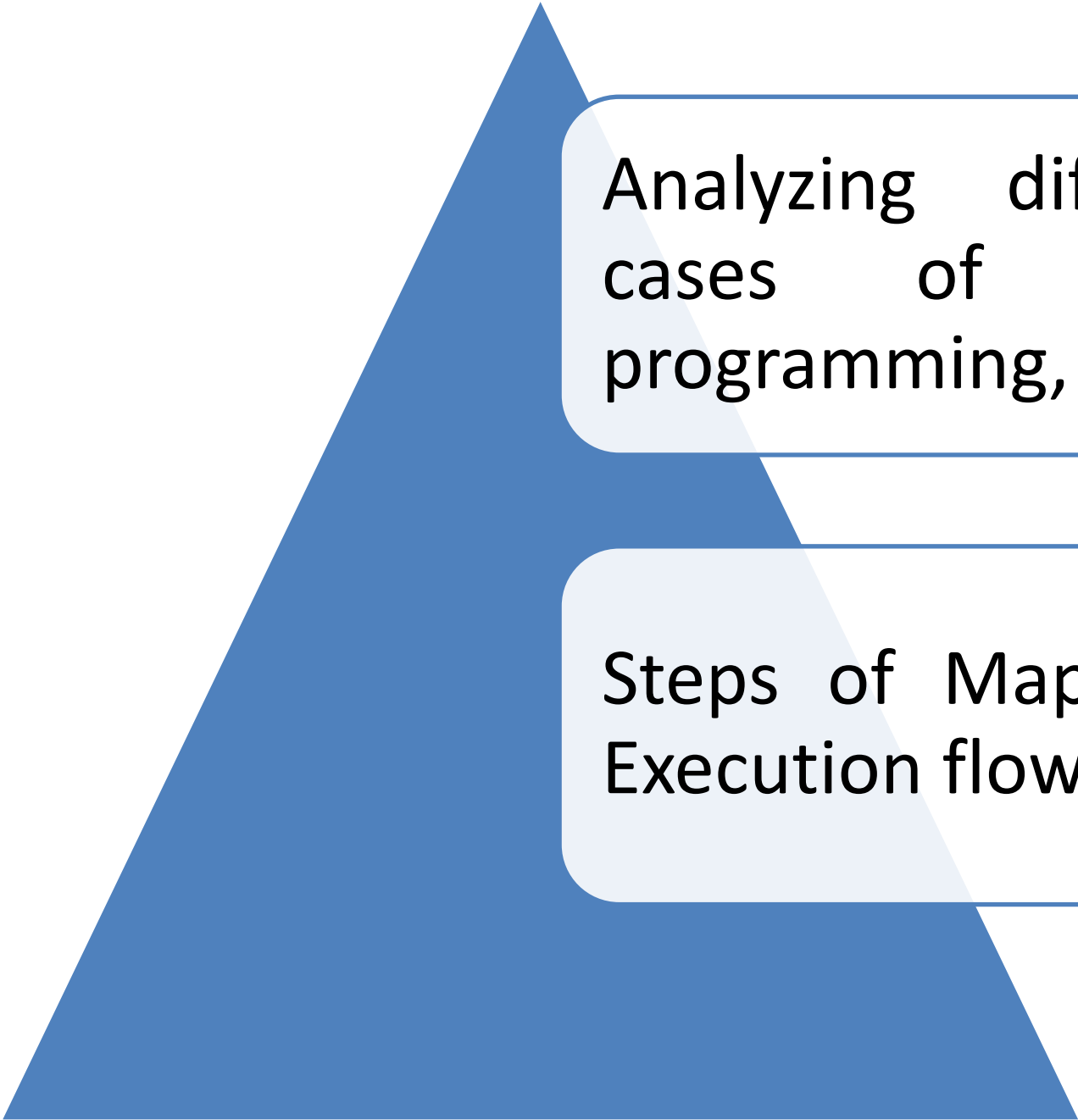


Hadoop Architecture and HDFS

Part 2

Dr. S. Srivastava



Analyzing different use cases of mapreduce programming,

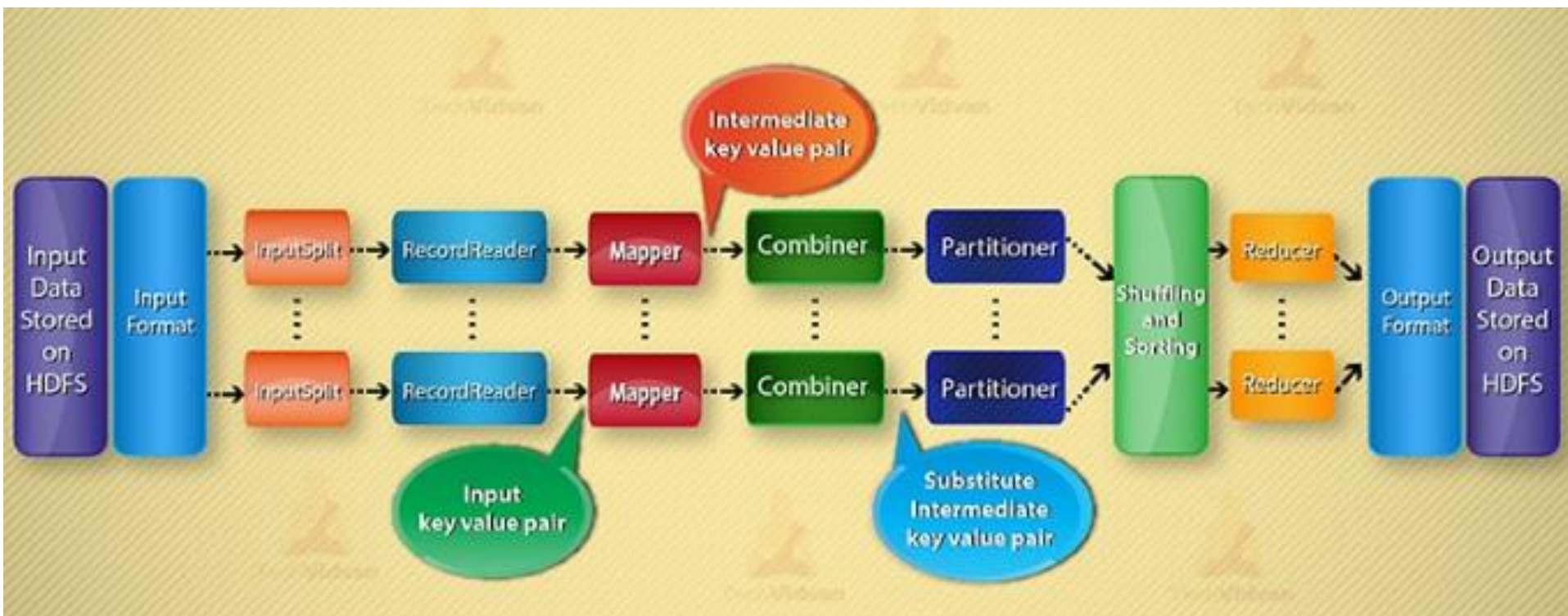
Steps of MapReduce Job Execution flow

MapReduce Use Case: YouTube Data Analysis

<https://acadgild.com/blog/mapreduce-use-case-youtube-data-analysis>

Steps of MapReduce Job Execution flow

MapReduce processes the data in various phases with the help of different components.



MapReduce Job Execution Flow

Input Files

- In input files data for MapReduce job is stored.
- In **HDFS**, input files reside.
- Input files format is arbitrary.
- Log files and binary format can also be used.

2. InputFormat

- **Hadoop InputFormat** describes the input-specification for execution of the Map-Reduce job.
- InputFormat describes how to split up and read input files.
- In MapReduce job execution, InputFormat is the first step.
- It is also responsible for creating the input splits and dividing them into records.

In MapReduce, InputFormat class is one of the fundamental classes which provides below functionality:

- InputFormat selects the files or other objects for input.
- It also defines the Data splits. It defines both the size of individual Map tasks and its potential execution server.
- Hadoop InputFormat defines the RecordReader. It is also responsible for reading actual records from the input files.

