Overview of MapReduce

Author: Peter Bugaj

Table of Content

Overview

MapReduce is a framework for processing large amouts of data, which it accomplishes by splitting the data into chunks and processing each chunk in parallel on multiple nodes. It then sorts the output and send it to the reducer tasks. For optimization purposes, MapReduce tries to compute the data on the same node it is stored on. Unlike other frameworks, data computed by MapReduce is not shared between nodes and each mapping and reducing task is able to restart without affecting the overall output. For this reason, external communication by a task, especially if it involves data changes, is not recommended.

MapReduce starts with an input and output specification for reading and outputting its' data to. It then runs through the data through the use of interfaces for which specific implementations are provided for. The most common interfaces includes the Map and Reduce interface. A class describing how to map the data along with a class defining how to reduce the data would implement these interfaces. A job configuration passed into the MapReduce framework would point to these classes.

Throughout the entire process, the MapReduce framework operates specifically on Key Value pairs. It first maps the data into zero or more key value pairs. Its then sorts the data, partitions it and combines it, before reducing the data into another set of zero or more key value pairs. The Keys and Values used must be serializable to allow writing to files and must implement the Writable Interface.

Main Components

- **Input Format**, which defines how the input data is split up per map task. It takes a list of input files, and breaks each file into InputSplits.

- **Input Split**, being a map task for reading a specific section of data.

- **Record Reader**, that defines how to read the specific section of data within the input split.

- **Mapper**, which takes input data from the record reader and transforms it into zero or more key value pairs.

- **Reducer**, which takes a list of values that share the same key, and outputs zero or more key value pairs as the final output.

- **Partitioner**, for portioning the intermediate key value pairs for each reducer.

- **Reporter**, for reporting progress of the map reduce operations.

- **Combiner**, which helps optimize the map reduce operations by combing key value pairs together before they are sent to the reducer.

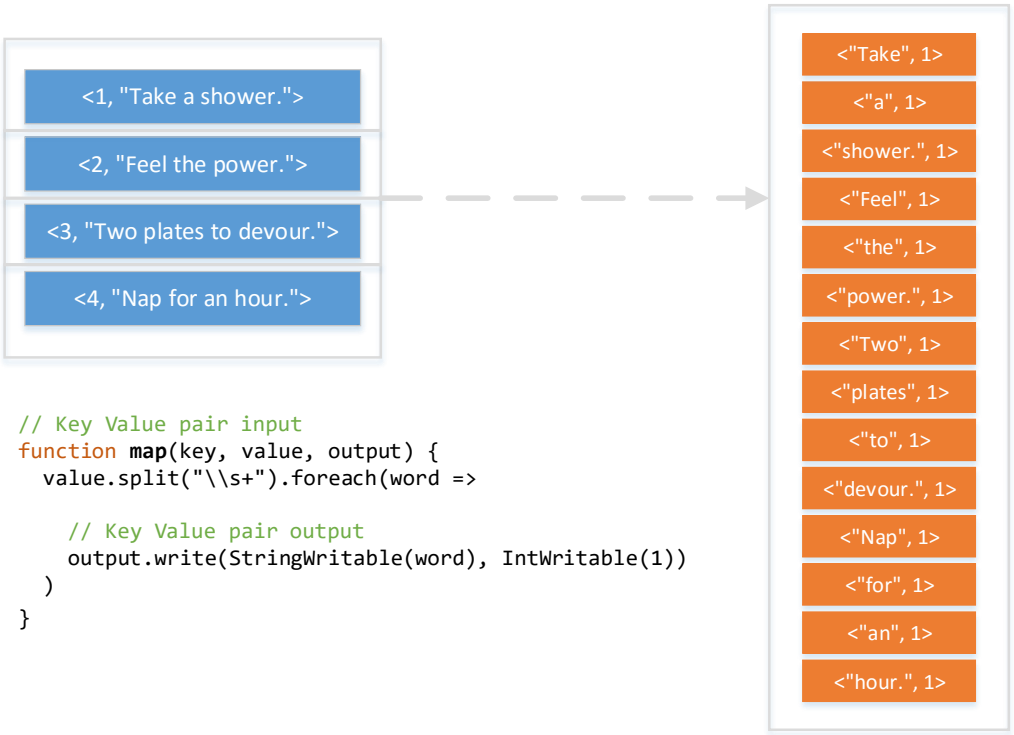
- **OutputCollector**, the interface for collecting output from the Mappers, Reducers, or Combiners.

