A Comprehensive Guide to the Total Order Sort design pattern in MapReduce

James G. Shanahan, Kyle Hamilton, Yiran Sheng

University of California, Berkeley

Abstract

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, Hadoop Streaming, MRJob and Spark all of which generally operate on data records consisting of key-value pairs. These frameworks, 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 such as Hadoop, Hadoop Streaming and MrJob; Spark works a little differently and will be discussed separately later. 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 these MapReduce frameworks. 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 the following MapReduce frameworks: Hadoop Streaming, MrJob, and Spark. 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, MapReduce, Spark

A Comprehensive Guide To the Total Order Sort design pattern in MapReduce

In this notebook we are going to demonstrate how to achieve Total Order Sort via three MapReduce frameworks: Hadoop Streaming, MRJob, and Spark. Hadoop Streaming and MRJob borrow 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 and MRJob. 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.

Lastly, we'll provide a Spark version of Total Order Sort.

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 MRJob implementation.

Anatomy of a MapReduce Job

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)

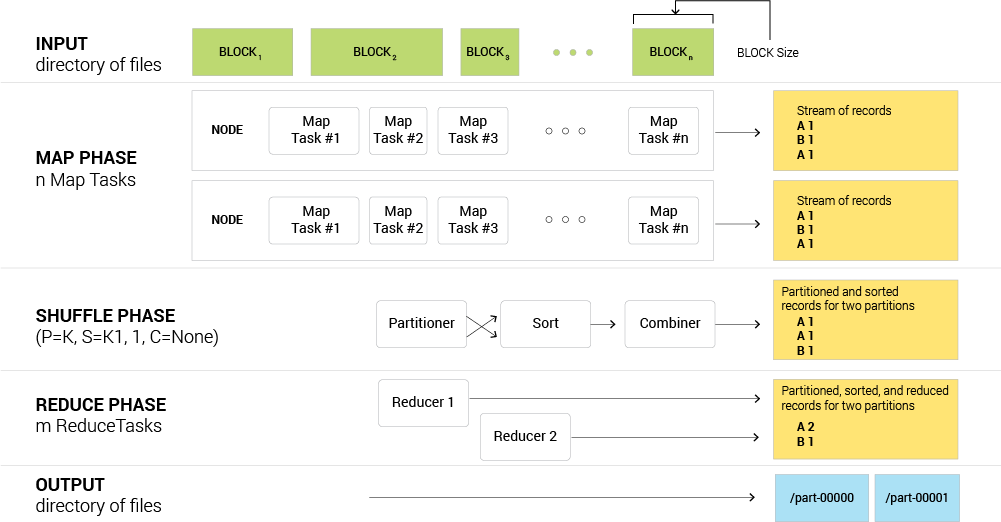
where there is no ordering between reducer values (V1, V2, ..., Vn).

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.

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

Terminology

**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.

**MRJob** is a Python library developed by Yelp to simplify writing Map/Reduce programs. It allows developers to test their code locally without installing Hadoop or run it on a cluster of choice. It also has extensive integration with Amazon Elastic Map Reduce. More information is available at <http://mrjob.readthedocs.io/en/latest/index.html>.

**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, Hadoop Streaming and MRJob.

**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 MapReduce 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*

Hadoop Streaming is a utility that comes with the Hadoop distribution. The utility allows a user to create and run MapReduce jobs with any executable or script as the mapper and/or the reducer.

[INSERT CODE-BLOCK-1]

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)*

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}.

[INSERT CODE-BLOCK-2]

Keys are assigned to buckets according to their numeric value. The result is that all keys between 20-30 end up in one bucket, keys between 10-20 end up in another bucket, and keys 0-10 end up yet in another bucket. Keys are sorted within each bucket. Here, partitions are assigned in sorted order, such that keys between 20-30 end up in the first bucket, keys between 10-20 end up in the second bucket, and keys 0-10 end up in the third bucket. We use the term buckets and partitions interchangeably.

Prepare Dataset

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.