2.1 How we get the data from Mapper?

- Methods to get the data from mapper are:
- **getsplits()**
- **createRecordReader()**

```
1. public abstract class InputFormat<K, V>
2. {
3.     public abstract List<InputSplit> getSplits(JobContext
        context)
4.     throws IOException, InterruptedException;
5.     public abstract RecordReader<K, V>
6.     createRecordReader(InputSplit split,
7.     TaskAttemptContext context) throws IOException,
8.     InterruptedException;
9. }
```

2.2. Types of InputFormat in MapReduce

- **FileInputFormat-**
- It is the base class for all file-based InputFormats.
- FileInputFormat also specifies input directory which has data files location.
- When we start a MapReduce job execution, FileInputFormat provides a path containing files to read. This InputFormat will read all files.
- Then it divides these files into one or more InputSplits.

TextInputFormat

- It is the default InputFormat. This InputFormat treats each line of each input file as a separate record. It performs no parsing.
- TextInputFormat is useful for unformatted data or line-based records like log files.
- **Key** – It is the byte offset of the beginning of the line within the file (not whole file one split). So it will be unique if combined with the file name.
- **Value** – It is the contents of the line. It excludes line terminators.

KeyValueTextInputFormat

- It is similar to TextInputFormat.
- This InputFormat also treats each line of input as a separate record.
- While the difference is that TextInputFormat treats entire line as the value, but the KeyValueTextInputFormat breaks the line itself into key and value by a tab character ('/t'). Hence,

Key – Everything up to the tab character.

Value – It is the remaining part of the line after tab character.

SequenceFileInputFormat

- It is an InputFormat which reads sequence files.
- Sequence files are binary files. These files also store sequences of binary key-value pairs.
- These are block-compressed and provide direct serialization and deserialization of several arbitrary data.

Hence,
Key & Value both are user-defined.

SequenceFileAsTextInputFormat

- It is the variant of SequenceFileInputFormat.
- This format converts the sequence file key values to Text objects.
- So, it performs conversion by calling '**toString()**' on the keys and values.
- Hence, SequenceFileAsTextInputFormat makes sequence files suitable input for streaming.

SequenceFileAsBinaryInputFormat

By using SequenceFileInputFormat we can extract the sequence file's keys and values as an opaque binary object.

NLineInputFormat

- Each mapper receives a variable number of lines of input with TextInputFormat and KeyValueTextInputFormat.
- The number depends on the size of the split. Also, depends on the length of the lines.
- So, if want our mapper to receive a fixed number of lines of input, then we use NLineInputFormat.

N- It is the number of lines of input that each mapper receives.

By default (N=1), each mapper receives exactly one line of input.

Suppose N=2, then each split contains two lines. So, one mapper receives the first two Key-Value pairs. Another mapper receives the second two key-value pairs

DBInputFormat

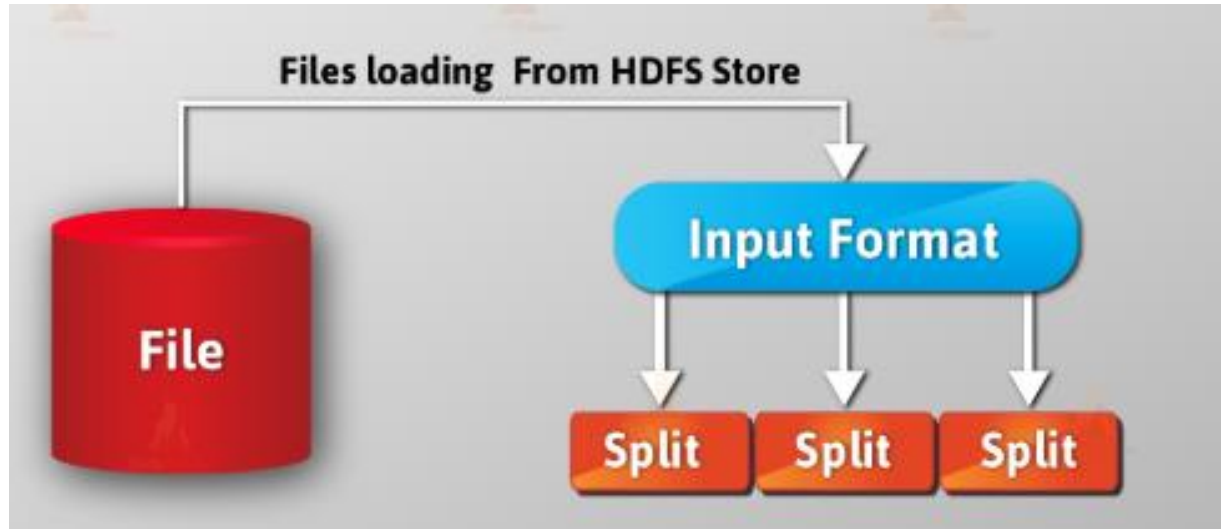
- This InputFormat reads data from a relational database, using JDBC.
- It also loads small datasets, perhaps for joining with large datasets from HDFS using MultipleInputs.
- Hence,