- **JobConfiguration**, being the main interface for defining map and reduce operations, and things like the Combiner, and Input and Output formats.

Mapper

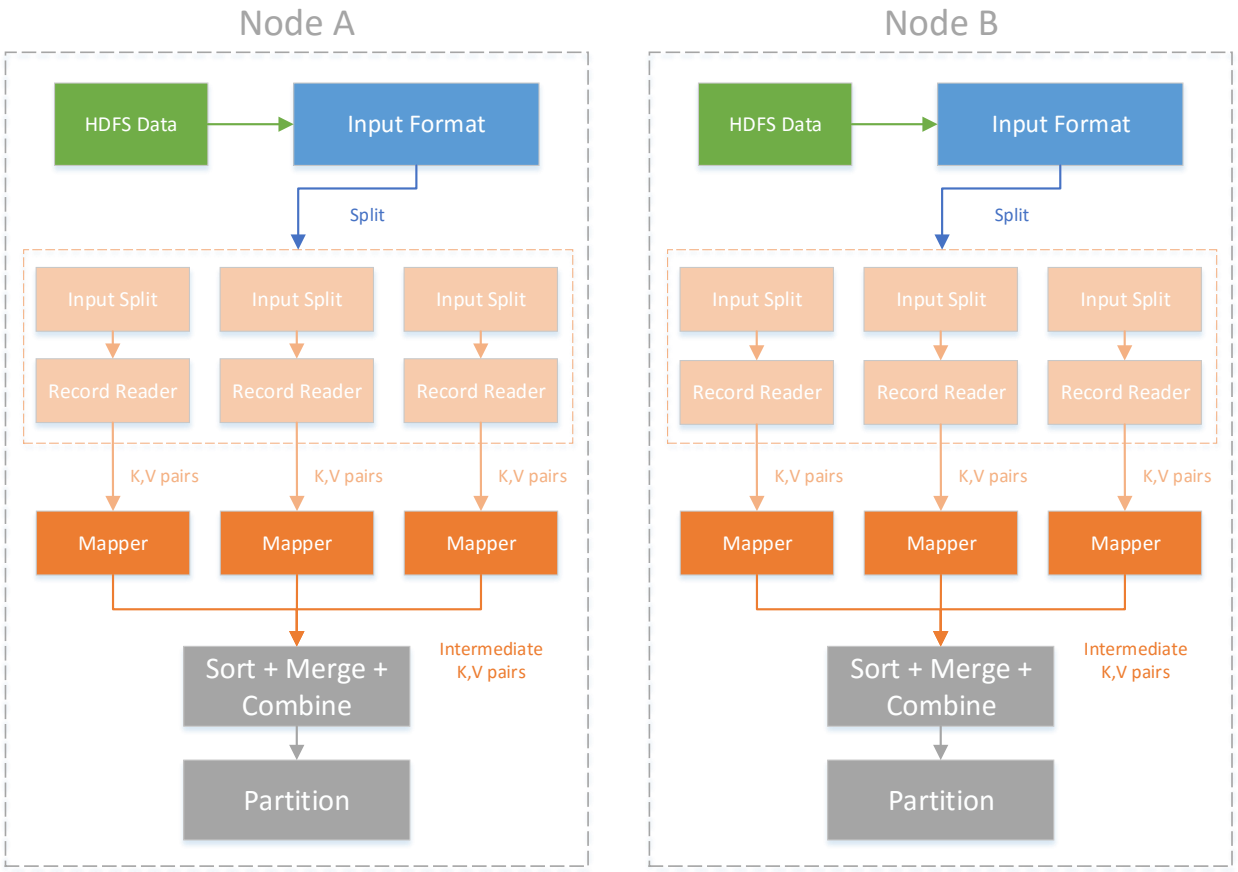
The mapper is the first process that runs. Its' job is to take input data, in the form of key value pairs, and to transform it into zero or more intermediate key value pairs. The input and intermediate pairs managed by the mapper do not need to be of the same type.

