one or more sort keys extracted from each line of input. Without customizing its sorting and partitioning, Hadoop Streaming treats implicitly each input line as a record consisting of a single key and value, separated by a "tab" character.

Just like the various options Unix sort offers, Hadoop Streaming can be customized to use multiple fields for sorting, sort records by numeric order or keys and sort in reverse order.

The following table provides an overview of relationships between Hadoop Streaming sorting and Unix sort:

	Unix `sort`	**Hadoop streaming**
Key Field Separator	`-t`	`-D stream.map.output.field.separat`
Number of Key Fields	Not Required	`-D stream.num.map.output.key.field`
Key Range	`-k, --key=POS1[,POS2]`	`-D mapreduce.partition.keycomparat` (same syntax as unix sort)
Numeric Sort	`-n, --numeric-sort`	`-D mapreduce.partition.keycomparat` (same syntax as unix sort)
Reverse Order	`-r --reverse`	`-D mapreduce.partition.keycomparat` (same syntax as unix sort)
Partitioner Class	Not Applicable	`-partitioner org.apache.hadoop.mapred.lib.Ke`
Comparator Class	Not Applicable	`-D mapreduce.job.output.key.compar`
Partition Key Fields	Not Applicable	`-D mapreduce.partition.keypartitic`

Therefore, given a distributed sorting problem, it is always helpful to start with a non-scalable solution that can be provided by Unix sort and work out the required Hadoop Streaming configurations from there.

Configure Hadoop Streaming: Prerequisites

Back to top | Back to Section II

```
  -D
mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapreduce.lib
 \
  -partitioner
org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

These two options instruct Hadoop Streaming to use two specific Hadoop Java library classes: KeyFieldBasedComparator and KeyFieldBasedPartitioner. They come in standard Hadoop distribution, and provide the required machinery.

Configure Hadoop Streaming: Step 1

Back to top | Back to Section II

Specify number of key fields and key field separator

```
-D stream.num.map.output.key.fields=4 \
-D mapreduce.map.output.key.field.separator=.
```

In Unix sort when input lines use a non-tab delimiter, we need to supply the -t separator option. Similarly in Hadoop Streaming, we need to specify the character to use as key separators. Common options include: comma",", period ".", and space " ".

One additional hint to Hadoop is the number of key fields, which is not required for Unix sort. This helps Hadoop Streaming to only parse the relevant parts of input lines, as in the end only keys are sorted (not values) – therefore, Hadoop can avoid performing expensive parsing and sorting on value parts of input line records.

Configure Hadoop Streaming: Step 2

Back to top | Back to Section II

Specify sorting options

```
-D mapreduce.partition.keycomparator.options=-k2,2nr
```

This part is very straightforward. Whatever one would do with Unix sort (eg. -k1,1 -k3,4nr), just mirror it for Hadoop Streaming. However it is crucial to remember that Hadoop only uses KeyFieldBasedComparator to sort records within partitions. Therefore, this step only helps achieve partial sort.

Configure Hadoop Streaming: Step 3

Back to top | Back to Section II

Specify partition key field

```
-D mapreduce.partition.keypartitioner.options=-k1,1
```

In this step, we need to specify which key field to use for partitioning. There's no equivalent in Unix sort. One critical detail to keep in mind is that, even though Hadoop Streaming uses Unix sort --key option's syntax for mapreduce.partition.keypartitioner.options., no sorting will actually be performed. It only uses expressions such as -k2,2nr for key extraction; the nr flags will be ignored.

In later sections we will discuss in detail how to incorporate sorting into the partitioner by custom partition key construction.

Summary of Common Practices for Sorting Related Configuration

Back to top | Back to Section II

```
  -D
mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapreduce.lib.
 \
  -D stream.num.map.output.key.fields=4 \
  -D map.output.key.field.separator=. \
  -D mapreduce.partition.keypartitioner.options=-k1,2 \
  -D mapreduce.job.reduces=12 \
  -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

At bare minimum, we typically need to specify:

1. Use KeyFieldBasedPartitioner
2. Use KeyFieldBasedComparator
3. Key field separator (can be omitted if TAB is used as separator)
4. Number of key fields
5. Key field separator again for mapper (under a different config option)
6. Partitioner options (Unix sort syntax)
7. Number of reducer jobs

See hadoop streaming official documentation for more information (hadoop version = 2.7.2).

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

Back to top | Back to Section II

Unix sort	Hadoop Streaming

	`-D mapreduce.job.output.key.comparator.class=\`
	`  org.apache.hadoop.mapreduce.lib.partition.KeyFieldBasedComparator`
	`\`

`sort -t"," -k1,1`	`-D stream.num.map.output.key.fields=2 \`
	`-D stream.map.output.field.separator="," \`
	`-D mapreduce.partition.keypartitioner.options=-k1,1\`
	`-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner\`
	`-D mapreduce.job.reduces=1`

`sort -k1,1 -`	`-D mapreduce.job.output.key.comparator.class=\`

```
k2,3nr              org.apache.hadoop.mapreduce.lib.partition.KeyFieldBasedComparator
                    \
                    -D stream.num.map.output.key.fields=3 \
                    -D mapreduce.partition.keypartitioner.options="-k1,1 -k2,3nr"\
                    -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner\
                    -D mapreduce.job.reduces=1
```

Note: in both examples we use only a single reduce job, which makes hadoop's partial sort equivalent to a total sort.

II.C. Hadoop Streaming implementation

Back to top | Back to Section II

You will need to install, configure, and start Hadoop. Brief instructions follow, but detailed instructions are beyond the scope of this notebook.

Start Hadoop

Back to top | Back to Section II

To run the examples in this notebook you must use the w261 class Docker container, either locally, or in AWS.

- AWS region: N. California region (us-west-1)
- public AMI: w261-linux-hadoop

In [9]:
```
# global vars (paths) - ADJUST AS NEEDED
JAR_FILE = "/usr/lib/hadoop-mapreduce/hadoop-streaming.jar"
HDFS_DIR = "/user/root/TOS"
```

In [10]:
```
from os import environ
PATH  = environ['PATH']
```

In [11]:
```
# should you need to regenerate the file and put it in hdfs a second time, make
!hdfs dfs -mkdir {HDFS_DIR}
```

In [12]:
```
# put the file in hdfs:
!hdfs dfs -mkdir {HDFS_DIR}/sort
!hdfs dfs -mkdir {HDFS_DIR}/output
!hdfs dfs -put generate_numbers.output {HDFS_DIR}/sort
```

In [13]:
```
# make sure it's really there:
!hdfs dfs -ls {HDFS_DIR}/sort/generate_numbers.output
```

```
-rw-r--r--   1 root supergroup        366 2020-05-21 10:55 /user/root/TOS/sort/g
enerate_numbers.output
```

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

Back to top | Back to Section II

Key points:

- Single reducer guarantees a single partition
- Partial sort becomes total sort
- No need for secondary sorting
- Single Reducer becomes scalibility bottleneck

Steps

In the mapper shuffle sort phase, the data is sorted by the primary key, and sent to a single reducer. By specifying `/bin/cat/` for the mapper and reducer, we are telling Hadoop Streaming to use the identity mapper and reducer which simply output the input (Key,Value) pairs.

Setup:

```
-D stream.num.map.output.key.fields=2
-D stream.map.output.field.separator="\t"
-D mapreduce.partition.keycomparator.options="-k1,1nr -k2,2"
```

First we'll specify the number of keys, in our case, 2. The count and the word are primary and secondary keys, respectively. Next we'll tell Hadoop Streaming that our field separator is a tab character. Lastly we'll use the keycompartor options to specify which keys to use for sorting. Here, -n specifies that the sorting is numerical for the primary key, and -r specifies that the result should be reversed, followed by k2 which will sort the words alphabetically to break ties. Refer to the Unix sort section above.

IMPORTANT: Hadoop Streaming is particular about the order in which options are specified.

(For more information, see the docs here: https://hadoop.apache.org/docs/r1.2.1/streaming.html#Hadoop+Comparator+Class)