Key – LongWritable

Value – DBWritable.

3. InputSplits

- It represents the data which will be processed by an individual **Mapper**.
- For each split, one map task is created.
- Thus the number of map tasks is equal to the number of InputSplits.
- Framework divide split into records, which mapper process.



MapReduce InputSplit length has measured in bytes. Every InputSplit has storage locations (hostname strings).

The MapReduce system places map tasks as close to the split's data as possible by using storage locations.

Framework processes Map tasks in the order of the size of the splits so that the largest one gets processed first (greedy approximation algorithm).

This minimizes the job run time. The main thing to focus is that Inputsplit does not contain the input data; it is just a reference to the data.

InputSplit is user defined. The user can also control split size based on the size of data in MapReduce program. Hence, In a MapReduce job execution number of map tasks is equal to the number of InputSplits.

By calling '**getSplit()**', the client calculate the splits for the job. Then it sent to the application master, which uses their storage locations to schedule map tasks that will process them on the cluster.

After that map task passes the split to the **createRecordReader()** method. From that it obtains RecordReader for the split. Then RecordReader generate record (**key-value pair**), which it passes to the map function.

4. RecordReader

- It communicates with the inputSplit. And then converts the data into **key-value pairs** suitable for reading by the Mapper.
- RecordReader by default uses TextInputFormat to convert data into a key-value pair.
- It communicates to the InputSplit until the completion of file reading. It assigns byte offset to each line present in the file.
- Then, these key-value pairs are further sent to the mapper for further processing.

