What is MapReduce?

- A MapReduce is a data processing tool which is used to process the data parallelly in a distributed form. It was developed in 2004, on the basis of paper titled as "MapReduce: Simplified Data Processing on Large Clusters," published by Google.
- The MapReduce is a paradigm which has two phases, the mapper phase, and the reducer phase. In the Mapper, the input is given in the form of a key-value pair. The output of the Mapper is fed to the reducer as input. The reducer runs only after the Mapper is over. The reducer too takes input in key-value format, and the output of reducer is the final output.

Example:

Imagine you have a large set of documents, and you want to count the occurrences of each word across all documents. The Map function processes each document and outputs intermediate key-value pairs where the key is a word and the value is 1. The Reduce function then aggregates these counts for each word.

ii. Apache MapReduce Terminologies

Map-Reduce is the data processing component of Hadoop. Map-Reduce programs transform lists of input data elements into lists of output data elements. A Map-Reduce program will do this twice, using two different list processing idioms-

- Map
- Reduce

In between Map and Reduce, there is small phase called Shuffle and Sort in MapReduce.
Let's understand basic terminologies used in Map Reduce.

- What is a MapReduce Job?

MapReduce Job or a A "full program" is an execution of a Mapper and Reducer across a data set. It is an execution of 2 processing layers i.e mapper and reducer. A MapReduce job is a work that the client wants to be performed. It consists of the input data, the MapReduce Program, and configuration info. So client needs to submit input data, he needs to write Map Reduce program and set the configuration info (These were provided during Hadoop setup in the configuration file and also we specify some configurations in our program itself which will be specific to our map reduce job).

- What is Task in Map Reduce?

A task in MapReduce is an execution of a Mapper or a Reducer on a slice of data. It is also called Task-In-Progress (TIP). It means processing of data is in progress either on mapper or reducer.

- What is Task Attempt?

Task Attempt is a particular instance of an attempt to execute a task on a node. There is a possibility that anytime any machine can go down. For example, while processing data if any node goes down, framework reschedules the task to some other node. This rescheduling of the task cannot be infinite. There is an upper limit for that as well. The default value of task attempt is 4. If a task (Mapper or reducer) fails 4 times, then the job is considered as a failed job. For high priority job or huge job, the value of this task attempt can also be increased.
Install Hadoop and play with MapReduce.

iii. Map Abstraction

Let us understand the abstract form of **Map** in MapReduce, the first phase of MapReduce paradigm, what is a map/mapper, what is the input to the mapper, how it processes the data, what is output from the mapper?
The **map** takes key/value pair as input. Whether data is in structured or unstructured format, framework converts the incoming data into key and value.

- Key is a reference to the input value.
- Value is the data set on which to operate.

Map Processing:

- A function defined by user – user can write custom business logic according to his need to process the data.
- Applies to every value in value input.

Map produces a new list of key/value pairs:

- An output of Map is called intermediate output.
- Can be the different type from input pair.
- An output of map is stored on the local disk from where it is shuffled to reduce nodes.

iv. Reduce Abstraction

Now let's discuss the second phase of MapReduce – **Reducer** in this MapReduce Tutorial, what is the input to the reducer, what work reducer does, where reducer writes output?
Reduce takes intermediate Key / Value pairs as input and processes the output of the mapper. Usually, in the reducer, we do aggregation or summation sort of computation.

- Input given to reducer is generated by Map (intermediate output)
- Key / Value pairs provided to reduce are sorted by key

Reduce processing:

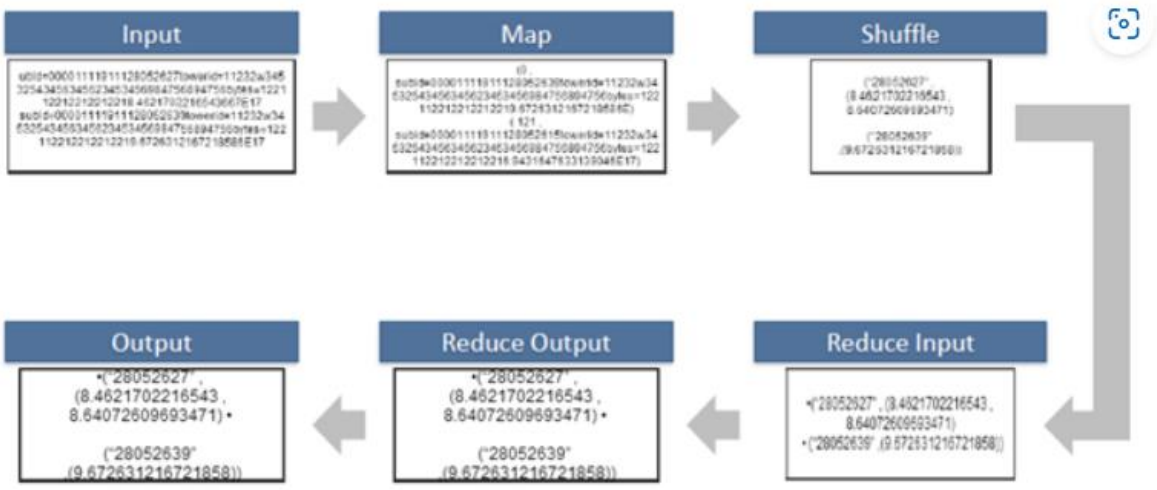
- A function defined by user – Here also user can write custom business logic and get the final output.
- Iterator supplies the values for a given key to the Reduce function.

