

## MAPREDUCE

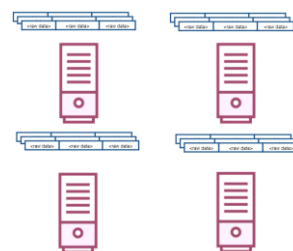
- ▢ Every click, every like, every share, every visit to the web page is tracked somewhere.
- ▢ This means that there is a processing of huge amounts of data.
- ▢ This requires running processes on many machines.
- ▢ Machines work together as distributed systems.



- ▢ MapReduce is a programming paradigm to allow processing on a distributed computing system.
- ▢ The data set map reduce works on is itself distributed on multiple machines in a cluster.
- ▢ Takes advantage of the inherent parallelism in data processing.
- ▢ Modern systems generate millions of records of raw data.
- ▢ Such datasets are petabytes in size.
- ▢ MapReduce breaks up the processing of tasks of this scale is processed in two stages.
  - φ Map
    - ∞ Run on multiple nodes in a cluster
  - φ Reduce
    - ∞ Takes the output of the map phase and further processes it to give the final result.

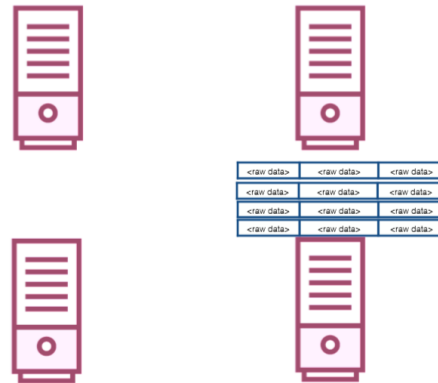
### Map

- ▢ The data set in the cluster will be partitioned across the data nodes.
- ▢ Some of the records will be present on some machines.
- ▢ Other records will be present on other machines.
- ▢ It will also be replicated for fault tolerance.
- ▢ Map processes work on the data that live on their own machines.
- ▢ Whatever output the map phase generates, is then collated together, transferred across the network within the cluster to one node typically where the reduce phase runs.



## Reduce

- ▮ Can also run on multiple nodes.



## MapReduce

- ▮ The developer only writes code for two functions
  - φ Map
  - φ Reduce
- ▮ Hadoop does the rest.
- ▮ Map - An operation performed in parallel, on small portions of the dataset
- ▮ Reduce - An operation to combine the results of the map step in a meaningful way.

## Map

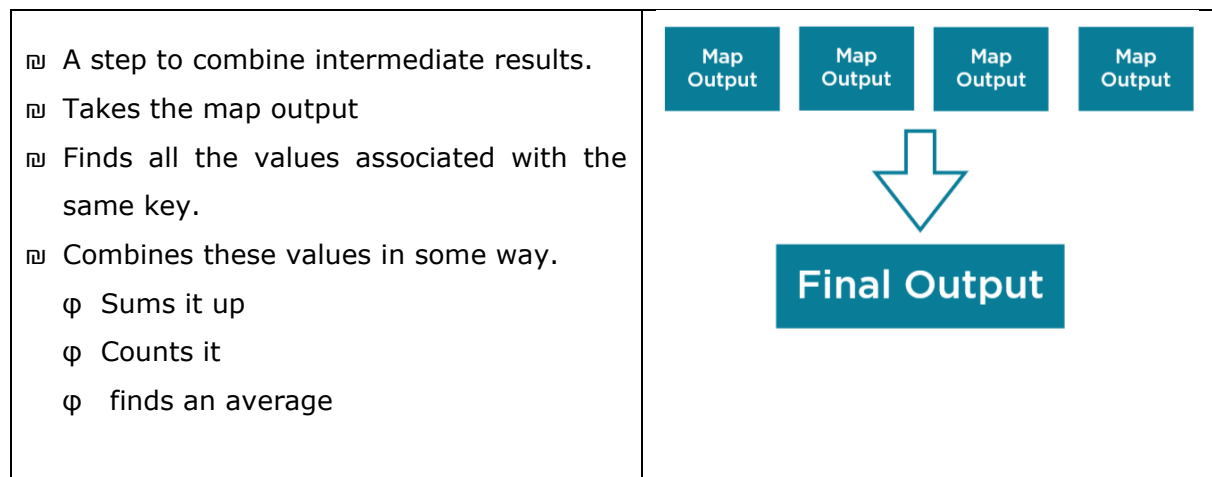
- ▮ A step that can be performed in parallel.
- ▮ Should process only one record.
- ▮ The output is in the form of key-value pair.