The transformation itself is done in parallel per map task. The number of such tasks is determined by the size of the input data divided by the configured block size. A task takes time to initialize and it is usually recommended that the number of tasks is small enough so to at least allow each task to execute for a bit longer than the time it took to initialize it. Mapping tasks do not communicate to each other and each work independently.



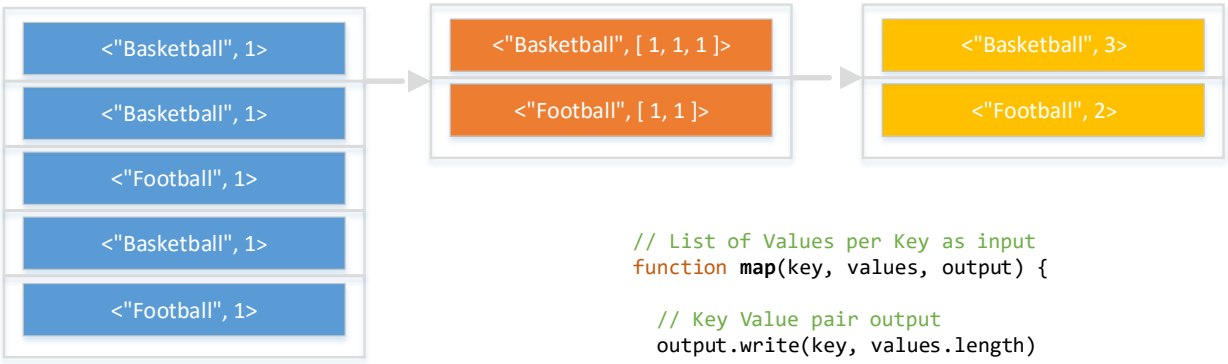

```
// Key Value pair input
function map(key, value, output) {
  value.split("\\s+").foreach(word =>

    // Key Value pair output
    output.write(StringWritable(word), IntWritable(1))
  )
}
```

After the intermediate pairs are created, they are locally sorted on each node by the framework and are partitioned. The number of partitions is dependent on the number of reducers specified to run in the job configuration as each reducer is then created and assigned to one partition. The partitioning of the data can be customized through the implementation of the Partitioner interface. If a combiner is specified, the data will also be combined, either each after it is spilled onto the disk, or after it is merged.