In [305…

```
!hdfs dfs -rm -r {HDFS_DIR}/sort/output
!hadoop jar {JAR_FILE} \
-D stream.num.map.output.key.fields=2 \
-D stream.map.output.field.separator="\t" \
-D mapreduce.partition.keypartitioner.options=-k1,1 \
-D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapred.lib.KeyFie
-D mapreduce.partition.keycomparator.options="-k1,1nr -k2,2" \
-mapper /bin/cat \
-reducer /bin/cat \
-input {HDFS_DIR}/sort/generate_numbers.output \
-output {HDFS_DIR}/sort/output
```

```
Deleted /user/root/TOS/sort/output
packageJobJar: [] [/usr/lib/hadoop-mapreduce/hadoop-streaming-2.6.0-cdh5.16.2.ja
r] /tmp/streamjob4497762318533089622.jar tmpDir=null
20/05/21 03:51:06 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.
0:8032
20/05/21 03:51:06 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.
0:8032
20/05/21 03:51:07 INFO mapred.FileInputFormat: Total input paths to process : 1
20/05/21 03:51:07 INFO mapreduce.JobSubmitter: number of splits:2
```

```
20/05/21 03:51:07 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_15
89906177930_0023
20/05/21 03:51:07 INFO impl.YarnClientImpl: Submitted application application_15
89906177930_0023
20/05/21 03:51:07 INFO mapreduce.Job: The url to track the job: http://docker.w2
61:8088/proxy/application_1589906177930_0023/
20/05/21 03:51:07 INFO mapreduce.Job: Running job: job_1589906177930_0023
20/05/21 03:51:13 INFO mapreduce.Job: Job job_1589906177930_0023 running in uber
mode : false
20/05/21 03:51:13 INFO mapreduce.Job:  map 0% reduce 0%
20/05/21 03:51:19 INFO mapreduce.Job:  map 50% reduce 0%
20/05/21 03:51:20 INFO mapreduce.Job:  map 100% reduce 0%
20/05/21 03:51:25 INFO mapreduce.Job:  map 100% reduce 100%
20/05/21 03:51:25 INFO mapreduce.Job: Job job_1589906177930_0023 completed succe
ssfully
20/05/21 03:51:26 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=457
                FILE: Number of bytes written=446728
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=794
                HDFS: Number of bytes written=391
                HDFS: Number of read operations=9
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=2
                Launched reduce tasks=1
                Data-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=6014
                Total time spent by all reduces in occupied slots (ms)=2813
                Total time spent by all map tasks (ms)=6014
                Total time spent by all reduce tasks (ms)=2813
                Total vcore-milliseconds taken by all map tasks=6014
                Total vcore-milliseconds taken by all reduce tasks=2813
                Total megabyte-milliseconds taken by all map tasks=6158336
                Total megabyte-milliseconds taken by all reduce tasks=2880512
        Map-Reduce Framework
                Map input records=30
                Map output records=30
                Map output bytes=391
                Map output materialized bytes=463
                Input split bytes=252
                Combine input records=0
                Combine output records=0
                Reduce input groups=30
                Reduce shuffle bytes=463
                Reduce input records=30
                Reduce output records=30
                Spilled Records=60
                Shuffled Maps =2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=118
                CPU time spent (ms)=2000
                Physical memory (bytes) snapshot=799264768
                Virtual memory (bytes) snapshot=4138995712
                Total committed heap usage (bytes)=822607872
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
```

```
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=542
        File Output Format Counters
                Bytes Written=391
20/05/21 03:51:26 INFO streaming.StreamJob: Output directory: /user/root/TOS/sor
t/output
```

In [306…

```
# Check to see that we have indeed generated a single output file
!hdfs dfs -ls {HDFS_DIR}/sort/output
```

```
Found 2 items
-rw-r--r--   1 root supergroup          0 2020-05-21 03:51 /user/root/TOS/sort/o
utput/_SUCCESS
-rw-r--r--   1 root supergroup        391 2020-05-21 03:51 /user/root/TOS/sort/o
utput/part-00000
```

In [307…

```
# Print the results
print("="*100)
print("Single Reducer Sorted Output - Hadoop Streaming")
print("="*100)
!hdfs dfs -cat {HDFS_DIR}/sort/output/part-00000
```

```
====================================================================================
====================
Single Reducer Sorted Output - Hadoop Streaming
====================================================================================
====================
30      center
30      corresponding
30      driver
29      cell
29      drivers
28      contour
28      descent
28      develop
26      dataset
24      consists
24      evaluate
23      clustering
23      experiements
20      def
19      efficient
17      computing
17      document
16      done
12      during
9       computational
8       do
4       compute
4       descent
3       current
2       change
2       cluster
1       creating
1       distributed
0       code
0       different
```

II.C.2. Hadoop Streaming Implementation - multiple reducers

Back to top | Back to Section II

Key points:

- Need to guarantee that every key in a single reducer is to be "pre-sorted" against all other reducers
- Requires knowledge of the distribution of values to be sorted - more about this later in the sampling section
- Uses secondary sort to order keys within each partition

What's new:

Now the mapper needs to emit an additional key for each record by which to partition. We partition by this new 'primary' key and sort by secondary and tertiary keys to break ties. Here, the partition key is the primary key.

The following diagram illustrates the steps required to perform total sort in a multi-reducer setting:

In [175…
```
Image("TotalSortSteps2.png")
```

Out[175…

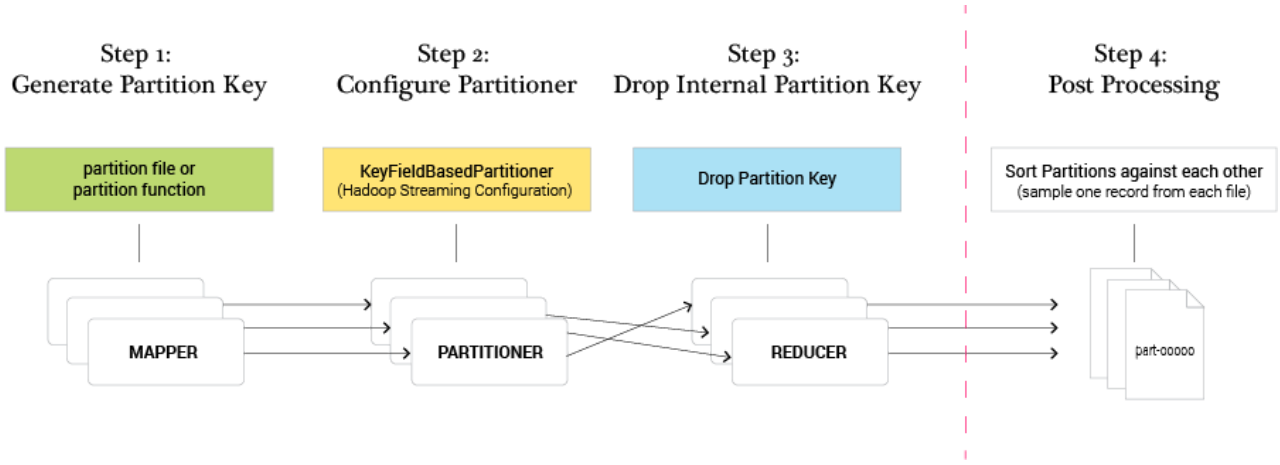


Figure 2. Total Order sort with multiple reducers

Multiple Reducer Overview

Back to top | Back to Section II

After the Map phase and before the beginning of the Reduce phase there is a handoff process, known as shuffle and sort. Output from the mapper tasks is prepared and moved to the nodes where the reducer tasks will be run. To improve overall efficiency, records from mapper output are sent to the physical node that a reducer will be running on as they are being produced - to avoid flooding the network when all mapper tasks are complete. What this means is that when we have more than one reducer in a MapReduce job, Hadoop no longer sorts the keys globally (total sort). Instead mapper outputs are partitioned while they're being produced, and before the reduce phase starts, records are sorted by the key within each partition. In other words, Hadoop's architecture only guarantees partial sort. Therefore, to achieve total sort, a

programmer needs to incorporate additional steps and supply the Hadoop framework additional aid during the shuffle and sort phase. Particularly, a partition file or partition function is required.

Version 1 - A two step approach

Step 1: Include a partition file or partition function inside mappers

Recall that we can use an identity mapper in single-reducer step up, which just echos back the (key, value) pair from input data. In a multi-reducer setup, we will need to add an additional "partition key" to instruct Hadoop how to partition records, and pass through the original (key, value) pair.

The partition key is derived from the input key, with the help of either a partition file (more on this in the sampling section) or a user-specified partition function, which takes a key as input, and produces a partition key. Different input keys can result in same partition key.

Now we have two keys (as opposed to just one), and one is used for partitioning, the other is used for sorting. The reducer needs to drop the partition key which is used internally to aid total sort, and recover the original (key, value) pairs.

Step 2: Post-processing step to order partitions

The MapReduce job output is written to HDFS, with the output from each partition in a separate file (usually named something such as: part-00000). These file names are indexed and ordered. However, Hadoop makes no attempt to sort partition keys – the mapping between partition key and partition index is not order-preserving. Therefore, while partition keys key_1, key_2 and key_1 < key_2, it's possible that the output of partition with key_1 could be written to file part-00006 and the output of partition with key_2 written to file part-00003.

Therefore, a post-processing step is required to finish total sort. Next, we'll provide an example of using linux to order the partition files. We will need to take one record from every (non-empty) partition output, sort them, and construct the appropriate ordering among partitions.

Introductory Example:

Back to top | Back to Section II

Consider the task of sorting English words alphabetically, for example, four words from our example dataset:

```
experiements
def
descent
compute
```

The expected sorted output is:

```
compute
def
descent
experiements
```

We can use the first character from each word as a partition key. The input data could potentially have billions of words, but we will never have more than 26 unique partition keys (assuming all words are lower-cased). In addition, a word starting with "a" will always have a lower alphabetical ordering compared to a word which starts with "z". Therefore, all words belonging to partition "a" will be "pre-sorted" against all words from partition "z". The technique described here is equivalent to the following partition function:

```
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 partition function is not valid (in sorting words in alphabetical order):

```
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:

```
In [177…  Image(filename="Partition2.png")
```

Out[177…

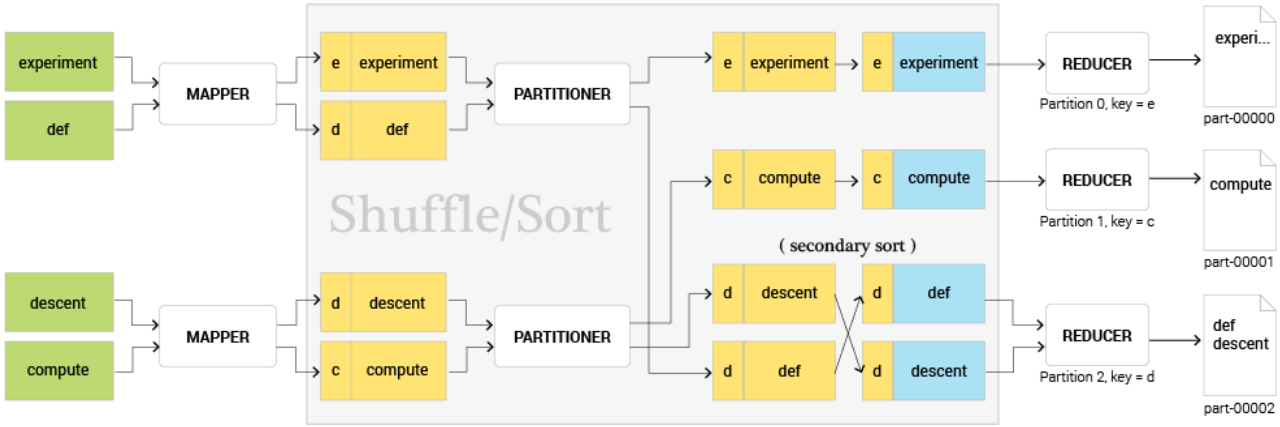


Figure 3. Partial Order Sort

Note that partition key "e" maps to partition 0, even if it is "greater than" key "d" and "c". This illustrates that the mapping between partition key and partition indices are not order preserving. In addition, sorting within partitions is based on the original key (word itself).