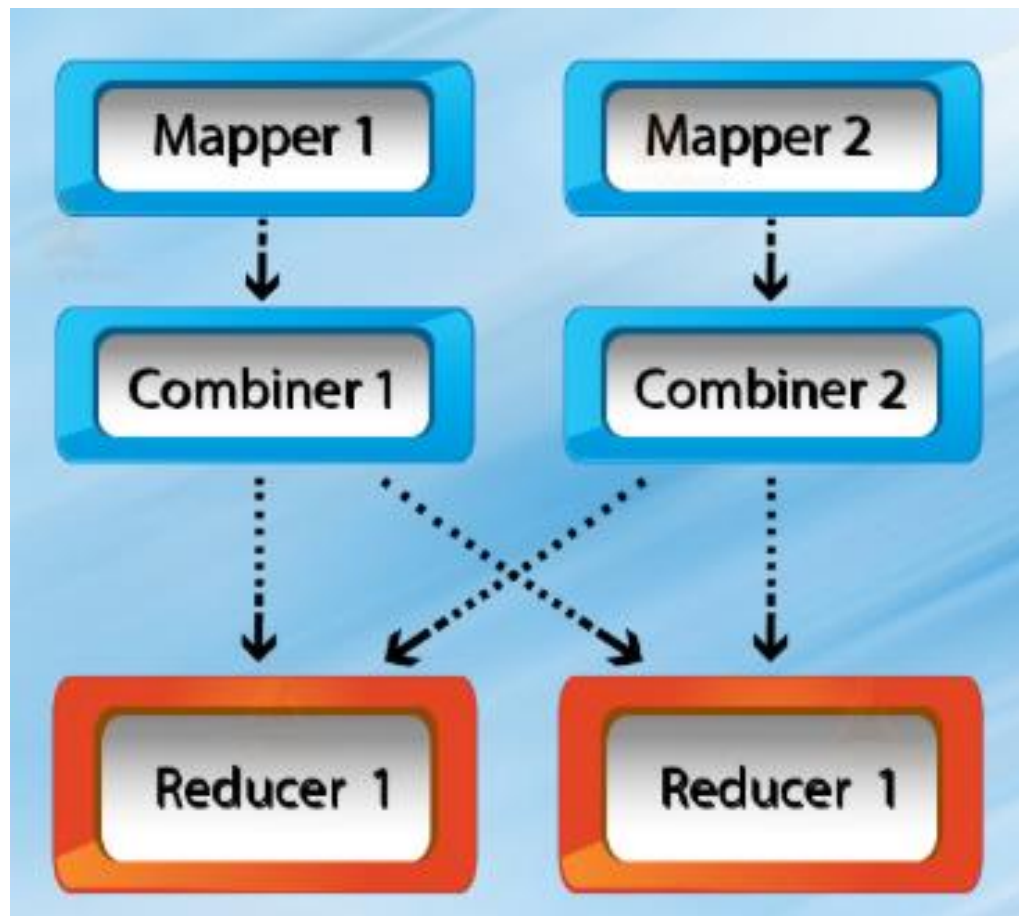
5. Mapper

- It processes input record produced by the RecordReader and generates intermediate key-value pairs.
- The intermediate output is completely different from the input pair. The output of the mapper is the full collection of key-value pairs. Hadoop framework doesn't store the output of mapper on HDFS.
- It doesn't store, as data is temporary and writing on HDFS will create unnecessary multiple copies. Then Mapper passes the output to the combiner for further processing.

6. Combiner

- Combiner is Mini-reducer which performs local aggregation on the mapper's output.
- It minimizes the data transfer between mapper and reducer.
- So, when the combiner functionality completes, framework passes the output to the partitioner for further processing.

Mapper generates large chunks of intermediate data. Then the framework passes this intermediate data on the Reducer for further processing. This leads to enormous network congestion. The Hadoop framework provides a function known as **Combiner** that plays a key role in reducing network congestion.



How does Combiner work in Hadoop?

- (As shown in figure a)
- If there is no combiner then Input is split into two mappers. The framework generates 9 keys from the mappers.
- So, now we have (9 key/value) intermediate data.
- Further mapper sends this **key-value** directly to the reducer. While sending data to the reducer, it consumes some network bandwidth. It takes more time to transfer data to reducer if the size of data is big

