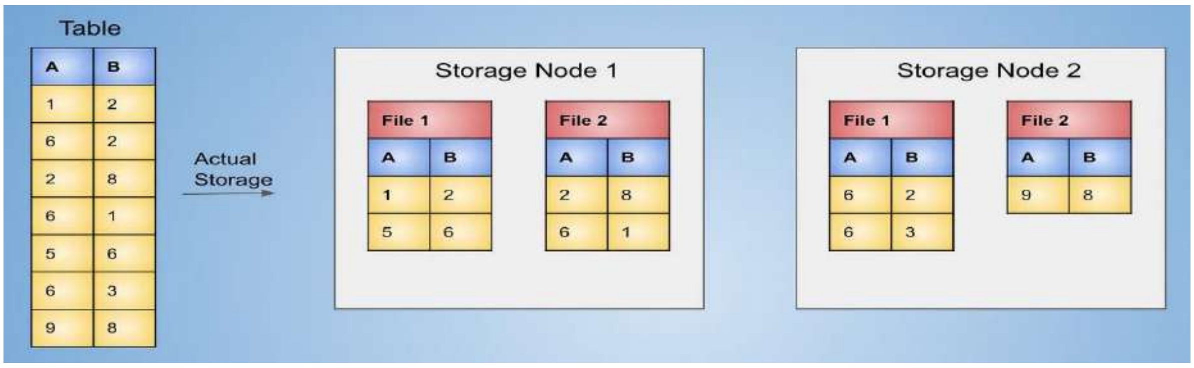
Module 2: Hadoop HDFS and MapReduce

Relational Algebra Operations

A relation represents a database table. In a distributed storage system the entire table isn't stored on a single node(computer), the most relevant reason being it doesn't fit completely on a single system because of its large size. So in order to store a table the table is partitioned into small files which are distributed across the nodes available in the system.



Table

A	B
1	2
6	2
2	8
6	1
5	6
6	3
9	8

Storage Node 1 — File 1

A	B
1	2
5	6

Storage Node 1 — File 2

A	B
2	8
6	1

Storage Node 2 — File 1

A	B
6	2
6	3

Storage Node 2 — File 2

A	B
9	8

Actual storage of a table on distributed file system

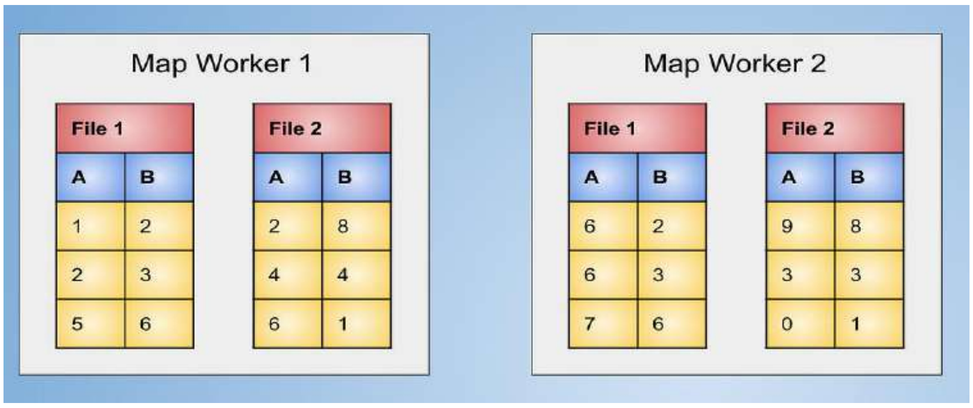
Computing Selections by MapReduce:

To perform selections using map reduce we need the following Map and Reduce functions:

- **Map Function**: For each row r in the table apply condition and produce a key value pair r, r if condition is satisfied else produce nothing. i.e. key and value are the same.
- **Reduce Function:** The reduce function has nothing to do in this case. It will simply write the value for each key it receives to the output.

For our example we will do Selection(B <= 3). Select all the rows where value of B is less than or equal to 3.

Let's consider the data we have initially distributed as files in Map Workers, And the data looks like the following figure



Map Worker 1 — File 1

A	B
1	2
2	3
5	6

Map Worker 1 — File 2

A	B
2	8
4	4
6	1

Map Worker 2 — File 1

A	B
6	2
6	3
7	6

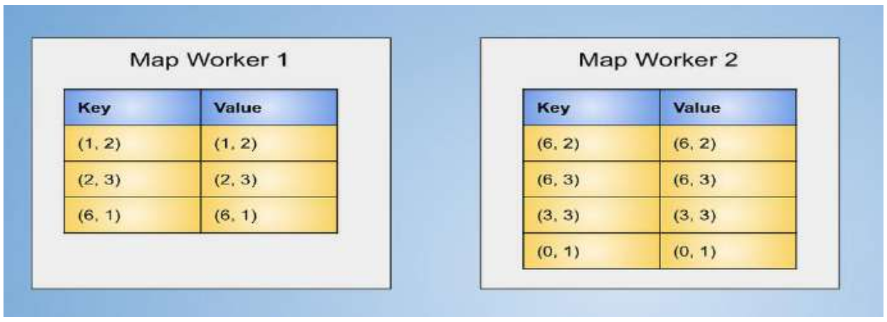
Map Worker 2 — File 2

A	B
9	8
3	3
0	1

Initial data distributed in files across map workers representing a single table

After applying the map function (And grouping, there are no common keys in this case as each row is unique) we will get the output as follows, **The tuples are constructed with 0th index containing values from A column and 1st index containing values from B.**

In actual implementations either this information can be sent as some extra metadata or within each value itself, making values and keys look something like ({A: 1}, {B: 2}), which does look somewhat inefficient.



