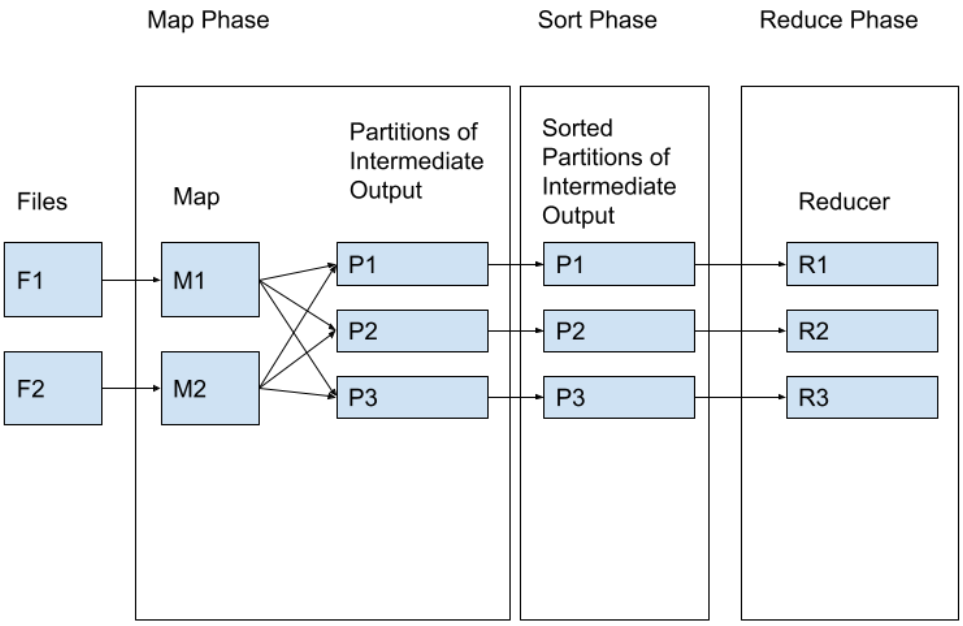
MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

In simple words, since one machine cannot fit all data in memory, we'd like multiple machines to split work among themselves and operate on portions of data at a time so as to generate required output. We'd go over all phases listed below one by one to discuss what each phase does:



Map Phase :

I. Each Mapper thread(M1 and M2 in figure) reads from 1 file. Say there are two threads and two files, then each thread will concurrently read data from 1 file assigned to it during the thread's creation. Let's say there are two mapper threads and 2 input files, mapper thread M1 will get input file F1 and M2 will get F2. You can think of passing function arguments to the mapper thread about the file it is supposed to read from.
II. Each mapper thread(M1 and M2 in figure) produces **intermediate output** to all partitions. Mapper thread will ask the partitioner function - say MR_DefaultHashPartition, to find the partition index of an input key. If you'd see the definition of MR_DefaultHashPartition, it returns an index corresponding to a key. That index will

determine the partition index for a particular input key. Now, one might wonder why partitioning is required. Data partitioning is done to divide and bring all relevant portions of data in one partition.

III. Now, since two mapper threads can write to the same partition, you may want to use pthread_mutex_t for each partition so that only one thread can write to one partition at a time. If you see any garbage values in a partition, this means locking is not working properly. Note that since lock is per-partition, M1 should be able to write to partition P1 and M2 should be able to write to partition P3 concurrently. Thus, different mappers should be able to concurrently access different partitions at a time.

Sort Phase

I. In this phase we simply need to sort one partition at a time. You just need to run a for-loop which sorts all contents of one partition at a time. Say, you'll pick partition - P2 and sort all entries of that partition.