MapReduce program without Combiner

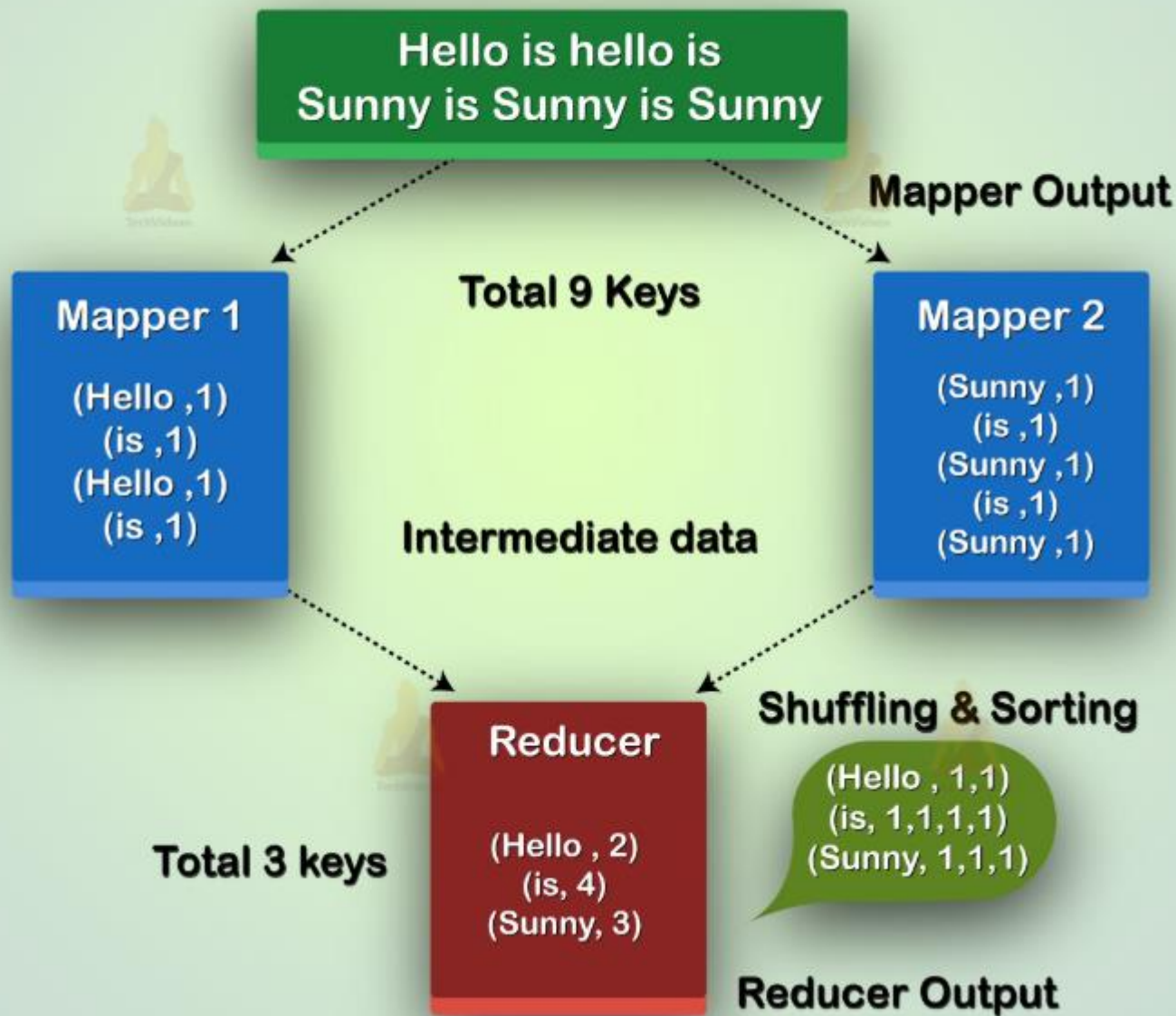


Fig. a. MapReduce program without Combiner

MapReduce program with Combiner

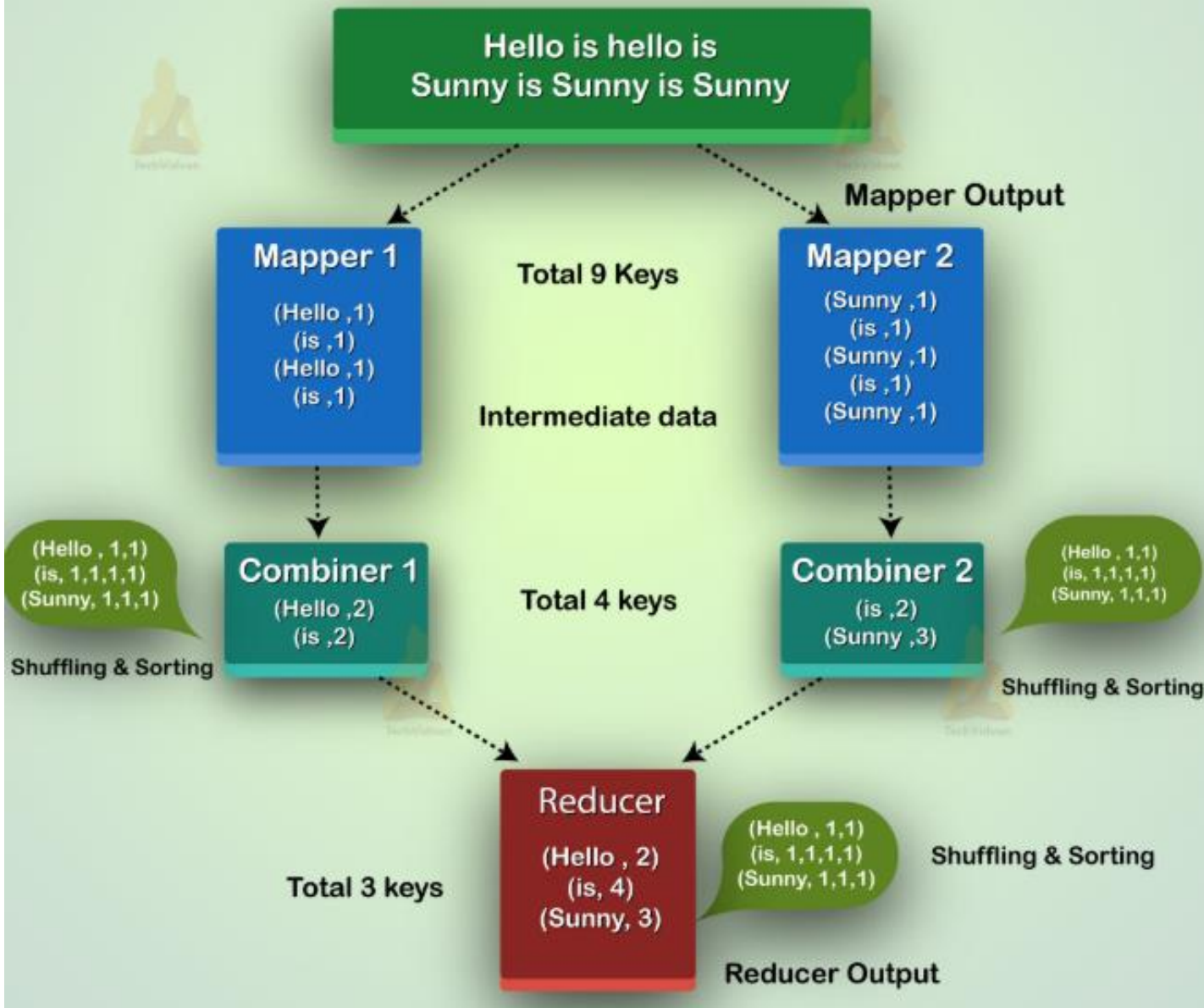


Fig b.
MapReduce
program with
combiner

- if we use a combiner in between mapper and reducer.
- Then combiner will shuffle 9 key/value before sending it to the reducer. And then generates 4 key/value pair as an output.