Reducer

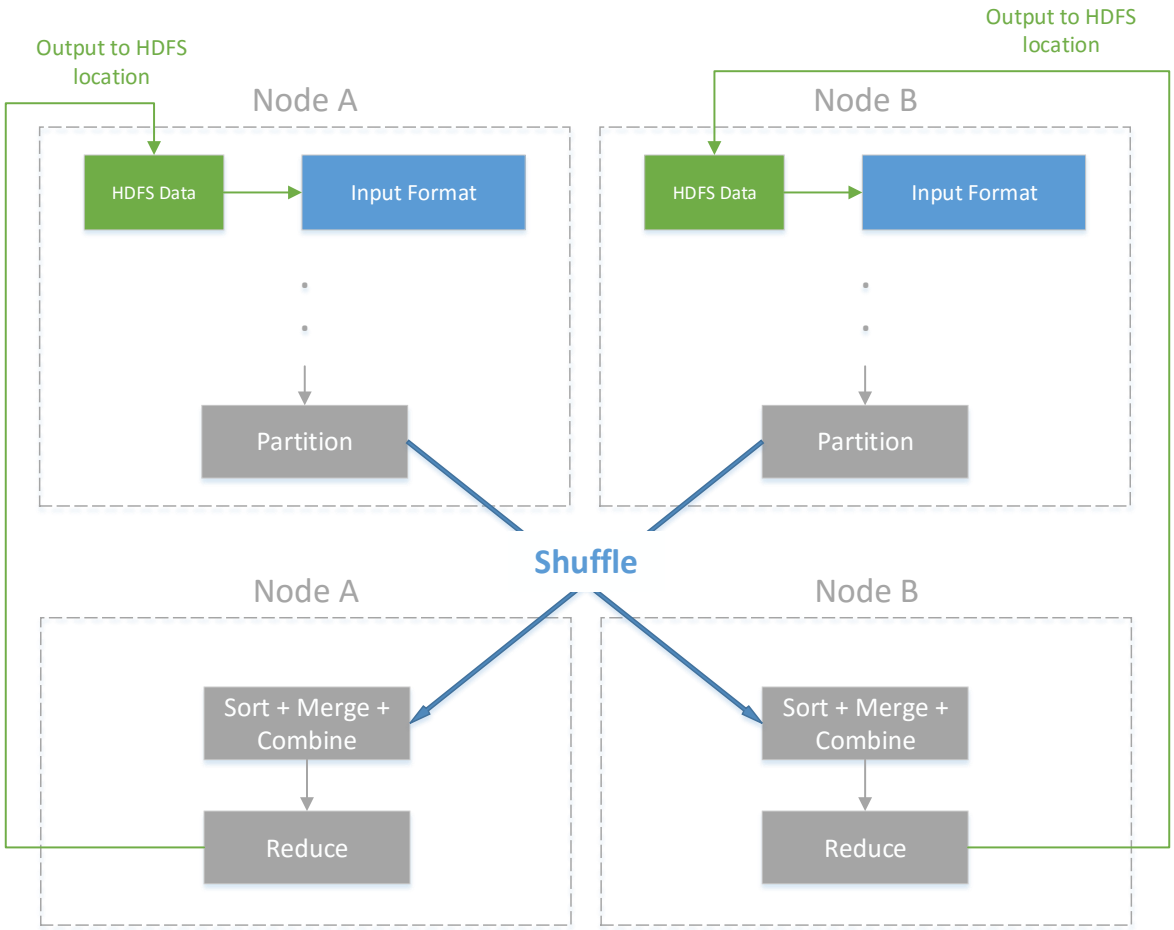
A reducer reduces a set of intermediate key value pairs into a smaller set. The input it receives is a list of values grouped by the same key, and it outputs another set of zero or more key value pairs. The output pairs don't have to be of the same type as the intermediate pairs either, but unlike the mapper, the reducer outputs the pairs to the specified output location where the data stops being processed.




```
// List of Values per Key as input
function map(key, values, output) {

  // Key Value pair output
  output.write(key, values.length)
}
```

The reducer first starts of by fetching the sorted list of key value pairs for its partition. It does this by calling the framework through through HTTP calls. This is called the Shuffle phase since key value pairs belonging to the one partition