[INSERT CODE\_BLOCK-3]

Section I - Understanding Unix Sort

*Importance of Unix Sort*

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*

# 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*

[INSERT CODE\_BLOCK-4]

[INSERT TABLE-TABS-2]

Section II - Hadoop Streaming

*II.A. Hadoop's Default Sorting Behavior*

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*

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:

[INSERT TABLE-1]

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*

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

-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*

Specify number of key fields and key field seperator:

-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*

Specify sorting options

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

This part is very straightforward. Whatever one would do with Unix sort (e.g. -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*

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 (Partitioning in MRJob), 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*

[INSERT CODE\_BLOCK-5]

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](https://hadoop.apache.org/docs/r2.7.2/hadoop-streaming/HadoopStreaming.html) for more information (Hadoop version = 2.7.2.)

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

[INSERT TABLE-2]

*II.C. Hadoop Streaming Implementation*

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

*Start Hadoop*

To run the examples in this notebook you must download and configure Hadoop on your local computer. Go to http://hadoop.apache.org/ for the latest downloads.

Everything you need to get up and running can be found on this page: https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/SingleCluster.html. There are also many websites with specialized instructions.

Once all components have been downloaded and installed, you can check that everything is running by running the `jps` command in your terminal. You should see output like this:

localhost:~ $ jps

83360 NodeManager

82724 DataNode

82488 NameNode

82984 SecondaryNameNode

83651 Jps

83118 ResourceManager

83420 JobHistoryServer

This notebook runs on the following setup:

Mac OX Yosemite 10.10.5

Java version "1.7.0\_51"

Hadoop version 2.7.2

[INSERT CODE\_BLOCK-6]

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

Key points:

* Single reducer guarantees a single partition
* Partial sort becomes total sort
* No need for secondary sorting
* Single Reducer becomes scalability 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*

[INSERT CODE\_BLOCK-7]

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>.)

[INSERT CODE\_BLOCK-8]

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

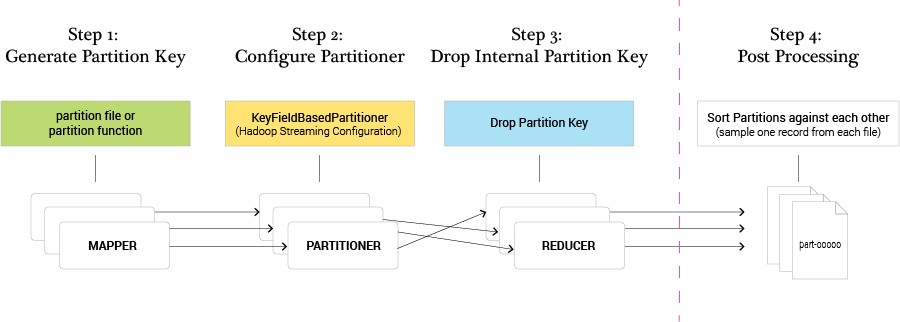
Key points:

* 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:



*Figure 2. Total Order sort with multiple reducers*

*Multiple Reducer Overview*

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.

Modification 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.

Modification 2: Drop internal partition key inside reducers

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.

Modification 3: 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. We will need to take one record from every (non-empty) partition output, sort them, and construct the appropriate ordering among partitions.

*Introductory Example*

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

experiments

def

descent

compute

The expected sorted output is:

compute

def

descent

experiments

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 lowercase). 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:

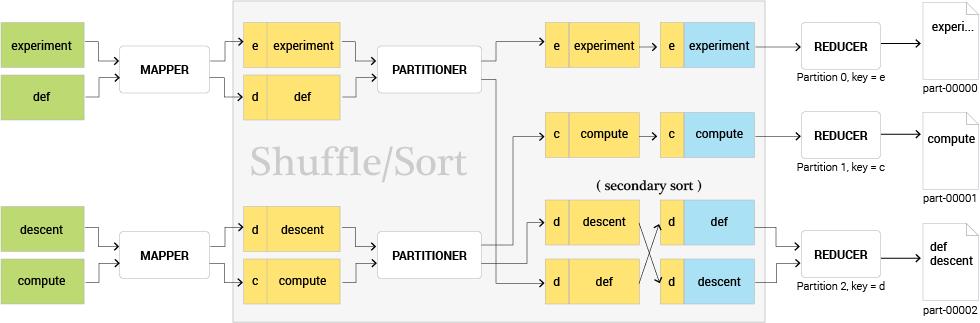
e experiments

d def

d descent

c compute

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



*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*

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 MRJob 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, 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.

