**Unit II: MAP REDUCE**

2.2.2 Grouping by Key

* As soon as the Map tasks have all completed successfully, the key-value pairs are grouped by key, and the values associated with each key are formed into a list of values.
* The grouping is performed by the system, regardless of what the Map and Reduce tasks do.
* The master controller process knows how many Reduce tasks there will be, say r such tasks.
* The user typically tells the MapReduce system what r should be.
* Then the master controller picks a hash function that applies to keys and produces a bucket number from 0 to r − 1.
* Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files.
* Each file is destined for one of the Reduce tasks.
* To perform the grouping by key and distribution to the Reduce tasks, the master controller merges the files from each Map task that are destined for a particular Reduce task and feeds the merged file to that process as a sequence of key-list-of-value pairs.
* That is, for each key k, the input to the Reduce task that handles key k is a pair of the form (k, [v1, v2, . . . , vn]), where (k, v1), (k, v2), . . . , (k, vn) are all the key-value pairs with key k coming from all the Map tasks.

2.2.3 The Reduce Tasks

* The Reduce function’s argument is a pair consisting of a key and its list of
* associated values.
* The output of the Reduce function is a sequence of zero or more key-value pairs.
* These key-value pairs can be of a type different from those sent from Map tasks to Reduce tasks, but often they are the same type.
* A Reduce task receives one or more keys and their associated value lists.
* That is, a Reduce task executes one or more reducers. The outputs from all the Reduce tasks are merged into a single file.
* Reducers may be partitioned among a smaller number of Reduce tasks is by hashing the keys and associating each Reduce task with one of the buckets of the hash function.(Eg mentioned in assignment book)

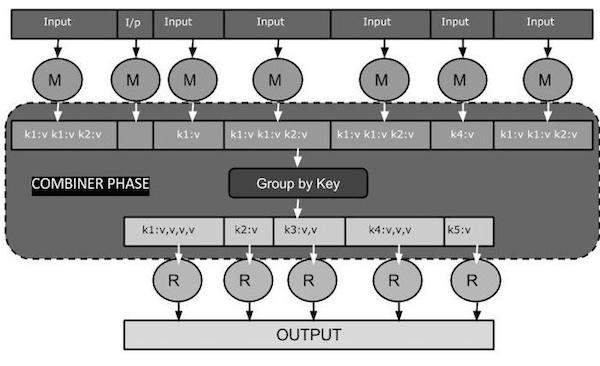
A Combiner, also known as a **semi-reducer,** is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

## **Combiner**

The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Usually, the output of the map task is large and the data transferred to the reduce task is high.

The following MapReduce task diagram shows the COMBINER PHASE.



## **How Combiner Works?**

Here is a brief summary on how MapReduce Combiner works −

* A combiner does not have a predefined interface and it must implement the Reducer interface’s reduce() method.
* A combiner operates on each map output key. It must have the same output key-value types as the Reducer class.
* A combiner can produce summary information from a large dataset because it replaces the original Map output.

Although, Combiner is optional yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.

## **MapReduce Combiner Implementation**

The following example provides a theoretical idea about combiners. Let us assume we have the following input text file named **input.txt** for MapReduce.

What do you mean by Object

What do you know about Java

What is Java Virtual Machine

How Java enabled High Performance

The important phases of the MapReduce program with Combiner are discussed below.

### **Record Reader**

This is the first phase of MapReduce where the Record Reader reads every line from the input text file as text and yields output as key-value pairs.

**Input** − Line by line text from the input file.

**Output** − Forms the key-value pairs. The following is the set of expected key-value pairs.

<1, What do you mean by Object>

<2, What do you know about Java>

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

### **Map Phase**

The Map phase takes input from the Record Reader, processes it, and produces the output as another set of key-value pairs.

**Input** − The following key-value pair is the input taken from the Record Reader.

<1, What do you mean by Object>

<2, What do you know about Java>

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

The Map phase reads each key-value pair, divides each word from the value using StringTokenizer, treats each word as key and the count of that word as value. The following code snippet shows the Mapper class and the map function.

public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>

{

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(Object key, Text value, Context context) throws IOException, InterruptedException

{

StringTokenizer itr = new StringTokenizer(value.toString());

while (itr.hasMoreTokens())

{

word.set(itr.nextToken());

context.write(word, one);

}

}

}

**Output** − The expected output is as follows −

<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>

<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>

<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>

<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>

### **Combiner Phase**

The Combiner phase takes each key-value pair from the Map phase, processes it, and produces the output as **key-value collection** pairs.

**Input** − The following key-value pair is the input taken from the Map phase.

<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>

<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>

<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>

<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>

The Combiner phase reads each key-value pair, combines the common words as key and values as collection. Usually, the code and operation for a Combiner is similar to that of a Reducer. Following is the code snippet for Mapper, Combiner and Reducer class declaration.