Reduce produces a final list of key/value pairs:

- An output of Reduce is called Final output.
- It can be a different type from input pair.
- An output of Reduce is stored in HDFS.

Steps in Map Reduce

- The map takes data in the form of pairs and returns a list of <key, value> pairs. The keys will not be unique in this case.
- Using the output of Map, sort and shuffle are applied by the Hadoop architecture. This sort and shuffle acts on these list of <key, value> pairs and sends out unique keys and a list of values associated with this unique key <key, list (values)>.
- An output of sort and shuffle sent to the reducer phase. The reducer performs a defined function on a list of values for unique keys, and Final output <key, value> will be stored/ displayed.



Sort and Shuffle

The sort and shuffle occur on the output of Mapper and before the reducer. When the Mapper task is complete, the results are sorted by key, partitioned if there are multiple reducers, and then written to

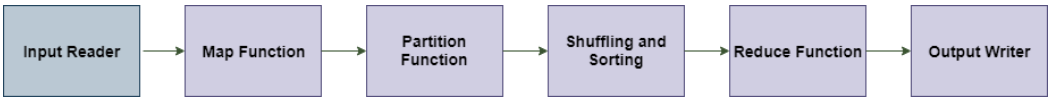
disk. Using the input from each Mapper <k2,v2>, we collect all the values for each unique key k2. This output from the shuffle phase in the form of <k2, list(v2)> is sent as input to reducer phase.

Usage of MapReduce

- It can be used in various applications like document clustering, distributed sorting, and web link-graph reversal.
- It can be used for distributed pattern-based searching.
- We can also use MapReduce in machine learning.
- It was used by Google to regenerate Google's index of the World Wide Web.
- It can be used in multiple computing environments such as multi-cluster, multi-core, and mobile environment.

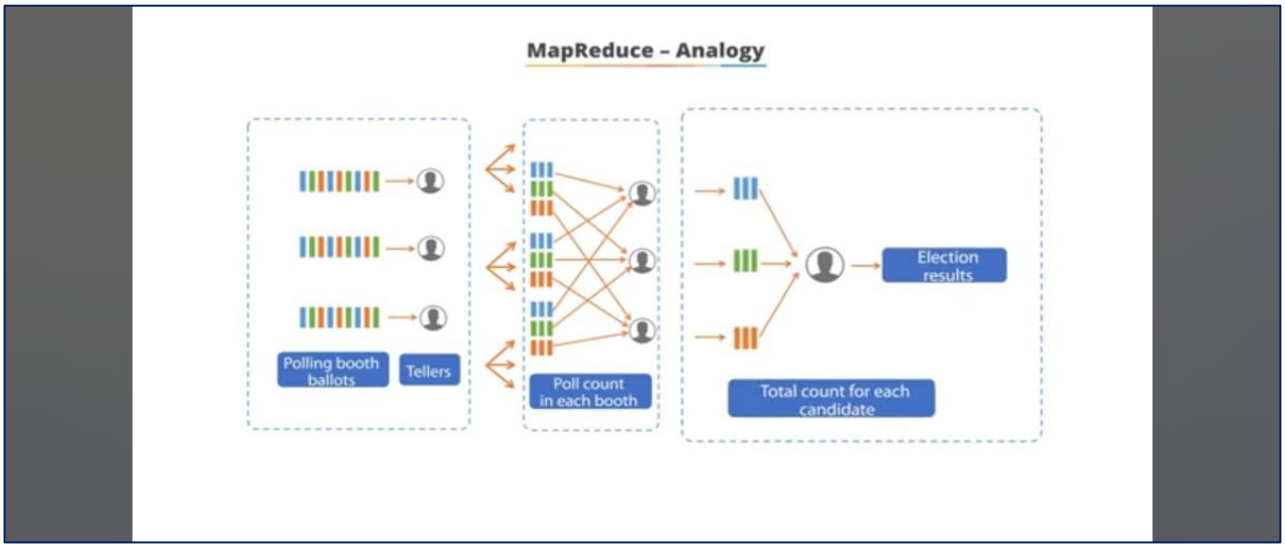
Data Flow in MapReduce

MapReduce is used to compute the huge amount of data. To handle the upcoming data in a parallel and distributed form, the data has to flow from various phases.