[IMSERT CODE\_BLOCK-9]

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

* -k1,1: partition key, one of {A,B,C}{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.

The following mapper is an identity mapper with a partition function included; it prepends an alphabetic key as partition key to input records.

[INSERT CODE\_BLOCK-10]

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

[INSERT CODE-BLOCK-11]

Check the output

[INSERT CODE-BLOCK-12]

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 MRJob implementation.

[INSERT CODE-BLOCK-13]

Section III - MRJob

For this section you will need the MRJob Python library. For installation instructions, go to: <https://github.com/Yelp/mrjob>

We'll first discuss a couple of key aspects of MRJob such as modes, protocols, and partitioning, before diving into the implementation. We'll also provide an illustrated example of partitioning.

*III.A. MRJob Modes*

MRJob has three modes that correspond to different Hadoop environments.

*Local mode*

Local mode simulates Hadoop Streaming, but does not require an actual Hadoop installation. This is great for testing out small jobs. However, local mode does not support '-k2,2nr' type of sorting, i.e. sorting by numeric value, as it is not capable of Hadoop .jar library files (such as KeyBasedComparator). A workaround is to make sure numbers are converted to strings with a fixed length, and sorted by reverse order of their values. For positive integers, this can be done by: sys.maxint – value. We include Local Mode implementation for completeness (see below).

*Hadoop mode*

MRJob is capable of dispatching runners in a environment where Hadoop is installed. User-authored MRJob Python files are treated as shell scripts, and submitted to Hadoop Streaming as MapReduce jobs. MRJob allows users to specify configurations supported by Hadoop Streaming via the jobconf dictionary, either as part of MRStep or MRJob itself (which will be applied to all steps). The Python dictionary is serialized into command line arguments, and passed to the Hadoop Streaming .jar file. (See https://python hosted.org/mrjob/guides/configs-hadoopy-runners.html for further documentation of Hadoop mode).

*EMR/Dataproc mode*

In addition, MRJob supports running MapReduce jobs on a vendor-provided Hadoop runtime environment such as AWS Elastic MapReduce or Google Dataproc. The configuration and setup is very similar to Hadoop mode (-r hadoop) with the following key differences:

* Vendor-specific credentials (such as AWS key and secret).
* Vendor-specific bootstrap process (for instance, configure Python version, and install third party libraries upon initialization).
* Use platform's native "step" management process (e.g. AWS Steps).
* Use vendor-provided data storage and transportation (e.g. use S3 for input/output on EMR).

The api surface for -r emr and -r hadoop are almost identical, but the performance profiles can be drastically different.

*III.B. MRJob Protocols*

At a high level, a Protocol is a gateway between the Hadoop Streaming world, and the MRJob/Python world. It translates raw bytes (text) into (key, value) pairs as some Python data structure, and vice versa. An MRJob protocol has the following interface: it is a class with a pair of functions read (which converts raw bytes to (key, value) pairs) and write (which converts (key, value) pairs back to bytes/text):

[INSERT CODE-BLOCK-14]