Map Worker 1

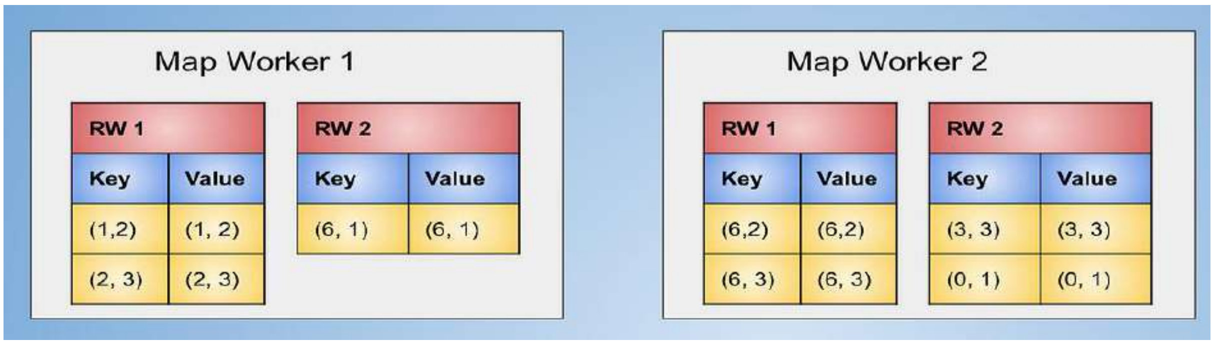
Key	Value
(1, 2)	(1, 2)
(2, 3)	(2, 3)
(6, 1)	(6, 1)

Map Worker 2

Key	Value
(6, 2)	(6, 2)
(6, 3)	(6, 3)
(3, 3)	(3, 3)
(0, 1)	(0, 1)

Data after applying Map function which filtered rows having B value less than 3

After this based on number or reduce workers (2 in our case). A hash function is applied as explained in the Hash Function section. The files for reduce workers on map workers will look like:



Map Worker 1

RW 1	
Key	Value
(1,2)	(1, 2)
(2, 3)	(2, 3)

RW 2	
Key	Value
(6, 1)	(6, 1)

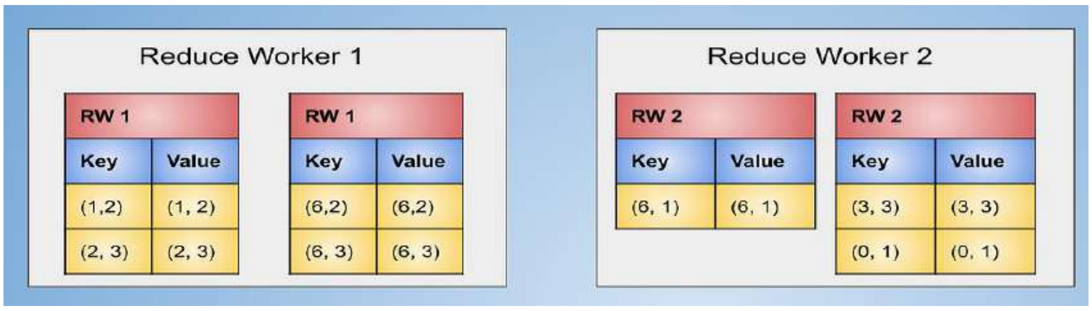
Map Worker 2

RW 1	
Key	Value
(6,2)	(6,2)
(6, 3)	(6, 3)

RW 2	
Key	Value
(3, 3)	(3, 3)
(0, 1)	(0, 1)

Files for reduce workers created at map worker based on hash function

After this step The files for reduce worker 1 are sent to that and reduce worker 2 are sent to that. The data at reduce workers will look like:



Reduce Worker 1

RW 1	
Key	Value
(1,2)	(1, 2)
(2, 3)	(2, 3)

RW 1	
Key	Value
(6,2)	(6,2)
(6, 3)	(6, 3)

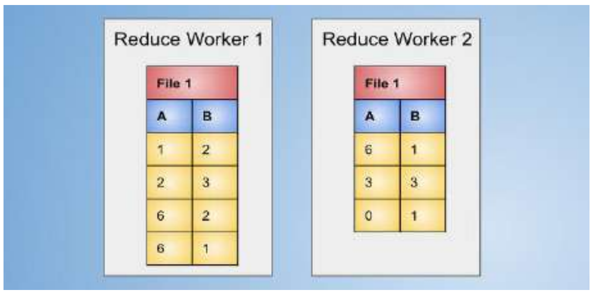
Reduce Worker 2

RW 2	
Key	Value
(6, 1)	(6, 1)

RW 2	
Key	Value
(3, 3)	(3, 3)
(0, 1)	(0, 1)

Data at reduce workers sent from map workers

The final output after applying the reduce function which ignores the keys and just consider values will look like:



Reduce Worker 1

File 1

A	B
1	2
2	3
6	2
6	1

Reduce Worker 2

File 1

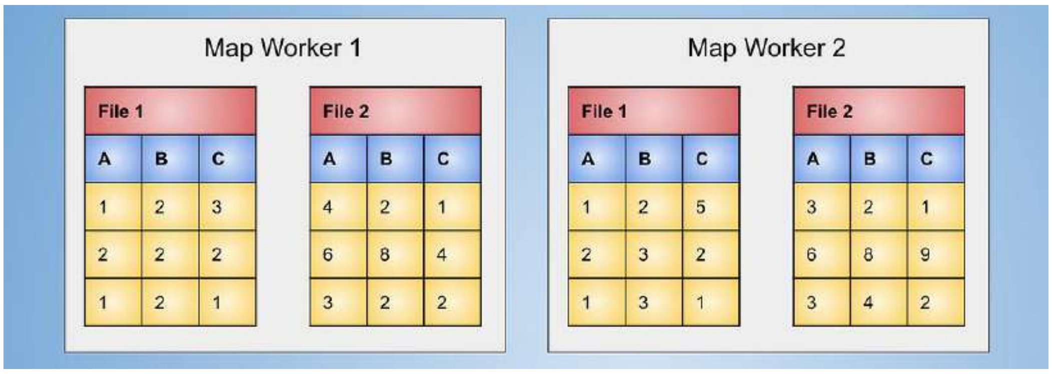
A	B
6	1
3	3
0	1

Output of selection(B $\leq$ 3)

Projection Using Map Reduce

- **Map Function:** For each row r in the table produce a key value pair r', r’, where r' only contains the columns which are wanted in the projection.
- **Reduce Function:** The reduce function will get outputs in the form of r' :[r', r', r', r', ...]. As after removing some columns the output may contain duplicate rows. So it will just take the value at 0th index, getting rid of duplicates.