II. In the end, you'll have internally sorted partitions to be consumed by reducer threads.

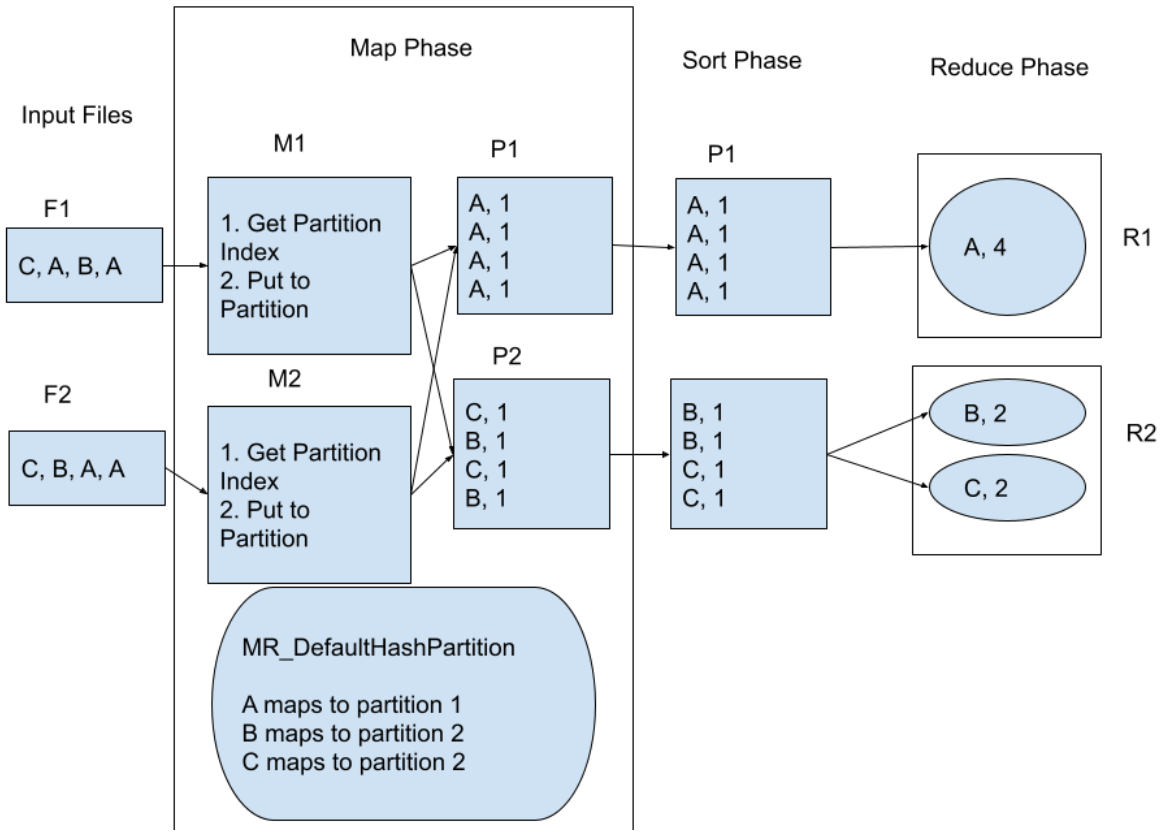
Reduce Phase

I. Note that the number of partitions are equal to the number of reducer threads.

II. Each reducer thread will first pick a key and read intermediate output entries of a key. It will run it's reduction function on a key. Since Reducer function is supplied as argument to your library, we - as library developers have no control over how final output is consumed.

Now, let's have a look at a concrete example - word count and see how this abstraction works with this example.

1. **Map phase :** File F1 has C, A, B, A and F2 has C, B, A, A. M1 thread reads all entries from F1, gets the index of each key(A, B or C) from MR_DefaultHashPartition and pushes intermediate outputs to partition 1 or 2. Note that all entries of A are in partition P1, all entries of B are in partition P2 and all entries of C are in partition P3
2. **Sort phase :** Simply sort all entries inside a partition. Note that all B's are before C's in pairtion P2.
3. **Reduce phase :** Reducer thread will pick one key at a time and run the user-provided reduction function on it. Note that reducer thread R2 will first pick key B, run reduction function on it and then pick key C and run reduction function on it. That's why we have two ovals in R2 indicating that it picks one key at a time.



Appendix : Why partitioning?

Now, let's talk about partitions. What is a partition? Why is it needed? Here we motivate why partitions are needed.
Let’s assume there are no partitions and all mappers write to a single list only, then the mappers’ execution becomes sequential because they’ll have to compete for taking lock on that single list. Mapper threads will waste time waiting for that lock.

So, to avoid the sequentialization, we should have multiple lists. Each list can be seen as a partition which is accessed by all the mappers and reducers.

However, we’re not simply partitioning data, we’re doing that in an intelligent way. MR_Emit() will decide in which partition the data should be stored. For example, the same key(word in example above) will always be stored in the same partition. That way, reducers work is lessened because reducer needs to **find all key-value pairs** for a particular key and it no longer needs to go through all partitions for finding the same. Our intelligent partitioning ensures **all key-value pairs of a particular key go to the same partition.** Also, there’s no need to perform global sort across all partitions. Note that global sort would be much more costly than several local per-partition sort. Our partitioning enables a reducer thread to access only one partition for reduction on a key.