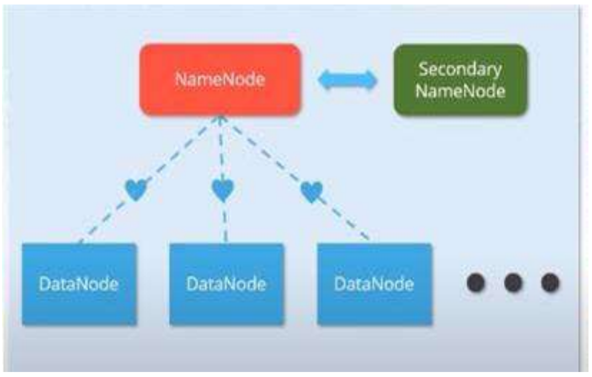
Module 2: Hadoop HDFS and MapReduce

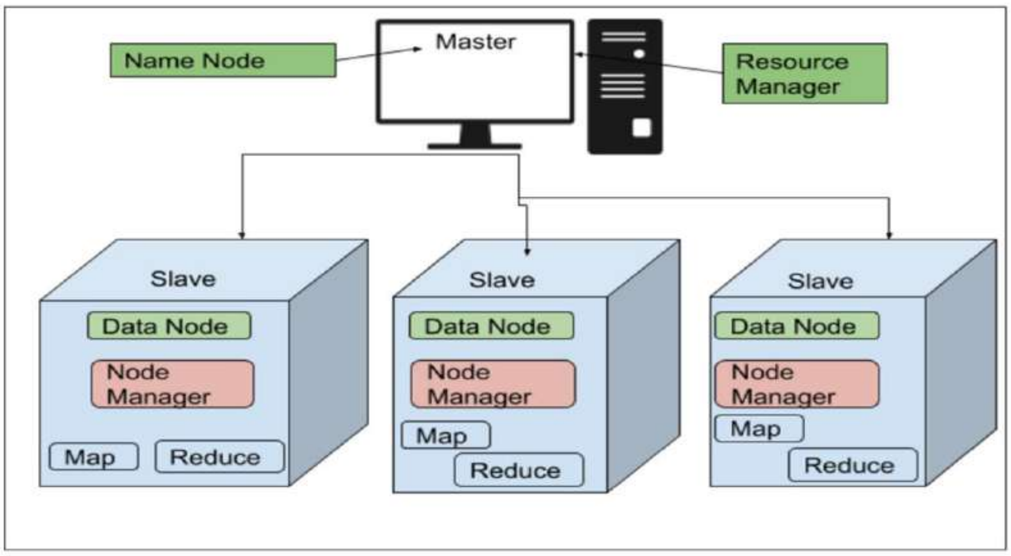
HDFS

HDFS(Hadoop Distributed File System) is utilized for storage permission. It is mainly designed for working on commodity Hardware devices (inexpensive devices), working on a distributed file system design. HDFS is designed in such a way that it believes more in storing the data in a large chunk of blocks rather than storing small data blocks. HDFS in Hadoop provides Fault-tolerance and High availability to the storage layer and the other devices present in that Hadoop cluster. Data storage Nodes in HDFS.

NameNode(Master)
DataNode(Slave)