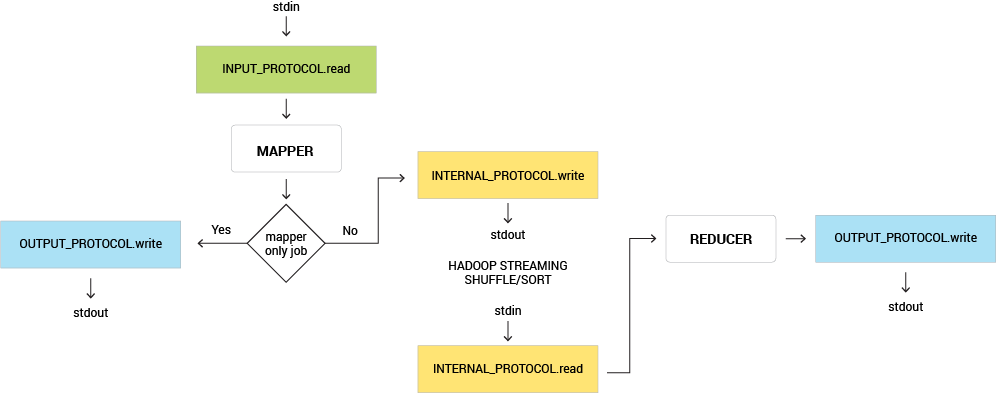
In addition, MRJob further abstracts away the differences between Python 2 and Python 3 (primarily in areas such as Unicode handling), and provides a unified interface across most Hadoop versions and Python versions.

When data enters MRJob components (mapper, reducer, combiner), the Protocol's read method (specified in MRJob class) is invoked, and supplies the (key, value) pair to the component. When data exits MRJob components, its Protocol's write method is invoked, converting (key, value) pair output from the component to raw bytes / text.

Consider a most generic MRJob job, consisting of one mapper step and one reduce step:

[INSERT CODE-BLOCK-15]

There are four contact points between Python scripts and Hadoop Streaming, which require three Protocols to be specified, as illustrated below:



*Figure 4. Protocols diagram*

*III.B.1 Types of Built-in Protocols*

MRJob provides a number of built-in protocols, all of which can all be used for INPUT\_PROTOCOL, INTERNAL\_PROTOCOL or OUTPUT\_PROTOCOL. By default, MRJob uses RawValueProtocol for INPUT\_PROTOCOL, and JSONProtocol for INTERNAL\_PROTOCOL and OUTPUT\_PROTOCOL.

The table below lists four commonly used Protocols and their signature. Some key observations are:

* RawProtocol and RawValueProtocol do not attempt to serialize and deserialize text data.
* JSONProtocol and JSONValueProtocol use the JSON encoder and decoder to convert between text (stdin/stdout) and Python data structures (Python runtime).
* Value Protocols always treat key as None and do not auto insert a tab character in their write method.

[INSERT TABLE-TABS-3]

The following example further illustrates the behaviors of these protocols: a single line of text input (single line text file containing: 1001 {"value":10}) is fed to four different single step MapReduce jobs. Each uses one of the four Protocols discussed above as INPUT\_PROTOCOL, INTERNAL\_PROTOCOL and OUTPUT\_PROTOCOL.



*Figure 5. Lifecycle of a single record.*

Corresponding protocols code examples:

[INSERT TABLE-TABS-4]

*III.B.2 Importance of RawProtocols in Context of Total Sorting*

While working with MRJob, it is critical to understand that MRJob is an abstraction layer above Hadoop Streaming, and does not extend or modify Hadoop Streaming's normal behaviors. Hadoop streaming works through Unix piping, and uses stdout and stderr as communication channels. In Unix environments, the protocol of data transfer is text streams – mappers, reducers and Hadoop libraries exchange data through text.

Therefore, data structures common in programming languages (e.g. dictionary in Python ) have to be serialized to text for Hadoop Streaming to understand and consume. The implication of this on total sorting is this: since Hadoop only sees text, custom data structures (as keys) will be sorted as strings in their serialized forms.

MRJob abstracts away the data serialization and deserialization processes by the concept of protocols. For instance, JSONProtocol is capable of serializing complex data structures, and is used by default as INTERNAL\_PROTOCOL. This frees the user from messy string manipulation and processing, but it does come with a cost.

One example is Python tuples:

# mapper

yield 1, ("value", "as", "tuple", True)

Inside the reducer however, when values are deserialized into Python objects, the (key, value) pair above will become:

1, ["value", "as", "tuple", True]

Namely, tuples become lists, due to the fact JSON does not support tuples.

