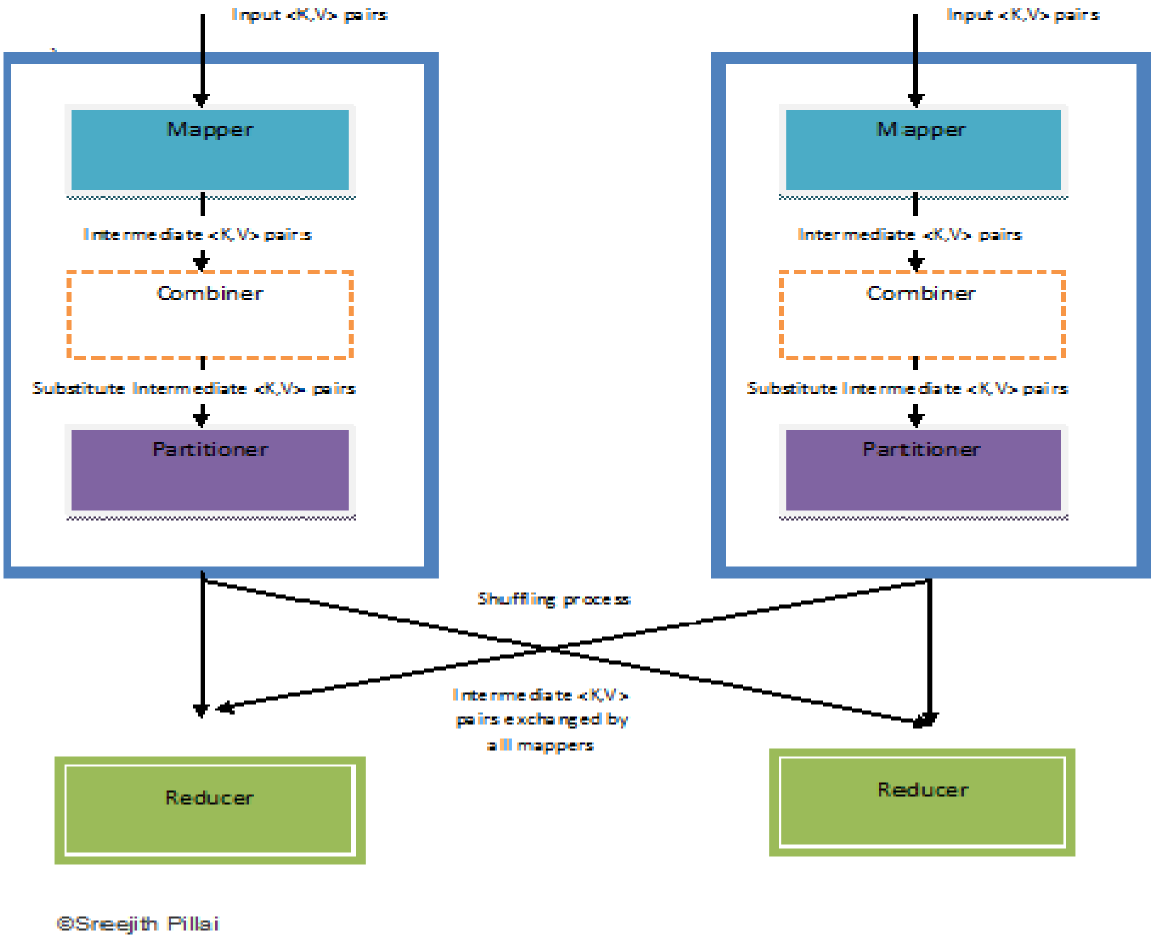
IMPLEMENTING PARTITIONERS AND COMBINERS FOR MAPREDUCE

Partitioners and Combiners in MapReduce:

Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. In other words, the partitioner specifies the task to which an intermediate key-value pair must be copied. Within each reducer, keys are processed in sorted order. Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase. The primary goal of combiners is to save as much bandwidth as possible by minimizing the number of key/value pairs that will be shuffled across the network between mappers and reducers With optimization I mean we can think of combiners as mini-reducers" that take place on the output of the mappers, prior to the shuffle and sort phase. Each combiner operates in isolation and therefore does not have access to intermediate output from other mappers. Let's talk about each separately; first we will go with **Partitioners**. A common misconception for first-time MapReduce programmers is to use only a single reducer. After all, a single reducer sorts all of your data before processing and would have stored output data in one single output file— and who doesn't like sorted data? It is easy to understand that such a constraint is a nonsense and that using more than one reducer is most of the time necessary, else the map/reduce concept would not be very useful. With multiple reducers, we need some way to determine the appropriate one to send a (key/value) pair outputted by a mapper. The default behavior is to hash the key to determine the reducer. The partitioning phase takes place after the map phase and before the reduce phase. The number of partitions is equal to the number of reducers. The data gets partitioned across the reducers according to the partitioning function. This approach improves the overall performance and allows mappers to operate completely independently. For all of its output key/value pairs, each mapper determines which reducer will receive them. Because all the mappers are using the same partitioning for any key, regardless of which mapper instance generated it, the destination partition is the same. Hadoop uses an interface called Partitioner to determine which partition a key/value pair will go to. A single partition refers to all key/value pairs that will be sent to a single reduce task. You can configure the number of reducers in a job driver by setting a number of reducers on the job object (job.setNumReduceTasks). Hadoop comes with a default partitioner implementation i.e. HashPartitioner, which hashes a record's key to determine which partition the record belongs in. Each partition is processed by a reduce task, so the number of partitions is equal to the number of reduce tasks for the job



```
1   job.setPartitionerClass(LogPartitioner.class);
```

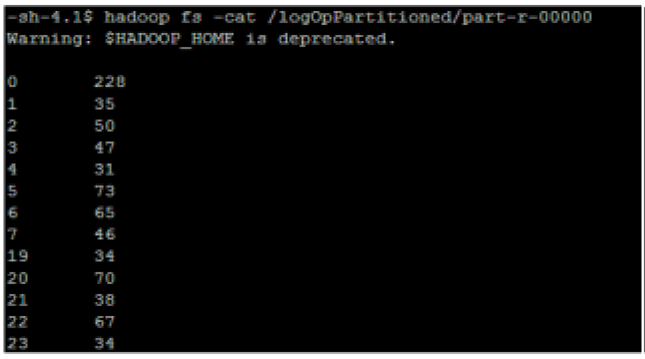
When the map function starts producing output, it is not simply written to disk. Each map task has a circular memory buffer that it writes the output to. When the contents of the buffer reach a certain threshold size, a background thread will start tospill the contents to disk. Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete. Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort. **Combiners:** Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output—the combiner function's output forms the input to the reduce function. Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer. One can think of Combiners as "mini-reducers" that take place on the output of the mappers, prior to the shuffle and sort phase. Each combiner operates in isolation and therefore does not have access to intermediate output from other mappers. The combiner is provided keys and values associated with each key (the same types as the mapper output keys and values). Critically, one cannot assume that a combiner will have the opportunity to process all values associated with the same key. The combiner can emit any number of key-value pairs, but the keys and values must be of the same type as the mapper output (same as the reducer input). In cases where an operation is both associative and commutative (e.g., addition or multiplication), reducers can directly serve as combiners. In general, however, reducers and combiners are not interchangeable.

```
job.setCombinerClass(LogReducer.class);
```