Implementation Walkthrough

Back to top | Back to Section II

Coming back to our original dataset, here we will sort and print the output in two steps. Step one will partition and sort the data, and step two will arrange the partitions in the appropriate order. In the implementation that follows, we'll build on this and demonstrate how to ensure that the partitions are created in the appropriate order to begin with.

1. Run the Hadoop command that prepends an alphabetic key to each row such that they end up in the appropriate partition, then shuffle and sort.
2. Combine the secondary sort output files in the appropriate order

Setup

The following options comprise the Hadoop Streaming configuration for the sort job. Notice the addition of the keypartitioner option, which tells Hadoop Streaming to partition by the primary key. Remember that the order of the options is important.

```
-D stream.num.map.output.key.fields=3 \
-D stream.map.output.field.separator="\t" \
-D mapreduce.partition.keypartitioner.options=-k1,1 \
-D
mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapred.lib.Ke
 \
-D mapreduce.partition.keycomparator.options="-k1,1 -k2,2nr -k3,3"
\
```

Here, "-k1,1 -k2,2nr -k3,3" performs secondary sorting. Our three key fields are:

- `-k1,1` : partition key, one of $\{A, B, C\}$ (this part is optional, since each partition will contain the same partition key)
- `-k2,2nr` : input key (number/count), and we specify `nr` flags to sort them numerically reverse
- `-k3,3` : input value (word), if two records have same count, we break the tie by comparing the words alphabetically

Function to prepend an alphabetic key to each row such that they end up in the appropriate partition

The following mapper is an identity mapper with a partition function included, it prepends an alphabetic key as partition key to input records. The letters A, B, and C are arbitrary, and as we'll see, do not correspond to any ordering of partition files.

In [310…

```
%%writefile prependPartitionKeyMapper.py
#!/usr/bin/env python
import sys
for line in sys.stdin:
    line = line.strip()
    key, value = line.split("\t")
```

```
    if int(key) < 10:
        print "%s\t%s\t%s" % ("A", key, value)
    elif int(key) < 20:
        print "%s\t%s\t%s" % ("B", key, value)
    else:
        print "%s\t%s\t%s" % ("C", key, value)
```

```
Overwriting prependPartitionKeyMapper.py
```

Step 1 - run the hadoop command specifying 3 reducers, the partition key, and the sort keys

In [18]:
```
!hdfs dfs -rm -r {HDFS_DIR}/sort/secondary_sort_output
!hadoop jar {JAR_FILE} \
    -D stream.num.map.output.key.fields=3 \
    -D stream.map.output.field.separator="\t" \
    -D mapreduce.partition.keypartitioner.options=-k1,1 \
    -D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapred.lib.Ke
    -D mapreduce.partition.keycomparator.options="-k1,1 -k2,2nr -k3,3" \
    -mapper prependPartitionKeyMapper.py \
    -reducer /bin/cat \
    -file prependPartitionKeyMapper.py \
    -input {HDFS_DIR}/sort/generate_numbers.output \
    -output {HDFS_DIR}/sort/secondary_sort_output \
    -numReduceTasks 3 \
    -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

```
rm: `/user/root/TOS/sort/secondary_sort_output': No such file or directory
20/05/19 17:35:01 WARN streaming.StreamJob: -file option is deprecated, please u
se generic option -files instead.
packageJobJar: [prependPartitionKeyMapper.py] [/usr/lib/hadoop-mapreduce/hadoop-
streaming-2.6.0-cdh5.16.2.jar] /tmp/streamjob5288394612382812866.jar tmpDir=null
20/05/19 17:35:02 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.
0:8032
20/05/19 17:35:02 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.
0:8032
20/05/19 17:35:03 INFO mapred.FileInputFormat: Total input paths to process : 1
20/05/19 17:35:03 INFO mapreduce.JobSubmitter: number of splits:2
20/05/19 17:35:03 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_15
89906177930_0003
20/05/19 17:35:03 INFO impl.YarnClientImpl: Submitted application application_15
89906177930_0003
20/05/19 17:35:03 INFO mapreduce.Job: The url to track the job: http://docker.w2
61:8088/proxy/application_1589906177930_0003/
20/05/19 17:35:03 INFO mapreduce.Job: Running job: job_1589906177930_0003
20/05/19 17:35:09 INFO mapreduce.Job: Job job_1589906177930_0003 running in uber
mode : false
20/05/19 17:35:09 INFO mapreduce.Job:  map 0% reduce 0%
20/05/19 17:35:15 INFO mapreduce.Job:  map 50% reduce 0%
20/05/19 17:35:16 INFO mapreduce.Job:  map 100% reduce 0%
20/05/19 17:35:21 INFO mapreduce.Job:  map 100% reduce 33%
20/05/19 17:35:22 INFO mapreduce.Job:  map 100% reduce 67%
20/05/19 17:35:23 INFO mapreduce.Job:  map 100% reduce 100%
20/05/19 17:35:23 INFO mapreduce.Job: Job job_1589906177930_0003 completed succe
ssfully
20/05/19 17:35:23 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=529
                FILE: Number of bytes written=751857
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=794
```

```
                HDFS: Number of bytes written=451
                HDFS: Number of read operations=15
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=6
        Job Counters
                Launched map tasks=2
                Launched reduce tasks=3
                Data-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=6166
                Total time spent by all reduces in occupied slots (ms)=10753
                Total time spent by all map tasks (ms)=6166
                Total time spent by all reduce tasks (ms)=10753
                Total vcore-milliseconds taken by all map tasks=6166
                Total vcore-milliseconds taken by all reduce tasks=10753
                Total megabyte-milliseconds taken by all map tasks=6313984
                Total megabyte-milliseconds taken by all reduce tasks=11011072
        Map-Reduce Framework
                Map input records=30
                Map output records=30
                Map output bytes=451
                Map output materialized bytes=547
                Input split bytes=252
                Combine input records=0
                Combine output records=0
                Reduce input groups=30
                Reduce shuffle bytes=547
                Reduce input records=30
                Reduce output records=30
                Spilled Records=60
                Shuffled Maps =6
                Failed Shuffles=0
                Merged Map outputs=6
                GC time elapsed (ms)=227
                CPU time spent (ms)=3980
                Physical memory (bytes) snapshot=1241333760
                Virtual memory (bytes) snapshot=6911700992
                Total committed heap usage (bytes)=1525153792
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=542
        File Output Format Counters
                Bytes Written=451
20/05/19 17:35:23 INFO streaming.StreamJob: Output directory: /user/root/TOS/sor
t/secondary_sort_output
```

Check the output

In [19]:

```
!hdfs dfs -ls {HDFS_DIR}/sort/secondary_sort_output
print ("="*100)
print ("/part-00000")
print ("="*100)
!hdfs dfs -cat {HDFS_DIR}/sort/secondary_sort_output/part-00000
print ("="*100)
print( "/part-00001")
print ("="*100)
!hdfs dfs -cat {HDFS_DIR}/sort/secondary_sort_output/part-00001
print ("="*100)
```

```
print ("/part-00002")
print ("="*100)
!hdfs dfs -cat {HDFS_DIR}/sort/secondary_sort_output/part-00002
```

```
Found 4 items
-rw-r--r--   1 root supergroup          0 2020-05-19 17:35 /user/root/TOS/sort/s
econdary_sort_output/_SUCCESS
-rw-r--r--   1 root supergroup         76 2020-05-19 17:35 /user/root/TOS/sort/s
econdary_sort_output/part-00000
-rw-r--r--   1 root supergroup        217 2020-05-19 17:35 /user/root/TOS/sort/s
econdary_sort_output/part-00001
-rw-r--r--   1 root supergroup        158 2020-05-19 17:35 /user/root/TOS/sort/s
econdary_sort_output/part-00002
================================================================================
====================
/part-00000
================================================================================
====================
B       19      efficient
B       17      computing
B       17      document
B       16      done
B       12      during
================================================================================
====================
/part-00001
================================================================================
====================
C       30      center
C       30      corresponding
C       30      driver
C       29      cell
C       29      drivers
C       28      contour
C       28      descent
C       28      develop
C       26      dataset
C       24      consists
C       24      evaluate
C       23      clustering
C       23      experiements
C       20      def
================================================================================
====================
/part-00002
================================================================================
====================
A       9       computational
A       8       do
A       4       compute
A       4       descent
A       3       current
A       2       change
A       2       cluster
A       1       creating
A       1       distributed
A       0       code
A       0       different
```

Step 2 - Combine the sorted output files in the appropriate order

The following code block peaks at the first line of each partition file to determine the order of partitions, and prints the contents of each partition in order, from largest to smallest. Notice

that, while the files are arranged in total order, the partition file names are not ordered. We'll tackle this issue in the next section.

In [ ]:

```
# The subprocess module allows you to spawn new (system) processes, connect to t
# and obtain their return codes. Ref: https://docs.python.org/2/library/subproce
import subprocess
import re