Another example related to sorting: suppose mappers emit custom data structures as composite keys, such as:

# mapper

yield [12, 1, 2016]

Assuming we want to use hadoop streaming's KeyFieldBasedComparator, when hadoop streaming sees the JSON string [2016, 12, 1], it is very difficult to instruct hadoop to understand the key consists of three fields (month, day, year), and sort them accordingly (e.g. we want sort by year, month, day).

Notice if we yield a date as [2016, 12, 01] instead, the string comparison coincidentally produces the same sort order as the underlying data (dates). However, to achieve this requires intimate knowledge of JSON (or other transportation formats); therefore, we recommend using RawProtocol or RawValueProtocol when advanced sorting is desired, which offers maximum flexibility to control key sorting and key hashing performed by Hadoop Streaming.

*III.C. Partitioning in MRJob*

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.
* MRJob uses Hadoop Streaming under the hood, therefore inherits the same problem

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*

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:

[INSERT CODE-BLOCK-16]

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 a string) by the same hashCode function, its modulus using a number of reduce tasks yields the partition index.

(See KeyBasedPartitioner [source code](https://github.com/apache/hadoop/blob/2e1d0ff4e901b8313c8d71869735b94ed8bc40a0/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/lib/partition/KeyFieldBasedPartitioner.java) for the actual implementations.)

*Inverse HashCode Function*

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 a partition key. This key, when supplied to the Hadoop framework (KeyBasedPartitioner), will hash to the correct partition index.

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

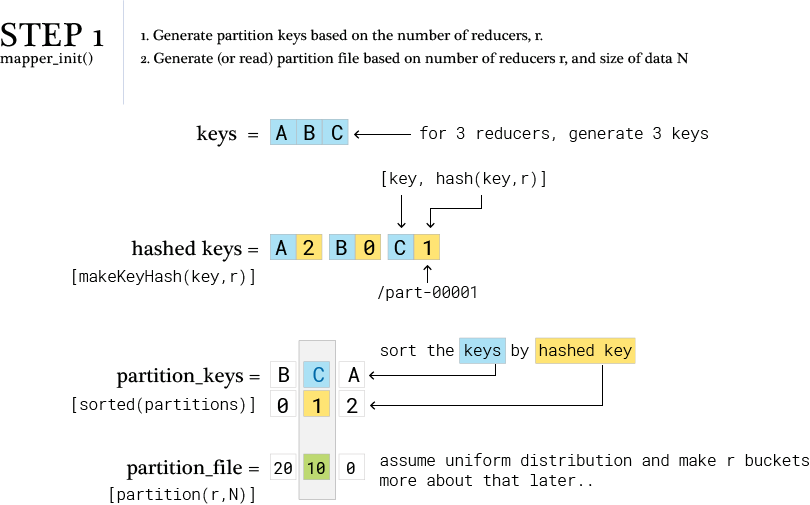
[INSERT CODE-BLOCK-17]

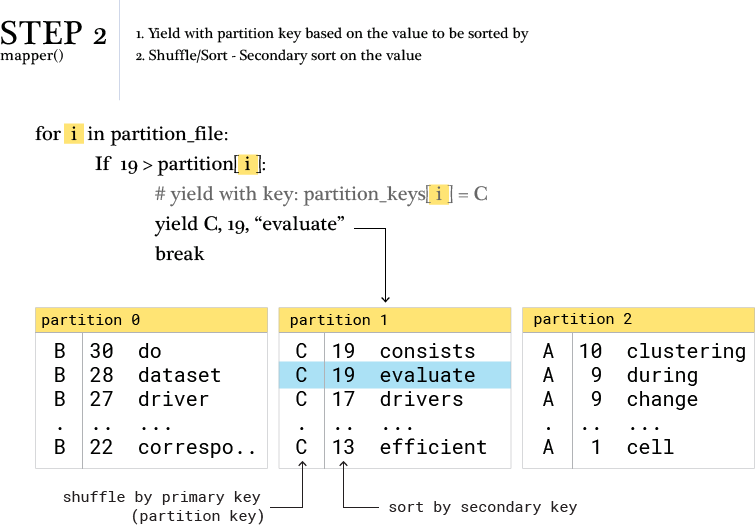
A simple strategy to implement an inverse hashCode function is to use a lookup table. For example, assuming we have 3 reducers, we can compute the partition index with makeKeyHash for keys "A", "B", and "C". The results are listed the the table below.