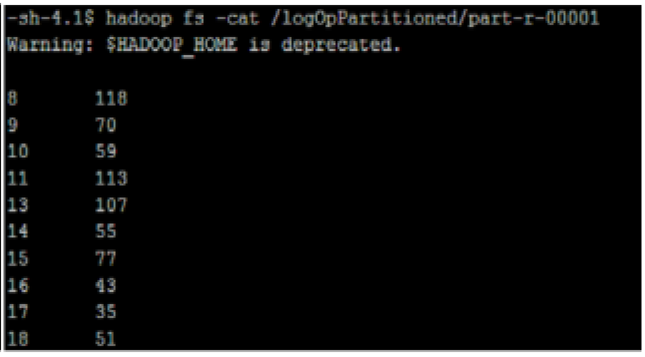
You can also find the whole project on Githubhttps://gist.github.com/sreejithpillai/6044673b4daeee2d1263 The difference between a partitioner and a combiner is that the partitioner divides the data according to the number of reducers so that all the data in a single partition gets executed by a single reducer. However, the combiner functions similar to the reducer and processes the data in each partition. The combiner is an optimization to the reducer. However it might be useful to partition the data according to some other function of the key or the value. A combiner doesn't necessarily improve performance. You should monitor the job's behavior to see if the number of records outputted by the combiner is meaningfully less than the number of records going in. The reduction must justify the extra execution time of running a combiner. You can easily check this through the JobTracker's Web UI. Below I am taking an example from my previous blog analyzing Apache web log using MapReduce. My requirement was to get all count of hits from 8th hour till 18th hour in a different output file and rest hours hot count in different log. To achieve the same I implemented Partitioner by writing the logic to send all key values in range of 8 to 18 to Reducer 1 and rest keys to Reducer 0. I changed my job configuration to implement two reducers

```
job.setNumReduceTasks(2);
```

Below is the output stored in HDFS which is partitioned among two reducers:

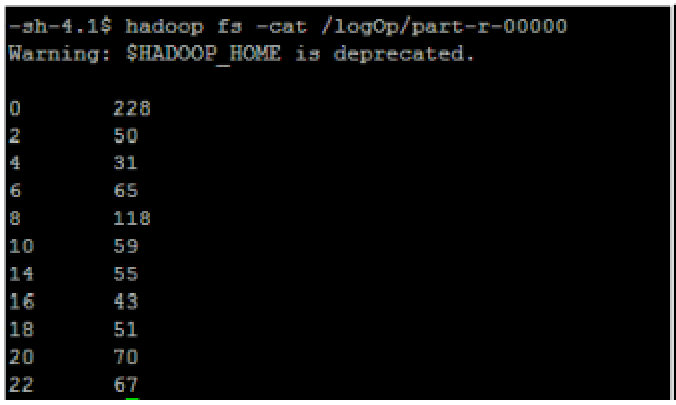


```
-sh-4.1$ hadoop fs -cat /logOpPartitioned/part-r-00000
Warning: $HADOOP_HOME is deprecated.

0       228
1       35
2       50
3       47
4       31
5       73
6       65
7       46
19      34
20      70
21      38
22      67
23      34
```

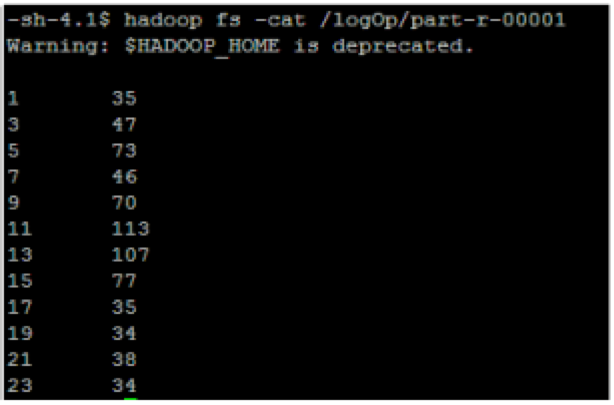


```
-sh-4.1$ hadoop fs -cat /logOpPartitioned/part-r-00001
Warning: $HADOOP_HOME is deprecated.

8       118
9       70
10      59
11      113
13      107
14      55
15      77
16      43
17      35
18      51
```

In same code I just removed job.setPartitionerClass(LogPartitioner.class); line and the output was different



```
-sh-4.1$ hadoop fs -cat /logOp/part-r-00000
Warning: $HADOOP_HOME is deprecated.

0       228
2       50
4       31
6       65
8       118
10      59
14      55
16      43
18      51
20      70
22      67
```



```
-sh-4.1$ hadoop fs -cat /logOp/part-r-00001
Warning: $HADOOP_HOME is deprecated.

1       35
3       47
5       73
7       46
9       70
11      113
13      107
15      77
17      35
19      34
21      38
23      34
```

The difference between both output was the partitioned output was stored as per our requirement i.e. keys in range 8-18 was stored in Reducer 1 and rest keys in Reducer 0. But the later used HashPartitioner(Default Partitioner) to differentiate they keys in both files.