'''
subprocess.Popen()
Opens a new subprocess and executes the unix command in the args array, passing
This is the equivalent of typing:
hdfs dfs -ls {HDFS_DIR}/sort/secondary_sort_output/part-*
in the unix shell prompt
Even though we cannot see this output it would look like this:
-rw-r--r--   1 koza supergroup        141 2016-08-20 19:25 /user/root/TOS/sort/s
-rw-r--r--   1 koza supergroup        164 2016-08-20 19:25 /user/root/TOS/sort/s
-rw-r--r--   1 koza supergroup        118 2016-08-20 19:25 /user/root/TOS/sort/s
'''

p = subprocess.Popen(["hdfs", "dfs", "-ls", "/user/root/TOS/sort/secondary_sort_
                      stdout=subprocess.PIPE, stderr=subprocess.STDOUT)


'''
Save the output of the above command to the 'lines' string variable by reading e
The resulting lines string should look like this:

'-rw-r--r--   1 koza supergroup        141 2016-08-20 19:25 /user/root/TOS/sort/

'''
lines=""
for line in p.stdout.readlines():
    lines = lines + str(line)


'''
The following regular expresion extracts the paths from 'lines', and appends eac
The resulting outputPARTFiles list should look like this:

['/user/root/TOS/sort/secondary_sort_output/part-00000',
 '/user/root/TOS/sort/secondary_sort_output/part-00001',
 '/user/root/TOS/sort/secondary_sort_output/part-00002']

'''
regex = re.compile('(\/user\/root\/TOS\/sort\/secondary_sort_output\/part-\d*)')
it = re.finditer(regex, lines)

outputPARTFiles=[]
for match in it:
    outputPARTFiles.append(match.group(0))

'''
Next is where we peek at the first line of each file and extract the key. The re
[19, 30, 9]

For each file f in outputPARTFiles
    int(...)                          <-- this will convert the key returned b
    ["hdfs", "dfs", "-cat", f]        <-- cat the file
    stdout=subprocess.PIPE            <-- to STDOUT
```

```
    stdout.read()                       <-- read the STDOUT into memory
    decode()                            <-- convert from bytes to string (for py
    splitlines()[0]                     <-- split that output into lines, and re
    split('\t')[1]                      <-- split that first line by tab charact
    strip()                             <-- remove trailing and leading spaces s

'''
partKeys=[]
for f in outputPARTFiles:
    partKeys.append(int(subprocess.Popen(["hdfs", "dfs", "-cat", f], stdout=subp


'''
create a dict d assoicating each key with its corresponding file path. The resul

{ 9: '/user/root/TOS/sort/secondary_sort_output/part-00002',
 19: '/user/root/TOS/sort/secondary_sort_output/part-00000',
 30: '/user/root/TOS/sort/secondary_sort_output/part-00001'}

 ^^ we now know that the largest key lives in part-00001, and we will display th

'''
d={}
for i in range(len(outputPARTFiles)):
    print ("part is %d, key is %d, %s" %(i, partKeys[i], outputPARTFiles[i]))
    d[partKeys[i]] = outputPARTFiles[i]

'''
Print the contents of each file in total sorted order, by sorting the d dict by

    sorted(d.items(), key=lambda x: x[0], reverse=True)   <-- sorts d dict by ke
    print "%d:%s"%(k[0], k[1])                            <-- print the key k[0]

    use a subprocess to read the contents of each file listed in d (k[1]) (see e
    and print each line omitting leading and trailing spaces
'''

#TOTAL Sort in decreasing order
for k in sorted(d.items(), key=lambda x: x[0], reverse=True):
    print ("="*100)
    print ("%d:%s"%(k[0], k[1]))
    print ("="*100)
    p = subprocess.Popen(["hdfs", "dfs", "-cat", k[1]],  stdout=subprocess.PIPE,
    for line in p.stdout.readlines():
        print (line.decode().strip())
```

II.C.3 Hadoop Streaming Implementation - multiple reducers with ordered partitions

Back to top

Keypoints:

- Challenge: which partition contains the largest/smallest values seems arbitrary
- Hadoop streaming `KeyFieldBasedPartitioner` does not sort partition keys, even though it seemingly accepts Unix `sort` compatible configurations

Solution:

- Understand the inner working of `HashPartitioner` and `KeyFieldBasedPartitioner`
- Relationship between `KeyFieldBasedPartitioner` and `HashPartitioner`:
  * `KeyFieldBasedPartitioner` applies `HashPartitioner` on configured key field(s)
- Create the inverse function of `HashPartitioner` and assign partition keys accordingly

Understanding HashPartitioner

Back to top | Back to Section 3

By default, Hadoop uses a library class `HashPartitioner` to compute the partition index for keys produced by mappers. It has a method called `getPartition`, which takes `key.hashCode() & Integer.MAX_VALUE` and finds the modulus using the number of reduce tasks. For example, if there are 10 reduce tasks, `getPartition` will return values 0 through 9 for all keys.

```
    // HashPartitioner

    partitionIndex = (key.hashCode() & Integer.MAX_VALUE) % numReducers
```

In the land of native hadoop applications (written in Java or JVM languages), keys can be any object type that is hashable (i.e. implements hashable interface). For Hadoop Streaming, however, keys are always string values. Therefore the hashCode function for strings is used:

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

When we configure Hadoop Streaming to use KeyBasedPartitioner, the process is very similar. Hadoop Streaming will parse command line options such as -k2,2 into key specs, and extract the part of the composite key (in this example, field 2 of many fields) and read in the partition key as a string. For example, with the following configuration:

```
    "stream.map.output.field.separator" : ".",
    "mapreduce.partition.keycomparator.options": "-k2,2",
```

Hadoop will extract '**a**' from a composite key **2.a.4** to use as the partition key.

The partition key is then hashed (as string) by the same `hashCode` function, its modulus using number of reduce tasks yields the partition index.

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:

In [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"]
```

Out[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
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".

Total Order Sort with ordered partitions - illustrated

In [8]:
```
Image(filename="tos-step1.png")
```

Out[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

1a

keys = A B C ← For 3 reducers, generate 3 (arbitrary) keys

[key, hash(key,r)]

indexed keys = A 2 B 0 C 1 ← For each key, generate indexes by taking the hash of the key modulo number of reducers.

[makeIndex(key,r)]
(returns hash(key) % r)

/part-00001

sort the keys by the index

partition_keys = B C A
0 1 2

1b

partition_file = 20 10 0
[partition(r,N)]

The partition file contains the split points for our example data set, where the values range from 0-30. For simplicity, we'll assume a uniform distribution (more about that in the sampling section..).

In [6]:
```
Image(filename="tos-step2.png")
```

Out[6]:

# STEP 2
mapper()

2a. Yield with partition key based on the value to be sorted by
2b. Shuffle/Sort - Secondary sort on the value

2a

```
for idx in partition_file:
    if key > partition_file[ idx ]:
        yield (partition_keys[ idx ] , key, value)
        break
```

Example:
partition_file: `[20, 10, 0]`
partition_keys: `[B, C, A]`
input data: `(19 evaluate)`

since `19 > partition_file[1]`, and `partition_keys[1] == C` then we yield:

(C 19 evaluate)

2b

part-00000		
B	30	do
B	28	dataset
B	27	driver
.	..	...
B	22	correspo..

part-00001		
C	19	consists
C	19	evaluate
C	17	drivers
.	..	...
C	13	efficient

part-00002		
A	10	clustering
A	9	change
A	9	during
.	..	...
A	1	cell

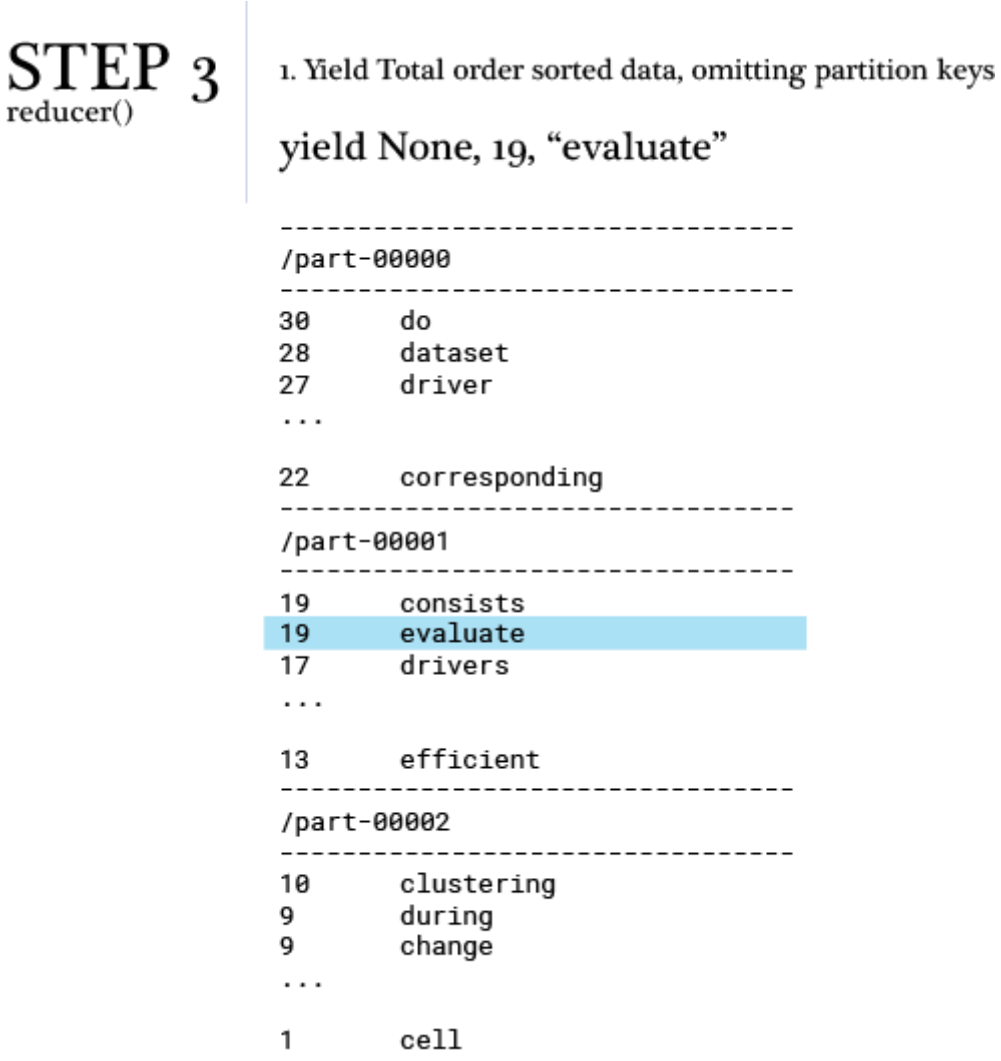
shuffle by the new primary key (partition key)

secondary sort on the original key

optionally, we can break ties by specifying additional secondary sorting on the value (ie, "consists" comes before "evaluate")