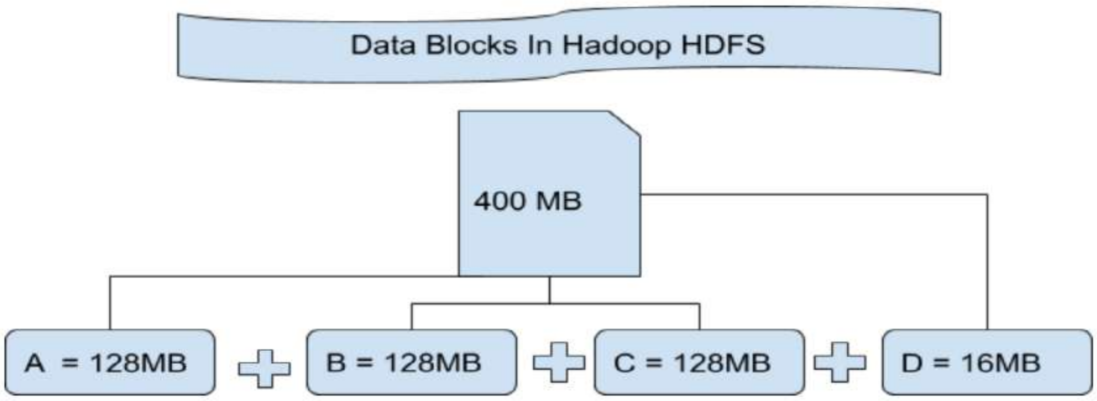


- ✓ NameNode: NameNode works as a Master in a Hadoop cluster that guides the Datanode(Slaves). Namenode is mainly used for storing the Metadata i.e. the data about the data. Meta Data can be the transaction logs that keep track of the user's activity in a Hadoop cluster. Meta Data can also be the name of the file, size, and the information about the location(Block number, Block ids) of Datanode that Namenode stores to find the closest DataNode for Faster Communication. Namenode instructs the DataNodes with the operation like delete, create, Replicate, etc.
- ✓ DataNode: DataNodes works as a Slave DataNodes are mainly utilized for storing the data in a Hadoop cluster, the number of DataNodes can be from 1 to 500 or even more than that. The more number of DataNode, the Hadoop cluster will be able to store more data. So it is advised that the DataNode should have High storing capacity to store a large number of file blocks.



✓

- ✓ File Block In HDFS: Data in HDFS is always stored in terms of blocks. So the single block of data is divided into multiple blocks of size 128MB which is default and you can also change it manually.



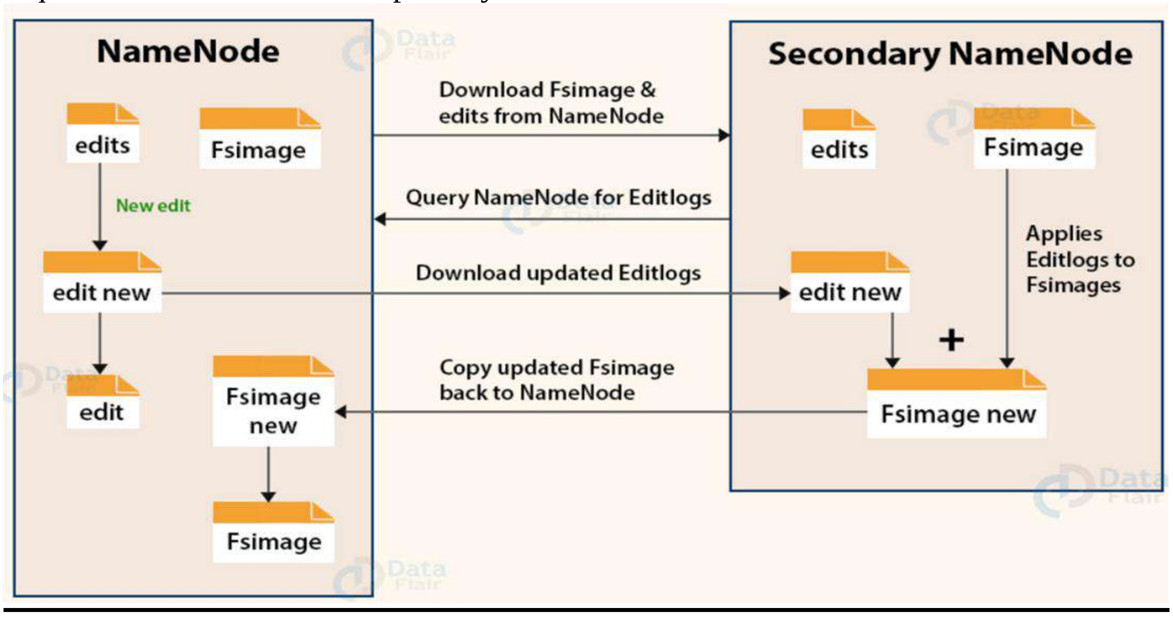
- ✓
- ✓ Let's understand this concept of breaking down of file in blocks with an example. Suppose you have uploaded a file of 400MB to your HDFS then what happens is this file got divided into blocks of 128MB+128MB+128MB+16MB = 400MB size. Means 4 blocks are created each of 128MB except the last one. Hadoop doesn't know or it doesn't care about what data is stored in these blocks so it considers the final file blocks as a partial record as it does not have any idea regarding it. In the Linux file system, the size of a file block is about 4KB which is very much less than the default size of file blocks in the Hadoop file system. As we all know Hadoop is mainly configured for storing the large size data which is in petabyte, this is what makes Hadoop file system different from other file systems as it can be scaled, nowadays file blocks of 128MB to 256MB are considered in Hadoop.
- ✓ Replication In HDFS Replication ensures the availability of the data. Replication is making a copy of something and the number of times you make a copy of that particular thing can be expressed as it's Replication Factor. As we have seen in File blocks that the HDFS stores the data in the form of various blocks at the same time Hadoop is also configured to make a copy of those file blocks.
- ✓ By default, the Replication Factor for Hadoop is set to 3 which can be configured means you can change it manually as per your requirement like in above example we have made 4 file blocks which means that 3 Replica or copy of each file block is made means total of 4×3 = 12 blocks are made for the backup purpose.