One Record



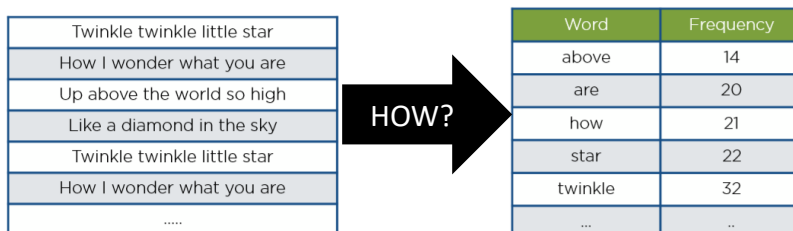
Key-Value Output

## Reduce



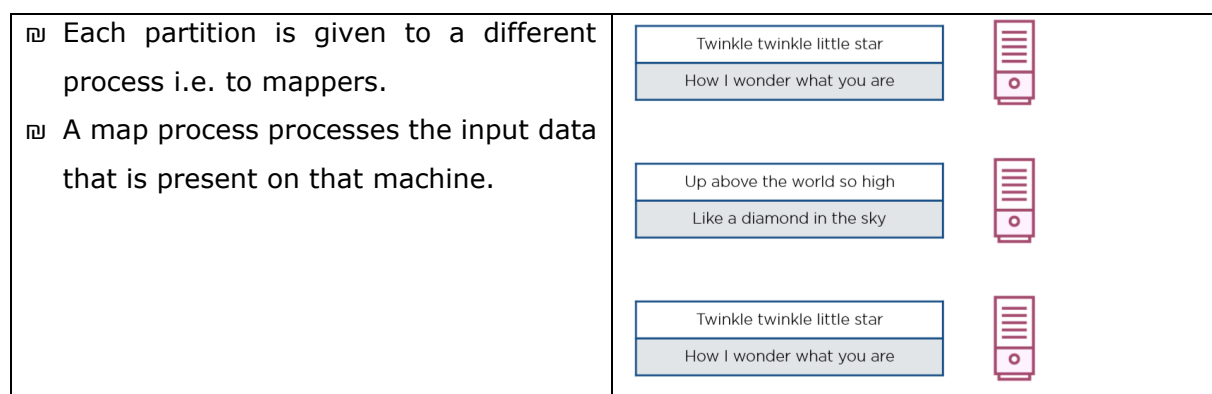
### Counting Word Frequencies

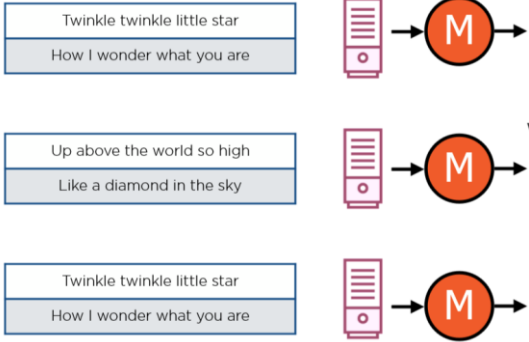
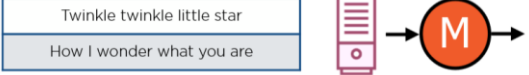
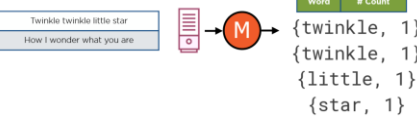
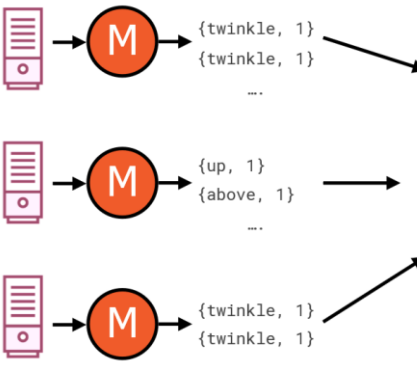
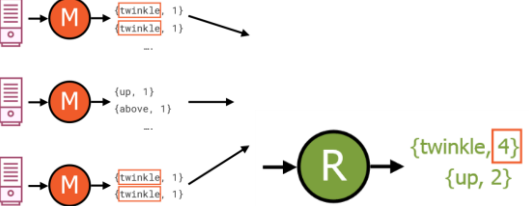
- ▣ Consider a large text file to find the word frequencies of that file.