In [111…
```
Image(filename="tos-step3.png")
```

Out[111…



Implementation

Back to top | Back to Section 3

Version 2 - The final Total Order Sort with ordered partitions

What's New

The solution we will delve into is very similar to the one discussed earlier in the Hadoop Streaming section. The only addition is Step 1B, where we take a desired partition index and craft a custom partition key such that Hadoop's KeyFieldBasedPartitioner hashes it back to the correct index.

In [179…
```
Image("TotalSortStepsComplete.png")
```

Out[179…

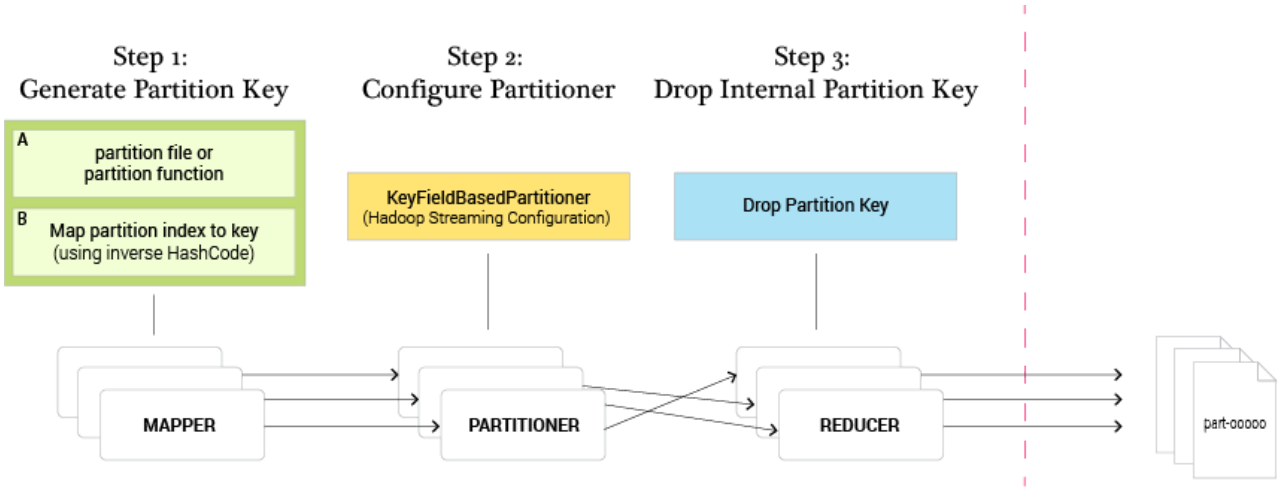


Figure 6. Total order sort with custom partitioning.

In [21]:

```
%%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


if os.getenv('mapreduce_job_reduces') == None:
    N = 1
else:
    N = int(os.getenv('mapreduce_job_reduces'))

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()

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
```

```
Overwriting multipleReducerTotalOrderSort_mapper.py
```

In [22]:

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

for line in sys.stdin:
    pkey, key, value = line.strip().split('\t')
    print(key,value)
```

```
Overwriting multipleReducerTotalOrderSort_reducer.py
```

In [23]:

```
!hdfs dfs -rm -r {HDFS_DIR}/sort/total_order_sort
!hadoop jar {JAR_FILE} \
    -D stream.num.map.output.key.fields=3 \
    -D stream.map.output.field.separator="\t" \
    -D mapreduce.partition.keypartitioner.options=-k1,1 \
    -D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapred.lib.Ke
    -D mapreduce.partition.keycomparator.options="-k2,2nr -k3,3" \
    -files multipleReducerTotalOrderSort_mapper.py,multipleReducerTotalOrderSort
    -mapper multipleReducerTotalOrderSort_mapper.py \
    -reducer multipleReducerTotalOrderSort_reducer.py \
    -input {HDFS_DIR}/sort/generate_numbers.output \
    -output {HDFS_DIR}/sort/total_order_sort \
    -numReduceTasks 3 \
    -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \
    -cmdenv PATH={PATH}
```

```
rm: `/user/root/TOS/sort/total_order_sort': No such file or directory
packageJobJar: [] [/usr/lib/hadoop-mapreduce/hadoop-streaming-2.6.0-cdh5.16.2.ja
r] /tmp/streamjob3403073017835678931.jar tmpDir=null
20/05/19 18:19:46 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.
0:8032
20/05/19 18:19:47 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.
0:8032
20/05/19 18:19:47 INFO mapred.FileInputFormat: Total input paths to process : 1
20/05/19 18:19:47 INFO mapreduce.JobSubmitter: number of splits:2
20/05/19 18:19:47 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_15
89906177930_0005
```

```
20/05/19 18:19:47 INFO impl.YarnClientImpl: Submitted application application_15
89906177930_0005
20/05/19 18:19:47 INFO mapreduce.Job: The url to track the job: http://docker.w2
61:8088/proxy/application_1589906177930_0005/
20/05/19 18:19:47 INFO mapreduce.Job: Running job: job_1589906177930_0005
20/05/19 18:19:53 INFO mapreduce.Job: Job job_1589906177930_0005 running in uber
mode : false
20/05/19 18:19:53 INFO mapreduce.Job:  map 0% reduce 0%
20/05/19 18:20:01 INFO mapreduce.Job:  map 50% reduce 0%
20/05/19 18:20:02 INFO mapreduce.Job:  map 100% reduce 0%
20/05/19 18:20:07 INFO mapreduce.Job:  map 100% reduce 33%
20/05/19 18:20:08 INFO mapreduce.Job:  map 100% reduce 67%
20/05/19 18:20:09 INFO mapreduce.Job:  map 100% reduce 100%
20/05/19 18:20:09 INFO mapreduce.Job: Job job_1589906177930_0005 completed succe
ssfully
20/05/19 18:20:09 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=498
                FILE: Number of bytes written=755320
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=794
                HDFS: Number of bytes written=368
                HDFS: Number of read operations=15
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=6
        Job Counters
                Launched map tasks=2
                Launched reduce tasks=3
                Data-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=6397
                Total time spent by all reduces in occupied slots (ms)=10867
                Total time spent by all map tasks (ms)=6397
                Total time spent by all reduce tasks (ms)=10867
                Total vcore-milliseconds taken by all map tasks=6397
                Total vcore-milliseconds taken by all reduce tasks=10867
                Total megabyte-milliseconds taken by all map tasks=6550528
                Total megabyte-milliseconds taken by all reduce tasks=11127808
        Map-Reduce Framework
                Map input records=30
                Map output records=28
                Map output bytes=424
                Map output materialized bytes=516
                Input split bytes=252
                Combine input records=0
                Combine output records=0
                Reduce input groups=28
                Reduce shuffle bytes=516
                Reduce input records=28
                Reduce output records=28
                Spilled Records=56
                Shuffled Maps =6
                Failed Shuffles=0
                Merged Map outputs=6
                GC time elapsed (ms)=181
                CPU time spent (ms)=3920
                Physical memory (bytes) snapshot=1246547968
                Virtual memory (bytes) snapshot=6891323392
                Total committed heap usage (bytes)=1459093504
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
```

```
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=542
        File Output Format Counters
                Bytes Written=368
20/05/19 18:20:09 INFO streaming.StreamJob: Output directory: /user/root/TOS/sor
t/total_order_sort
```

In [24]:

```
print ("="*100)
print ("Total Order Sort with multiple reducers - notice that the part files are
print ("="*100)
print ("/part-00000")
print ("-"*100)
!hdfs dfs -cat {HDFS_DIR}/sort/total_order_sort/part-00000
print ("-"*100)
print ("/part-00001")
print ("-"*100)
!hdfs dfs -cat {HDFS_DIR}/sort/total_order_sort/part-00001
print ("-"*100)
print ("/part-00002")
print ("-"*100)
!hdfs dfs -cat {HDFS_DIR}/sort/total_order_sort/part-00002
```

```
====================================================================================
====================
Total Order Sort with multiple reducers - notice that the part files are also in
order.
====================================================================================
====================
/part-00000
------------------------------------------------------------------------------------
--------------------
30  center
30  corresponding
30  driver
29  cell
29  drivers
28  contour
28  descent
28  develop
26  dataset
24  consists
24  evaluate
23  clustering
23  experiements
------------------------------------------------------------------------------------
--------------------
/part-00001
------------------------------------------------------------------------------------
--------------------
20  def
19  efficient
17  computing
17  document
16  done
12  during
------------------------------------------------------------------------------------
--------------------
/part-00002
------------------------------------------------------------------------------------
--------------------
9  computational
8  do
```

