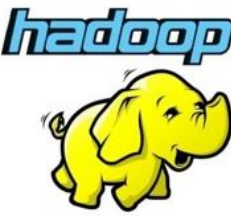


2CEIT702: Big Data Analytics

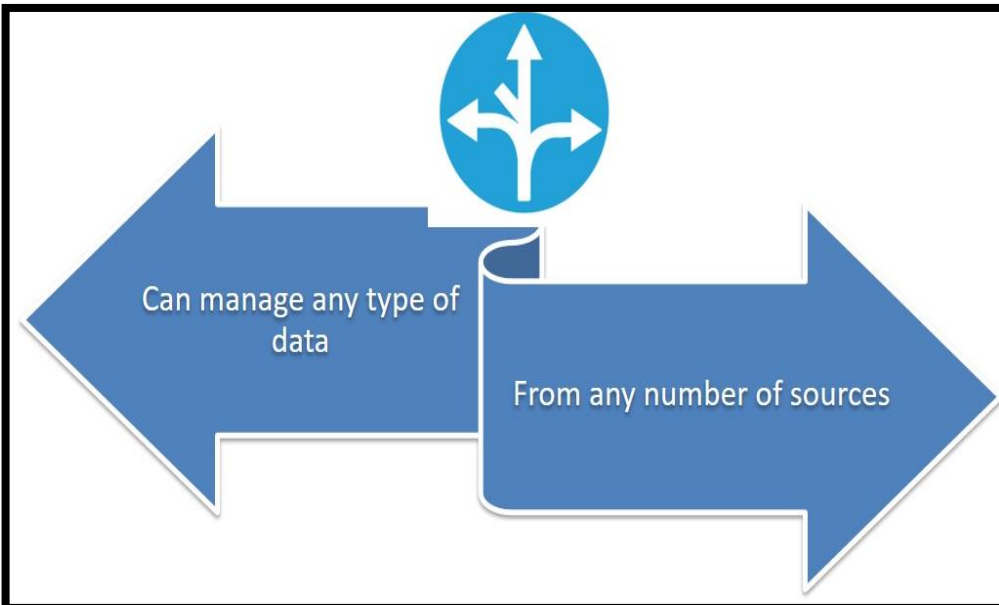
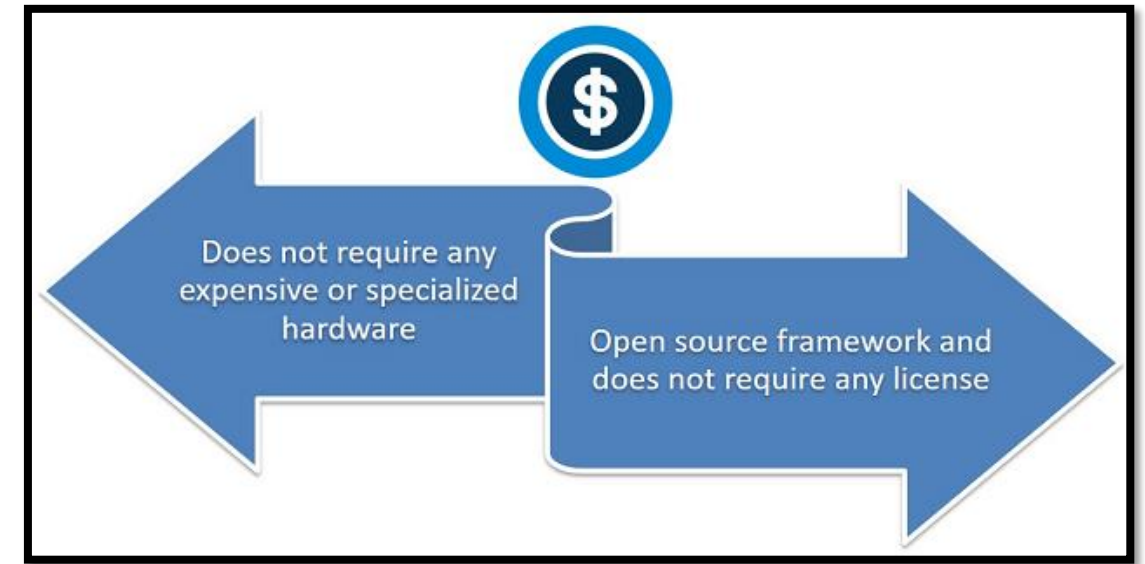


MapReduce

(Hadoop Processing Component)

Hadoop



- Open Source Framework licensed under Apache
- Stores huge volume of data
- Process the data using simple programming language



Challenge: Data is too big store in one computer

Hadoop Solution: Data is stored in multiple computer.

Challenge: Very high end machines are expensive

Hadoop solution: Run on commodity hardware

Challenge: commodity hardware will fail.

Hadoop Solution: Software is intelligent enough to deal with hardware failure.

Challenge: hardware failure may lead to data loss

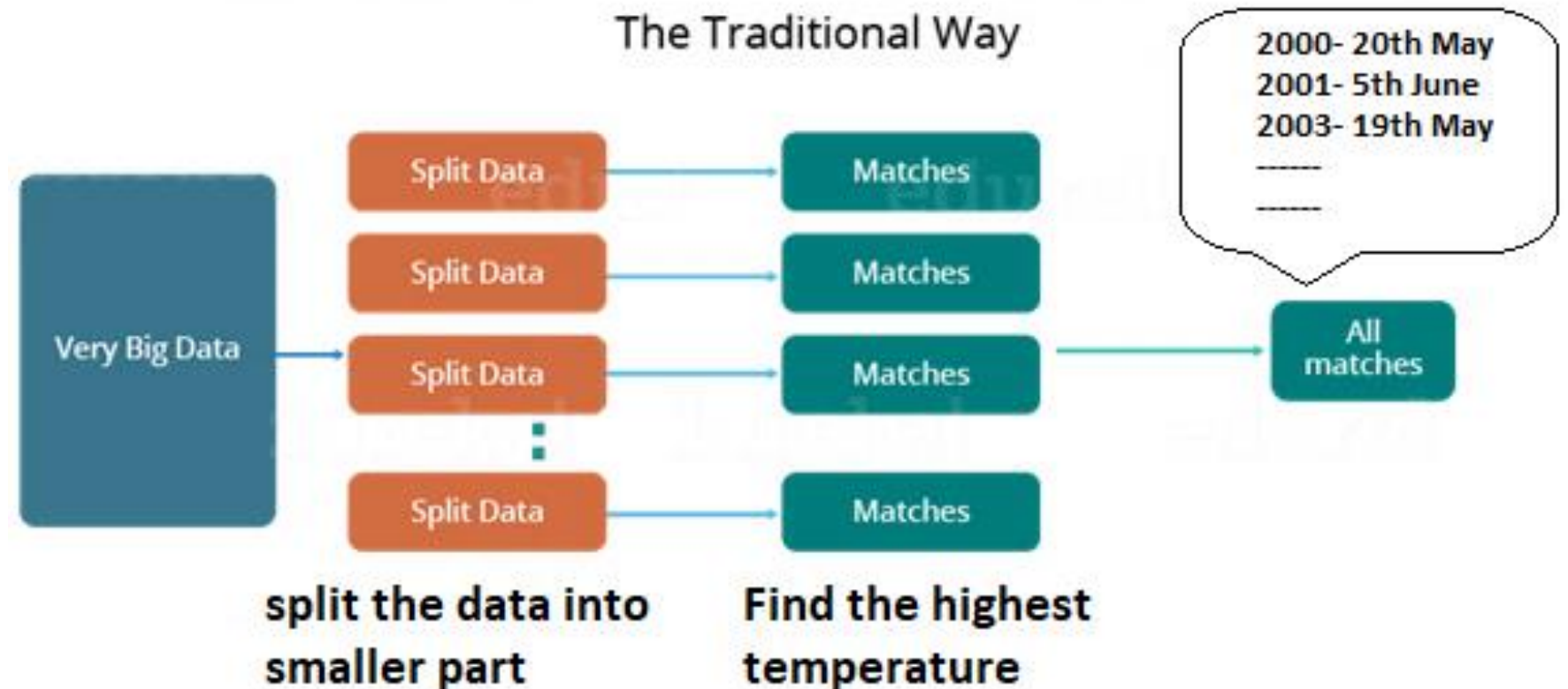
Hadoop Solution: Replicate data

Challenge: how will the distributed nodes co-ordinate among themselves

Hadoop solution: Master node that co-ordinates all the worker nodes

Distributed processing (Traditional way)

- When the MapReduce framework was not there, how parallel and distributed processing used to happen in a traditional way.
- **Example-1:** Weather dataset (Daily average temperature of the years from 2000 to 2018). **Query:** want to calculate the day having the highest temperature in each year.
- **In traditional way:**