### Input

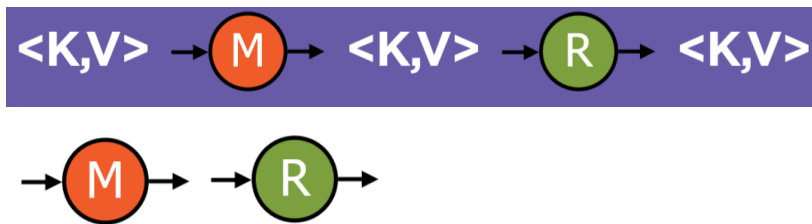
- ▣ The raw data is really large (potentially in PetaBytes)
- ▣ It's distributed across many machines in a cluster.
- ▣ Entire file is not present in one machine.
- ▣ Each machine holds a partition of data



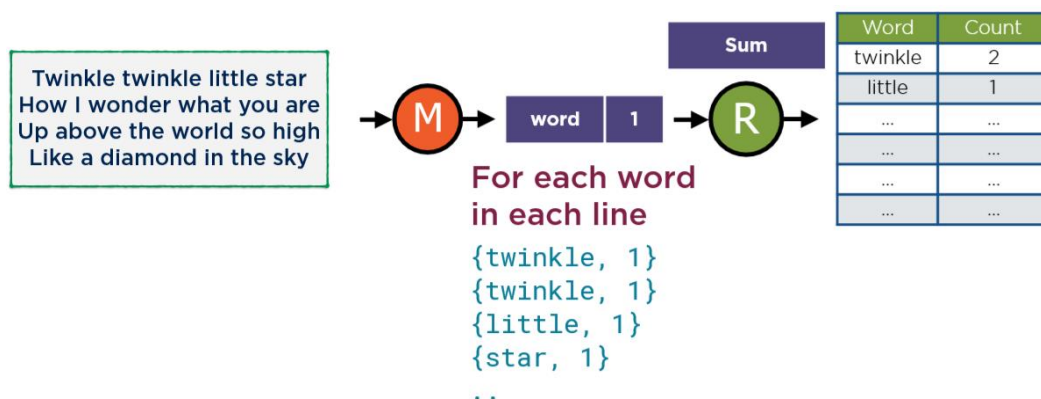
|  |  |
|--|--|
| <p>Each mapper works in parallel.</p>  |    |
| <p>Within each mapper, the rows are processed serially.</p> <p>∅ i.e. they are processed one record at a time.</p> <p>Each row emits {key, value} pairs.</p>       |    |
| <p>This key, value pair depends on what output is needed.</p> <p>In this example, each word of the record is associated with a count of one.</p>                   |    |
| <p>Multiple mappers process the input available to them and produce each individual result.</p> <p>The results are passed on to another process i.e. a reducer</p> |  |
| <p>The reducer combines the values with the same key</p>   |  |

### Key Insight behind MapReduce

- Each mapper works in parallel.
  - ∅ This is a widely acceptable format in data processing.
- Many data processing tasks can be expressed in this form



- ▢ MapReduce program boils down to answering two key questions
  - φ What {key, value} pairs should be emitted in the map phase such that using those keys and values, the reduce phase can produce the final result?
  - φ How should values with the same key be combined?
- ▢ What is the processing required to combine values which have the same key?
- ▢ In the counting word frequencies, the output of the mapper is every word encountered in the input with a frequency count of 1.
- ▢ Every word has exactly 1 as its associated value.
- ▢ Now this is summed up in the reducer.
- ▢ Sum up the counts for every word to get a frequency histogram.