- Now, Reducer needs to process only 4 key/value pair data which are generated from 2 combiners.
- Therefore reducer gets executed only 4 times to produce the final output. Thus, this increases the overall performance.

Advantages of Combiner in MapReduce

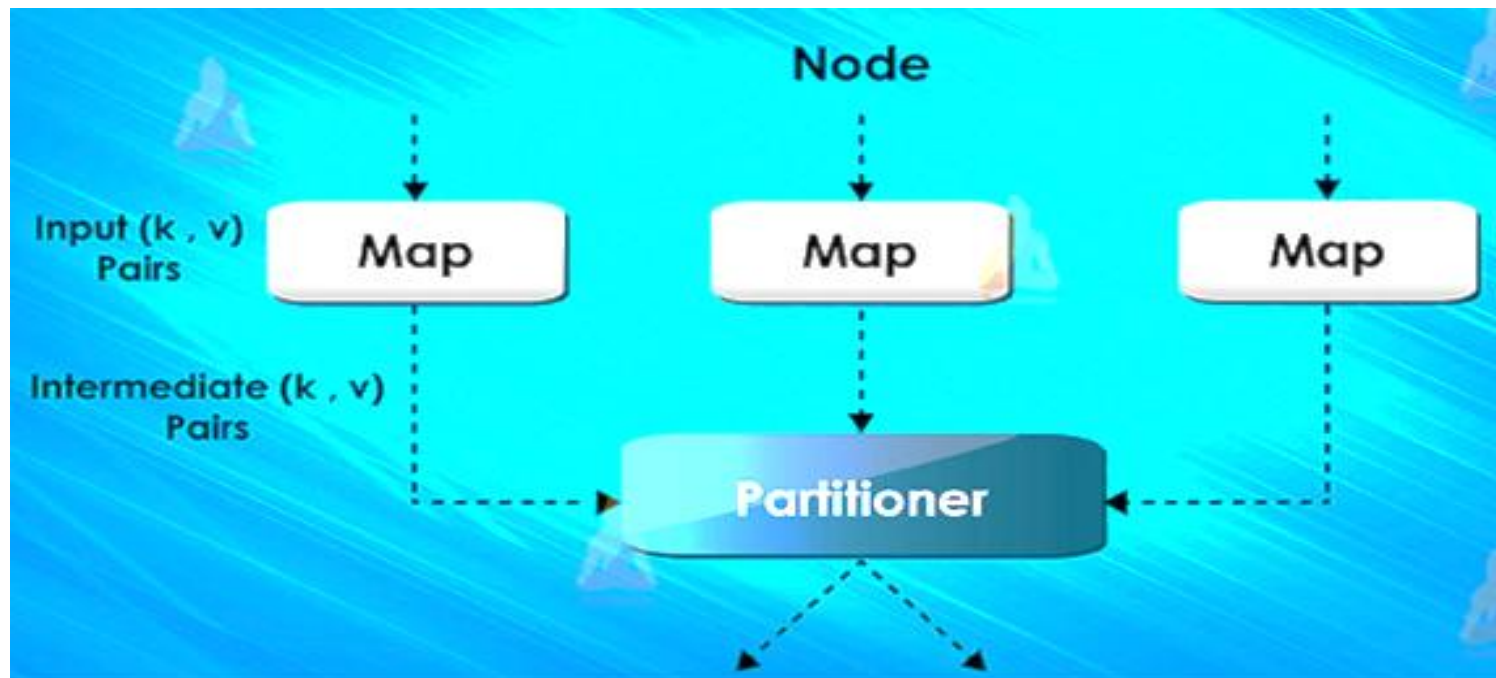
- Use of combiner reduces the time taken for data transfer between mapper and reducer.
- Combiner improves the overall performance of the reducer.
- It decreases the amount of data that reducer has to process.