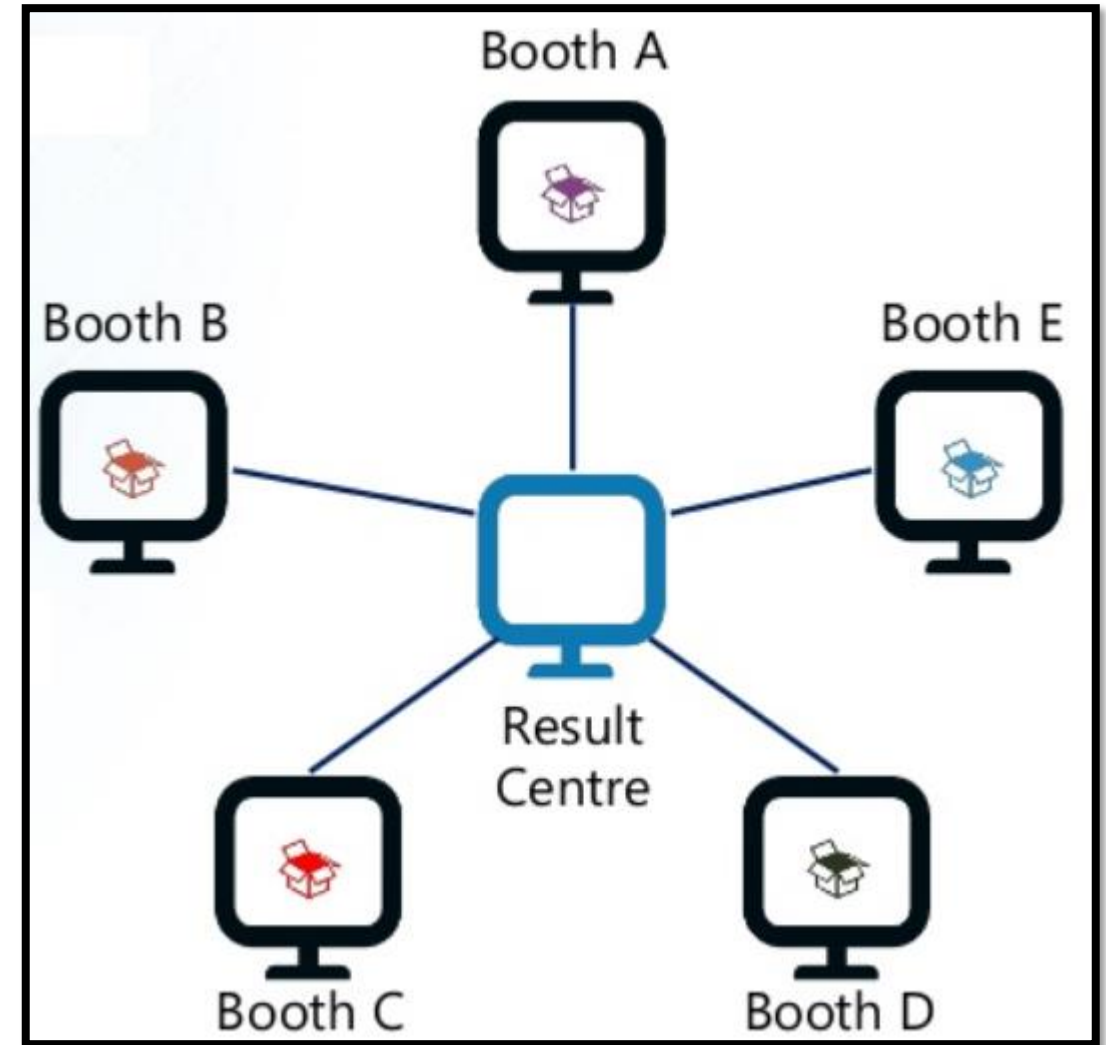


Example-2: Counting election votes

Data →     

Election Votes Casting

- Votes is stored at different Booths
- Result Centre has the details of all the Booths

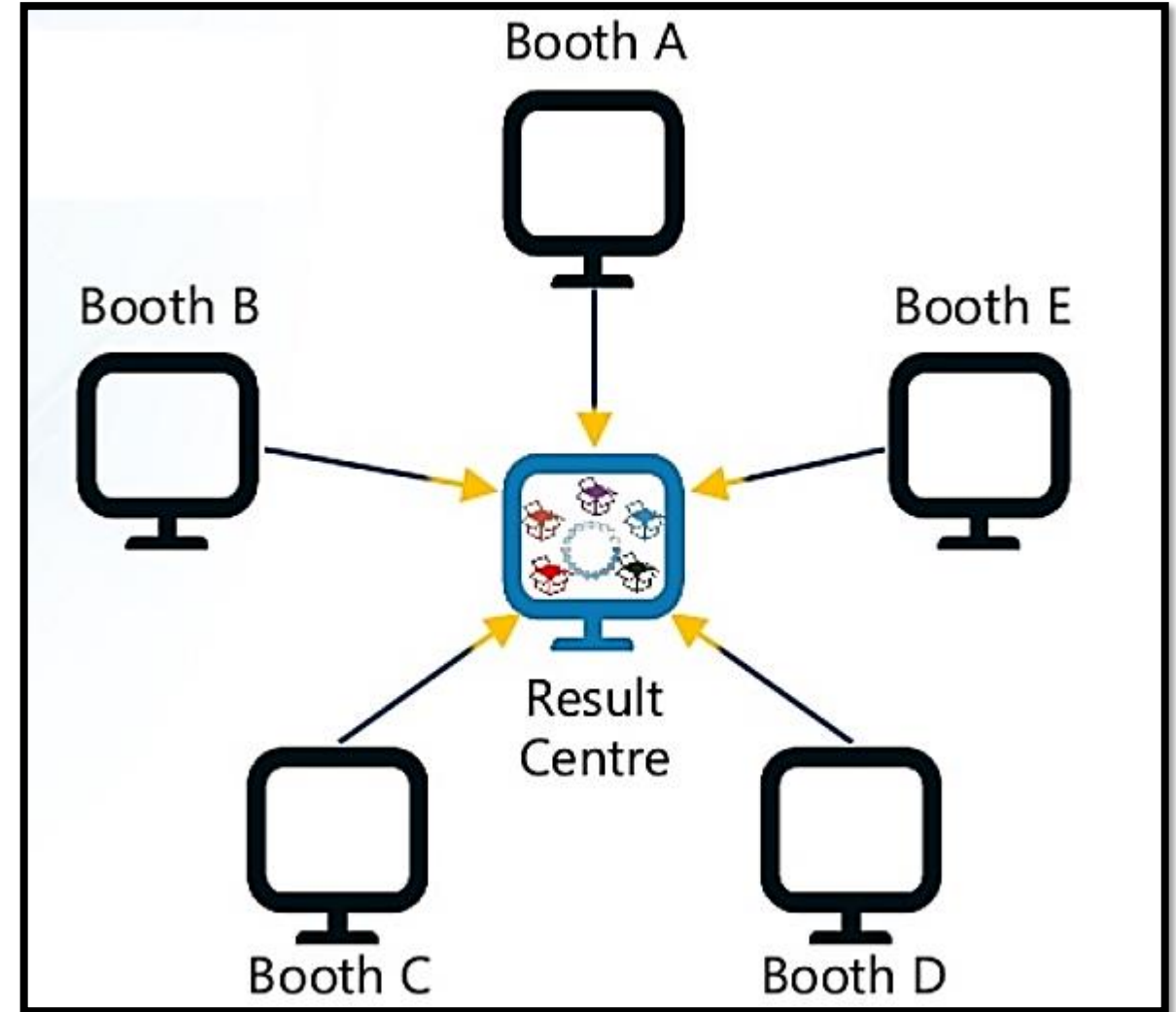


Example

Data → 

Counting – Traditional Approach

- Votes are moved to Result Centre for counting
- Moving all the votes to Centre is costly
- Result Centre is over-burdened
- Counting takes time



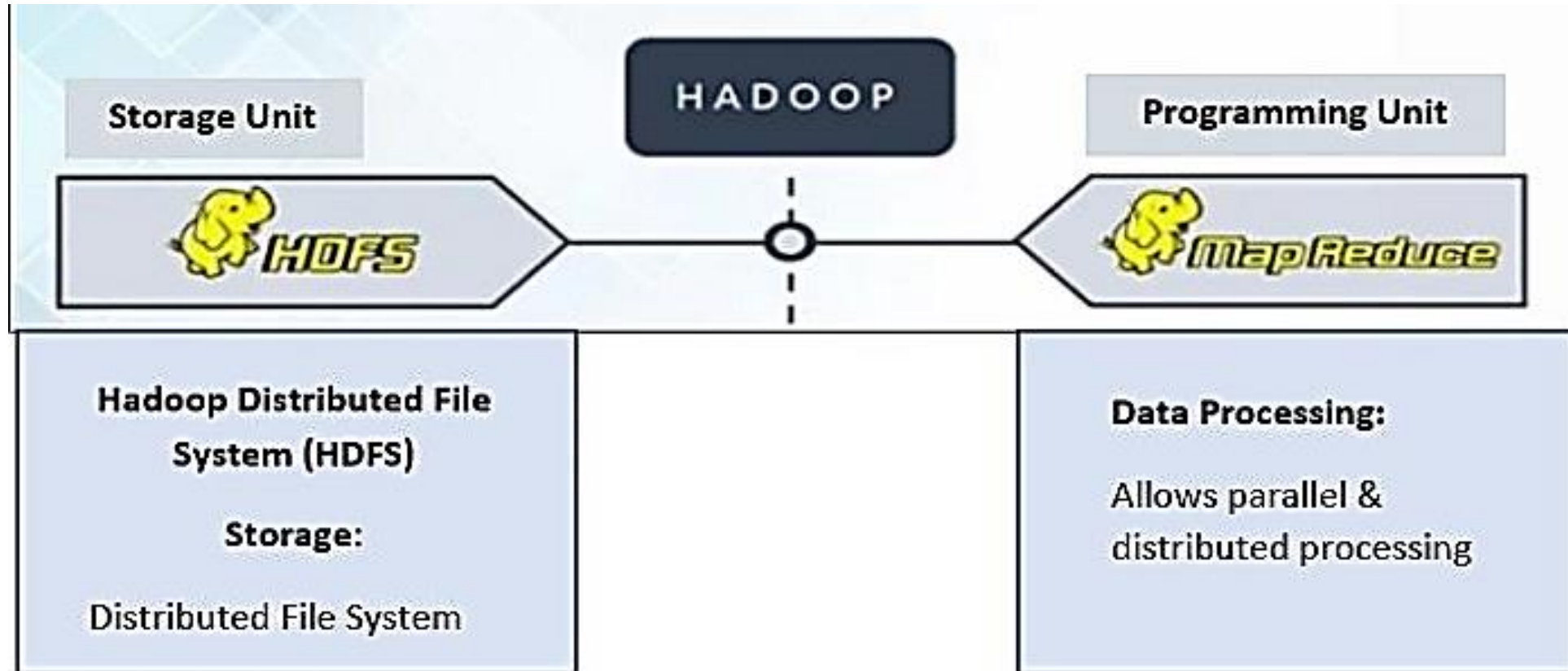
Challenges-Distributed processing (Traditional way)

- If, any of the machines delay the job, the whole work gets delayed (Critical path problem).
- What if, any of the machines which are working with a part of data fails? The management of this failover becomes a challenge (Reliability problem).
- how to equally divide the data such that no individual machine is overloaded or underutilized (Equal split issue).
- There should be a mechanism to aggregate the result generated by each of the machines to produce the final output (Aggregation of the result).