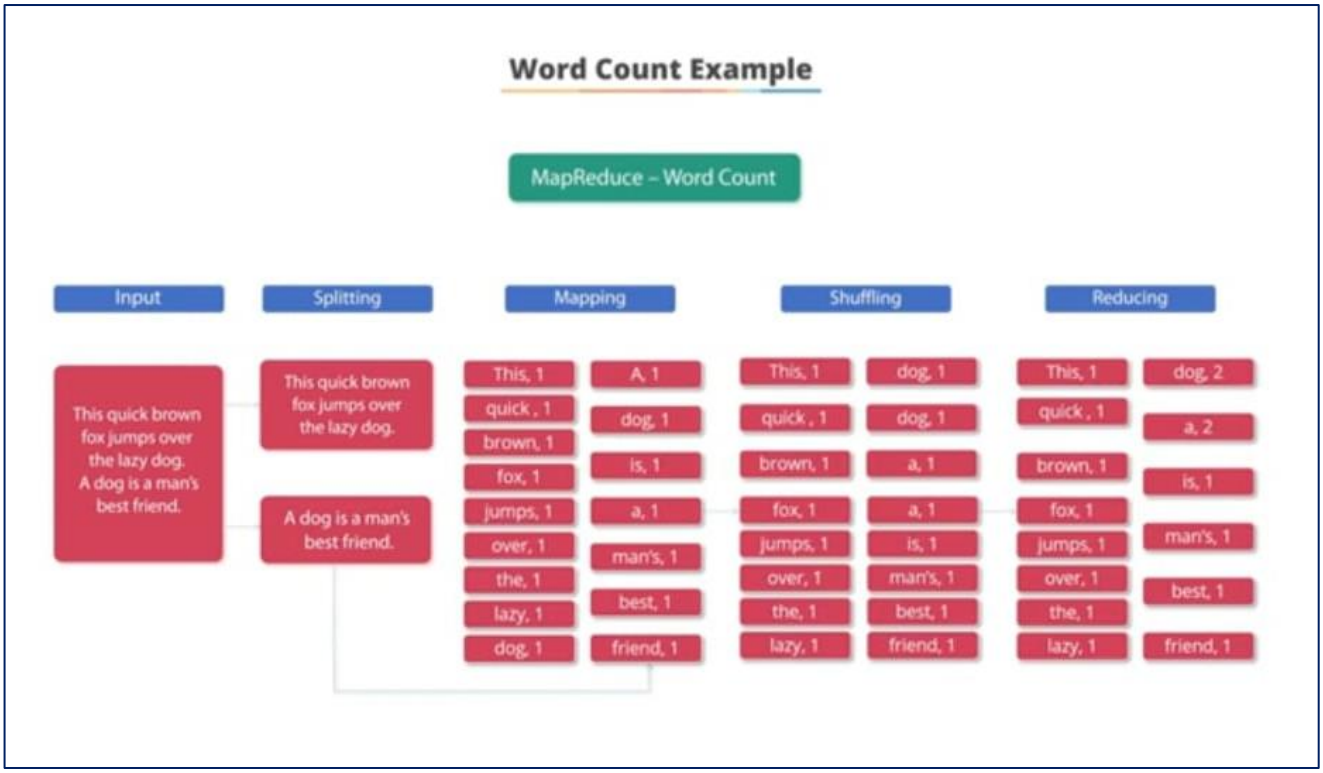


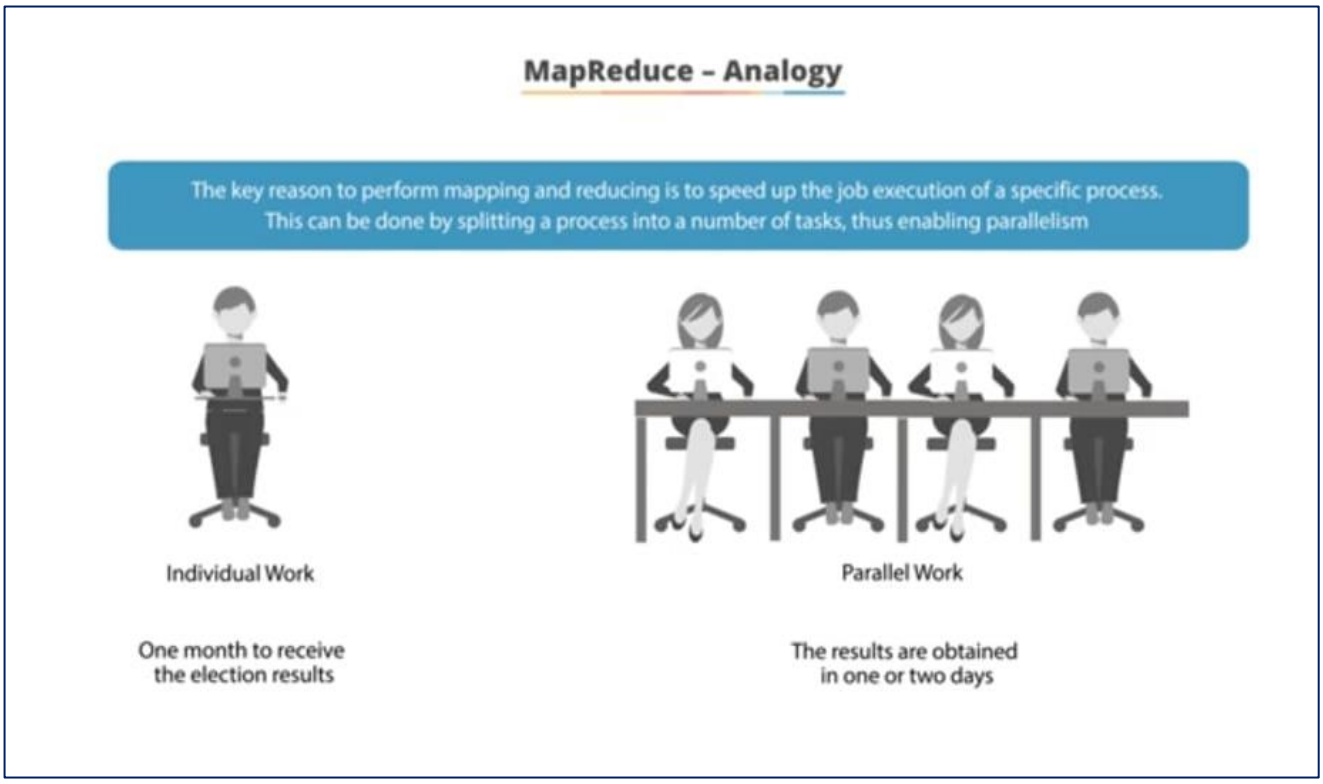