Disadvantages of Combiner in MapReduce

- .
- In the local filesystem, when Hadoop stores the key-value pairs and run the combiner later this will cause expensive disk IO.
- MapReduce jobs can't depend on the combiner execution as there is no guarantee in its execution.

7. Partitioner

- Partitioner comes into the existence if we are working with more than one reducer.
- It takes the output of the combiner and performs partitioning.
- Partitioning of output takes place on the basis of the key in MapReduce.
- By hash function, key (or a subset of the key) derives the partition.
- On the basis of key value in MapReduce, partitioning of each combiner output takes place. And then the record having the same key value goes into the same partition. After that, each partition is sent to a reducer.



Hash Partitioner is the default Partitioner. It computes a hash value for the key. It also assigns the partition based on this result.

The total number of Partitioner depends on the number of reducers. Hadoop Partitioner divides the data according to the number of reducers.

It is set by **JobConf.setNumReduceTasks()** method.

Thus the single reducer processes the data from single partitioner. The important thing to notice is that the framework creates partitioner only when there are many reducers.

8. Shuffling and Sorting

- After partitioning, the output is shuffled to the reduce node.
- The shuffling is the physical movement of the data which is done over the network.
- As all the mappers finish and shuffle the output on the reducer nodes. Then framework merges this intermediate output and sort.
- This is then provided as input to reduce phase.

9. Reducer

- Reducer then takes set of intermediate key-value pairs produced by the mappers as the input.
- After that runs a reducer function on each of them to generate the output.
- The output of the reducer is the final output. Then framework stores the output on HDFS

Phases of Hadoop Reducer

- Three phases of Reducer are as follows:
- **1. Shuffle Phase-** This is the phase in which sorted output from the mapper is the input to the reducer. The framework with the help of HTTP fetches the relevant partition of the output of all the mappers in this phase.
- **2. Sort Phase-** This is the phase in which the input from different mappers is again sorted based on the similar keys in different Mappers.
- Both Shuffle and Sort occur concurrently.
- **3. Reduce Phase-** This phase occurs after shuffle and sort. Reduce task aggregates the key-value pairs.
With the ***OutputCollector.collect()*** property, the output of the reduce task is written to the FileSystem. Reducer output is not sorted.

10. RecordWriter

- It writes these output key-value pair from the Reducer phase to the output files.

11.OutputFormat

- OutputFormat defines the way how RecordReader writes these output key-value pairs in output files.
- So, its instances provided by the Hadoop write files in HDFS. Thus OutputFormat instances write the final output of reducer on HDFS.