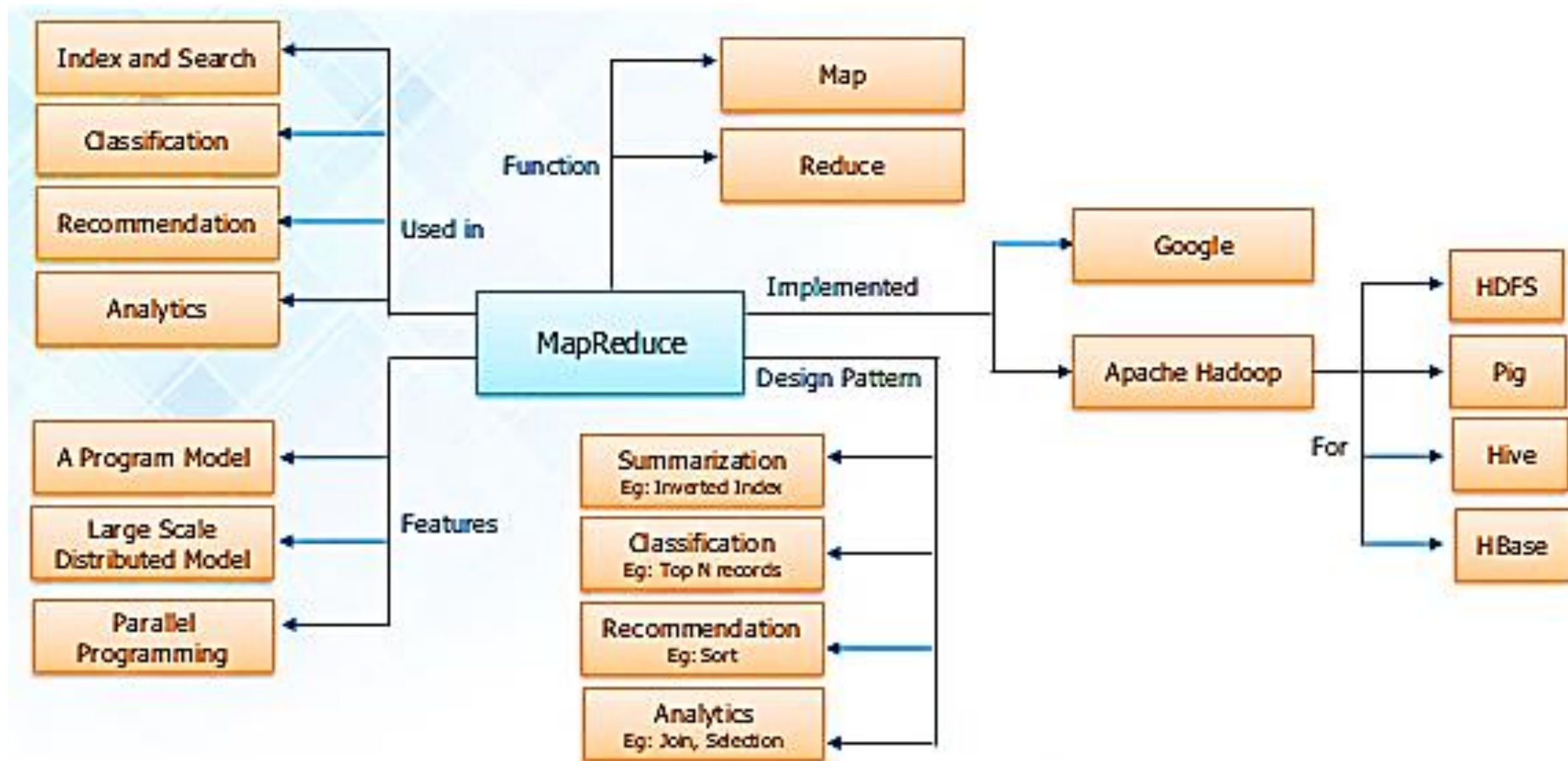
Above are the issues which we will have to take care individually while performing parallel processing of huge data sets when using traditional approaches.

To overcome these issues, we have the **MapReduce framework** which allows us to perform such parallel computations without bothering about the issues like reliability, fault tolerance etc.

Main Hadoop Components



MapReduce



MapReduce: What It Is and Why It Is Important

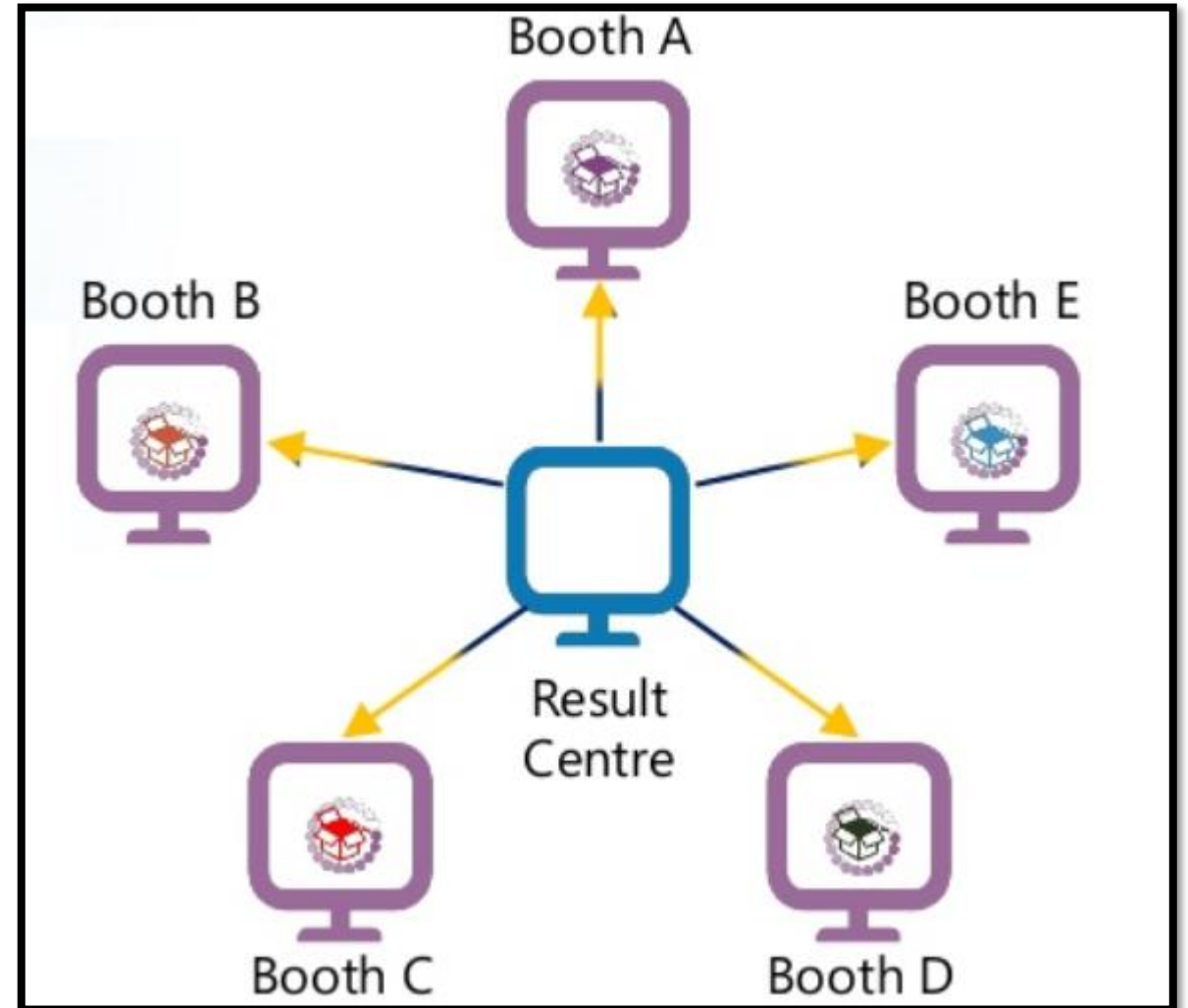
- MapReduce is a Distributed Data Processing Algorithm, introduced by Google in its MapReduce Tech Paper (2004).
- MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.
- MapReduce is a programming model that allows us to perform parallel and distributed processing on huge data sets.