Secondary Name-Node

- It is not a backup Name Node. Name node stores metadata in main memory as well as in disks for persistent storage. This information is stored in two files:
- ✓ Edit logs- It keeps track of each and every change to HDFS.
- ✓ Fsimage- It stores the snapshot of the HDFS at a certain point of time.
- ✓ Any changes done to HDFS gets noted to edit logs, the file size increases and size of fs image remains same. This will not cause any problem until we restart the server. Only at startup, NameNode merges FsImage and EditLog files, the edits log file might get very large over time on a busy cluster.
- ✓ Whenever a NameNode is restarted, the latest status of FsImage is built by applying edits records on last saved copy of Fs Image. That means, if the EditLog is very large,NameNode restart process result in some considerable delay in the availability of file system.
- ✓ So, it is important keep the edits log as small as possible which is one of the main functions of

Secondary NameNode.

- ✓ Secondary NameNode in hadoop is a specially dedicated node in HDFS cluster whose main function is to take checkpoints of the file system metadata present on Name Node. It just checkpoints Name node's file system namespace. Usually, the new fsimage from merge operation is called as a checkpoint.
- ✓ Create periodic checkpoints of file system metadata by merging edits file with fsimage file.
- ✓ The Secondary Name Node is a helper to the primary Name Node but not replacement for primary Name Node. As the Name Node is the single point of failure in HDFS, if Name Node fails entire HDFS file system is lost. So in order to overcome this, Hadoop implemented Secondary NameNode whose main function is to store a copy of Fs Image file and edits log file.
- ✓ It stores the latest checkpoint in a directory that has the same structure as the Namenode's directory.
- ✓ This permits the check pointed image to be always available for reading by the NameNode if necessary. It usually runs on a different machine than the primary NameNode since its memory requirements are same as the primary NameNode.


NameNode
edits
Fsimage
New edit
edit new
edit
Fsimage new
Fsimage
Download Fsimage & edits from NameNode
Query NameNode for Editlogs
Download updated Editlogs
Copy updated Fsimage back to NameNode
Secondary NameNode
edits
Fsimage
Applies Editlogs to Fsimages
edit new
+
Fsimage new

Features of HDFS

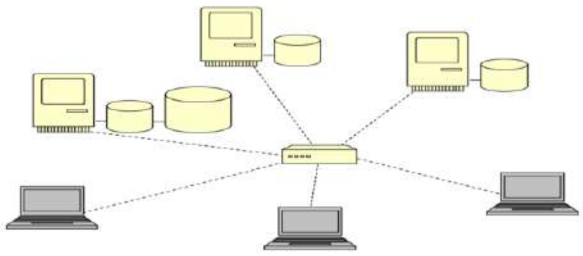
1. Cost-effective: In HDFS architecture, the DataNodes, which stores the actual data are inexpensive commodity hardware, thus reduces storage costs.
2. Large Datasets/ Variety and volume of data: HDFS can store data of any size (ranging from megabytes to petabytes) and of any formats (structured, unstructured).
3.Replication: Data Replication is one of the most important and unique features of HDFS. In HDFS replication of data is done to solve the problem of data loss in unfavorable conditions like crashing of a node, hardware failure, and so on. The data is replicated across a number of machines in the cluster by creating replicas of blocks. Hence whenever any machine in the cluster gets crashed, the user can access their data from other machines that contain the blocks of that data. Hence there is no possibility of a loss of user data.
4. Fault Tolerance and reliability: HDFS is highly fault-tolerant and reliable. HDFS creates replicas of file blocks depending on the replication factor and stores them on different machines. If any of the machines containing data blocks fail, other DataNodes containing the replicas of that data blocks are available. Thus, ensuring no loss of data and makes the system reliable even in unfavorable conditions.
5. High Availability: The High availability feature of Hadoop ensures the availability of data even during NameNode or DataNode failure. HDFS creates replicas of data blocks, if any of the DataNodes goes down, the user can access his data from the other DataNodes. If the active NameNode goes down, the passive node takes the responsibility of the