- ▢ The questions have to be answered to parallelize any task.
- ▢ The code for MapReduce is implemented in Java.
- ▢ Hadoop is built in Java and it has a whole bunch of standard libraries that can be used to implement the MapReduce program.
- ▢ Any MapReduce job comprises of three basic components
  - φ Map
  - φ Reduce
  - φ Main
- ▢ Each of this corresponds to individual classes when written in Java.

### Implementing in Java

- ▢ Map
  - φ A class where map logic is implemented

- φ A process that is run on multiple data nodes, processing subsets of the file.
- ▢ Reduce
  - φ A class where reduce logic is implemented
  - φ A class that implements the combining operation on the key-value pairs produced by the numerous map processes which run.
- ▢ Main
  - φ A driver program that sets up the job.

#### Map Step

|   |  |
|---|--|
| <ul style="list-style-type: none"> <li>▢ The map logic run by the mapper processes is typically implemented in a single class.                             <ul style="list-style-type: none"> <li>φ This single class may use other utility classes to process the data.</li> </ul> </li> <li>▢ The map class extends from a base class which is present in the Hadoop library called the mapper class.</li> <li>▢ The mapper class has a whole bunch of helper methods, has only one method that is needed to be overridden.</li> <li>▢ The map logic is implemented in a class that extends the Mapper Class</li> </ul> | <div style="border: 1px solid blue; padding: 10px; text-align: center;"> <p><b>Map Class</b></p> <div style="border: 1px solid blue; padding: 5px; display: inline-block; margin-top: 20px;"> <p><b>Mapper Class</b></p> </div> </div>   |
| <ul style="list-style-type: none"> <li>▢ Map class is a generic class with four type parameters</li> <li>▢ The type parameters specify the                             <ul style="list-style-type: none"> <li>φ Data types of the input keys and values that are fed into the map phase</li> <li>φ Output keys and values that are output from the map phase.</li> </ul> </li> </ul>  | <div style="border: 1px solid blue; padding: 10px; text-align: center;"> <p><b>Map Class</b></p> <p>&lt;input key type,<br/>input value type,<br/>output key type,<br/>output value type&gt;</p> <div style="border: 1px solid blue; padding: 5px; display: inline-block; margin-top: 20px;"> <p><b>Mapper Class</b></p> </div> </div> |
| <ul style="list-style-type: none"> <li>▢ Reduce logic will be returned in a single class.</li> <li>▢ It can access helper classes if needed.</li> <li>▢ This reduce class extends the reducer.java class from the Hadoop library that is provided.</li> <li>▢ Reducer.java has a whole bunch of helper methods.</li> <li>▢ The method that is needed to be overridden is the reduce method in this class.</li> </ul>  | <div style="border: 1px solid blue; padding: 10px; text-align: center;"> <p><b>Reduce Class</b></p> <div style="border: 1px solid blue; padding: 5px; display: inline-block; margin-top: 20px;"> <p><b>Reducer Class</b></p> </div> </div>   |