Example: Counting the votes

Votes →     

Counting – MapReduce Approach

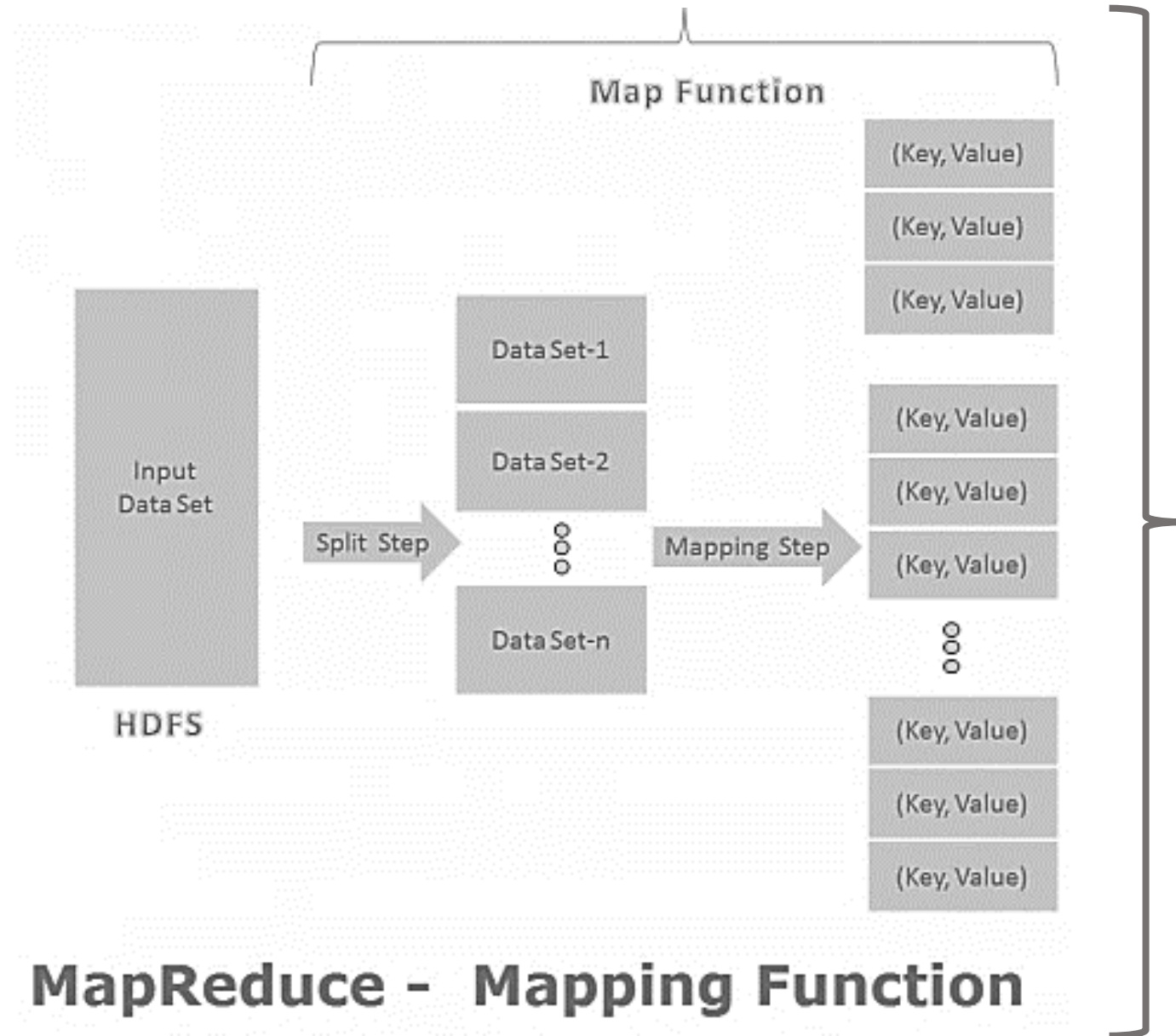
- Votes are counted at individual booths
- Booth-wise results are sent back to the result centre
- Final Result is declared easily and quickly using this way



How does MapReduce work?

- Three main steps of MapReduce: **Map Function**, **Shuffle Function**, **Reduce Function**
- **Map Function**
 - Map Function is the first step in MapReduce Algorithm.
 - Map Function performs the following two sub-steps:
 - **Splitting** (takes input DataSet from Source and divide into smaller Sub-Datasets)
 - **Mapping** (On Sub-Datasets, perform required computation on each Sub-DataSet.)
 - The output of this Map Function is a **set of key and value pairs as <Key, Value>**.

How does MapReduce work?



MapReduce First Step Output:

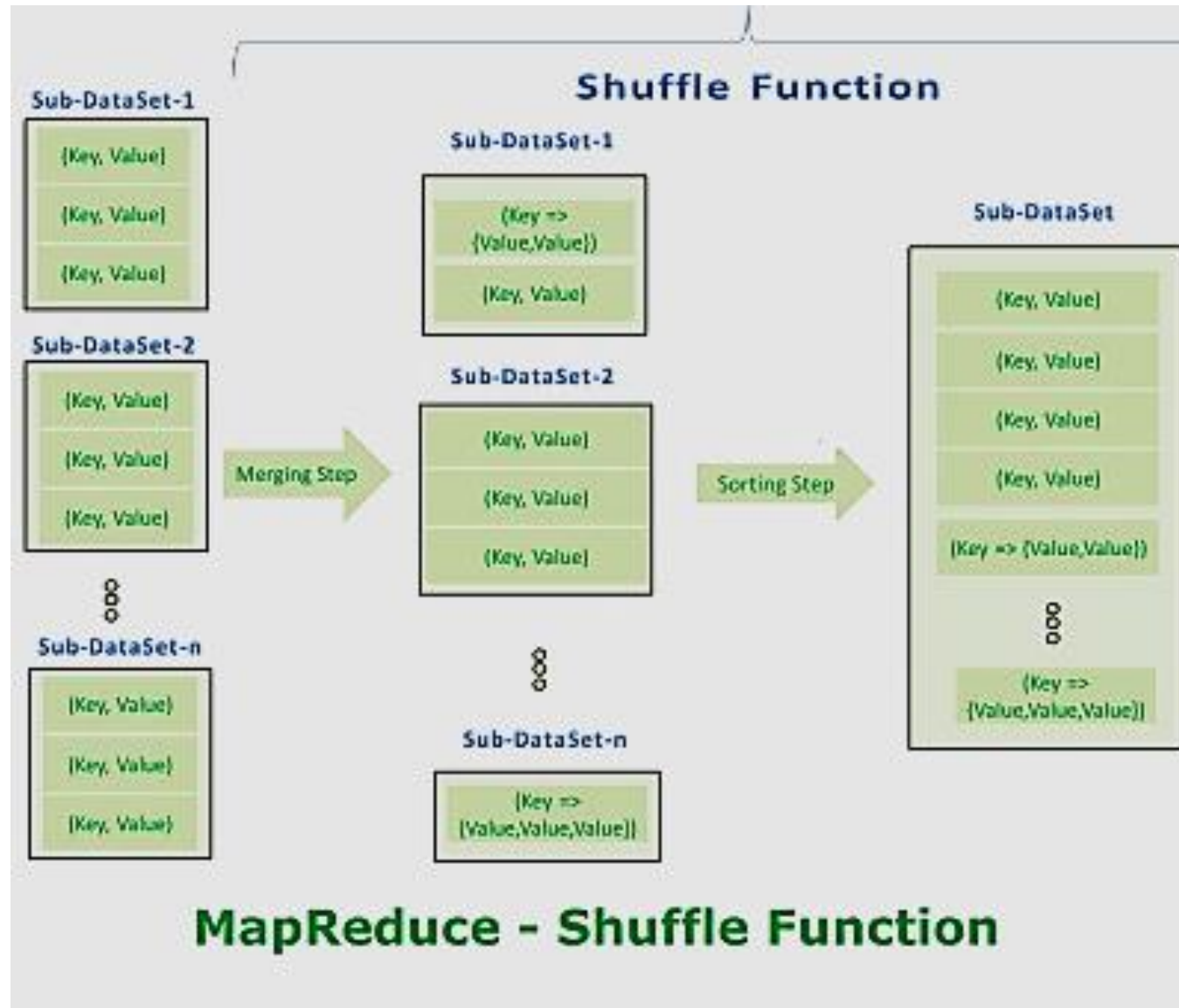
Map Function Output = List of <Key, Value> Pairs

How does MapReduce work?

- **Shuffle Function**

- It is the second step in MapReduce Algorithm. Shuffle Function is also known as “Combine Function”.
- It takes a list of outputs coming from “Map Function” and performs following two sub-steps on each and every key-value pair.
 - **Merging:** this step combines all key-value pairs which have same keys (that is grouping key-value pairs by comparing “Key”). This step returns $\langle \text{Key}, \text{List}\langle \text{Value} \rangle \rangle$.
 - **Sorting:** this step takes input from Merging step and sorts all key-value pairs by using Keys. This step also returns $\langle \text{Key}, \text{List}\langle \text{Value} \rangle \rangle$ output but with sorted key-value pairs.
- Finally, Shuffle Function returns a list of $\langle \text{Key}, \text{List}\langle \text{Value} \rangle \rangle$ sorted pairs to next step.

How does MapReduce work?