Let’s see it in action, by computing projection(A, B) for the following table:



Map Worker 1

File 1

A	B	C
1	2	3
2	2	2
1	2	1

File 2

A	B	C
4	2	1
6	8	4
3	2	2

Map Worker 2

File 1

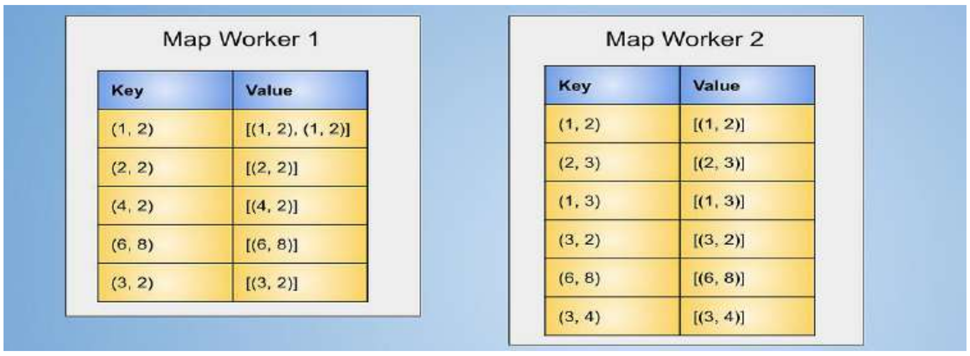
A	B	C
1	2	5
2	3	2
1	3	1

File 2

A	B	C
3	2	1
6	8	9
3	4	2

Initial Data distributed on map workers

After application of map function (ignoring values in C column) and grouping the keys the data will look like:



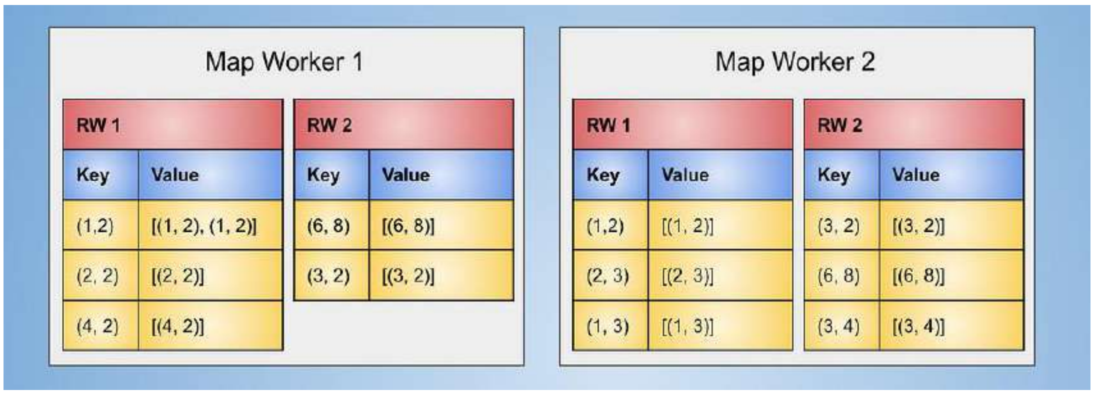
Map Worker 1

Key	Value
(1, 2)	[(1, 2), (1, 2)]
(2, 2)	[(2, 2)]
(4, 2)	[(4, 2)]
(6, 8)	[(6, 8)]
(3, 2)	[(3, 2)]

Map Worker 2

Key	Value
(1, 2)	[(1, 2)]
(2, 3)	[(2, 3)]
(1, 3)	[(1, 3)]
(3, 2)	[(3, 2)]
(6, 8)	[(6, 8)]
(3, 4)	[(3, 4)]

The keys will be partitioned using a hash function as was the case in selection. The data will look like:



Map Worker 1

RW 1

Key	Value
(1,2)	[(1, 2), (1, 2)]
(2, 2)	[(2, 2)]
(4, 2)	[(4, 2)]

RW 2

Key	Value
(6, 8)	[(6, 8)]
(3, 2)	[(3, 2)]

Map Worker 2

RW 1

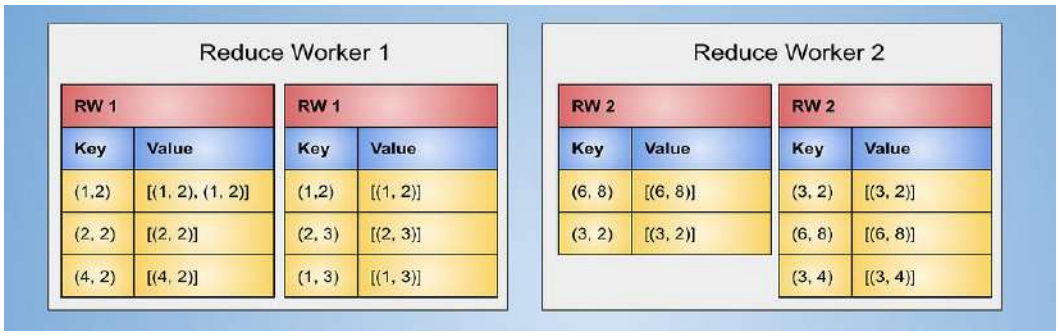
Key	Value
(1,2)	[(1, 2)]
(2, 3)	[(2, 3)]
(1, 3)	[(1, 3)]

RW 2

Key	Value
(3, 2)	[(3, 2)]
(6, 8)	[(6, 8)]
(3, 4)	[(3, 4)]

Files generated for reduce workers

The data at the reduce workers will be:



Reduce Worker 1

RW 1

Key	Value
(1,2)	[(1, 2), (1, 2)]
(2, 2)	[(2, 2)]
(4, 2)	[(4, 2)]

RW 1

Key	Value
(1,2)	[(1, 2)]
(2, 3)	[(2, 3)]
(1, 3)	[(1, 3)]

Reduce Worker 2

RW 2

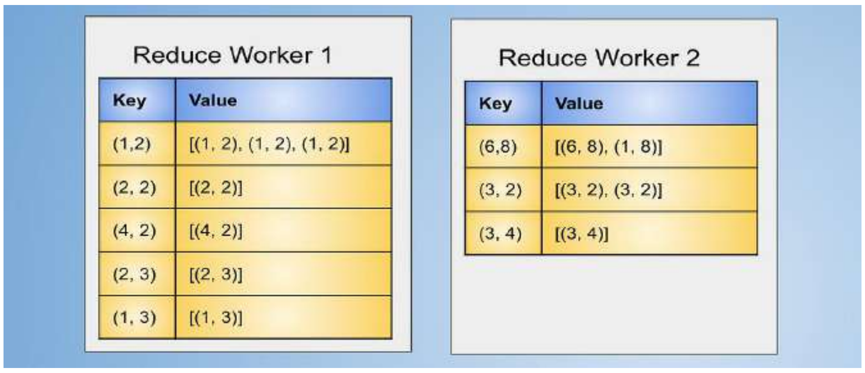
Key	Value
(6, 8)	[(6, 8)]
(3, 2)	[(3, 2)]

RW 2

Key	Value
(3, 2)	[(3, 2)]
(6, 8)	[(6, 8)]
(3, 4)	[(3, 4)]

Data at reduce workers

At the reduce node the keys will be aggregated again as same keys might have occurred at multiple map workers. As we already know the reduce function operates on values of each key only once.