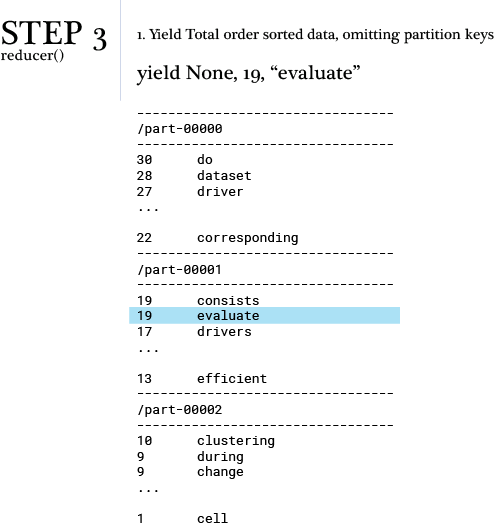
[INSERT TABLE-3]

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*







*III.D. MRJob Implementations*

III.D.1 MRJob implementation - single reducer - local mode

[INSERT CODE-BLOCK-18]

III.D.2. MRJob implementation - single reducer - Hadoop mode

[INSERT CODE\_BLOCK-19]

III.D.3. MRJob Multiple Reducers - With Un-Ordered Partiton

[INSERT CODE\_BLOCK-20]

Without special partitioning, the output is not organized by partition number.

(i.e., part-00000 does not contain the "A" key)

[CODE-BLOCK-21]

III.D.4. MRJob Multiple Reducers - With Ordered Partitons

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.



*Figure 6. Total order sort with custom partitioning.*

[INSERT CODE-BLOCK-22]

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., because we hashed the keys (A,B,C) to those file names.

Section IV - Sampling Key Spaces

Keypoints:

* Random Sampling - Easy implementation when we know the total size of the data.
* Reservoir Sampling - A method to sample the data with equal probability 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

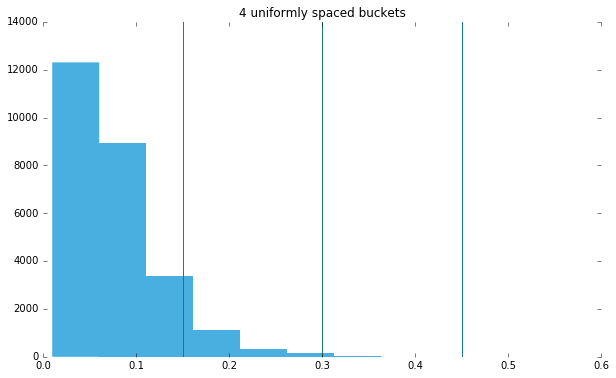
flip coin whether to replace the existing d with new d

(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>.)

Consider the following example in which we assumed a uniform distribution of the data. For simplicity, we made our partition file based on that assumption. 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:

[INSERT CODE\_BLOCK-23]



*Figure 7. Uniform partitions over skewed data.*

If we made a uniform partition file the (above example has 4 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 first generate a sample of our data; then, based on this sample, create a partition file that distributes the keys more evenly.

*IV.A. Random Sample implementation*

[INSERT CODE\_BLOCK-24]

*IV.B. Custom partition file implementation*

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 find appropriate percentiles based on the number of desired partitions. A basic implementation using NumPy is provided below:

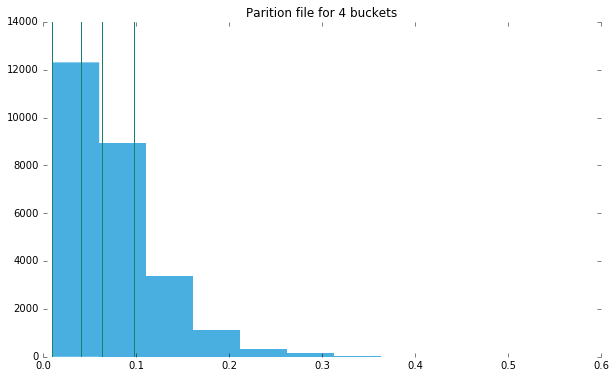
[INSERT CODE\_BLOCK-25]