the reducer is looking for might exist on multiple nodes after the mapping phrase. In this case they are sent to the same node the reducer is making the calls from. Afterwards the reducer sorts the data by key. The input from each mapper is already sorted, however sorting is required again due to the mappers being on different nodes.

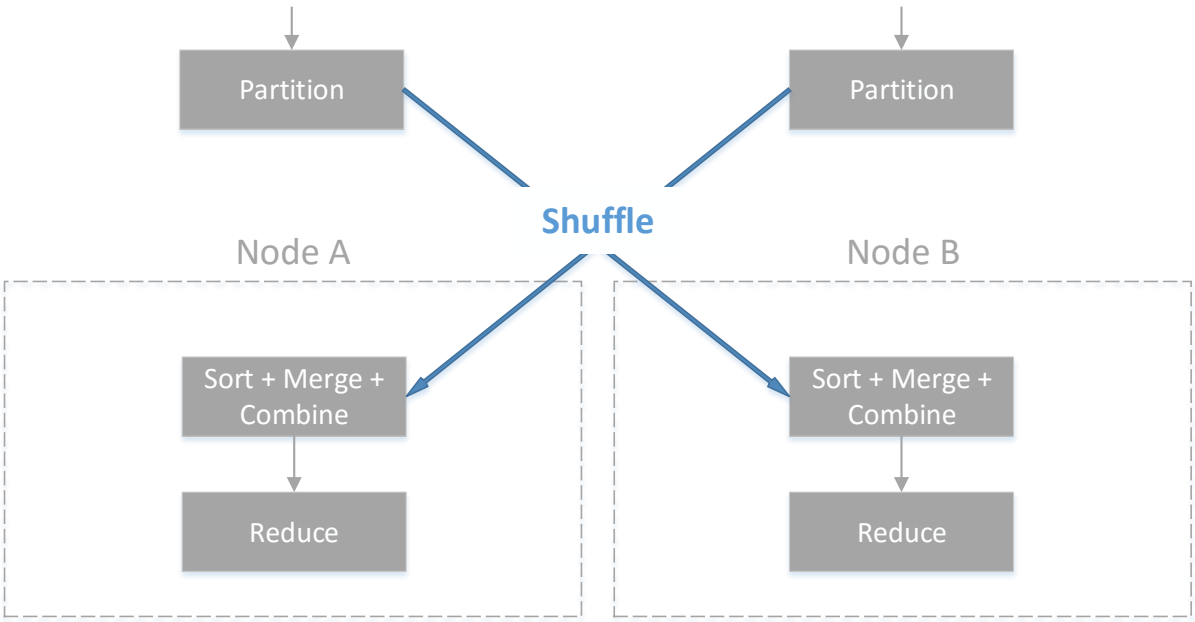
Once sorted, the reducer takes a list of all values for each key, and for each list it outputs zero or more key value pairs, which it then sends to the specified output location. Unlike the mapper, it no longer sorts the output. Like the mapper, one or more reducers can run on one node, although scheduling more reducers can have performance issues due to the communication required in the shuffling process. Reducers also work independently and do not talk to each other.