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

**Output** − The expected output is as follows −

<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,1,1,1>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>

**Coping With Node Failures**

* The worst thing that can happen is that the compute node at which the Master is executing fails.
* In this case, the entire MapReduce job must be restarted.
* But only this one node can bring the entire process down; other failures will be managed by the Master, and the MapReduce job will complete eventually.
* Suppose the compute node at which a Map worker resides fails.
* This failure will be detected by the Master, because it periodically pings the Worker processes.
* All the Map tasks that were assigned to this Worker will have to be redone, even if they had completed.
* The reason for redoing completed Map tasks is that their output destined for the Reduce tasks resides at that compute node, and is now unavailable to the Reduce tasks.
* The Master sets the status of each of these Map tasks to idle and will schedule them on a Worker when one becomes available.
* The Master must also inform each Reduce task that the location of its input from that Map task has changed.
* Dealing with a failure at the node of a Reduce worker is simpler.
* The Master simply sets the status of its currently executing Reduce tasks to idle.
* These will be rescheduled on another reduce worker later.

**Union and Intersection**

* First, consider the union of two relations. Suppose relations R and S have the same schema.
* The Map Function: Turn each input tuple t into a key-value pair (t, t).
* The Reduce Function: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.
* To compute the intersection, we can use the same Map function.
* However,the Reduce function must produce a tuple only if both relations have the tuple.
* If the key t has a list of two values [t, t] associated with it, then the Reduce task for t should produce (t, t).
* However, if the value-list associated with key t is just [t], then one of R and S is missing t, so we don’t want to produce a tuple for the intersection.
* The Map Function: Turn each tuple t into a key-value pair (t, t).
* The Reduce Function: If key t has value list [t, t], then produce (t, t). Otherwise, produce nothing.
* The Difference R − S requires a bit more thought. The only way a tuple t can appear in the output is if it is in R but not in S.
* The Map function can pass tuples from R and S through, but must inform the
* Reduce function whether the tuple came from R or S. We shall thus use the relation as the value associated with the key t.
* Here is a specification for the two functions.
* The Map Function: For a tuple t in R, produce key-value pair (t,R), and for a tuple t in S, produce key-value pair (t, S).
* Note that the intent is that the value is the name of R or S (or better, a single bit indicating whether the relation is R or S), not the entire relation.
* The Reduce Function: For each key t, if the associated value list is [R], then
* produce (t, t). Otherwise, produce nothing.

**2.3.7 Computing Natural Join by MapReduce**

* The idea behind implementing natural join via MapReduce can be seen if we look at the specific case of joining R(A,B) with S(B,C).
* We must find tuples that agree on their B components, that is the second component from tuples of R and the first component of tuples of S. We shall use the B-value of tuples from either relation as the key.
* The value will be the other component and the name of the relation, so the Reduce function can know where each tuple came from.
* **The Map Function:** For each tuple (a, b) of R, produce the key-value pair 􀀀b, (R, a)\_. For each tuple (b, c) of S, produce the key-value pair 􀀀b, (S, c)\_.
* **The Reduce Function:** Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct all pairs consisting of one with first component R and the other with first component S, say (R, a) and (S, c). The output from this key and value list is a sequence of key-value pairs.
* The key is irrelevant. Each value is one of the triples (a, b, c) such that (R, a) and (S, c) are on the input list of values.
* The same algorithm works if the relations have more than two attributes. You can think of A as representing all those attributes in the schema of R but not S. B represents the attributes in both schemas, and C represents attributes only in the schema of S.
* The key for a tuple of R or S is the list of values in all the attributes that are in the schemas of both R and S. The value for a tuple of R is the name R together with the values of all the attributes belonging to R but not to S, and the value for a tuple of S is the name S together with the values of the attributes belonging to S but not R.
* The Reduce function looks at all the key-value pairs with a given key and combines those values from R with those values of S in all possible ways. From each pairing, the tuple produced has the values from R, the key values, and the values from S.

**2.3.8 Grouping and Aggregation by MapReduce**

* As with the join, we shall discuss the minimal example of grouping and aggregation, where there is one grouping attribute and one aggregation. Let R(A,B,C) be a relation to which we apply the operator γA,θ(B)(R). Map will perform the grouping, while Reduce does the aggregation.
* **The Map Function:** For each tuple (a, b, c) produce the key-value pair (a, b).
* **The Reduce Function:** Each key a represents a group. Apply the aggregation operator θ to the list [b1, b2, . . . , bn] of B-values associated with key a. The output is the pair (a, x), where x is the result of applying θ to the list.
* For example, if θ is SUM, then x = b1 + b2 + · · · + bn, and if θ is MAX, then x is the largest of b1, b2, . . . , bn.
* If there are several grouping attributes, then the key is the list of the values of a tuple for all these attributes. If there is more than one aggregation, then the Reduce function applies each of them to the list of values associated with a given key and produces a tuple consisting of the key, including components for all grouping attributes if there is more than one, followed by the results of each of the aggregations.