```
4  compute
4  descent
3  current
2  change
2  cluster
1  creating
1  distributed
```

Total Order Sort results

We now have exactly what we were looking for. Total Order Sort, with the added benefit of ordered partitions. Notice that the top results are stored in part-00000, the next set of results is stored in part-00001, etc..

Section III - Sampling Key Spaces

Back to top

Previously, we used a partition file which assumed our key space was uniformly distributed. In reality this is rarely the case, and we should make our partition file based on the actual distribution of the data to avoid bottlenecks. A bottleneck would occur if the majority of our data resided in a single bucket, as could happen with a typical power law distribution.

Consider the following example:

In [15]:
```
import random
import numpy as np

# Create a skewed distribution
# make 750,000 numbers between 0-10
# make 200,000 numbers between 10-20
# make 50,000 numbers between 20-30

A = np.random.uniform(0,10,750000)
B = np.random.uniform(10,20,200000)
C = np.random.uniform(20,30,50000)

ALL = np.hstack((A,B,C))
print(ALL.shape)
```

```
(1000000,)
```

In [119…
```
# Visualizae data distribution
%matplotlib inline
import pylab as pl
fig, ax = pl.subplots(figsize=(10,6))

ax.hist(ALL,color="#48afe0",edgecolor='none')

xcoords = [10,20,30]
for xc in xcoords:
    pl.axvline(x=xc,color="#197f74", linewidth=1)

ax.spines['top'].set_visible(False)
```

```
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)


pl.title("3 Uniformly Spaced Buckets")
pl.show()
```

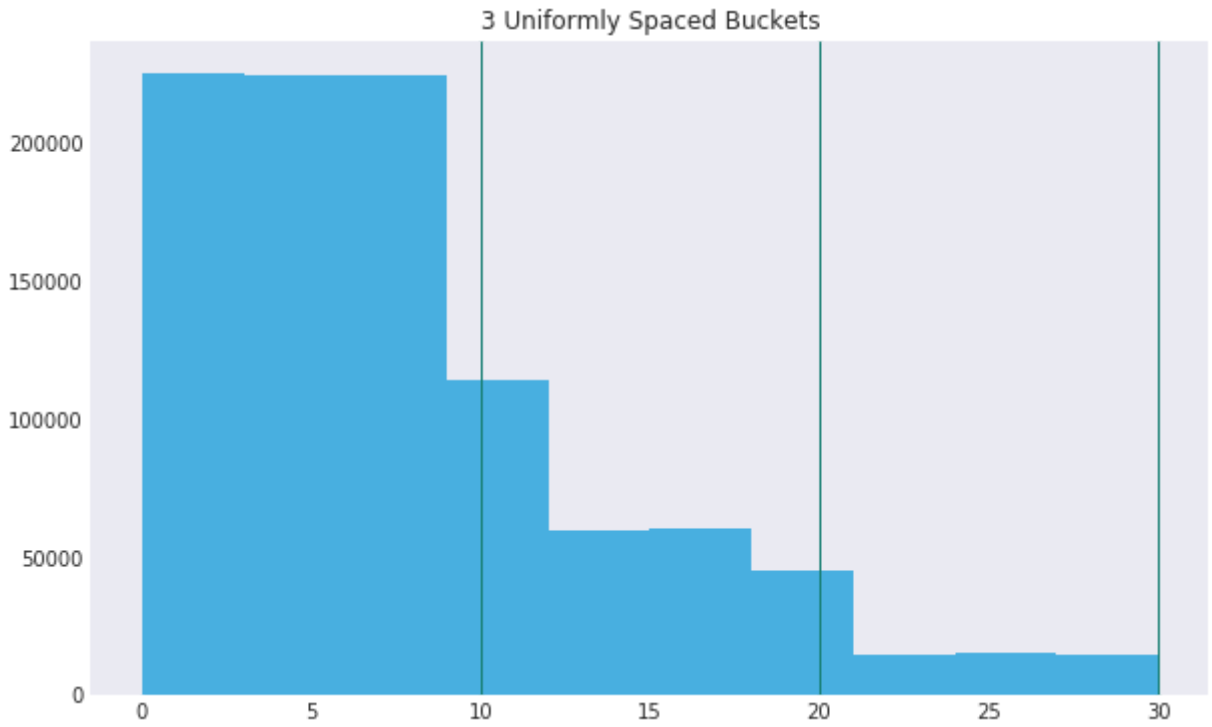


Figure 7. Uniform partitions over skewed data.

If we made a uniform partition file the (above example has 3 buckets), we would end up with most of the data in a single reducer, and this would create a bottle neck. To fix this, we must create a partition file that distributes the keys more evenly.

The challenge is that when our dataset set is large, creating the partition file can be costly. But as long as we know the shape of the distribution, we don't need the whole dataset to create resonable partitions. Next we'll look at two ways to take a random sample of a large datset.

Key points:

1. **Simple Random Sampling** - Easy implementation when we know the total size of the data
2. **Reservoir Sampling** - A method to sample the data with equal probablity for all data points when the size of the data is unknown. The algorithm works as follows:

```
n = desired sample size
reservoir = []
for d in data
    if reservoir size < n
        add d to reservoir
    else:
        choose random location in reservoir
        flip coin whether to replace the exisiting d with new d
```

  (This paper has a nice explanation of reservoir sampling, see: 2.2 Density-Biased Reservoir Sampling http://science.sut.ac.th/mathematics/pairote/uploadfiles/weightedkm-temp2_EB.pdf.)

III.A. Random Sample implementation

Back to top

In [17]:
```
# write numpy aray to file
np.savetxt('randomNumbers.out', ALL, fmt='%.18f')
```

In [18]:
```
# put file in HDFS
!hdfs dfs -rm -r {HDFS_DIR}/sort/random_numbers
!hdfs dfs -put randomNumbers.out {HDFS_DIR}/sort/random_numbers
```

```
rm: `/user/root/TOS/sort/random_numbers': No such file or directory
```

In [19]:
```
%%writefile RandomSample.py
#!/usr/bin/env python

import sys
import numpy as np

##########################################################
#  Emit a random sample of 1/100th of the data
##########################################################

for line in sys.stdin:
    s = np.random.uniform(0,1)
    if s < .01:
        print(line.strip())
```

```
Overwriting RandomSample.py
```

In [20]:
```
!hdfs dfs -rm -r {HDFS_DIR}/sort/sampleData
!hadoop jar {JAR_FILE} \
    -files RandomSample.py \
    -mapper RandomSample.py \
    -input {HDFS_DIR}/sort/random_numbers \
    -output {HDFS_DIR}/sort/sampleData \
    -numReduceTasks 0 \
    -cmdenv PATH={PATH}
```

```
rm: `/user/root/TOS/sort/sampleData': No such file or directory
packageJobJar: [] [/usr/lib/hadoop-mapreduce/hadoop-streaming-2.6.0-cdh5.16.2.jar] /tmp/streamjob2041614838548135011.jar tmpDir=null
20/05/21 10:56:47 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
20/05/21 10:56:47 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
20/05/21 10:56:48 INFO mapred.FileInputFormat: Total input paths to process : 1
20/05/21 10:56:48 INFO mapreduce.JobSubmitter: number of splits:2
20/05/21 10:56:49 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1590009070447_0002
20/05/21 10:56:50 INFO impl.YarnClientImpl: Submitted application application_1590009070447_0002
20/05/21 10:56:50 INFO mapreduce.Job: The url to track the job: http://docker.w261:8088/proxy/application_1590009070447_0002/
```

```
20/05/21 10:56:50 INFO mapreduce.Job: Running job: job_1590009070447_0002
20/05/21 10:57:02 INFO mapreduce.Job: Job job_1590009070447_0002 running in uber
mode : false
20/05/21 10:57:02 INFO mapreduce.Job:  map 0% reduce 0%
20/05/21 10:57:13 INFO mapreduce.Job:  map 100% reduce 0%
20/05/21 10:57:13 INFO mapreduce.Job: Job job_1590009070447_0002 completed succe
ssfully
20/05/21 10:57:14 INFO mapreduce.Job: Counters: 30
        File System Counters
                FILE: Number of bytes read=0
                FILE: Number of bytes written=297514
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=21254330
                HDFS: Number of bytes written=223326
                HDFS: Number of read operations=10
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=4
        Job Counters
                Launched map tasks=2
                Data-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=15289
                Total time spent by all reduces in occupied slots (ms)=0
                Total time spent by all map tasks (ms)=15289
                Total vcore-milliseconds taken by all map tasks=15289
                Total megabyte-milliseconds taken by all map tasks=15655936
        Map-Reduce Framework
                Map input records=1000000
                Map output records=10034
                Input split bytes=234
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=127
                CPU time spent (ms)=3700
                Physical memory (bytes) snapshot=422944768
                Virtual memory (bytes) snapshot=2779865088
                Total committed heap usage (bytes)=452984832
        File Input Format Counters
                Bytes Read=21254096
        File Output Format Counters
                Bytes Written=223326
20/05/21 10:57:14 INFO streaming.StreamJob: Output directory: /user/root/TOS/sor
t/sampleData
```

In [21]:
```
!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/s
ampleData/_SUCCESS
-rw-r--r--   1 root supergroup     111430 2020-05-21 10:57 /user/root/TOS/sort/s
ampleData/part-00000
-rw-r--r--   1 root supergroup     111896 2020-05-21 10:57 /user/root/TOS/sort/s
ampleData/part-00001
```

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

In [23]:
```
!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
```

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