active NameNode. Thus, data will be available and accessible to the user even during a machine crash.
6 scalability: As HDFS stores data on multiple nodes in the cluster, when requirements increase, we can scale the cluster. There are two scalability mechanism available:
Vertical scalability - add more resources (CPU, Memory, Disk) on the existing nodes of the cluster.
Horizontal scalability - Add more machines in the cluster. The horizontal way is preferred since we can scale the cluster from 10s of nodes to 100s of nodes on the fly without any downtime.
7. Data Integrity: Data integrity refers to the correctness of data. HDFS ensures data integrity by constantly checking the data against the checksum calculated during the write of the file. While file reading, if the checksum does not match with the original checksum, the data is said to be corrupted. The client then opts to retrieve the data block from another DataNode that has a replica of that block. The NameNode discards the corrupted block and creates an additional new replica.
8. High Throughput : Hadoop HDFS stores data in a distributed fashion, which allows data to be processed parallely on a cluster of nodes. This decreases the processing time and thus high throughput.
9. Data Locality: Data locality means moving computation logic to the data rather than moving data to the computational unit. In the traditional system, the data is brought at the application layer and then gets processed. But in the present scenario, due to the massive volume of data, bringing data to the application layer degrades the network performance. In HDFS, we bring the computation part to the Data Nodes where data resides. This feature reduces the bandwidth utilization in a system.

01
Cost-effective
02
Large Datasets
03
Replication
04
Fault Tolerance
05
High Availability
06
Scalability
07
Data Integrity
08
High Throughput
09
Data Locality

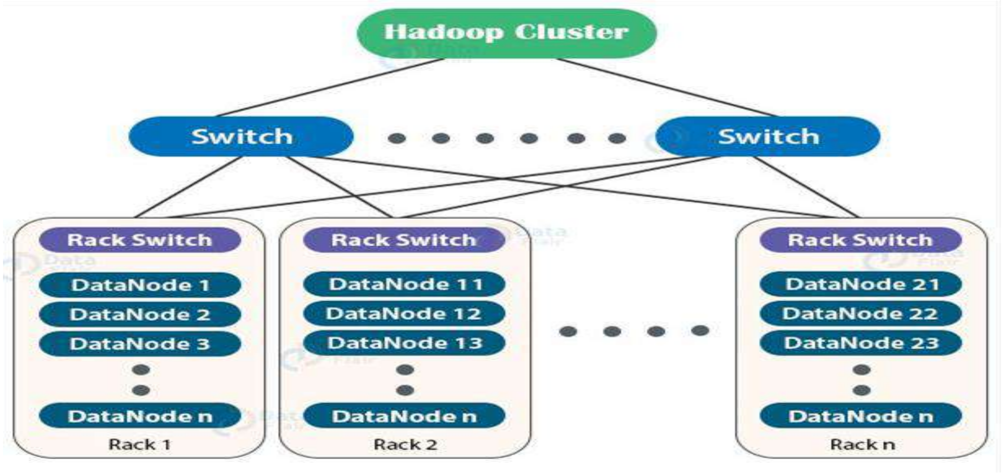
Distributed File System:

A distributed file system (DFS) is a file system that is distributed on various file servers and locations
A DFS manages set of dispersed storage devices
 It allows programs to access or store isolated files as they do with the local ones, allowing programmers to access files from any network or computer.
The main purpose is to allows users of physically distributed systems to share their data and resources by using a Common File System.
DFS has two properties:
Location Transparency –achieves through the namespace component.
File names are identified by the prefixes fs/or _fs/.
For example the name fs/etc/motd identifies the file motd which is stored in the /etc directory.
Redundancy – is done through a file replication component.



Physical organization of computer nodes

The new parallel-computing architecture, sometimes called cluster computing. Cluster organized as compute nodes are stored on racks, perhaps 8-64 on a rack. The Rack is the collection of around 40-50 DataNodes connected using the same network switch. If the network goes down, the whole rack will be unavailable. A large Hadoop cluster is deployed in multiple racks.



The nodes on a single rack are connected by a network, typically gigabit Ethernet. There can be many racks of compute nodes, and racks are connected by another level of network or a switch. The bandwidth of inter-rack communication is somewhat greater than the intra rack Ethernet.