|   |   |
|---|---|
| <ul style="list-style-type: none"> <li>▣ It is generic class with four type parameters.</li> <li>▣ The type parameters specify the                         <ul style="list-style-type: none"> <li>φ Data types of the input keys and values that are fed into the reduce phase</li> <li>φ Output keys and values of the final result that the reduce phase produces.</li> </ul> </li> </ul>   | <div style="border: 1px solid blue; padding: 10px;"> <p style="color: red; text-align: center;"><b>Reduce Class</b></p> <p style="text-align: center;">&lt;input key type,<br/>input value type,<br/>output key type,<br/>output value type&gt;</p> <div style="border: 1px solid blue; padding: 5px; text-align: center; margin-top: 10px;"> <p style="color: red;">Reducer Class</p> </div> </div>  |
| <ul style="list-style-type: none"> <li>▣ The entire concept of the MapReduce is that the output of the mapper is processes and then fed as the input to the reducer.</li> <li>▣ The output data types of the mapper have to match the input data types of the reduce phase.</li> </ul>  | <div style="border: 1px solid blue; padding: 10px; margin-bottom: 10px;"> <p style="color: red; text-align: center;"><b>Map Class</b></p> <p style="text-align: center;">output key type,<br/>output value type&gt;</p> <div style="border: 1px solid blue; padding: 5px; text-align: center; margin-top: 10px;"> <p style="color: red;">Mapper Class</p> </div> </div> <div style="border: 1px solid blue; padding: 10px;"> <p style="color: red; text-align: center;"><b>Reduce Class</b></p> <p style="text-align: center;">&lt;input key type,<br/>input value type,</p> <div style="border: 1px solid blue; padding: 5px; text-align: center; margin-top: 10px;"> <p style="color: red;">Reducer Class</p> </div> </div> |
| <ul style="list-style-type: none"> <li>▣ The main class is the driver program.</li> <li>▣ The main class sets up a job instance which contains the configuration parameters of the MapReduce and serves as the entry point of the MapReduce job.</li> <li>▣ The main class instantiates a job which has the configuration parameters and it is this job object that's submitted to the Hadoop cluster in order to get the MapReduce running.</li> </ul> | <div style="border: 1px solid blue; padding: 10px; text-align: center;"> <p style="color: red; font-size: 1.2em;"><b>Main Class</b></p> <div style="border: 1px solid blue; padding: 10px; margin: 10px auto; width: 80%;"> <p style="color: red; font-size: 1.2em;"><b>Job Object</b></p> </div> </div>  |
| <ul style="list-style-type: none"> <li>▣ The job object has a bunch of properties that needs to be configured for MapReduce to run successfully.                         <ul style="list-style-type: none"> <li>φ Where are input files kept?</li> <li>φ Where should the output parts be stored?</li> <li>φ What are the key-value data types that the mapper and reducer use?</li> </ul> </li> </ul>  | <div style="border: 1px solid blue; padding: 10px;"> <p style="color: red; text-align: center;"><b>Main Class<br/>Job Object</b></p> <div style="border: 1px solid blue; padding: 5px; margin-top: 10px;"> <p>Input filepath</p> <p>Output filepath</p> <p>Mapper class</p> <p>Reducer class</p> <p>Output data types</p> </div> </div>   |

|   |  |
|---|--|
| ☐ Where is the logic of the actual MapReduce that has to run? |  |
|---|--|

## Process of MapReduce

### ☐ Input Phase

- ☐ Here there is a Record Reader that translates each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.

### ☐ Map

- ☐ Map is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate zero or more key-value pairs.

### ☐ Intermediate Keys

- ☐ The key-value pairs generated by the mapper are known as intermediate keys.

### ☐ Combiner

- ☐ A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets.
- ☐ It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper.
- ☐ It is not a part of the main MapReduce algorithm; it is optional.

### ☐ Shuffle and Sort

- ☐ The Reducer task starts with the Shuffle and Sort step.
- ☐ It downloads the grouped key-value pairs onto the local machine, where the Reducer is running.
- ☐ The individual key-value pairs are sorted by key into a larger data list.
- ☐ The data list groups the equivalent keys together so that their values can be iterated easily in the Reducer task.