Example of Map Reduce

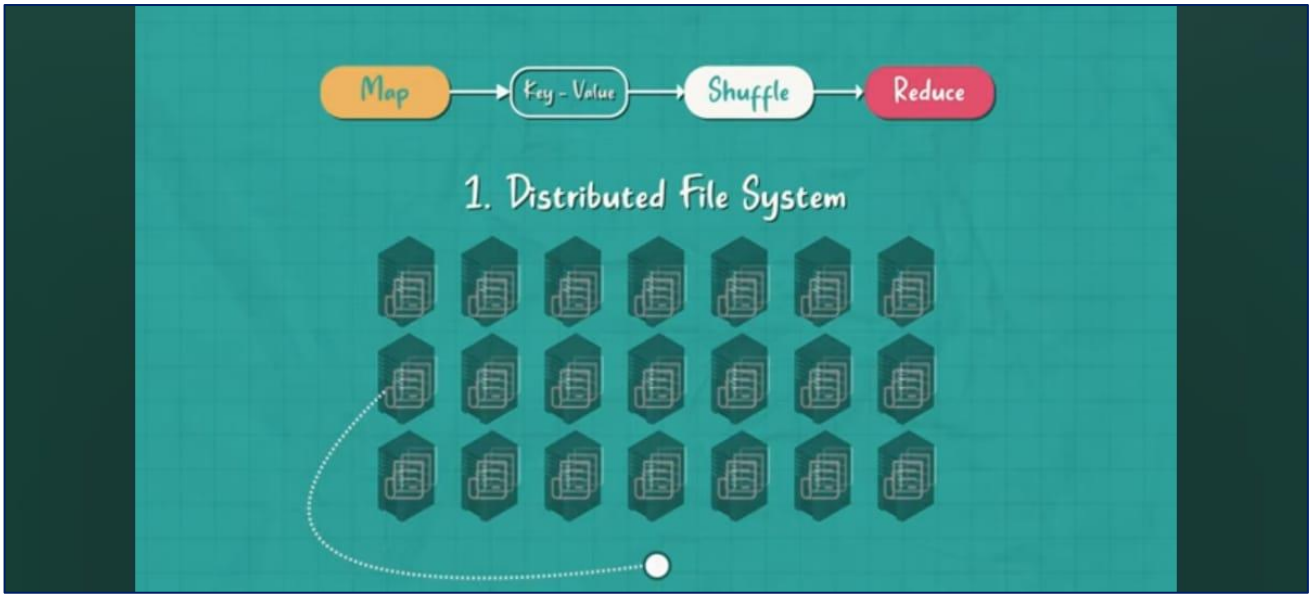
Example 1:

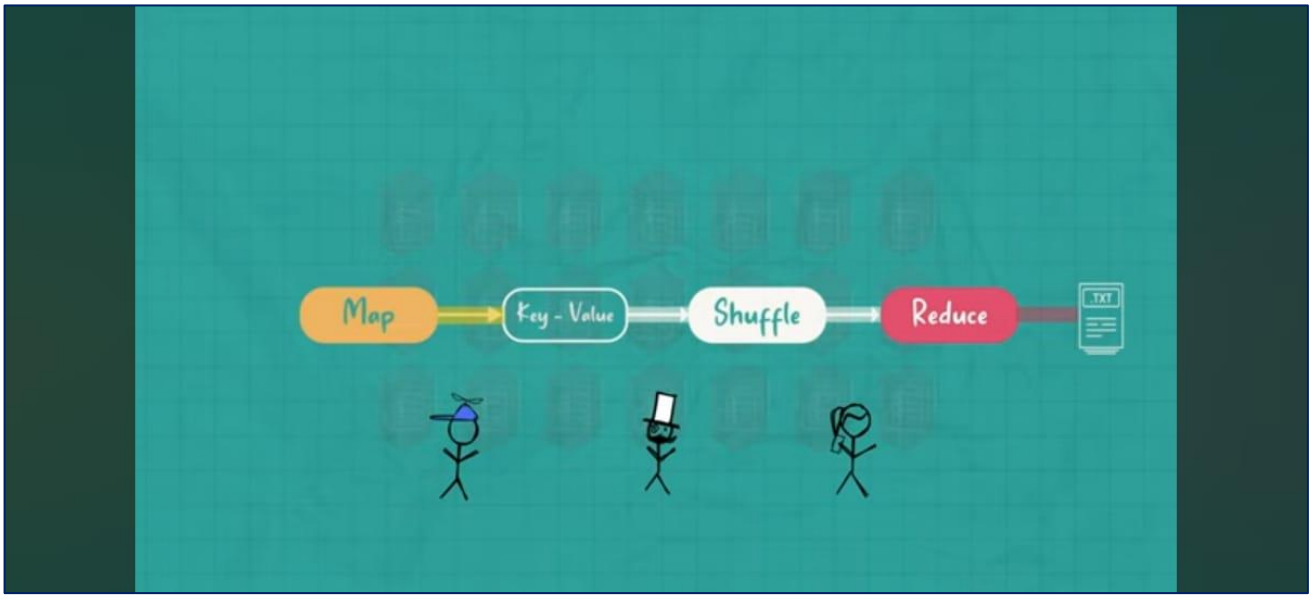


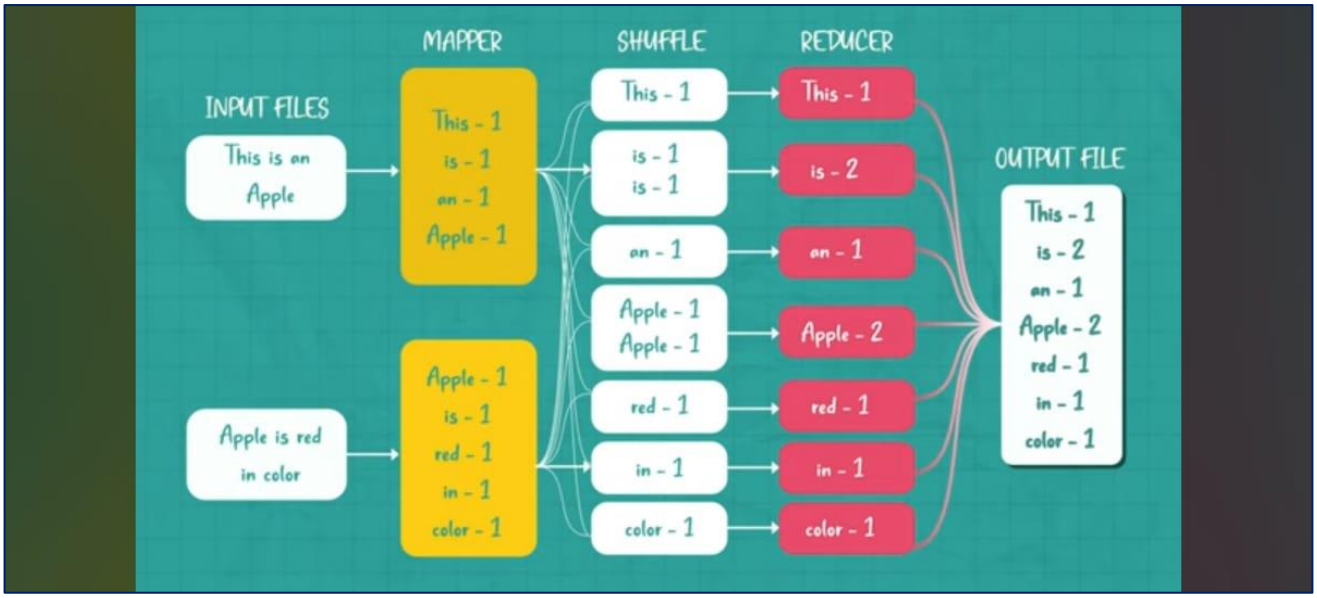
Example 2:


Word Count Example
MapReduce – Word Count
Input
Splitting
Mapping
Shuffling
Reducing
This quick brown fox jumps over the lazy dog. A dog is a man's best friend.
This quick brown fox jumps over the lazy dog.
A dog is a man's best friend.
This, 1
quick , 1
brown, 1
fox, 1
jumps, 1
over, 1
the, 1
lazy, 1
dog, 1
A, 1
dog, 1
is, 1
a, 1
man's, 1
best, 1
friend, 1
This, 1
quick , 1
brown, 1
fox, 1
jumps, 1
over, 1
the, 1
lazy, 1
dog, 1
dog, 1
a, 1
a, 1
is, 1
man's, 1
best, 1
friend, 1
This, 1
quick , 1
brown, 1
fox, 1
jumps, 1
over, 1
the, 1
lazy, 1
dog, 2
a, 2
is, 1
man's, 1
best, 1
friend, 1