Rack Awareness

- To reduce the network traffic during file read/write, NameNode chooses the closest DataNode for serving the client read/write request. NameNode maintains rack ids of each DataNode to achieve this rack information. This concept of choosing the closest Data Node based on the rack information is known as Rack Awareness.
- The reasons for the Rack Awareness in Hadoop are:

✓ To reduce the network traffic while file read/write, which improves the cluster performance.
✓ To achieve fault tolerance, even when the rack fails.
✓ Achieve high availability of data so that data is available even in unfavorable conditions.
✓ To reduce the latency, that is, to make the file read/write operations done with lower delay.

Rack Awareness Algorithm

To ensure that all the replicas of a block are not stored on the same rack or a single rack, NameNode follows a rack awareness algorithm to store replicas and provide latency and fault tolerance.

✓ Not more than one replica be placed on one node.
✓ Not more than two replicas are placed on the same rack.
✓ Also, the number of racks used for block replication should always be smaller than the number of replicas.

- **Suppose if the replication factor is 3, then according to the rack awareness algorithm:**

✓ The first replica will get stored on the local rack.
✓ The second replica will get stored on the other Data Node in the same rack.
✓ The third replica will get stored on a different rack, near to first rack, so as to have higher bandwidth and low latency.

<u>LARGE SCALE FILE SYSTEM ORGANIZATION</u>

In cluster computing, files must look and behave somewhat differently from the conventional file systems found on single computers.

This new file system, often called a distributed file system or DFS is typically used as follows:

- ➢ Files can be enormous, possibly a terabyte in size
- ➢ If you have only small files, there is no point using a DFS for them.
- ➢ Files are rarely updated
- ➢ Rather, they are read as data for some calculation, and possibly additional data is appended to files from time to time.

Files are divided into chunks, which are typically 64 or 128 megabytes in size.
Chunks are replicated, perhaps three times, at three different compute nodes.
The nodes holding copies of one chunk should be located on different racks, so we don't lose all copies due to a rack failure.
There are several distributed file systems:
1. The Google File System (GFS), the original of the class.
2. Hadoop Distributed File System (HDFS), an open-source DFS used with Hadoop, an implementation of map-reduce and distributed by the Apache Software Foundation.
3. CloudStore, an open-source DFS originally developed by Kosmix.racks, so we don’t lose all copies due to a rack failure.
Normally, both the chunk size and the degree of replication can be decided by the user.
To find the chunks of a file, there is another small metadata file maintained by the master node or name node for that file.
All participants using the DFS know where the chunk copies are.

MAPREDUCE:

1. MapReduce is a **software framework**.
2. MapReduce is the data processing layer of Hadoop.
3. Similar to HDFS, MapReduce also exploits master/slave architecture in which JobTracker runs on master node and TaskTracker runs on each salve node.
4. Task Trackers are processes running on data nodes.
5. These monitors the maps and reduce tasks executed on the node and coordinates with Job tracker.
6. Job Tracker monitors the entire MR job execution.
7. JobTracker and TaskTracker are 2 essential process involved in MapReduce execution in MRv1 (or Hadoop version 1).
8. Both processes are now deprecated in MRv2 (or Hadoop version 2) and replaced by Resource Manager, Application Master and Node Manager Daemons.

JOB TRACKER:

1. JobTracker is an essential Daemon for MapReduce execution in MRv1.
2. It is replaced by Resource Manager / Application Master in **MRv2**.
3. JobTracker receives the requests for MapReduce execution from the client.
4. JobTracker talks to the NameNode to determine the location of the data.
5. JobTracker finds the best Task Tracker nodes to execute tasks based on the data locality (proximity of the data) and the available slots to execute a task on a given node.
6. JobTracker monitors the individual Task Trackers and the submits back the overall status of the job back to the client.
7. If a task fails, the JobTracker can reschedule it on a different TaskTrackers.
8. When the JobTracker is down, HDFS will still be functional but the MapReduce execution cannot be started and the existing MapReduce jobs will be halted.

TASKTRACKER:

1. TaskTracker **runs on DataNode**.
2. TaskTracker is replaced by Node Manager in MRv2.
3. Mapper and Reducer tasks are executed on DataNodes administered by TaskTrackers.
4. TaskTrackers will be assigned Mapper and Reducer tasks to execute by JobTracker.
5. TaskTracker will be in constant communication with the JobTracker signalling the progress of the task in execution.
6. TaskTracker failure is not considered fatal. When a TaskTracker becomes unresponsive, JobTracker will assign the task executed by the TaskTracker to another node.