Partitioner

The partitioner divides the intermediate key value pairs produced by the mapper into subsets, where each reducer would then process the key value pairs within the partitions assigned to it. All key value pairs with the same key always belong to the same partition.

Before the reducer processes the key value pairs for each of its assigned partitions, it first makes a call to the MapReduce framework to get the pairs from all the different nodes where the mapping of the data took place. This process is known as shuffling. Shuffling of the mapped output is necessary because each node might possibly produce a key value pair belonging to a partition the reducer is interested in. Since a reducer has to grab the key value pair from all the nodes, the shuffling phase can be expensive, especially if there are a lot of nodes to grab the data from. This cost increases if multiple reducers exist per node and for this reason having more than two reducers per node starts to become less optimal.

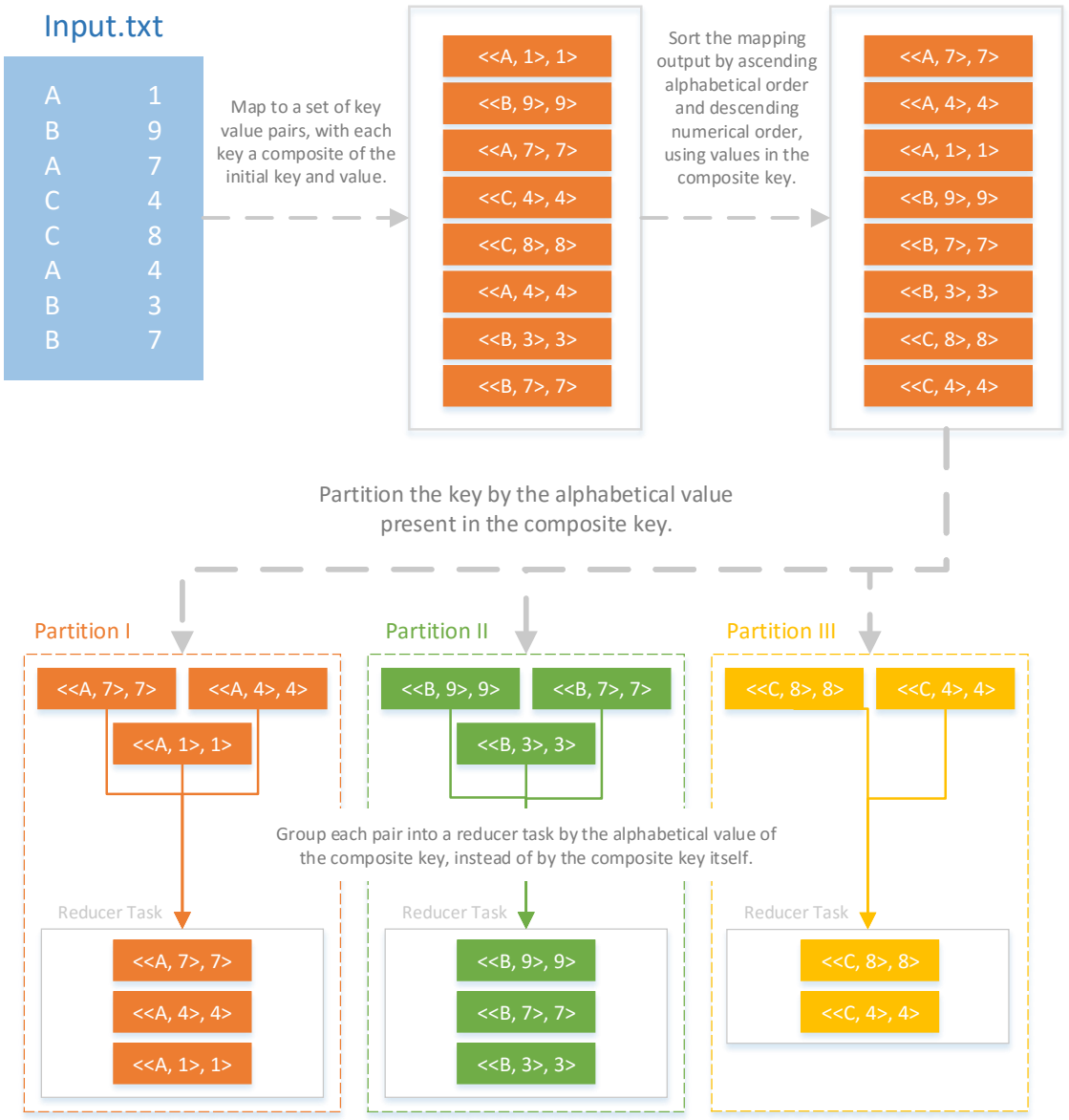


The key value pairs coming from each node are already sorted, since each mapper produces a sorted list of key value pairs. However since this sorted data is coming from multiple nodes, it has to be remerged when it reaches the reducer so that the order is kept. While the data is being merged, a combiner, if specified, can be applied after each merge operation, to keep the number of key value pairs small. In other words, the combiner runs both during the local merging phase on each node after the mapper produces its output, and during the shuffling phase when data is sent and merged on the node for each reducer.

Sort and Grouping Comparators

A sort comparators defines how keys are sorted after an output from the mapper. By default the keys will use a default sorter based on the type, either being sorted alphabetically, numerically, or by a hash value. However it might also be necessary to sort the keys by specific logic in the case where the key is a composite key of more than one value that can be used in the sorting.

A grouping comparator defines how key value pairs should be combined in the reducer. In certain cases a partitioner might group different keys for one reducer, however each key will still have its values processed by one reducer task. A grouping comparator allows values belonging to multiple keys to be sent to the same reducer task.


Input.txt
A 1
B 9
A 7
C 4
C 8
A 4
B 3
B 7
Map to a set of key value pairs, with each key a composite of the initial key and value.
<<A, 1>, 1>
<<B, 9>, 9>
<<A, 7>, 7>
<<C, 4>, 4>
<<C, 8>, 8>
<<A, 4>, 4>
<<B, 3>, 3>
<<B, 7>, 7>
Sort the mapping output by ascending alphabetical order and descending numerical order, using values in the composite key.
<<A, 7>, 7>
<<A, 4>, 4>
<<A, 1>, 1>
<<B, 9>, 9>
<<B, 7>, 7>
<<B, 3>, 3>
<<C, 8>, 8>
<<C, 4>, 4>
Partition the key by the alphabetical value present in the composite key.
Partition I
Partition II
Partition III
Group each pair into a reducer task by the alphabetical value of the composite key, instead of by the composite key itself.
Reducer Task
Reducer Task
Reducer Task

Combiner

Combiners help optimize map reduce operations by reducing the number of tuples getting passed between the map and reduce operations. This is done by combining them together. Hence the term “combiner”.

It is important to note that MapReduce might choose to use a combiner on the tuples either once, multiple times over, or it might choose to never use it at all. The output from the reducer must also be the same, regardless of the cases of how the input is optimized. This therefore requires that the combiners are to be both associative and commutative operators.

Fault Tolerance

MapReduce provides fault tolerance by monitoring tasks through task trackers, and by monitoring the task trackers themselves through job trackers. In the case of a job tracker failure, the whole job has to be restarted.