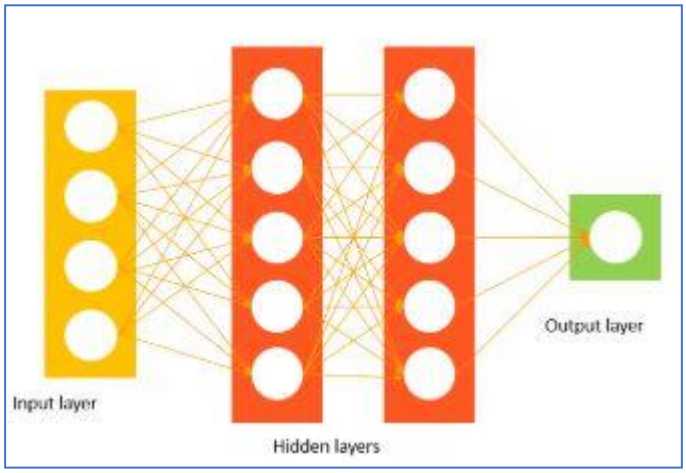

MapReduce – Analogy
The key reason to perform mapping and reducing is to speed up the job execution of a specific process.
This can be done by splitting a process into a number of tasks, thus enabling parallelism
Individual Work
One month to receive the election results
Parallel Work
The results are obtained in one or two days

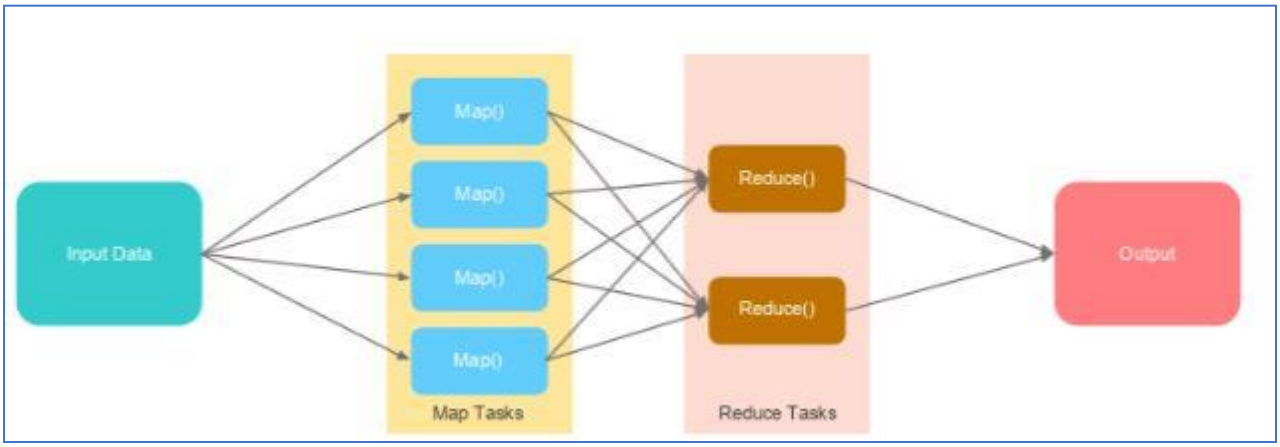

Map
Key - Value
Shuffle
Reduce
1. Distributed File System


Map
Key - Value
Shuffle
Reduce
.TXT



Phases of MapReduce data flow





Input reader

The input reader reads the upcoming data and splits it into the data blocks of the appropriate size (64 MB to 128 MB). Each data block is associated with a Map function.

Once input reads the data, it generates the corresponding key-value pairs. The input files reside in HDFS.

Map function

The map function process the upcoming key-value pairs and generated the corresponding output key-value pairs. The map input and output type may be different from each other.

Partition function

The partition function assigns the output of each Map function to the appropriate reducer. The available key and value provide this function. It returns the index of reducers.

Shuffling and Sorting

The data are shuffled between/within nodes so that it moves out from the map and get ready to process for reduce function. Sometimes, the shuffling of data can take much computation time.

The sorting operation is performed on input data for Reduce function. Here, the data is compared using comparison function and arranged in a sorted form.

Reduce function

The Reduce function is assigned to each unique key. These keys are already arranged in sorted order. The values associated with the keys can iterate the Reduce and generates the corresponding output.

Output writer

Once the data flow from all the above phases, Output writer executes. The role of Output writer is to write the Reduce output to the stable storage.

Methods of Reducer Class

void cleanup(Context context)	This method called only once at the end of the task.
void map(KEYIN key, Iterable<VALUEIN> values, Context context)	This method called only once for each key.
void run(Context context)	This method can be used to control the tasks of the Reducer.
void setup(Context context)	This method called only once at the beginning of the task.