A WORD COUNT EXAMPLE OF MAPREDUCE:

1. Let us understand, how a MapReduce works by taking an example where I have a text file called example.txt whose contents are as follows:

Dog, Cat, Mouse, Dog, Dog, Cat, Dog, Cat and Duck

2. Now, suppose, we have to perform a word count on the sample.txt using MapReduce.
3. So, we will be finding the unique words and the number of occurrences of those unique words.

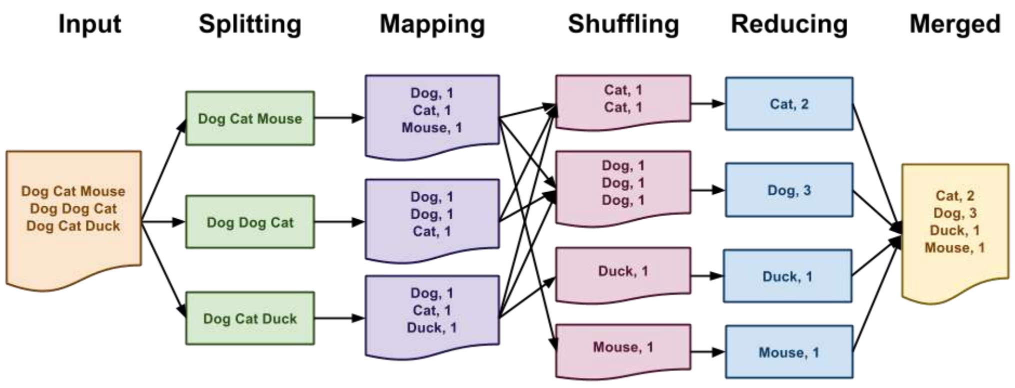


Figure 2.1: A WORD COUNT EXAMPLE OF MAPREDUCE

4. First, we divide the input into three splits as shown in the figure 2.1.
5. This will distribute the work among all the map nodes.
6. Then, we tokenize the words in each of the mappers and give a hardcoded value (1) to each of the tokens or words.
7. The rationale behind giving a hardcoded value equal to 1 is that every word, in itself, will occur once.
8. Now, a list of key-value pair will be created where the key is nothing, but the individual words and value is one.
9. So, for the first line (Dog Cat Mouse) we have 3 key-value pairs – Dog, 1; Cat, 1; Mouse, 1.
10. The mapping process remains the same on all the nodes.
11. After the mapper phase, a partition process takes place where sorting and shuffling happen so that all the tuples with the same key are sent to the corresponding reducer.
12. So, after the sorting and shuffling phase, each reducer will have a unique key and a list of values corresponding to that very key. For example, Cat, [1,1]; Dog, [1,1,1].., etc.
13. Now, each Reducer counts the values which are present in that list of values.
14. As shown in the figure 2.1, reducer gets a list of values which is [1,1] for the key Cat.
15. Then, it counts the number of ones in the very list and gives the final output as – Cat, 2.
16. Finally, all the output key/value pairs are then collected and written in the output file.

Function of Map Tasks in the Map Reduce framework

1. In the MapReduce framework, the Map task is one of the two main stages in the data processing pipeline, along with the Reduce task.
2. The Map task is responsible for processing and transforming input data into a set of intermediate key-value pairs.

The function of the Map task can be described as follows:

1. Input Data Splitting: The input data is divided into smaller chunks, which are then assigned to different Map tasks for parallel processing. Each Map task operates on a separate portion of the input data.
2. Data Transformation: The Map task takes each input record and applies a user-defined map function to it. The map function takes the input record and generates one or more key-value pairs as output. It can perform any required processing or computation on the input data.
3. Intermediate Key-Value Pair Generation: The map function produces a set of intermediate key-value pairs as its output. These key-value pairs are not the final result but are used as input for the subsequent Reduce tasks.
4. Sorting and Shuffling: The intermediate key-value pairs generated by different Map tasks are sorted based on their keys and then shuffled across the network. The purpose of sorting and shuffling is to group together the values associated with each unique key. This allows the subsequent Reduce tasks to process all the values for a particular key in one place.
5. Intermediate Key-Value Pair Storage: The sorted and shuffled intermediate key-value pairs are typically stored on disk, partitioned into multiple files based on the keys. This storage facilitates efficient retrieval during the Reduce stage.