Reduce Worker 1

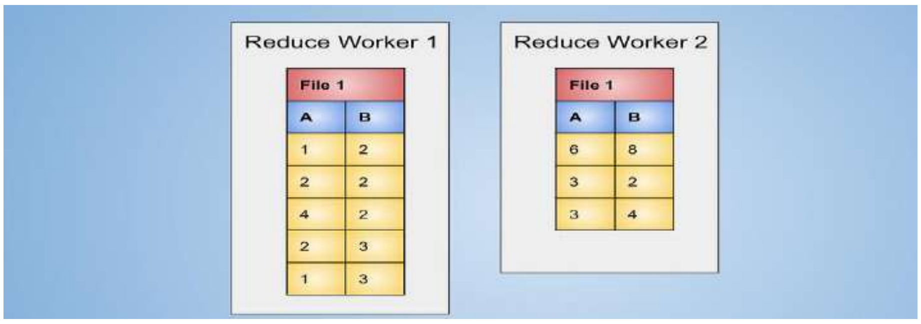
Key	Value
(1,2)	[(1, 2), (1, 2), (1, 2)]
(2, 2)	[(2, 2)]
(4, 2)	[(4, 2)]
(2, 3)	[(2, 3)]
(1, 3)	[(1, 3)]

Reduce Worker 2

Key	Value
(6,8)	[(6, 8), (1, 8)]
(3, 2)	[(3, 2), (3, 2)]
(3, 4)	[(3, 4)]

Data after aggregation by key at reduce workers

The reduce function is applied which will consider only the first value of the values list and ignore rest of the information.



Reduce Worker 1

File 1

A	B
1	2
2	2
4	2
2	3
1	3

Reduce Worker 2

File 1

A	B
6	8
3	2
3	4

Output of projection(A, B)

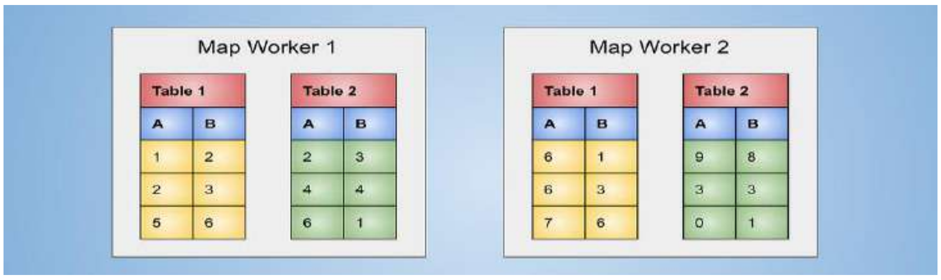
The points to remember are that here the reduce function is required for duplicate elimination

Union Using Map Reduce

Both selection and projection are operations that are applied on a single table, whereas Union, intersection and difference are among the operations that are applied on two or more tables. Let's consider that schemas of the two tables are the same, and columns are also ordered in same order.

- **Map Function:** For each row r generate key-value pair (r, r) .
- **Reduce Function:** With each key there can be one or two values (As we don't have duplicate rows), in either case just output first value.

This operations has the **map function of the selection** and **reduce function of projection**. Let's see the working using an example. Here yellow colour represents one table and green colour is used to represent the other one stored at two map workers.



Map Worker 1

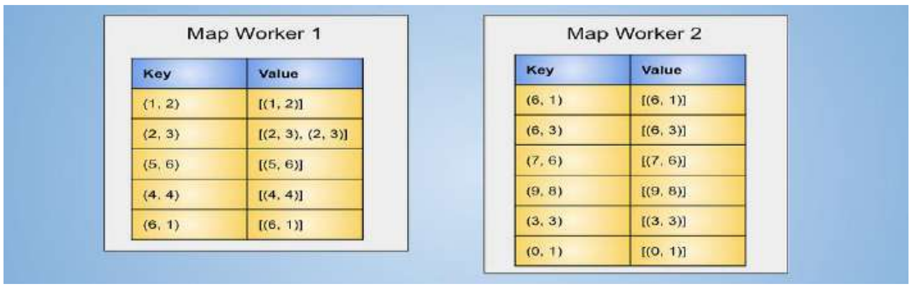
Table 1		Table 2	
A	B	A	B
1	2	2	3
2	3	4	4
5	6	6	1

Map Worker 2

Table 1		Table 2	
A	B	A	B
6	1	9	8
6	3	3	3
7	6	0	1

Initial data at map workers

After applying the map function and grouping the keys we will get output as:



Map Worker 1

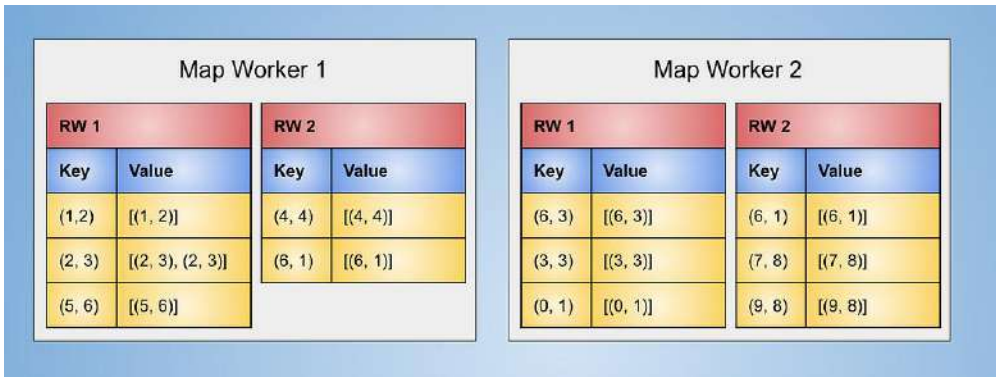
Key	Value
(1, 2)	[(1, 2)]
(2, 3)	[(2, 3), (2, 3)]
(5, 6)	[(5, 6)]
(4, 4)	[(4, 4)]
(6, 1)	[(6, 1)]

Map Worker 2

Key	Value
(6, 1)	[(6, 1)]
(6, 3)	[(6, 3)]
(7, 6)	[(7, 6)]
(9, 8)	[(9, 8)]
(3, 3)	[(3, 3)]
(0, 1)	[(0, 1)]

Map and grouping the keys

The data to be sent to reduce workers will look like:



Map Worker 1

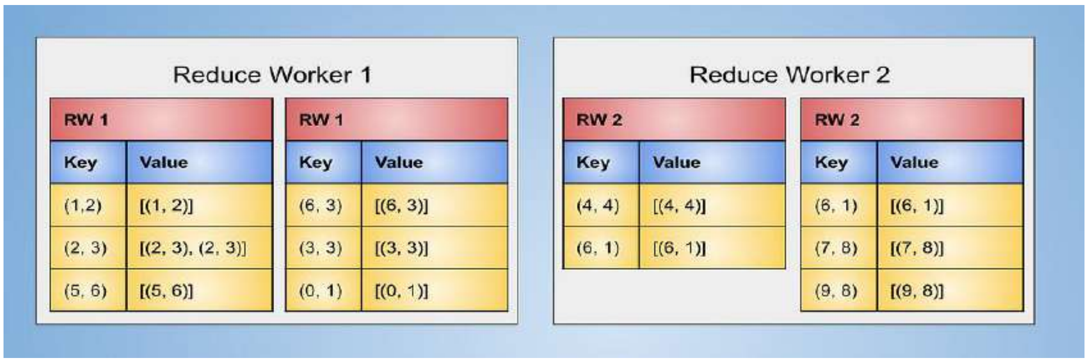
RW 1		RW 2	
Key	Value	Key	Value
(1,2)	[(1, 2)]	(4, 4)	[(4, 4)]
(2, 3)	[(2, 3), (2, 3)]	(6, 1)	[(6, 1)]
(5, 6)	[(5, 6)]		

Map Worker 2

RW 1		RW 2	
Key	Value	Key	Value
(6, 3)	[(6, 3)]	(6, 1)	[(6, 1)]
(3, 3)	[(3, 3)]	(7, 8)	[(7, 8)]
(0, 1)	[(0, 1)]	(9, 8)	[(9, 8)]

Files to be sent to reduce workers

Data at reduce workers after will be:



Reduce Worker 1

RW 1	
Key	Value
(1,2)	[(1, 2)]
(2, 3)	[(2, 3), (2, 3)]
(5, 6)	[(5, 6)]

RW 1	
Key	Value
(6, 3)	[(6, 3)]
(3, 3)	[(3, 3)]
(0, 1)	[(0, 1)]

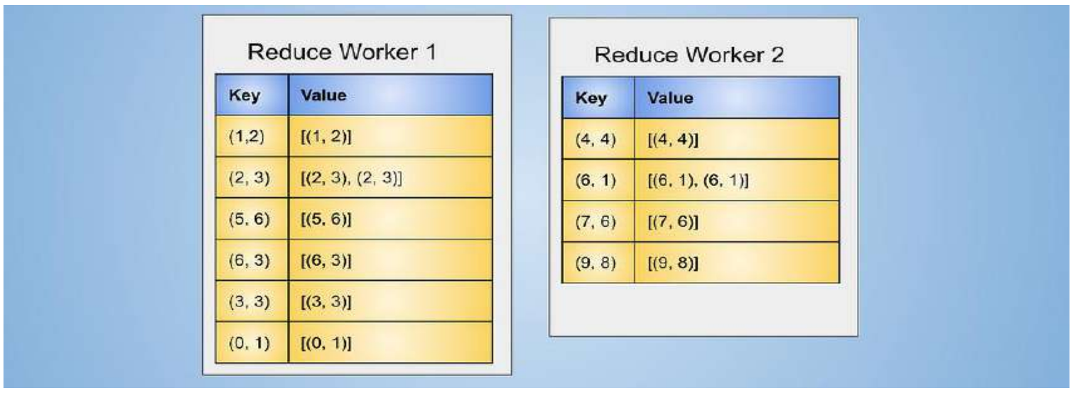
Reduce Worker 2

RW 2	
Key	Value
(4, 4)	[(4, 4)]
(6, 1)	[(6, 1)]

RW 2	
Key	Value
(6, 1)	[(6, 1)]
(7, 8)	[(7, 8)]
(9, 8)	[(9, 8)]

Files At reduce workers

At reduce workers aggregation on keys will be done.



Reduce Worker 1

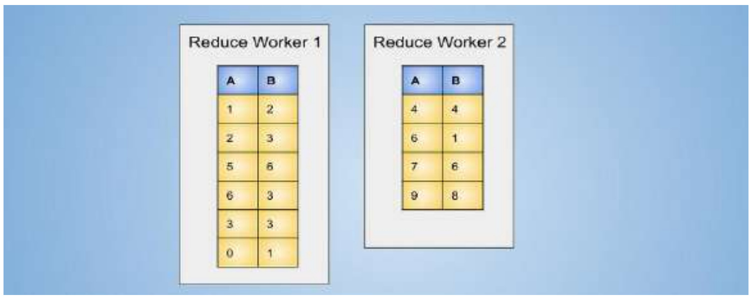
Key	Value
(1,2)	[(1, 2)]
(2, 3)	[(2, 3), (2, 3)]
(5, 6)	[(5, 6)]
(6, 3)	[(6, 3)]
(3, 3)	[(3, 3)]
(0, 1)	[(0, 1)]

Reduce Worker 2

Key	Value
(4, 4)	[(4, 4)]
(6, 1)	[(6, 1), (6, 1)]
(7, 6)	[(7, 6)]
(9, 8)	[(9, 8)]

Aggregated data at reduce workers

The final output after applying the reduce function which takes only the first value and ignores everything else is as follows:



Reduce Worker 1

A	B
1	2
2	3
5	6
6	3
3	3
0	1

Reduce Worker 2

A	B
4	4
6	1
7	6
9	8

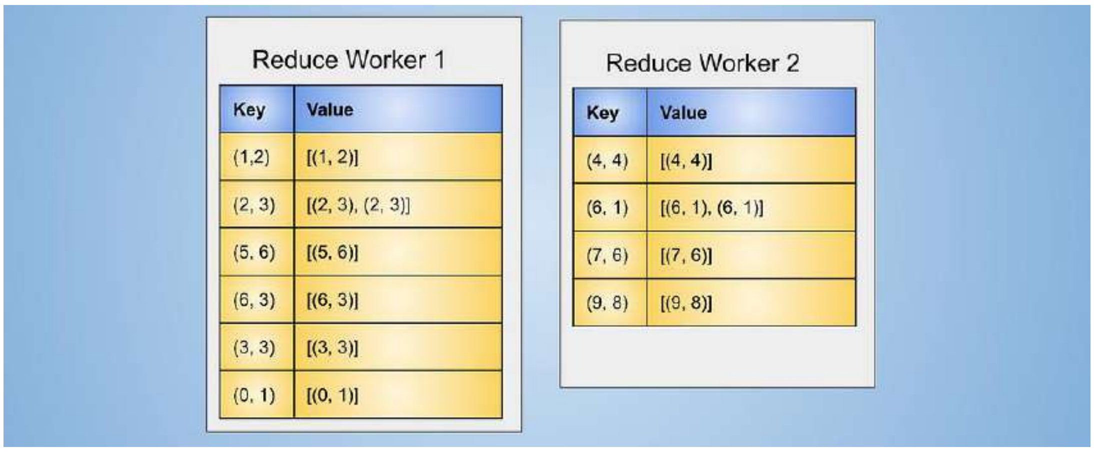
Final table after union

Intersection Using Map Reduce

For intersection, let's consider the same data we considered for union and just change the map and reduce functions

- **Map Function:** For each row r generate key-value pair (r, r) (Same as union).
- **Reduce Function:** With each key there can be one or two values (As we don't have duplicate rows), in case we have length of list as 2 we output first value else we output nothing.

