MapReduce is a user friendly configurable model to do computation on data intensive real world tasks in a distributed environment. The users only specify function map and reduce to do the computation while all the other tasks (parallel processing, fault tolerance, scheduling, efficient network and disk usage) is managed by the framework.

The abstraction designed by the authors was inspired by the map and reduce primitive present in many functional language (like Lisp). The authors have shown that, each problem can be designed in a way to map the input records to Map functions to generate intermediate key/value pairs and then apply Reduce functions to those pairs to combine derived data. The functional model with user specified Map and Reduce function facilitate the parallelization of large computation where re-execution is used as the primary mechanism for fault tolerance.

Let's consider the following word count problem, here the Map functions emit each word(key) with number of occurrence as count (value) and the Reduce functions combine all the unique words (keys) to get the total count of occurrence in documents.

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word in value:
    EmitIntermediate(w, 1);
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
    result += ParseInt(v);
    Emit(AsString(result));
```

Map and Reduce function defined by users have associate types. Depending on the environment, right version need to be chosen from different MapReduce implementations.

At the beginning of execution cycle MapReduce library splits the input files and start Map task worker node in cluster, the map tasks store the intermediate result in memory and finally to local disk which information is notified to the Reduce worker node through the Master node. Based on the notification, the Reduce worker fetch data from remote Map workers machine. Then the reduce worker sort the intermediate keys to group the same keys together. External sort is used in case of memory shortage. Then the Reduce worker iterates over the sorted intermediate data and apply Reduce function to each key/value pair. The output of the Reduce function is appended to the final output file. After finishing all the map task and reduce task, the master node wakes up the user program and return back to the user code. The output of the MapReduce function is available in R output files which can be combined to one if needed.

Master node keeps the identity of worker machine and also the state of Map, Reduce task (idle, in-progress and completed). It pings every worker node periodically to find the status (active or failed). If a failed worker node found, all map task (completed or in-progress) and reduce task (in-progress) executed by that particular node are reset to idle and becomes available for rescheduling. Re-execution is done in case of completed map task to generate the local disk intermediate result and then notified to all the Reduce worker node. Completed Reduce tasks do not need re-execution as data is stored in the global file system.

Deterministic Map and Reduce task produce same result as it would have been done on a sequential manner. It is ensured by the atomic manner of Map and Reduce task. Though the same Map and Reduce task can be completed by multiple worker node, the framework ensures consistency.

For the nondeterministic Map and Reduce task, in case of machine failure result might not be consistent as nondeterministic task does not necessarily produce same result on each cycle. To solve this problem, Google has an ultra reliable replicated file system (it's called Colossus) to make physical disk failures irrelevant [1].

MapReduce master node consider the locally stored data information to optimize usage of network bandwidth. The Map and Reduce phase are divided into M and R pieces. Generally this (M + R) value is much larger than the workers available and distributed to task (M and R task). Each worker can complete multiple tasks based on availability, but there should be a limit of this number (M + R). As the MapReduce master node keep O(M+R) scheduling decision and O(M*R) state in memory.

Sometimes couple of machine in the cluster might take longer time w.r.t others which lengthens the total time taken for a MapReduce operation. It is called straggler. To alleviate this problem, at the end stage of MapReduce operation the master node schedules backup execution for the remaining in-progress tasks. Whichever tasks (backup or in-progress) finish first the master node marks those as complete ignoring the others. This might increase the usage of computation resources but it reduces the execution time significantly.

User specified configuration in partition functions (mapping intermediate key to R reduce workers), combiner functions (combination of same intermediate key in the same map task to reduce data transfer across network), ordering guarantees in R reduce partition, defining custom input/output type formats and debugging flexibility (run in a single machine) is also possible in MapReduce.

[1] Ref