Raw implementation without using NumPy

[INSERT CODE\_BLOCK-26]

Visualize Partition

[INSERT CODE\_BLOCK-27]



*Figure 8. Percentile based partitions over skewed data.*

Section V - Spark implementation

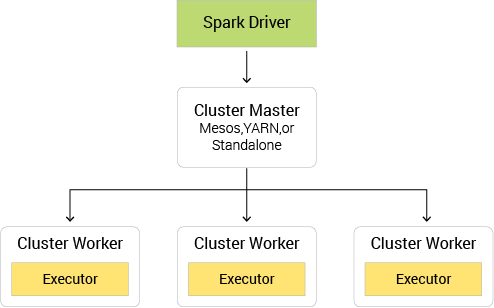
For this section, you will need to install Spark. We are using a local installation, version 2.0

[INSERT CODE-BLOCK-28]

SPARK - Key features

From the Spark website (http://spark.apache.org/):

* Spark is a fast and general engine for large-scale data processing.
* Spark runs on Hadoop, Mesos, standalone, or in the cloud.
* Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
* Write applications quickly in Java, Scala, Python, R.
* Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.
* Apache Spark has an advanced Directed Acyclic Graph (DAG) execution engine that supports cyclic data flow and in-memory computing.



*Figure 9. Spark data flow.*

At the core of Spark are "Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner" (<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf>).

An RDD is simply a distributed collection of elements (Key-Value records).

* In Spark all work is expressed as either creating new RDDs, running (lazy) transformations on existing RDDs, or performing actions on RDDs to compute a result.
* Under the hood, Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them.

*Total Sort in pyspark (Spark Python API)*

As before, to achieve Total Order Sort, we must first partition the data such that it ends up in appropriately ordered buckets (partitions, filenames), and then sort within each partition. There are a couple of ways to do this in Spark, but they are not all created equal.

repartition & sortByKey VS repartitionAndSortWithinPartitions

<https://spark.apache.org/docs/1.6.2/programming-guide.html#working-with-key-value-pairs>

**repartition**: Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

**sortByKey**: When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

The sortByKey function reshuffles all the data a second time!

Spark’s sortByKey transformation results in two jobs and three stages.

Sample stage: Sample the data to create a range-partitioner that will result in an even partitioning.

“Map” stage: Write the data to the destined shuffle bucket for reduce stage.

“Reduce” stage: Get the related shuffle output and merge/sort on the specific partition of dataset.

**repartitionAndSortWithinPartitions**: Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

[INSERT CODE\_BLOCK-29]

By using glom we can see each partition in its own array. We also have a secondary sort on the "word" (in fact this is the latter part of a complex key) in ascending order.

[INSERT CODE\_BLOCK-30]

Final Remarks

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.

References

http://wiki.apache.org/hadoop/

http://hadoop.apache.org/docs/stable1/streaming.html#Hadoop+Streaming

http://mrjob.readthedocs.io/en/latest/index.html

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

https://hadoop.apache.org/docs/r2.7.2/hadoop-streaming/HadoopStreaming.html

http://hadoop.apache.org/

https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/SingleCluster.html

https://hadoop.apache.org/docs/r1.2.1/streaming.html#Hadoop+Comparator+Class

https://github.com/Yelp/mrjob

https://pythonhosted.org/mrjob/guides/configs-hadoopy-runners.html

http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-steps.html

https://github.com/apache/hadoop/blob/2e1d0ff4e901b8313c8d71869735b94ed8bc40a0/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/lib/partition/KeyFieldBasedPartitioner.java

http://science.sut.ac.th/mathematics/pairote/uploadfiles/weightedkm-temp2\_EB.pdf

http://spark.apache.org/

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf

https://spark.apache.org/docs/1.6.2/programming-guide.html#working-with-key-value-pairs

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