Task trackers look after map and reduce tasks, and have them rescheduled by the job tracker to other task trackers whenever one fails. In the case of a map task failing, only that one task has to be rescheduled. However in the case of a reduce task, sometimes all the previous map tasks have to be re-executed as well, as the intermediate output might be derived from multiple nodes no longer accessible.

A task is marked as having failed if it either exists with an error code, or if it does not respond to the task tracker for a certain period of time. The default is set to ten minutes, but can this be reconfigured or even completely disabled – although the latter option is not recommended. This is set in **mapred.task tracker.expiry.interval**.

If the same task fails many times (set in **mapred.map.max.attempts** and **mapred.reduce.max.attempts**), the task will no longer be rescheduled by the job tracker and the job itself will fail. A MapReduce job might also want to allow a certain percentage of tasks to fail, before it considers the result unreliable and fails the job. This is set in **mapred.max.map.failures.percent** and **mapred.max.reduce.failures.percent**.

The failure of a task tracker is handled in a similar fashion. If a task tracker does not respond to the job tracker for a given period of time, it will be marked as having failed and tasks in that task tracker will be reassigned elsewhere. Usually this reassignment is a lot more expensive because a lot more task will have to be restarted. The job tracker has the option to mark the task tracker as faulty. If it fails too many times (the number is set in **mapred.max.tracker.blacklist)s**, the job tracker will blacklist that task tracker for a certain period of time.

Exercises

Combiners

1. Run an example where a few mapping outputs are created, and an example where many mapping outputs are created (enough to use up all the allocated RAM). Do not use a combiner and see how increases the number of outputs decreases the overall performance.

2. Repeat step one, but this time use a combiner. See how the performance improves for a large number of mapping outputs.

3. Create an example where the combiner and reducer are the same.

4. Create an example where the combiner and reducer are different.

5. Create an example where theh combiner and reducer are different, but where the reducer can be reused as the combiner as well.

Partitioner and Shufflers

6. Create an example with a custom combiner.

7. Create an example where all the mapping output belongs to one key and where data is forced to go onto a single partition for a one reducer. See how this differs if the data split into multiple keys.

Mappers

8. See the performance difference when too many mappers are created. To do this, increase the hadoop block size to decrease the number of mapping tasks produced for a set of input files, or decrease the block size to increase the number of mappings taks.

   Since the mapper uses the Hadoop block size by default, these can be overwritten with the setting `mapred.min.split.size`

Reducers

9. Change the specification for the number of partitions created to affect how many reducers get created. See how adding or removing reducers (more than two reducers per node, or less than one reducer per node) affects the performance.

Optimization

10. See the performance difference when changing things like mapred.job.shuffle.merge.percent for shuffling data, or mapred.job.shuffle.input.buffer.percent or for storing map outputs. See how this affects the use of combiners.

Online References

Overviews

- https://hadoop.apache.org/docs/r2.8.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html
- https://www.slideshare.net/andreaiacono/mapreduce-34478449
- https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

Spark Comparison

- https://blog.cloudera.com/blog/2014/09/how-to-translate-from-mapreduce-to-apache-spark/

Fault Tolerance

- http://sungsoo.github.io/2014/04/07/failures-in-classic-mapreduce.html
- https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920/
- https://stackoverflow.com/questions/5813665/fault-tolerance-in-mapreduce

Combiners

- http://hadooptutorial.info/combiner-in-mapreduce/
- https://www.tutorialspoint.com/map_reduce/map_reduce_combiners.htm

Partitioners

- https://sreejithrpillai.wordpress.com/2014/11/24/implementing-partitioners-and-combiners-for-mapreduce/
- https://community.hortonworks.com/questions/14328/what-is-the-difference-between-partitioner-combine.html

Scheduling

- http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html

Examples

- https://github.com/milesegan/scala-hadoop-example/blob/master/WordCount.scala
- http://developersnightmare.blogspot.ca/2011/08/hadoop-with-scala-wordcount-example.html
- https://www.linkedin.com/pulse/spark-mapreduce-scala-underscore-vishnu-viswanath
- http://hbase.apache.org/0.94/book/mapreduce.example.html