```
28.039338755785088608
25.345290806645600412
26.379113474443627751
29.636694380239692492
26.722319640328059620
29.877321118626916530
28.115086968783938204
24.174899488857747087
21.898206840176925425
26.753814102971556821
```

III.A. Reservoir Sample implementation

Back to top

Review of Reservoir Sampling

Reservoir sampling ensures that each datum in the stream will be sampled with the same equal probability, and this is property makes the process true random sampling.

Let N be the unknown length of the stream, and suppose we desire a random sample of size $n < N$. Furthermore, let R denote the set of element indices which are in the reservoir at the end of the sampling process.

`Description` : To begin, the first n elements, elements $1, 2, 3 \ldots$ up to element n get put in the reservoir in the form of a list. Next, the algorithm iterates through each element i that comes after the n^{th} element and considers putting it in the reservoir. When the algorithm is considering whether to place an element i with $i > n$ in the reservoir, the algorithm takes a random index J_i between 1 and i inclusively, that is $J_i \sim Unif\{1, \ldots, i\}$. If $J_i \leq n$ then the currrent J_i^{th} element of reservoir list is swapped with element i. If $J_i > n$ the algorithm doesn't do any swapping and proceeds to consider element $i + 1$.

`Claim` : Each element in the stream has chance $\frac{n}{N}$ of being sampled (this means $k \in R$ by the end), thus making the sample a truly random sample.

`Proof` : Consider the k^{th} element of the stream.

> Case 1: Suppose that $k \in \{1, 2, \ldots, n\}$. So element k is one of the first elements to fill the reservoir to begin with. For element k to end up in the final sample, element k must not be replaced by any incoming elements of the stream. Let's find the chance that by the end of the process, k is in the sample, $\mathbb{P}(k \in R)$:

$$
\begin{aligned}
\mathbb{P}(k \in R) &= \cap_{i=n+1}^{N} \mathbb{P}(J_i \neq k) \\
&= \cap_{i=n+1}^{N} \frac{i-1}{i} \\
&= \frac{(n)(n+1)(n+2)\cdots(N-1)}{(n+1)(n+2)\cdots(N)} \\
&= \frac{n}{N}
\end{aligned}
$$

> Case 2: Suppose that $k > n$. So k is not in the reservoir to begin with. When element k is being considered for the reservoir, it must be swapped with some element that is currently in the reservoir in order to have k end up in the final sample. After k has replaced another element in the reservoir, element k must not be replaced by any element that follows it.

$$
\begin{aligned}
\mathbb{P}(k \in R) &= \mathbb{P}(J_k \leq n) \cap_{i=k+1}^{N} \mathbb{P}(J_i \neq k) \\
&= \frac{n}{k} \cap_{i=k+1}^{N} \frac{i-1}{i} \\
&= \frac{n}{k} \cdot \frac{(k)(k+1)(k+2)\cdots(N-1)}{(k+1)(k+2)\cdots(N)} \\
&= \frac{n}{k} \cdot \frac{k}{N} \\
&= \frac{n}{N}
\end{aligned}
$$

The following code block creates a small data set for unit/systems testing:

In [140…
```
%%writefile numberPopulationTest.txt
3
55
6
4
10
2
1
67
3
20
9
19
11
0
88
```

```
Overwriting numberPopulationTest.txt
```

Naive Implementation

In its most basic form, a reservoir sampling algorithm keeps a list of datapoints (the reservoir) in memory. Unfrotunately, this version doesn't scale to multiple mappers, and thus is not fit for purpose. Here it is just for reference:

In [143…
```
%%writefile ReservoirSample.py
#!/usr/bin/env python
```

```
import sys
import numpy as np

########################################################
#  Sample n numbers uniformly at random
########################################################

print(sys.argv)

R = 5 # size of the reservoir
res = [] # reservoir

for i, line in enumerate(sys.stdin):
    num = line.strip()
    if len(res) < R:
        res.append(num)
    else:
        j = np.random.randint(i)
        if j < R:
            res[j] = num

# print
for j in res:
    print(j)
```

```
Overwriting ReservoirSample.py
```

In [145…

```
# Systems test
!python ReservoirSample.py < numberPopulationTest.txt
```

```
['ReservoirSample.py']
11
20
6
2
10
```

Parallel & Scalable Implementation Using Hadoop MapReduce

To implement reservoir sampling in a scalable way, we need to avoid holding the reservoir in memory. The core insight behind reservoir sampling is that picking a random sample of size R is equivalent to generating a random permutation (ordering) of the elements and picking the top R elements.

In the implementation below, leveraging secondary sort on random variate `i`, we can gaurantee that the top R elements are chosen by the reducers.

This can be done by first mapping over each data point `n` and emitting:

1. the data point itself, `n`
2. the index `j` in the reservoir which `n` would occupy (if it were chosen to be swapped into the reservoir)
3. the index `i` of the data point in the stream

Note that if `i` is less than the sample size, the data point should be automatically added to the reservoir at index `i`. So for these numbers, the mapper should emit `n`, `i`, `i`.

Next the reducers need to determine which data point will occupy the `j` location (for every location `j` in the reservoir) from among the data points that were assigned to swap into the `j` position in the reservoir. In the original algorithm, the data point that ultimately occupies the `j` position is the last data point in the stream that is assigned to swap into the `j` position. Therefore, in the reduce phase, the data point which gets printed for the `j` position should be, from among the data points with reservoir index `j`, the data point with the highest `i`, because this data point would have been the last element in the original stream.

In order for the reducers to be able to achieve this in a stateless manner, the intermediate key-value pairs should be partitioned by `j`, and then they should be sorted primarily by `j` and secondarily by `i` within each partition.

An implementation of the required mapper and reducer for scalable and parallizable reservoir sampling is below:

In [146…
```
%%writefile ReservoirSample_mapper.py
#!/usr/bin/env python
"""
INPUT:
    data point
    R -> size of the reservoir
OUTPUT:
    data point \t reservoirIndex(j) \t streamIndex(i)
"""
import os
import sys
import numpy as np


if os.getenv('R') == None:
    R = 3
else:
    R = int(os.getenv('R'))

for i, line in enumerate(sys.stdin):
    num = line.strip()

    if i < R:
        print(f'{num}\t{i}\t{i}')
    else:
        j = np.random.randint(i)
        if j < R:
            print(f'{num}\t{j}\t{i}')
```

```
Overwriting ReservoirSample_mapper.py
```

In [147…
```
%%writefile ReservoirSample_reducer.py
#!/usr/bin/env python

"""
INPUT:
    data point \t reservoirIndex(j) \t streamIndex(i)
OUTPUT:
    data point

Requirement: the inputs must be sorted with the
following option: -k2,2 -k3,3n. This way all of
```

```
records associated with position j in the reservoir
are consecutive and the one that will occupy position
j in the reservoir at the end is the one with the
highest stream index (the one that came last in
the original stream)
"""


import sys

current_j = None
current_num = None

for i, line in enumerate(sys.stdin):
    num, j, i = line.strip().split('\t')

    if current_j == j:
        current_num = num
    else:
        if current_j is not None:
            print(current_num)
        current_j = j
        current_num = num

print(current_num)
```

```
Overwriting ReservoirSample_reducer.py
```

In [148…
```
!chmod +x ReservoirSample_mapper.py
!chmod +x ReservoirSample_reducer.py
```

In [180…
```
!cat numberPopulationTest.txt | ./ReservoirSample_mapper.py | sort -k2,2n -k3,3n
```

In [181…
```
!cat mapper.out
```

```
3       0       0
10      0       4
2       0       5
20      0       9
55      1       1
3       1       8
6       2       2
4       2       3
```

In [182…
```
!cat mapper.out | ./ReservoirSample_reducer.py
```

```
20
3
4
```

In [183…
```
!hdfs dfs -rm -r {HDFS_DIR}/sort/resSampleData
!hadoop jar {JAR_FILE} \
    -D stream.num.map.output.key.fields=3 \
    -D stream.map.output.field.separator="\t" \
    -D mapreduce.partition.keypartitioner.options=-k2,2n \
    -D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapred.lib.Ke
    -D mapreduce.partition.keycomparator.options="-k2,2n -k3,3n" \
    -files ReservoirSample_mapper.py,ReservoirSample_reducer.py \
    -mapper ReservoirSample_mapper.py \
    -reducer ReservoirSample_reducer.py \
    -input {HDFS_DIR}/sort/random_numbers \
    -output {HDFS_DIR}/sort/resSampleData \
```

```
    -numReduceTasks 4 \
    -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \
    -cmdenv R=1000 \
    -cmdenv PATH={PATH}
```