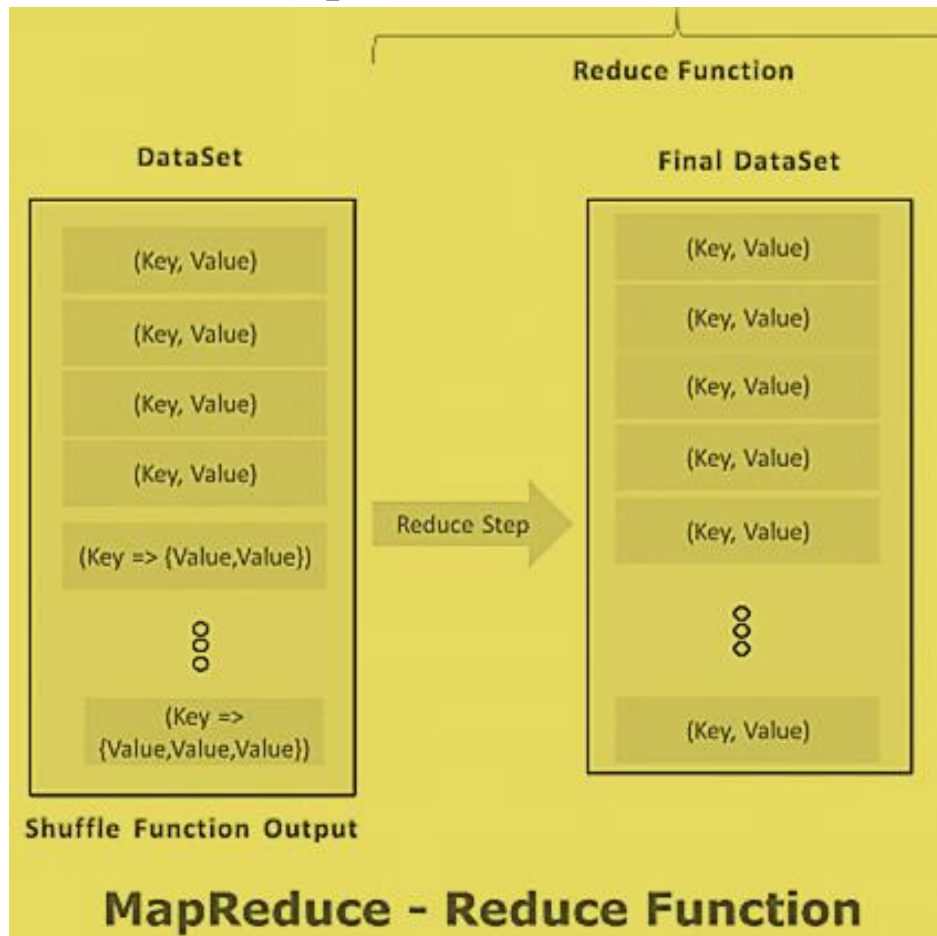
MapReduce Second Step Output:

Shuffle Function Output = List of <Key, List<Value>> Pairs

How does MapReduce work?

- **Reduce Function**

- It is the final step in MapReduce Algorithm. It performs only one step : Reduce step.
- It takes list of $\langle \text{Key}, \text{List}\langle \text{Value} \rangle \rangle$ sorted pairs from Shuffle Function and perform reduce operation as shown below.



MapReduce Final Step Output:

Reduce Function Output = List of $\langle \text{Key}, \text{Value} \rangle$ Pairs

- Final step output looks like first step output.
- However final step $\langle \text{Key}, \text{Value} \rangle$ pairs are different than first step $\langle \text{Key}, \text{Value} \rangle$ pairs.
- Final step $\langle \text{Key}, \text{Value} \rangle$ pairs are computed and sorted pairs.

MapReduce Example – Word Count

- **Problem Statement:**

- Count the number of occurrences of each word available in a DataSet.

- **Input DataSet:**

1	Red Blue Red Blue Green Red Blue Green
2	White Black
3	Red White Black
4	Orange Green
5	Red Blue Red
6	Blue Green Red Blue
7	Green White Black

- **Client Required Final Result:**

1	Black = 3
2	Blue = 6
3	Green = 5
4	Orange = 1
5	Red = 7
6	White = 3

- MapReduce – Map Function (Split Step)

1	Red Blue Red Blue Green Red Blue Green
2	White Black
3	Red White Black
4	Orange Green
5	Red Blue Red
6	Blue Green Red Blue
7	Green White Black

Input DataSet

Split Step

1	Sub-DataSet-1
2	-----
3	Red Blue Red Blue Green Red Blue Green
4	White Black
5	Red White Black

1	Sub-DataSet-2
2	-----
3	Orange Green
4	Red Blue Red
5	Blue Green Red Blue
6	Green White Black

Split Step Result

- MapReduce – Map Function (Mapping Step)

```
1 Sub-DataSet-1
2 -----
3 Red Blue Red Blue Green Red Blue Green
4 White Black
5 Red White Black
```

```
1 Sub-DataSet-2
2 -----
3 Orange Green
4 Red Blue Red
5 Blue Green Red Blue
6 Green White Black
```

Split Step Result

Mapping Step

```
1 Sub-DataSet-1
2 -----
3 Red = 1
4 Blue = 1
5 Red = 1
6 Blue = 1
7 Green = 1
8 Red = 1
9 Blue = 1
10 Green = 1
11 White = 1
12 Black = 1
13 Red = 1
14 White = 1
15 Black = 1
```

```
1 Sub-DataSet-2
2 -----
3 Orange = 1
4 Green = 1
5 Red = 1
6 Blue = 1
7 Red = 1
8 Blue = 1
9 Green = 1
10 Red = 1
11 Blue = 1
12 Green = 1
13 White = 1
14 Black = 1
```


- **MapReduce – Shuffle Function (Merge Step)**

```
1 Sub-DataSet-1
2 -----
3 Red = 1
4 Blue = 1
5 Red = 1
6 Blue = 1
7 Green = 1
8 Red = 1
9 Blue = 1
10 Green = 1
11 White = 1
12 Black = 1
13 Red = 1
14 White = 1
15 Black = 1
```

```
1 Sub-DataSet-2
2 -----
3 Orange = 1
4 Green = 1
5 Red = 1
6 Blue = 1
7 Red = 1
8 Blue = 1
9 Green = 1
10 Red = 1
11 Blue = 1
12 Green = 1
13 White = 1
14 Black = 1
```

Merging Step-1

```
1 Sub-DataSet-1
2 -----
3 Red = {1,1,1,1}
4 Blue = {1,1,1}
5 Green = {1,1}
6 White = {1,1}
7 Black = {1,1}
8
```

```
1 Sub-DataSet-2
2 -----
3 Orange = {1}
4 Green = {1,1,1}
5 Red = {1,1,1}
6 Blue = {1,1,1}
7 White = {1}
8 Black = {1}
```

- MapReduce – Shuffle Function (Merge Step)

```
1 Sub-DataSet-1
2 -----
3 Red = {1,1,1,1}
4 Blue = {1,1,1}
5 Green = {1,1}
6 White = {1,1}
7 Black = {1,1}
8
```

```
1 Sub-DataSet-2
2 -----
3 Orange = {1}
4 Green = {1,1,1}
5 Red = {1,1,1}
6 Blue = {1,1,1}
7 White = {1}
8 Black = {1}
```

Merging Step-2

```
1 DataSet
2 -----
3 Red = {1,1,1,1,1,1,1}
4 Blue = {1,1,1,1,1,1}
5 Green = {1,1,1,1,1}
6 White = {1,1,1}
7 Black = {1,1,1}
8 Orange = {1}
```

- **MapReduce – Shuffle Function (Sorting Step)**

```
1 DataSet
2 -----
3 Red = {1,1,1,1,1,1,1}
4 Blue = {1,1,1,1,1,1}
5 Green = {1,1,1,1,1}
6 White = {1,1,1}
7 Black = {1,1,1}
8 Orange = {1}
```

Sorting Step

```
1 DataSet
2 -----
3 Black = {1,1,1}
4 Blue = {1,1,1,1,1,1}
5 Green = {1,1,1,1,1}
6 Orange = {1}
7 Red = {1,1,1,1,1,1,1}
8 White = {1,1,1}
```

MapReduce - Sorting Step

- **MapReduce – Reduce Function (Reduce Step)**

1	DataSet
2	-----
3	Black = {1,1,1}
4	Blue = {1,1,1,1,1,1}
5	Green = {1,1,1,1,1}
6	Orange = {1}
7	Red = {1,1,1,1,1,1,1}
8	White = {1,1,1}