As the map function is same as union and we are considering the same data lets skip to the part before reduce function is applied.



Reduce Worker 1

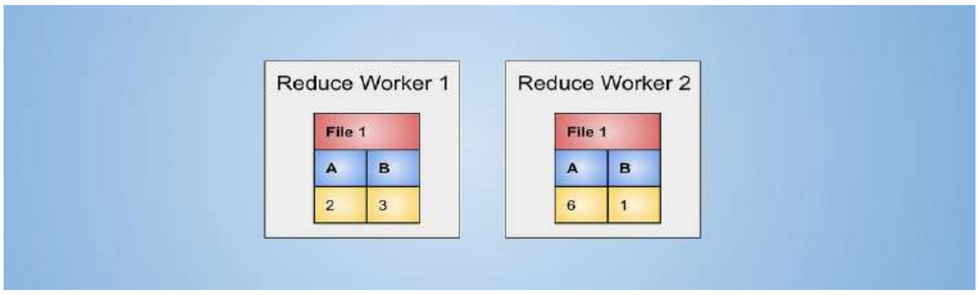
Key	Value
(1,2)	[(1, 2)]
(2, 3)	[(2, 3), (2, 3)]
(5, 6)	[(5, 6)]
(6, 3)	[(6, 3)]
(3, 3)	[(3, 3)]
(0, 1)	[(0, 1)]

Reduce Worker 2

Key	Value
(4, 4)	[(4, 4)]
(6, 1)	[(6, 1), (6, 1)]
(7, 6)	[(7, 6)]
(9, 8)	[(9, 8)]

Data at reduce workers

Now we just apply the reduce operation which will output only rows if list has a length of 2.



Reduce Worker 1

File 1

A	B
2	3

Reduce Worker 2

File 1

A	B
6	1

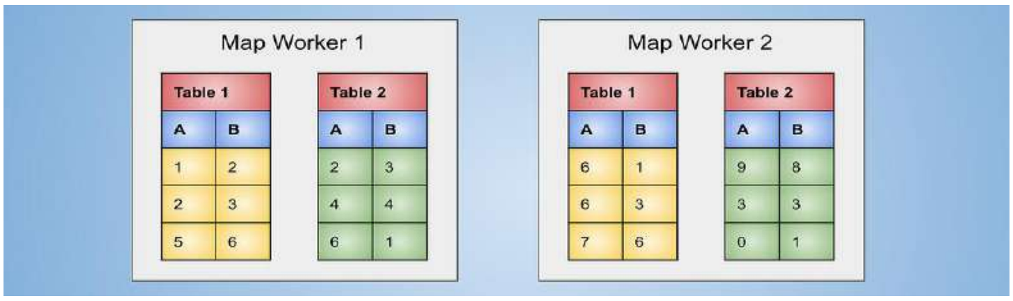
Output of intersection

Difference Using Map Reduce

Let's again consider the same data. The difficulty with difference arises with the fact that we want to output a row only if it exists in the first table but not the second one. So the reduce function needs to keep track on which tuple belongs to which relation. To visualize that easier we will keep those rows **green** which come from 2nd table and **yellow** for which come from 1st table and **purple** which comes from both tables.

- **Map Function:** For each row r create a key-value pair (r, T1) if row is from table 1 else product key-value pair (r, T2).
- **Reduce Function:** Output the row if and only if the value in the list is T1 , otherwise output nothing.

The data taken initially is the same as it was for union



Map Worker 1

Table 1

A	B
1	2
2	3
5	6

Table 2

A	B
2	3
4	4
6	1

Map Worker 2

Table 1

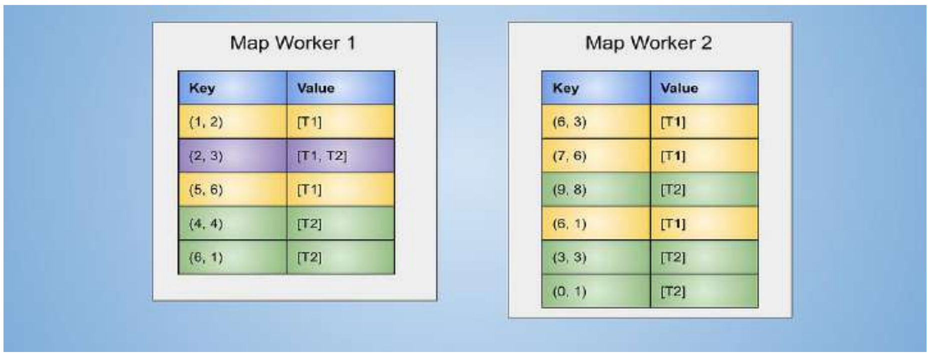
A	B
6	1
6	3
7	6

Table 2

A	B
9	8
3	3
0	1

Initial Data

After applying the map function and grouping the keys the data looks like the following figure



Map Worker 1

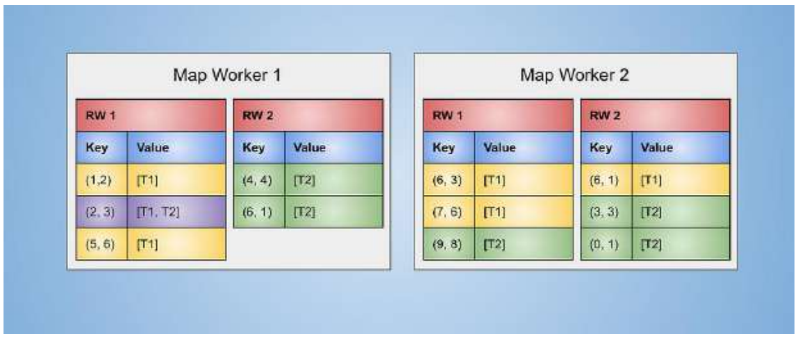
Key	Value
(1, 2)	[T1]
(2, 3)	[T1, T2]
(5, 6)	[T1]
(4, 4)	[T2]
(6, 1)	[T2]

Map Worker 2

Key	Value
(6, 3)	[T1]
(7, 6)	[T1]
(9, 8)	[T2]
(6, 1)	[T1]
(3, 3)	[T2]
(0, 1)	[T2]

Data after applying map function and grouping keys

After applying map function files for reduce workers will be created based on hashing keys as has been the case so far.