Here's an example to illustrate the Map task:

Let's say we have a large collection of documents, and we want to count the occurrences of each word across all the documents. We can use the MapReduce framework to achieve this.

In the Map stage:

1. Each Map task processes a portion of the input documents.
2. The map function takes a document as input and tokenizes it into individual words.
3. For each word encountered, the map function emits a key-value pair where the word is the key and the value is 1 (indicating the occurrence of the word).
4. For instance, if the Map task processes the document "Hello world, hello MapReduce," it would generate the following intermediate key-value pairs:
   ("Hello", 1)
   ("world", 1)
   ("hello", 1)
   ("MapReduce", 1)
5. The Map tasks operate in parallel, processing different portions of the documents and emitting intermediate key-value pairs.
6. Once the Map stage is complete, the intermediate key-value pairs are sorted, shuffled, and grouped by key, preparing them for the subsequent Reduce stage.
7. The Reduce tasks then process the grouped intermediate key-value pairs and perform further computations to produce the final output, such as aggregating the word counts.
8. Overall, the Map task in the MapReduce framework performs the initial data processing, transformation,

and intermediate key-value pair generation, preparing the data for the subsequent Reduce tasks to perform more complex computations.

COMBINERS:

1. A combiner is also known as a **semi-reducer**.
2. It is one of mediator between the mapper phase & the reducer phase.
3. The use of combiners is totally optional.
4. It accepts the output of map phase as an input and pass the key-value pair to the reduce operation.
5. The main function of a Combiner is to summarize the map output records with the same key.
6. It is also known as **grouping by key**.
7. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.
8. 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.
9. Usually, the output of the map task is large and the data transferred to the reduce task is high.
10. The following figure 2.2 shows position and working mechanism of combiner.

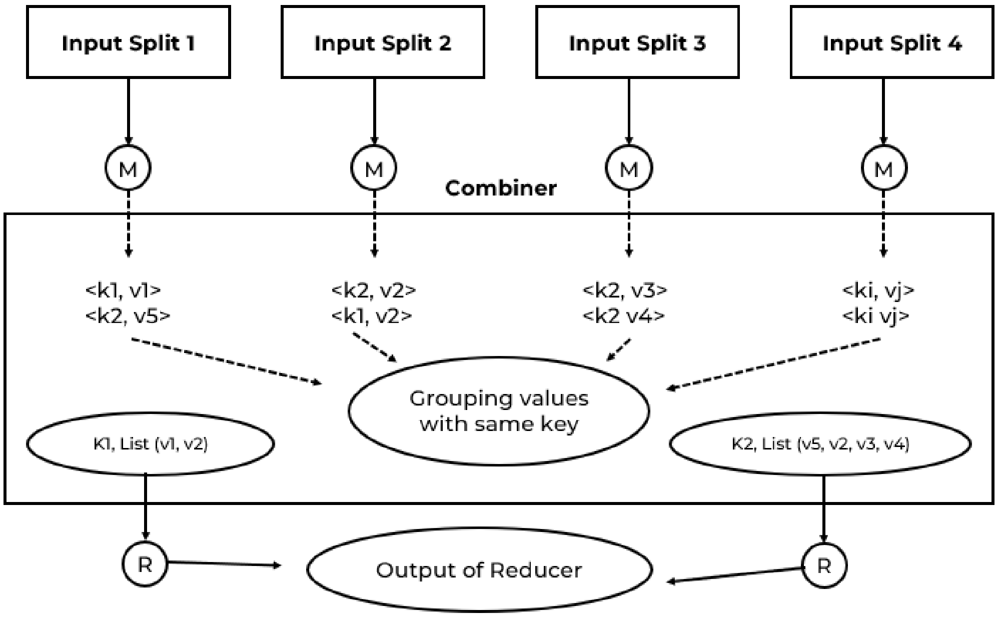


Figure 2.2: Position and working mechanism of combiner M = Mapper and R = Reducer

Working:

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

MATRIX VECTOR MULTIPLICATION:

Algorithm 1: The Map Function

```
1 for each element $m_{ij}$ of $M$ do
2     produce $(key, value)$ pairs as $((i,k),(M,j,m_{ij}))$, for $k = 1,2,3,..$ up
      to the number of columns of $N$
3 for each element $n_{jk}$ of $N$ do
4     produce $(key, value)$ pairs as $((i,k),(N,j,n_{jk}))$, for $i = 1,2,3,...$ up
      to the number of rows of $M$
5 return Set of (key, value) pairs that each key, $(i,k)$, has a list with
  values $(M,j,m_{ij})$ and $(N,j,n_{jk})$ for all possible values of $j$
```

Algorithm 2: The Reduce Function

```
1 for each key (i,k) do
2     sort values begin with $M$ by $j$ in $list_M$
3     sort values begin with $N$ by $j$ in $list_N$
4     multiply $m_{ij}$ and $n_{jk}$ for $j_{th}$ value of each list
5     sum up $m_{ij} * n_{jk}$
6 return $(i,k), \sum_{j=1} m_{ij} * n_{jk}$
```

Let us consider the matrix multiplication example to visualize MapReduce.
Consider the following matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

2×2 matrices A and B

Here matrix A is a 2×2 matrix which means the number of rows(i)=2 and the number of columns(j)=2. Matrix B is also a 2×2 matrix where number of rows(j)=2 and number of columns(k)=2. Each cell of the matrix is labelled as Aij and Bij. Ex. element 3 in matrix A is called A21 i.e. 2nd-row 1st column. Now One step matrix multiplication has 1 mapper and 1 reducer. The Formula is:

Mapper for Matrix A (k, v)=((i, k), (A, j, Aij)) for all k
Mapper for Matrix B (k, v)=((i, k), (B, j, Bjk)) for all i

Therefore computing the mapper for Matrix A:

```
# k, i, j computes the number of times it occurs.
# Here all are 2, therefore when k=1, i can have
# 2 values 1 & 2, each case can have 2 further
# values of j=1 and j=2. Substituting all values
# in formula

k=1  i=1  j=1   ((1, 1), (A, 1, 1))
          j=2   ((1, 1), (A, 2, 2))
     i=2  j=1   ((2, 1), (A, 1, 3))
          j=2   ((2, 1), (A, 2, 4))

k=2  i=1  j=1   ((1, 2), (A, 1, 1))
          j=2   ((1, 2), (A, 2, 2))
     i=2  j=1   ((2, 2), (A, 1, 3))
          j=2   ((2, 2), (A, 2, 4))
```

Computing the mapper for Matrix B

```
i=1  j=1  k=1   ((1, 1), (B, 1, 5))
          k=2   ((1, 2), (B, 1, 6))
     j=2  k=1   ((1, 1), (B, 2, 7))
          k=2   ((1, 2), (B, 2, 8))

i=2  j=1  k=1   ((2, 1), (B, 1, 5))
          k=2   ((2, 2), (B, 1, 6))
     j=2  k=1   ((2, 1), (B, 2, 7))
          k=2   ((2, 2), (B, 2, 8))
```

The formula for Reducer is:

Reducer(k, v)=(i, k)=>Make sorted Alist and Blist
*(i, k) => Summation (Aij * Bjk)) for j*
Output =>((i, k), sum)

Therefore computing the reducer:

```
# We can observe from Mapper computation
# that 4 pairs are common (1, 1), (1, 2),
# (2, 1) and (2, 2)
# Make a list separate for Matrix A &
# B with adjoining values taken from
# Mapper step above:

(1, 1) =>Alist ={(A, 1, 1), (A, 2, 2)}
        Blist ={(B, 1, 5), (B, 2, 7)}
        Now Aij x Bjk: [(1*5) + (2*7)] =19 -------(i)

(1, 2) =>Alist ={(A, 1, 1), (A, 2, 2)}
        Blist ={(B, 1, 6), (B, 2, 8)}
        Now Aij x Bjk: [(1*6) + (2*8)] =22 -------(ii)

(2, 1) =>Alist ={(A, 1, 3), (A, 2, 4)}
        Blist ={(B, 1, 5), (B, 2, 7)}
        Now Aij x Bjk: [(3*5) + (4*7)] =43 -------(iii)

(2, 2) =>Alist ={(A, 1, 3), (A, 2, 4)}
        Blist ={(B, 1, 6), (B, 2, 8)}
        Now Aij x Bjk: [(3*6) + (4*8)] =50 -------(iv)

From (i), (ii), (iii) and (iv) we conclude that
((1, 1), 19)
((1, 2), 22)
((2, 1), 43)
((2, 2), 50)
```

Therefore the Final Matrix is:

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$