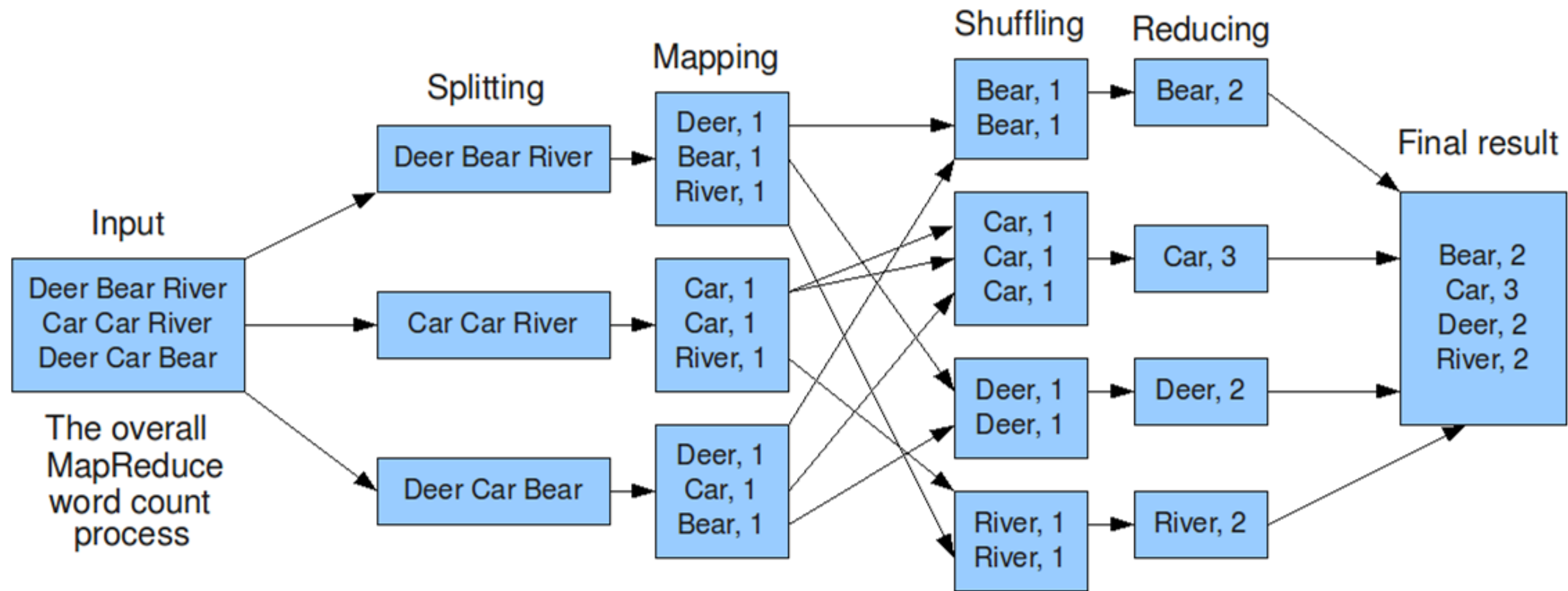
Reduce Step

1	Black = 3
2	Blue = 6
3	Green = 5
4	Orange = 1
5	Red = 7
6	White = 3

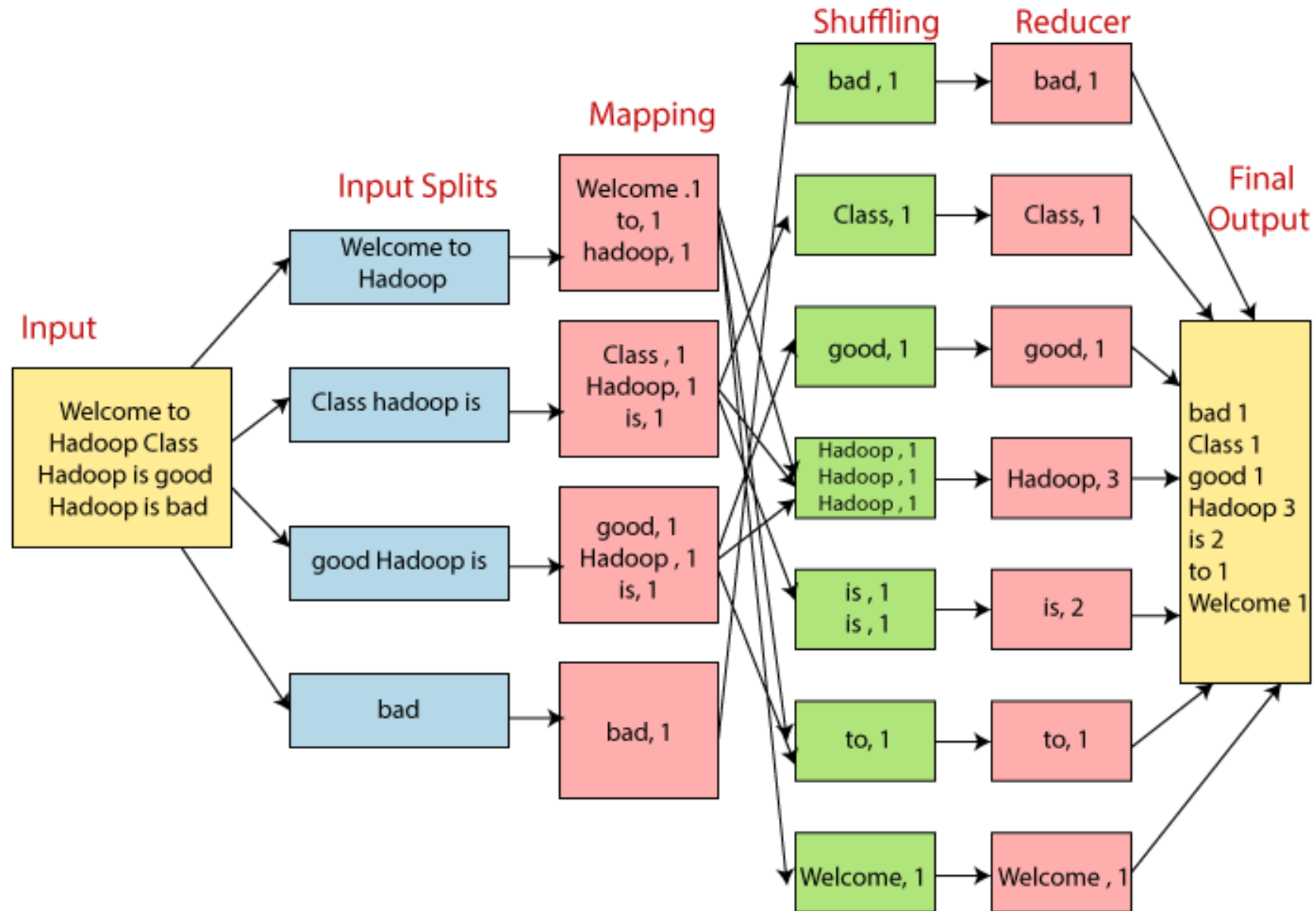
Final Output

MapReduce - Reduce Step

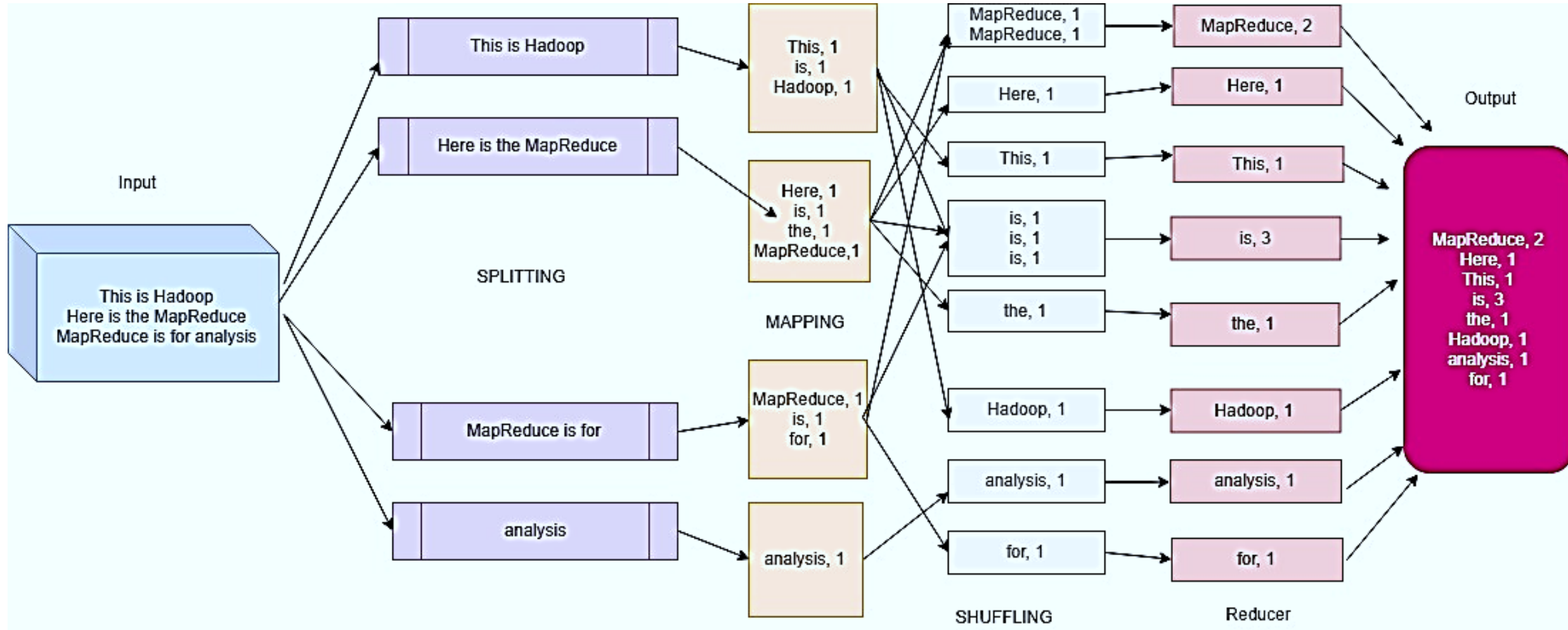
Example-2



Example-3



Example-4



How does MapReduce Work ?

From start to finish, there are four fundamental transformations.

Data is :

- 1. Transformed from the input files and fed into the mappers.
 - 2. Transformed by the mappers.
 - 3. Sorted, merged, and presented to the reducer.
 - 4. Transform by reducers and written to output files.
- Most of the **computing takes place on nodes with data on local disks** that reduces the network traffic. After completion of the given tasks, the **cluster collects and reduces the data to form an appropriate result**, and sends it back to the Hadoop server.