Map Worker 1

RW 1

Key	Value
(1,2)	[T1]
(2, 3)	[T1, T2]
(5, 6)	[T1]

RW 2

Key	Value
(4, 4)	[T2]
(6, 1)	[T2]

Map Worker 2

RW 1

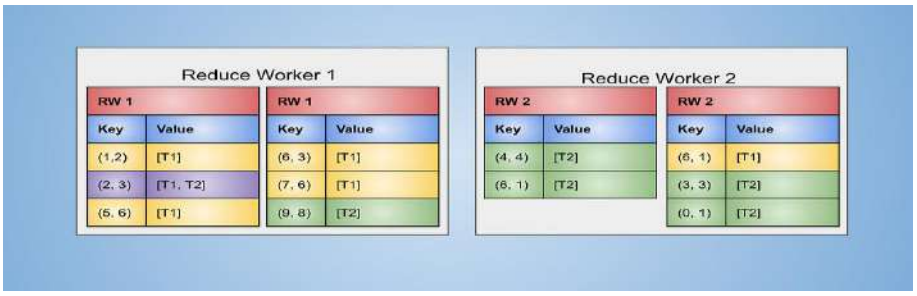
Key	Value
(6, 3)	[T1]
(7, 6)	[T1]
(9, 8)	[T2]

RW 2

Key	Value
(6, 1)	[T1]
(3, 3)	[T2]
(0, 1)	[T2]

Files for reduce workers

The data at the reduce workers will look like



Reduce Worker 1

RW 1

Key	Value
(1,2)	[T1]
(2, 3)	[T1, T2]
(5, 6)	[T1]

RW 1

Key	Value
(6, 3)	[T1]
(7, 6)	[T1]
(9, 8)	[T2]

Reduce Worker 2

RW 2

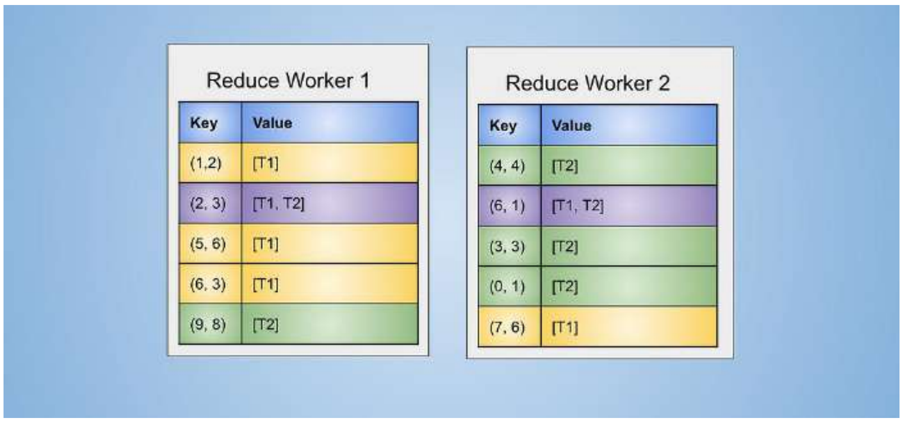
Key	Value
(4, 4)	[T2]
(6, 1)	[T2]

RW 2

Key	Value
(6, 1)	[T1]
(3, 3)	[T2]
(0, 1)	[T2]

Files at reduce workers

After aggregation of the keys at reduce workers the data looks like:



Reduce Worker 1

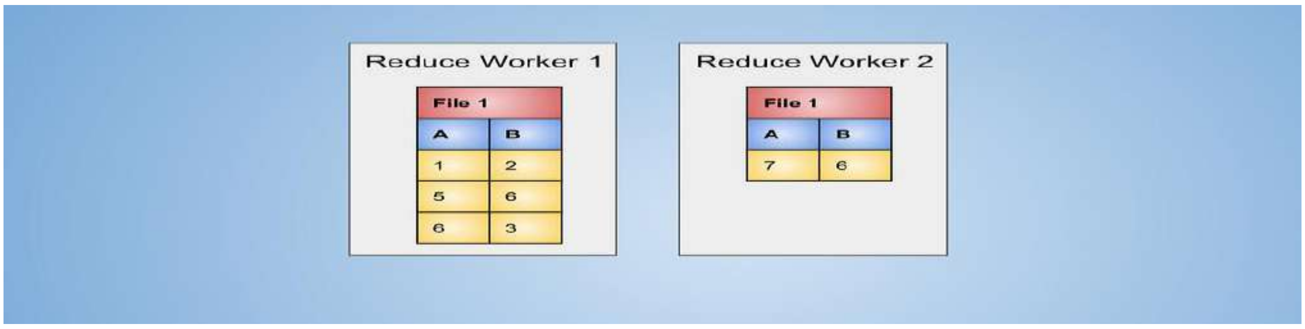
Key	Value
(1,2)	[T1]
(2, 3)	[T1, T2]
(5, 6)	[T1]
(6, 3)	[T1]
(9, 8)	[T2]

Reduce Worker 2

Key	Value
(4, 4)	[T2]
(6, 1)	[T1, T2]
(3, 3)	[T2]
(0, 1)	[T2]
(7, 6)	[T1]

Data after aggregation of keys at reduce workers

The final output is generated after applying the reduce function over the output.



Reduce Worker 1

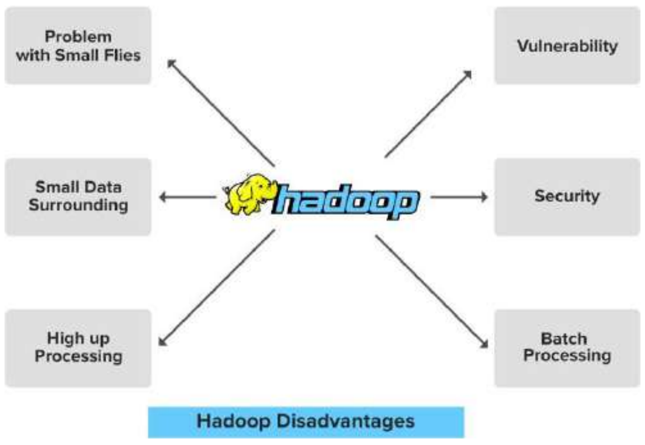
File 1	
A	B
1	2
5	6
6	3

Reduce Worker 2

File 1	
A	B
7	6

Output of difference of the tables

HADOOP LIMITATION:



I) Problem with Small files:

1. Hadoop can efficiently perform over a small number of files of large size.
2. Hadoop stores the file in the form of file blocks which are from 128MB in size (by default) to 256MB.
3. Hadoop fails when it needs to access the small size file in a large amount.
4. This so many small files surcharge the Namenode and make it difficult to work.