```
Deleted /user/root/TOS/sort/resSampleData
packageJobJar: [] [/usr/lib/hadoop-mapreduce/hadoop-streaming-2.6.0-cdh5.16.2.ja
r] /tmp/streamjob7123320903685861172.jar tmpDir=null
20/05/26 14:08:03 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.
0:8032
20/05/26 14:08:03 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.
0:8032
20/05/26 14:08:04 INFO mapred.FileInputFormat: Total input paths to process : 1
20/05/26 14:08:04 INFO mapreduce.JobSubmitter: number of splits:2
20/05/26 14:08:04 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_15
90009070447_0047
20/05/26 14:08:05 INFO impl.YarnClientImpl: Submitted application application_15
90009070447_0047
20/05/26 14:08:05 INFO mapreduce.Job: The url to track the job: http://docker.w2
61:8088/proxy/application_1590009070447_0047/
20/05/26 14:08:05 INFO mapreduce.Job: Running job: job_1590009070447_0047
20/05/26 14:08:14 INFO mapreduce.Job: Job job_1590009070447_0047 running in uber
mode : false
20/05/26 14:08:14 INFO mapreduce.Job:  map 0% reduce 0%
20/05/26 14:08:28 INFO mapreduce.Job:  map 50% reduce 0%
20/05/26 14:08:29 INFO mapreduce.Job:  map 100% reduce 0%
20/05/26 14:08:38 INFO mapreduce.Job:  map 100% reduce 25%
20/05/26 14:08:40 INFO mapreduce.Job:  map 100% reduce 50%
20/05/26 14:08:41 INFO mapreduce.Job:  map 100% reduce 100%
20/05/26 14:08:42 INFO mapreduce.Job: Job job_1590009070447_0047 completed succe
ssfully
20/05/26 14:08:42 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=476364
                FILE: Number of bytes written=1857040
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=21254330
                HDFS: Number of bytes written=22356
                HDFS: Number of read operations=18
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=8
        Job Counters
                Killed reduce tasks=1
                Launched map tasks=2
                Launched reduce tasks=4
                Data-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=22567
                Total time spent by all reduces in occupied slots (ms)=30004
                Total time spent by all map tasks (ms)=22567
                Total time spent by all reduce tasks (ms)=30004
                Total vcore-milliseconds taken by all map tasks=22567
                Total vcore-milliseconds taken by all reduce tasks=30004
                Total megabyte-milliseconds taken by all map tasks=23108608
                Total megabyte-milliseconds taken by all reduce tasks=30724096
        Map-Reduce Framework
                Map input records=1000000
                Map output records=14206
                Map output bytes=447928
                Map output materialized bytes=476388
                Input split bytes=234
                Combine input records=0
                Combine output records=0
                Reduce input groups=14206
```

```
                Reduce shuffle bytes=476388
                Reduce input records=14206
                Reduce output records=1000
                Spilled Records=28412
                Shuffled Maps =8
                Failed Shuffles=0
                Merged Map outputs=8
                GC time elapsed (ms)=343
                CPU time spent (ms)=10670
                Physical memory (bytes) snapshot=1347829760
                Virtual memory (bytes) snapshot=8285179904
                Total committed heap usage (bytes)=1423441920
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=21254096
        File Output Format Counters
                Bytes Written=22356
20/05/26 14:08:42 INFO streaming.StreamJob: Output directory: /user/root/TOS/sor
t/resSampleData
```

In [184…
```
!hdfs dfs -ls {HDFS_DIR}/sort/resSampleData/
```

```
Found 5 items
-rw-r--r--   1 root supergroup          0 2020-05-26 14:08 /user/root/TOS/sort/r
esSampleData/_SUCCESS
-rw-r--r--   1 root supergroup       5599 2020-05-26 14:08 /user/root/TOS/sort/r
esSampleData/part-00000
-rw-r--r--   1 root supergroup       5610 2020-05-26 14:08 /user/root/TOS/sort/r
esSampleData/part-00001
-rw-r--r--   1 root supergroup       5572 2020-05-26 14:08 /user/root/TOS/sort/r
esSampleData/part-00002
-rw-r--r--   1 root supergroup       5575 2020-05-26 14:08 /user/root/TOS/sort/r
esSampleData/part-00003
```

In [185…
```
!hdfs dfs -cat {HDFS_DIR}/sort/resSampleData/part-* > resSampleData.txt
```

In [186…
```
!head -n 5 resSampleData.txt
```

```
5.574918167903911481
1.571505454171698846
0.752479853907217500
11.253497108987883024
13.504271654126098312
```

In [187…
```
!wc -l resSampleData.txt
```

```
1000 resSampleData.txt
```

III.B. Custom partition file implementation

Back to top

Percentile Based Partitioning

Once we have a (small) sampled subset of data, we can compute partition boundaries by examining the distribution of this subset, and finding approporiate percentiles based on number of desired partitions. A basic implementation using NumPy is provided below:

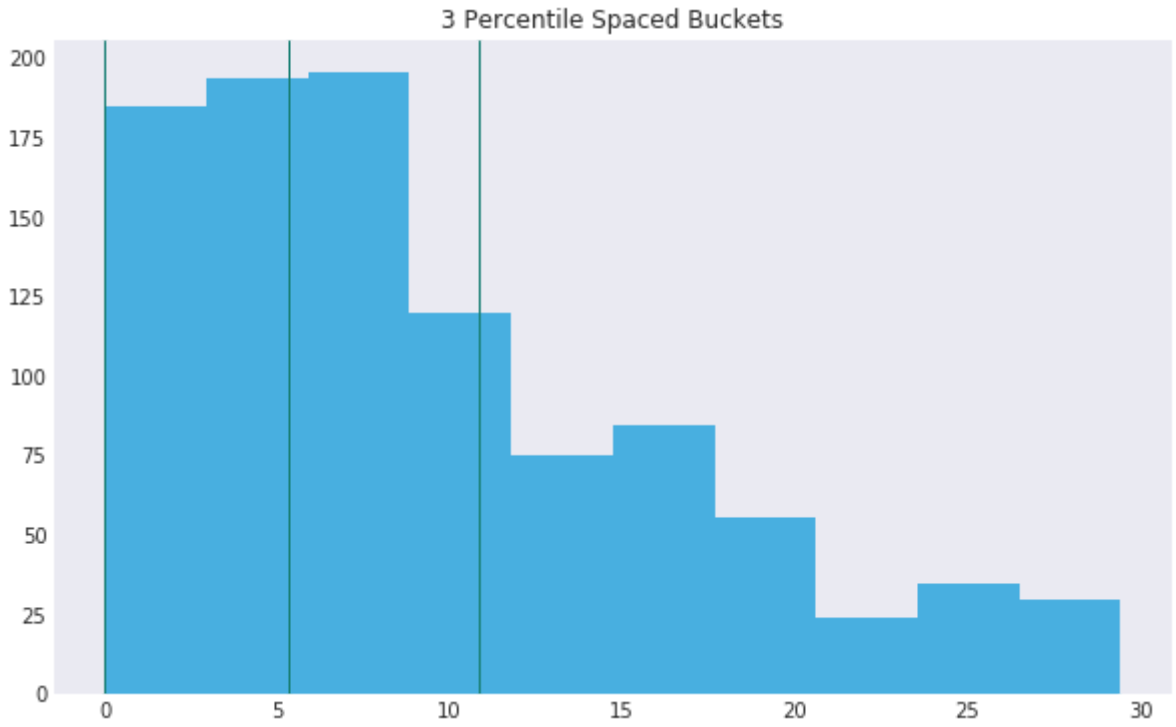
In [188…
```
def readSampleData(filePath):
    # A sample of the data is stored in a single file in sampleData/part-00000
    # from the previous step (RandomSample.py)
    sampleData = []

    with open(filePath,"r") as f:
        lines = f.readlines()
        for line in lines:
            line = line.strip().split("\t")[0]
            sampleData.append(float(line))

    return sampleData
```

In [189…
```
from numpy import array, percentile, linspace, random

def partition(data, num_of_partitions=3, return_percentiles=False):
    # remove percentile 100
    qs = linspace(0, 100, num=num_of_partitions, endpoint=False)
    if not return_percentiles:
        return percentile(data, qs)
    return percentile(data, qs), qs
```

In [190…
```
sampleData = readSampleData('resSampleData.txt')
partitionFile, percentiles = partition(sampleData, 3, return_percentiles=True)
```

Visualize Partition

In [191…
```
# Visualizae Partition File
%matplotlib inline
import pylab as pl

num_buckets = 3

fig, ax = pl.subplots(figsize=(10,6))

ax.hist(sampleData,color="#48afe0",edgecolor='none')

xcoords = partitionFile
for xc in xcoords:
    pl.axvline(x=xc,color="#197f74", linewidth=1)

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)


pl.title(str(num_buckets)+" Percentile Spaced Buckets")
pl.show()

print ("Sample Data min", min(sampleData))
print ("Sample Data max", max(sampleData))
print ("Partition file", partitionFile)
print ("Percentiles", percentiles)
```




```
Sample Data min 0.001046762963181624
Sample Data max 29.43125395253612
Partition file [1.04676296e-03 5.30686396e+00 1.08147038e+01]
Percentiles [ 0.         33.33333333 66.66666667]
```

Figure 8. Percentile based partitions over skewed data.

We could now replace the hard-coded partition file in our Total Order Sort implementation above with a custom file built from a sample of the data.

Final Remarks

Back to top

A note on TotalSortPartitioner: Hadoop has built in TotalSortPartitioner, which uses a partition file _partition.lst to store a pre-built order list of split points.TotalSortPartitioner uses binary search / Trie to look up the ranges a given record falls into.

https://github.com/facebookarchive/hadoop-20/blob/master/src/mapred/org/apache/hadoop/mapred/lib/TotalOrderPartitioner.java

References

Back to top

1. http://wiki.apache.org/hadoop/
2. http://hadoop.apache.org/docs/stable1/streaming.html#Hadoop+Streaming
3. http://www.theunixschool.com/2012/08/linux-sort-command-examples.html
4. https://hadoop.apache.org/docs/r2.7.2/hadoop-streaming/HadoopStreaming.html

5. http://hadoop.apache.org/
6. https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/SingleCluster.html
7. https://hadoop.apache.org/docs/r1.2.1/streaming.html#Hadoop+Comparator+Class
8. http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-steps.html
9. https://github.com/apache/hadoop/blob/2e1d0ff4e901b8313c8d71869735b94ed8bc40a0/had mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/lib/partition/KeyFieldBasedPartitioner.java
10. http://science.sut.ac.th/mathematics/pairote/uploadfiles/weightedkm-temp2_EB.pdf
11. http://spark.apache.org/
12. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf
13. https://spark.apache.org/docs/1.6.2/programming-guide.html#working-with-key-value-pairs
14. https://github.com/facebookarchive/hadoop-20/blob/master/src/mapred/org/apache/hadoop/mapred/lib/TotalOrderPartitioner.java