	Input	Output
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)

Map and Reduce Phases:

- A MapReduce program in Hadoop is called a Hadoop job.
- Map task is broken into Phases like:
 - **RecordReader**,
 - **Mapper**,
 - **Combiner**,
 - **Partitioner**
- Reducer task is broken into phases like:
 - **Shuffle**,
 - **Sort**,
 - **Reducer**,
 - **O/P format**

Technical details of Map and Reduce Phases

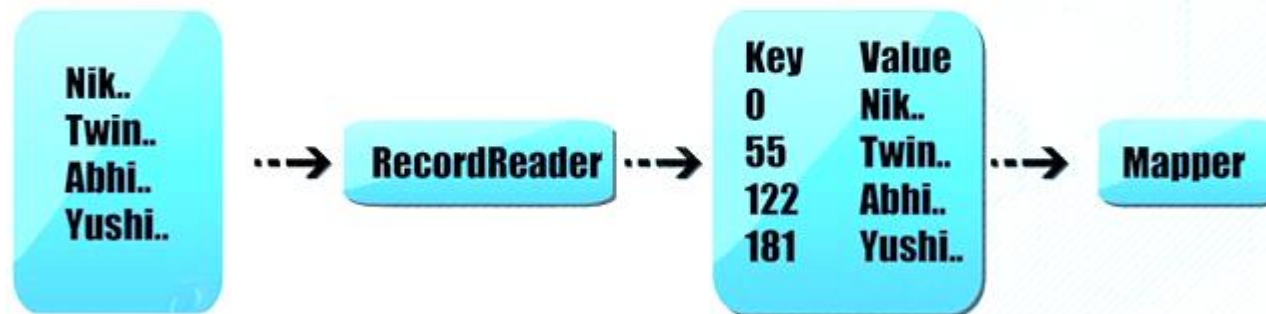
- **Input Files:** The data for a Map Reduce task is stored in input files and these input files are generally stored in HDFS.
- **InputFormat:**
 - Take the file from HDFS and divides the data into the number of splits (typically 64/128mb).
 - InputFormat defines how the input files are split and read.
 - InputFormat create the InputSplit.
- **InputSplit:**
 - InputSplit is the logical representation of data.
 - It represents the data which is processed by an individual Mapper.
 - One map task is created for each Input Split. (Note: Number of InputSplits = Number of map tasks).
 - The split is divided into records and each record will be processed by the mapper.

Technical details of Map and Reduce Phases

- **RecordReader:**

- It communicates with InputSplit in and converts the data into key-value pairs suitable for reading by the mapper.
- By default, it uses TextInputFormat for converting data into a key-value pair.
- It assigns byte offset (unique number) to each line present in the file. Then, these key-value pairs are sent to the mapper for further processing.

RecordReader in Hadoop



It Converts Data into key Value Format

Map and Reduce Phases:

- **Mapper**

- Map function works on the key-value pair produced by RecordReader and **generate zero or more intermediate key-value pairs.**
- Mappers output is passed to the combiner for further process.

- **Combiner**

- It is optional function but provide high performance in terms of n/w bandwidth and disk space.
- Like map output in some stage is **<1,10>, <1,15>, <1,20>, <2,5>, <2,60>** and the **purpose of map-reduce job is to find the maximum value** corresponding to each key.
- In combiner you can reduce this data to **<1,20> , <2,60>** as 20 and 60 are maximum value for key 1 and key 2 respectively.
- It is an optimization technique for MapReduce Job.
- Output generated by combiner is intermediate data and it is passed to the reducer.

Map and Reduce Phases:

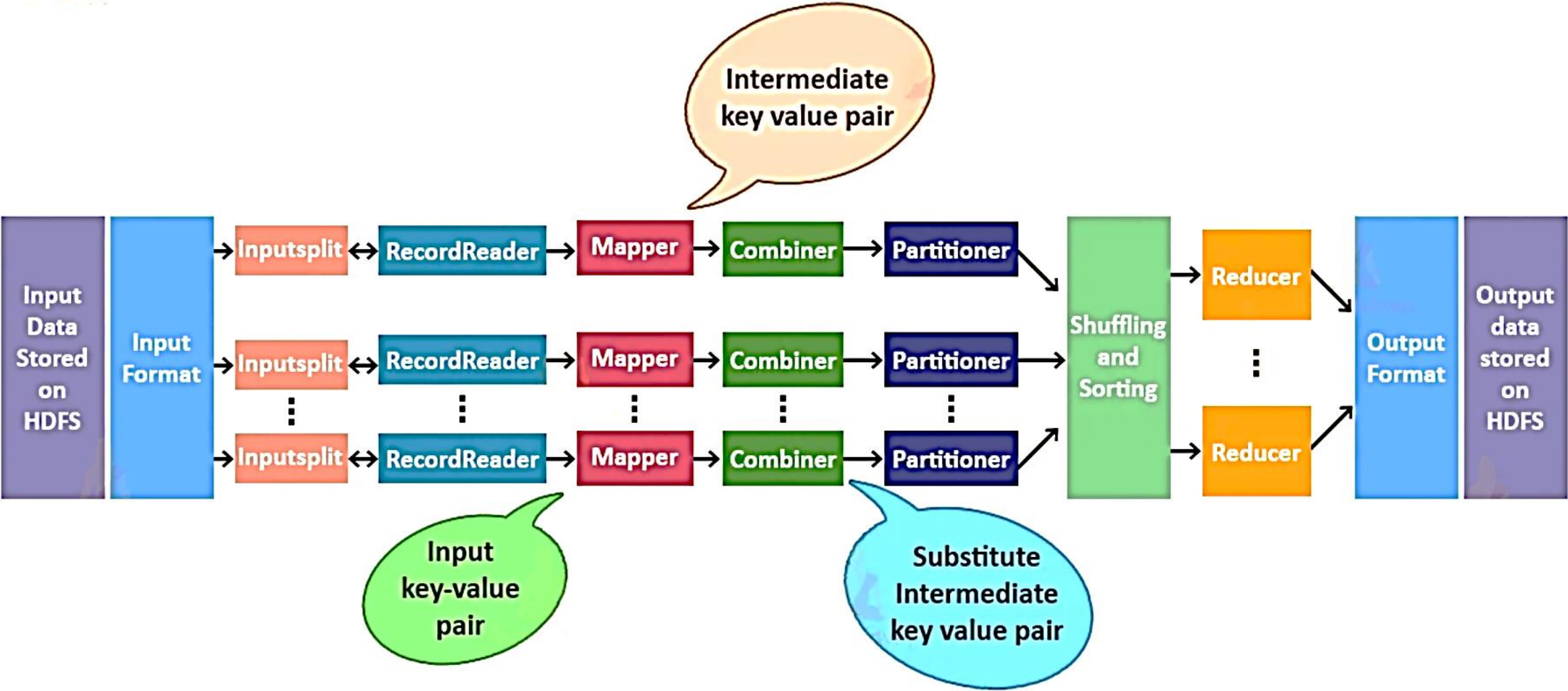
- **Partitioner**

- It happens after map phase and before reduce phase.
- Map task returns output in <key,value> form.
- Partitioner is club the data which should go to the same reducer based on keys.

- **Example:**

- If map output is <1,10> , <1,15> , <1,20> , <2, 13> , <2,6> , <4,8> , <4,20> etc.
- We can see that there are 3 different keys which are 1, 2 and 4.
- In map-reduce **number of reduce tasks are fixed and each reduce task should handle all the data related to one key.**
- That means map output like <1,10>, <1,15>, <1,20> should be handled by same reduce tasks.
- It is not possible that <1,10> is handled by one reduce task and <1,15> is handled by another reduce task as key which is 1 is same.

Entire workflow of a job in Hadoop.