MapReduce Job Class

The Job class is used to configure the job and submits it. It also controls the execution and query the state. Once the job is submitted, the set method throws IllegalStateException.

Methods of Job Class

Methods	Description
Counters getCounters()	This method is used to get the counters for the job.
long getFinishTime()	This method is used to get the finish time for the job.

Job getInstance()	This method is used to generate a new Job without any cluster.
Job getInstance(Configuration conf)	This method is used to generate a new Job without any cluster and provided configuration.
Job getInstance(Configuration conf, String jobName)	This method is used to generate a new Job without any cluster and provided configuration and job name.
String getJobFile()	This method is used to get the path of the submitted job configuration.
String getJobName()	This method is used to get the user-specified job name.
JobPrioritygetPriority()	This method is used to get the scheduling function of the job.
void setJarByClass(Class<?> c)	This method is used to set the jar by providing the class name with .class extension.
void setJobName(String name)	This method is used to set the user-specified job name.
void setMapOutputKeyClass(Class<?> class)	This method is used to set the key class for the map output data.
void setMapOutputValueClass(Class<?> class)	This method is used to set the value class for the map output data.
void setMapperClass(Class<? extends Mapper> class)	This method is used to set the Mapper for the job.
void setNumReduceTasks(int tasks)	This method is used to set the number of reduce tasks for the job
void setReducerClass(Class<? extends Reducer> class)	This method is used to set the Reducer for the job.

3.2 MapReduce Programming Basics

3.2.1 Input and Output Formats in MapReduce

Input Formats: These define how input data is split and read. Common input formats include:

- **TextInputFormat**: Default format; reads lines of text files.

- **KeyValueTextInputFormat**: Reads lines, treating the first tab-separated part as the key and the rest as the value.
- **SequenceFileInputFormat**: For binary key-value pairs.

Output Formats: These define how output data is written. Common output formats include:

- **TextOutputFormat**: Default format; writes each key-value pair as a line of text.
- **SequenceFileOutputFormat**: Writes binary key-value pairs.

Example:

For a log processing job, you might use TextInputFormat to read log lines and TextOutputFormat to write the processed results.

3.2.2 Mapper and Reducer Functions

- **Mapper**: The Mapper function processes input data and emits key-value pairs. It runs once for each input split.

```
public class TokenizerMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable 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);
      }
   }
}
```

Reducer: The Reducer function processes intermediate key-value pairs generated by the Mapper and emits final key-value pairs.

```
public class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
         sum += val.get();
      }
      context.write(key, new IntWritable(sum));
   }
}
```

3.2.3 Combiner Functions

Combiner: An optional component that performs local aggregation of intermediate data before sending it to the Reducer, reducing the amount of data shuffled across the network.

Example:

Using a Combiner in the word count example can reduce network traffic by summing counts locally before the Reduce phase.

3.2.4 Partitioners in MapReduce

Partitioner: Determines the partition (or Reducer) for each key. By default, Hadoop uses `HashPartitioner`, which ensures an even distribution of keys across Reducers.

Example:

Custom Partitioner to ensure specific keys are sent to particular Reducers.

```
public class CustomPartitioner extends Partitioner<Text, IntWritable> {
   @Override
   public int getPartition(Text key, IntWritable value, int numReduceTasks) {
      if (key.toString().startsWith("a")) {
         return 0; // Send to Reducer 0
      } else {
         return 1; // Send to Reducer 1
      }
   }
}
```

3.3 Advanced MapReduce Programming Concepts

3.3.1 Hadoop Streaming

Hadoop Streaming: Allows writing MapReduce jobs in any language that can read from standard input and write to standard output. It is useful for developers who prefer scripting languages like Python or Ruby.

Example:

A Python script for word count using Hadoop Streaming.

```
import sys
for line in sys.stdin:
   words = line.strip().split()
   for word in words:
      print(f"{word}\t1")
```

3.3.2 Multi-Stage MapReduce Jobs

Multi-Stage MapReduce Jobs: Involve chaining multiple MapReduce jobs where the output of one job becomes the input of the next.

Example:

A data processing pipeline where the first job filters data and the second job aggregates it.

3.3.3 MapReduce Joins, Counters, Optimization

- **Joins**: Techniques to combine data from different sources within a MapReduce job, such as Reduce-Side Join or Map-Side Join.
- **Counters**: Useful for tracking job-specific metrics or debugging purposes.

context.getCounter("MyCounterGroup", "MyCounterName").increment(1);

Optimization: Techniques include using combiners, tuning the number of mappers and reducers, and leveraging data locality.

3.4 Data Locality in MapReduce

Data Locality: Refers to the principle of moving computation closer to where the data resides, minimizing data transfer across the network, thus improving efficiency.

Example:

Hadoop attempts to schedule map tasks on nodes where the data block is present to take advantage of data locality.