**2.3.10 Matrix Multiplication with One MapReduce Step**

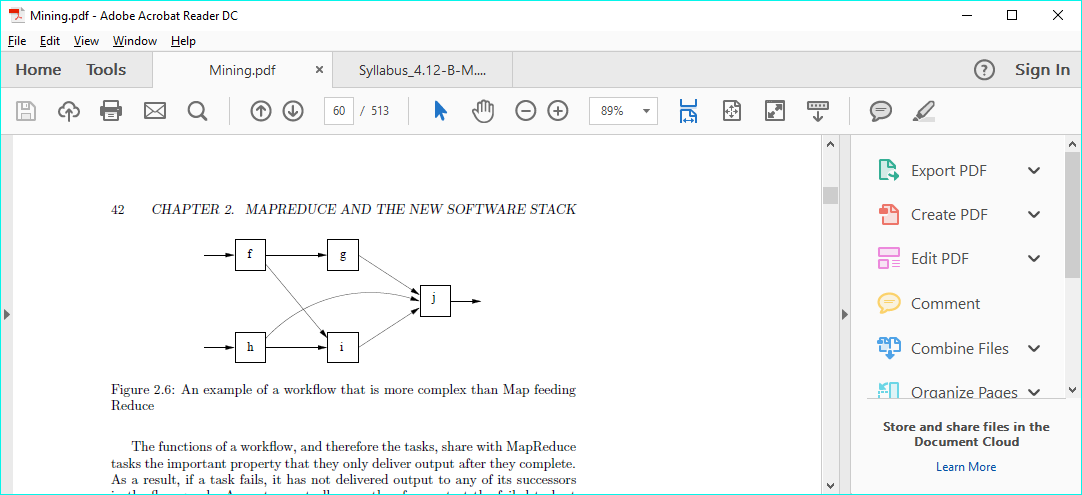
* There often is more than one way to use MapReduce to solve a problem. You may wish to use only a single MapReduce pass to perform matrix multiplicationP = MN.
* It is possible to do so if we put more work into the two functions.
* Start by using the Map function to create the sets of matrix elements that are needed to compute each element of the answer P. Notice that an element of M or N contributes to many elements of the result, so one input element will be turned into many key-value pairs.
* The keys will be pairs (i, k), where i is a row of M and k is a column of N. Here is a synopsis of the Map and Reduce functions.
* **The Map Function:** For each element mij of M, produce all the key-value pairs (i, k), (M, j,mij )\_ for k = 1, 2, . . ., up to the number of columns of N. Similarly, for each element njk of N, produce all the key-value pairs (i, k), (N, j, njk)\_ for i = 1, 2, . . ., up to the number of rows of M.
* **The Reduce Function:** Each key (i, k) will have an associated list with all the values (M, j,mij ) and (N, j, njk), for all possible values of j. The Reduce function needs to connect the two values on the list that have the same value of j, for each j. An easy way to do this step is to sort by j the values that begin with M and sort by j the values that begin with N, in separate lists. The jth values on each list must have their third components, mij and njk extracted and multiplied. Then, these products are summed and the result is paired with (i, k) in the output of the Reduce function.

**2.4.2 Recursive Extensions to MapReduce**

* Many large-scale computations are really recursions. An important example is PageRank.
* That computation is simple. The computation of the fixed point of a matrix-vector multiplication.
* It is computed under MapReduce systems by the iterated application of the matrix-vector multiplication algorithm or by a more complex strategy.
* The iteration typically continues for an unknown number of steps, each step being a MapReduce job, until the results of two consecutive iterations are sufficiently close that we believe convergence has occurred.
* The reason recursions are normally implemented by iterated MapReduce jobs is that a true recursive task does not have the property necessary for independent restart of failed tasks.
* It is impossible for a collection of mutually recursive tasks, each of which has an output that is input to at least some of the other tasks, to produce output only at the end of the task.
* If they all followed that policy, no task would ever receive any input, and nothing could be accomplished.
* As a result, some mechanism other than simple restart of failed tasks must be implemented in a system that handles recursive workflows (flow graphs that are not acyclic).

**Workflow Systems**

* In analogy to Map and Reduce functions, each function of a workflow can be executed by many tasks, each of which is assigned a portion of the input to the function.
* A master controller is responsible for dividing the work among the tasks that implement a function, usually by hashing the input elements to decide on the proper task to receive an element.
* Thus, like Map tasks, each task implementing a function f has an output file of data destined for each of the tasks that implement the successor function(s) off.
* These files are delivered by the Master at the appropriate time – after the task has completed its work.
* The functions of a workflow, and therefore the tasks, share with MapReduce tasks the important property that they only deliver output after they complete.
* As a result, if a task fails, it has not delivered output to any of its successors in the flow graph.
* A master controller can therefore restart the failed task at another compute node, without worrying that the output of the restarted task will duplicate output that previously was passed to some other task.



* Many applications of workflow systems such as Clustera or Hyracks are
* cascades of MapReduce jobs. An example would be the join of three relations,where one MapReduce job joins the first two relations, and a second MapReduce job joins the third relation with the result of joining the first two relations.
* There is an advantage to implementing such cascades as a single workflow.
* For example, the flow of data among tasks, and its replication, can be managed by the master controller, without need to store the temporary file that is output of one MapReduce job in the distributed file system. By locating tasks at compute nodes that have a copy of their input, we can avoid much of the communication that would be necessary if we stored the result of one MapReduce job and then initiated a second MapReduce job (although Hadoop and other MapReduce systems also try to locate Map tasks where a copy of their input is already present).