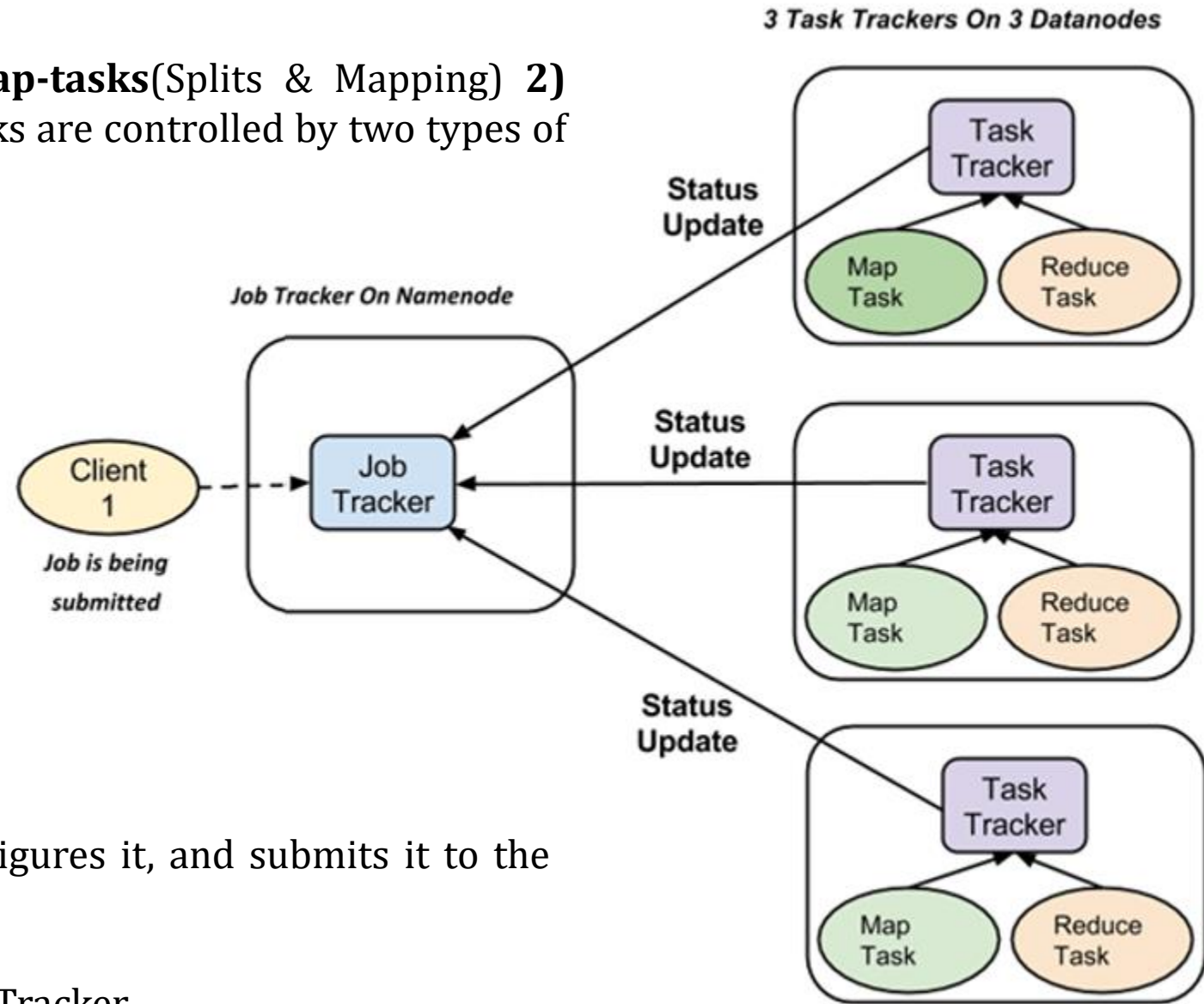


How MapReduce Organizes Work?

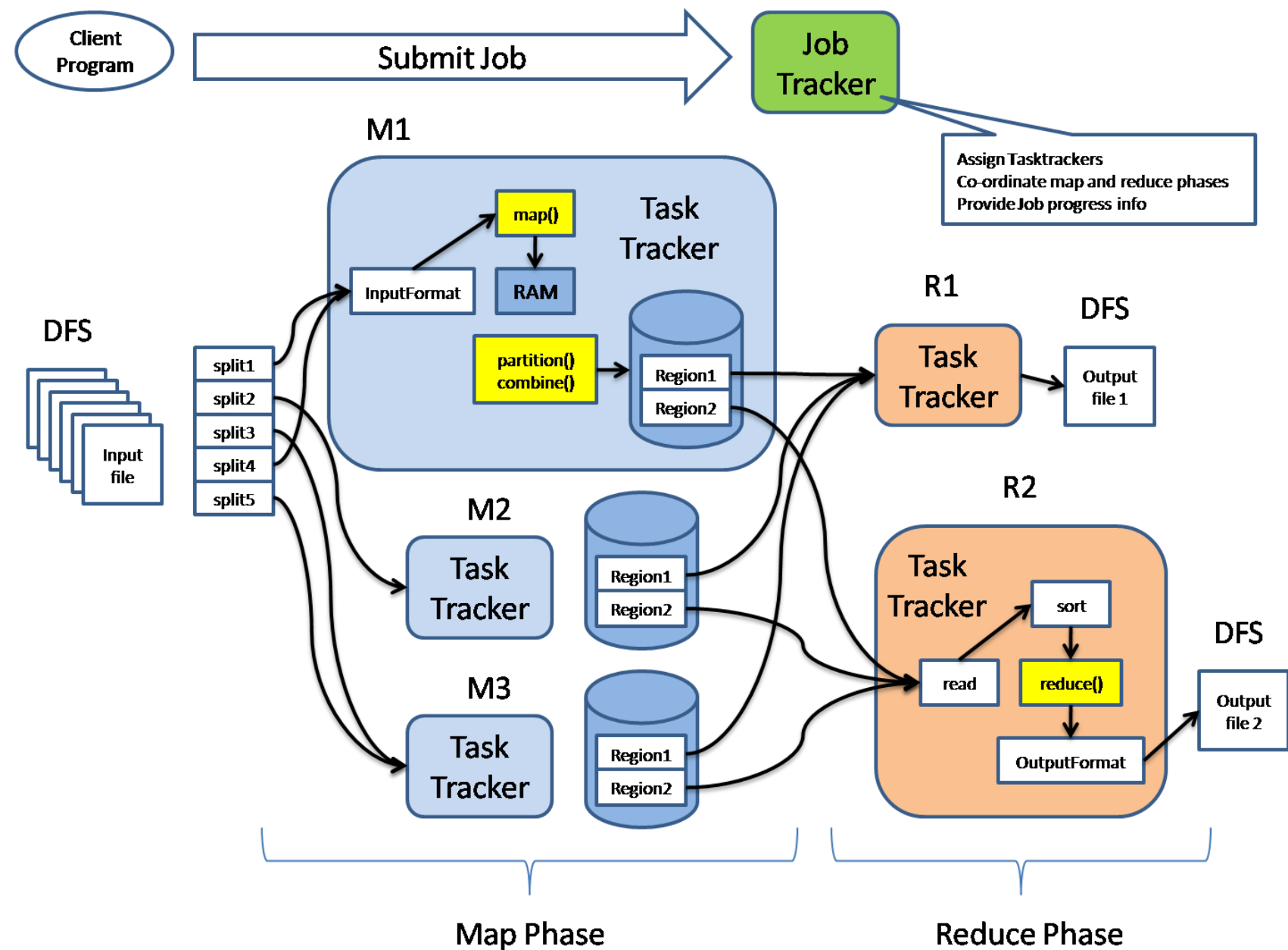
- ❖ Hadoop divides the job into two tasks. **1) Map-tasks**(Splits & Mapping) **2) Reduce-tasks**(Shuffling, Reducing). These two tasks are controlled by two types of entities:
- ❖ **JobTracker**: Act as **single-master** (responsible for complete execution of submitted job). Each job, there is **one Jobtracker** that resides on **Namenode**.
- ❖ **TaskTrackers**: Act as **one slaves/Cluster Node**, each of them performing the job as directed by the master. Each job, there are **multiple Tasktrackers** which reside on **Datanode**.

Job submission process is as follows:

- ❖ Client (i.e., driver program) creates a job, configures it, and submits it to the JobTracker
- ❖ JobClient computes input splits (on client end)
- ❖ Job data (jar, configuration XML) are sent to JobTracker
- ❖ JobTracker puts job data in a shared location, enqueues tasks
- ❖ TaskTrackers poll for tasks



Hadoop Map Reduce Architecture



How MapReduce Organizes Work?

- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.
- In addition, task tracker periodically sends '**heartbeat**' signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

MapReduce Architecture explained in detail

- One map task is created for each split which then executes map function for each record in the split.
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Map output is intermediate output which is processed by reduce tasks to produce the final output.
- Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.

MapReduce Architecture explained in detail

- In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
- An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine, the output is merged and then passed to the user-defined reduce function.
- Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

Terminology used in Map Reduce

Pay Load	Applications implement the Map and the Reduce functions, and form the core of the job.
Mapper	Mapper maps the input key/value pairs to a set of intermediate key/value pair.
Named Node	Node that manages the Hadoop Distributed File System (HDFS).
DataNode	Node where data is presented in advance before any processing takes place.
Master Node	Node where JobTracker runs and which accepts job requests from clients.
Slave Node	Node where Map and Reduce program runs.
Job Tracker	Schedules jobs and tracks the assign jobs to Task tracker.
Task Tracker	Tracks the task and reports status to JobTracker.
Job	A program is an execution of a Mapper and Reducer across a dataset.
Task	An execution of a Mapper or a Reducer on a slice of data.
Task Attempt	A particular instance of an attempt to execute a task on a SlaveNode.

MapReduce key benefits

Benefit	Description
Simplicity	Developers have freedom to write applications in their language of choice, such as Java, C++ or Python, and MapReduce jobs are easy to run.
Scalability	MapReduce can process huge amount of data (in terms of petabytes and more) , stored in HDFS on one cluster
Speed	It uses parallel processing it means that MapReduce can take problems that used to take days to solve which can be solved in hours or minutes
Recovery	MapReduce also takes care of failures. If a machine with one copy of the data is unavailable than another machine has a copy of the same key/value pair, which can be used to solve the same sub-task. The JobTracker keeps track of these all processes.
Minimal data motion	MapReduce moves are able to compute processes to the data on HDFS and not the other way around. Processing tasks can occur on the physical node where the data resides. This significantly reduces the network I/O patterns and contributes to Hadoop's processing speed.

- Business and other organizations run calculations to:
 - Determine the price for their products that yields the highest profits.
 - Know precisely how effective their advertising is and where they should spend their ad dollars.
 - Make long-range weather predictions.
 - Web clicks, sales records purchased from retailers, and Twitter trending topics to determine what new products the company should produce in the upcoming season.

MapReduce Use Case: Global Warming

Query: we want to know how much global warming has raised the ocean's temperature.

- Input temperature readings from thousands of OceanSignals all over the globe.
(OceanSignals, DateTime, longitude, latitude, lowTemp, highTemp)
- Run a map over every OceanSignals -dateTime reading and add the average temperature as a field:
(OceanSignals, DateTime, longitude, latitude, lowTemp, highTemp, **Average**)
- Drop DateTime column and produce one average temperature for each OceanSignals
(OceanSignal N, Average) //like, (OceanSignal 1, Average) , (OceanSignal 2, Average)
- Then the reduce operation runs.
$$\text{Ocean Average Temperature} = \frac{\text{Average (OceanSignal n)} + \text{Average (OceanSignal n-1)} + \dots + \text{Average (OceanSignal 2)} + \text{Average (OceanSignal 1)}}{\text{number of OceanSignals}}$$