3.4.1 MapReduce Compression

MapReduce Compression: Involves compressing intermediate data to reduce the volume of data transferred between Map and Reduce phases, improving job performance.

Example:

Configuring job to use compression.

```
conf.set("mapreduce.map.output.compress", "true");
conf.set("mapreduce.map.output.compress.codec", "org.apache.hadoop.io.compress.SnappyCodec");
```

3.4.2 MapReduce Sort and Shuffle

Sort and Shuffle: The process of sorting intermediate data by key and transferring it from mappers to reducers. It is a critical step that significantly impacts the performance of a MapReduce job.

Example:

Tuning the shuffle phase by adjusting buffer sizes and spill thresholds to optimize performance.

By understanding and leveraging these MapReduce programming concepts, one can write efficient and scalable distributed data processing applications using Hadoop.

Word Count Example:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;
import java.util.StringTokenizer;

public class WordCount {

   // Mapper Class
   public static class TokenizerMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map(LongWritable 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);
         }
      }
   }

   // Reducer Class
   public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
      public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException {
```

```
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
  }

  // Driver Class
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);  // Using reducer as combiner for local aggregation
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Example:

A program that counts the occurrences of different weather conditions from a dataset. Each line in the dataset contains a weather record, including a weather condition like "Sunny", "Rainy", "Cloudy", etc.

Here's how we can create a Hadoop MapReduce program to perform this task:

1. Define the Mapper Class

The Mapper will read each line of the dataset and emit each weather condition with a count of 1.

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WeatherConditionMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text weatherCondition = new Text();

   public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
       // Assuming the weather condition is the first word in each line
       String line = value.toString();
       String[] parts = line.split("\\s+"); // Split the line by whitespace
       if (parts.length > 0) {
           weatherCondition.set(parts[0]); // Set the weather condition
```

```
            context.write(weatherCondition, one); // Emit the weather condition with count 1
        }
    }
}
```

2. Define the Reducer Class

The Reducer will sum the counts for each weather condition.

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WeatherConditionReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum)); // Emit the weather condition and its total count
    }
}
```

Define the Driver Class

The Driver sets up and configures the MapReduce job.

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WeatherConditionCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "weather condition count");
        job.setJarByClass(WeatherConditionCount.class);
        job.setMapperClass(WeatherConditionMapper.class);
        job.setCombinerClass(WeatherConditionReducer.class); // Using reducer as combiner
        job.setReducerClass(WeatherConditionReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
```

```
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Explanation:

1. **WeatherConditionMapper Class**:
    - **Line 1**: Imports necessary Hadoop libraries.
    - **Line 2**: Declares the Mapper class extending Mapper<LongWritable, Text, Text, IntWritable>.
    - **Line 3**: Defines a constant one with a value of 1.
    - **Line 4**: Declares a Text instance weatherCondition to hold the weather condition.
    - **Line 5**: Overrides the map method.
    - **Line 6**: Converts the input line to a string and splits it into parts based on whitespace.
    - **Line 7**: Sets the weather condition to the first part of the line.
    - **Line 8**: Emits the weather condition with a count of 1.
2. **WeatherConditionReducer Class**:
    - **Line 1**: Imports necessary Hadoop libraries.
    - **Line 2**: Declares the Reducer class extending Reducer<Text, IntWritable, Text, IntWritable>.
    - **Line 3**: Overrides the reduce method.
    - **Line 4**: Initializes the sum to 0.
    - **Line 5**: Iterates over the values and sums them up.
    - **Line 6**: Emits the weather condition and its total count.
3. **WeatherConditionCount (Driver) Class**:
    - **Line 1**: Imports necessary Hadoop libraries.
    - **Line 2**: Declares the main class.
    - **Line 3**: Sets up the Hadoop configuration.
    - **Line 4**: Creates a new job instance.
    - **Line 5**: Sets the jar class.
    - **Line 6**: Sets the Mapper class.
    - **Line 7**: Sets the Combiner class (optional, for local aggregation).
    - **Line 8**: Sets the Reducer class.
    - **Line 9**: Sets the output key and value types.
    - **Line 10**: Sets the input and output paths.
    - **Line 11**: Submits the job and waits for its completion.

Running the Program:

1. **Compile the Program**:
    - Ensure you have Hadoop installed and set up properly.
    - Compile the Java code using a Java compiler.
2. **Package the Program**:
    - Package the compiled classes into a jar file.
3. **Execute the Program**:
    - Run the jar file using the Hadoop command line, specifying input and output paths:

```
hadoop jar WeatherConditionCount.jar WeatherConditionCount /path/to/input /path/to/output
```

Replace /path/to/input and /path/to/output with actual HDFS paths.

This program will read the weather data, count the occurrences of each weather condition, and write the results to the output path. For example, if the input data contains lines like:

Sunny
Rainy
Cloudy
Sunny
Sunny
Rainy

The output will be:

Sunny 3

Rainy 2

Cloudy 1

This demonstrates how to use Hadoop MapReduce to process and analyze different types of data.