II) Vulnerability:

1. Hadoop is a framework that is written in java, and java is one of the most commonly used programming languages which makes it more insecure as it can be easily exploited by any of the cyber-criminal.

III) Low Performance In Small Data Surrounding:

1. Hadoop is mainly designed for dealing with large datasets, so it can be efficiently utilized for the organizations that are generating a massive volume of data.
2. It’s efficiency decreases while performing in small data surroundings.

IV) **<u>Lack of Security:</u>**

1. Data is everything for an organization, by default the security feature in Hadoop is made un-available.
2. So the Data driver needs to be careful with this security face and should take appropriate action on it.
3. Hadoop uses Kerberos for security feature which is not easy to manage.
4. Storage and network encryption are missing in Kerberos which makes us more concerned about it.

V) **<u>High Up Processing:</u>**

1. Read/Write operation in Hadoop is immoderate since we are dealing with large size data that is in TB or PB.
2. In Hadoop, the data read or write done from the disk which makes it difficult to perform in-memory calculation and lead to processing overhead or High up processing.

VI) **<u>Supports Only Batch Processing:</u>**

1. The batch process is nothing but the processes that are running in the background and does not have any kind of interaction with the user.
2. The engines used for these processes inside the Hadoop core is not that much efficient.
3. Producing the output with low latency is not possible with it.

HDFS more suited for applications having large datasets and not when there are small files

HDFS (Hadoop Distributed File System) is specifically designed to handle large datasets and is not as well-suited for small files. This is mainly due to the following reasons:

1. <u>Block Size:</u>
   a. HDFS stores data in large blocks, typically ranging from tens of megabytes to several gigabytes in size.
   b. By storing data in large blocks, HDFS reduces the overhead associated with managing a large number of small files.
   c. However, when dealing with small files, the block size is still fixed, leading to potential inefficiencies in storage utilization.
   d. For example, if a small file occupies only a fraction of a block, the remaining space in that block is wasted.
2. <u>Namenode Overhead:</u>
   a. HDFS utilizes a master-slave architecture, where the Namenode acts as the central metadata repository and oversees the file system namespace.
   b. Each file and directory in HDFS is represented as an inode, and the Namenode holds the metadata information for all the files and directories.
   c. When dealing with a large number of small files, the Namenode's memory and processing overhead can become a bottleneck.
   d. This is because each small file requires an entry in the metadata, leading to increased memory consumption and potential performance degradation.
3. <u>Data Locality:</u>
   a. HDFS is designed to maximize data locality, meaning that it tries to perform computations on the same nodes where the data resides.
   b. This reduces network overhead and improves performance.

c. However, when dealing with small files, the amount of data stored on each node might be relatively small, resulting in reduced data locality benefits.
d. The overhead of transferring small files across the network can outweigh the advantages of data locality.

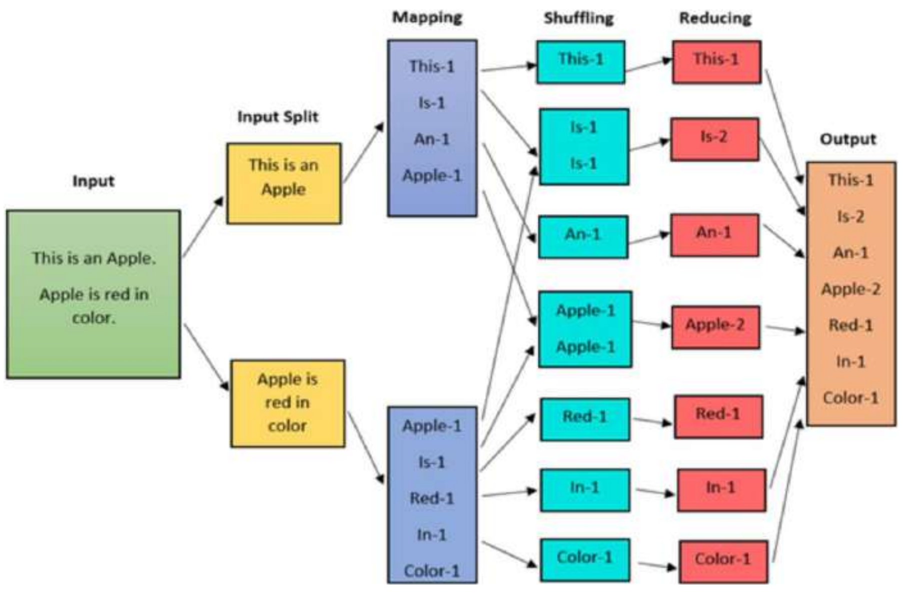
4. NameNode Scalability:

a. The scalability of the Namenode can be challenging when dealing with a large number of small files.
b. As the number of files increases, the Namenode's memory requirements and processing load also increase.
c. Eventually, the Namenode's capacity to handle the metadata for numerous small files may be limited, hindering the scalability of the system.

5. Metadata Operations:

a. HDFS performs metadata operations (e.g., file creation, deletion, and listing) through interactions with the Namenode.
b. With small files, these metadata operations become more frequent, resulting in additional overhead and potential performance degradation due to increased communication and coordination with the Namenode.

Map reduce pseudo code for word count problem. Apply map reduce working on the following document: "This is an apple. Apple is red in color"



1. The input data is divided into multiple segments, then processed in parallel to reduce processing time.

2. In this case, the input data will be divided into two input splits so that work can be distributed over all the map nodes.
3. The Mapper counts the number of times each word occurs from input splits in the form of key-value pairs where the key is the word, and the value is the frequency.
4. For the first input split, it generates 4 key-value pairs:
   This, 1; is, 1; an, 1; apple, 1;
5. And for the second, it generates 5 key-value pairs:
   Apple, 1; is, 1; red, 1; in, 1; color.
6. It is followed by the shuffle phase, in which the values are grouped by keys in the form of key-value pairs.
7. Here we get a total of 6 groups of key-value pairs.
8. The same reducer is used for all key-value pairs with the same key.
9. All the words present in the data are combined into a single output in the reducer phase.
10. The output shows the frequency of each word.
11. Here in the example, we get the final output of key-value pairs as This, 1; is, 2; an, 1; apple, 2; red, 1; in, 1; color, 1.
12. The record writer writes the output key-value pairs from the reducer into the output files, and the final output data is by default stored on HDFS.