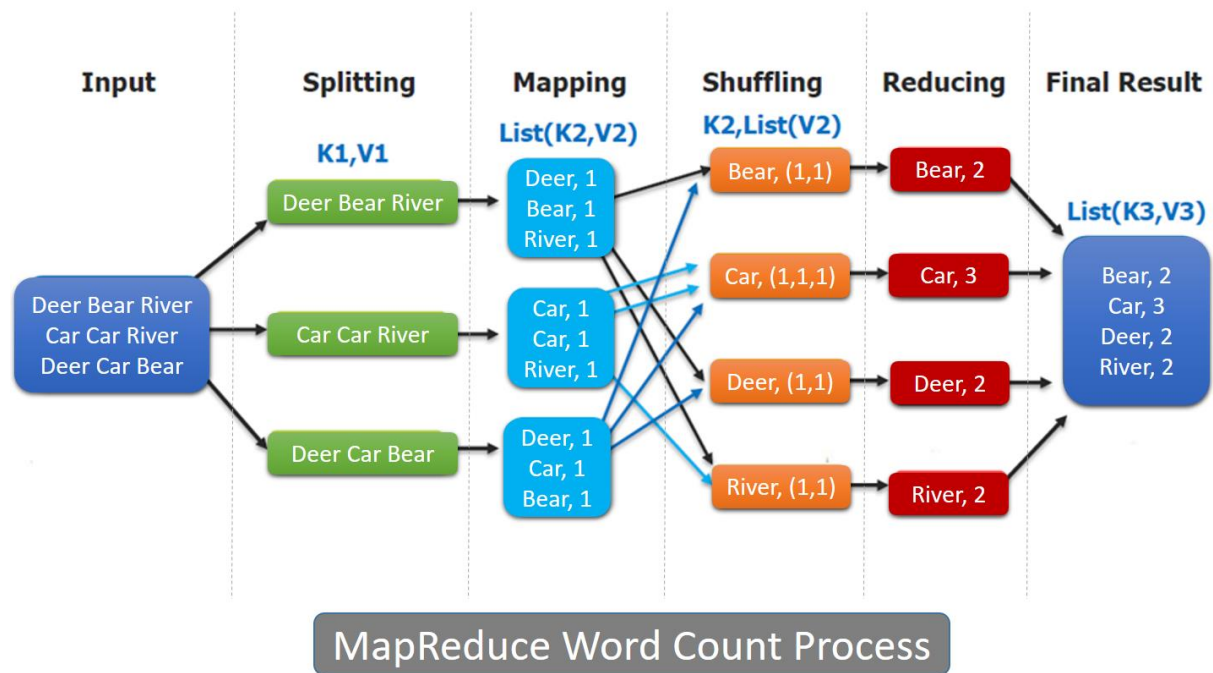
### ☐ Reducer

- ☐ The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them.
- ☐ Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing.
- ☐ Once the execution is over, it gives zero or more key-value pairs to the final step.

### ☐ Output Phase

- ☐ In the output phase, there is an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer.





### On what basis is a key-value pair generated in Hadoop?

- ▣ Generation of a key-value pair in Hadoop depends on the data set and the required output.
- ▣ In general, the key-value pair is specified in 4 places
  - φ Map input
    - ∞ Map-input by default will take the line offset as the key and the content of the line will be the value as Text.
    - ∞ It can also be customized.
  - φ Map output
    - ∞ The basic responsibility of map is to filter the data and provide the environment for grouping of data based on the key.
    - ∞ On what basis is a key-value pair generated in Hadoop?
    - ∞ Key
      - ▣ It will be the field/ text/ object on which the data has to be grouped and aggregated on the reducer side.
    - ∞ Value
      - ▣ It will be the field/ text/ object which is to be handled by each individual reduce method.
  - φ Reduce input
    - ∞ The output of Map is the input for reduce, so it is same as Map-Output.
  - φ Reduce output

- ∞ It depends on the required output.

### **Input Format**

- ▣ The MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.
- ▣ The key and value classes have to be serializable by the framework and hence need to implement the Writable interface.
- ▣ Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.
- ▣ Input and Output types of a MapReduce job:
  - φ (input) <k1, v1> -> map -> <k2, v2> -> combine -> <k2, v2> -> reduce -> <k3, v3> (output)
- ▣ The InputFormat class is one of the fundamental classes in the Hadoop MapReduce framework which provides the following functionality:
  - φ The files or other objects that should be used for input is selected by the InputFormat.
  - φ InputFormat defines the Data splits, which defines both the size of individual Map tasks and its potential execution server.
  - φ InputFormat defines the RecordReader, which is responsible for reading actual records from the input files.
- ▣ Types of Input format
  - φ FileInputFormat
  - φ TextInputFormat
  - φ KeyValueTextInputFormat
  - φ SequenceFileInputFormat
  - φ SequenceFileAsTextInputFormat
  - φ NLineInputFormat
  - φ SequenceFileAsBinaryInputFormat
  - φ DBInputFormat

### **FileInputFormat**

- ▣ It is the base class for all file-based InputFormats.
- ▣ Hadoop FileInputFormat specifies input directory where data files are located.
- ▣ When a Hadoop job is started, FileInputFormat is provided with a path containing files to read.
- ▣ FileInputFormat will read all files and divides these files into one or more InputSplits.

### **TextInputFormat**

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

### **KeyValueTextInputFormat**

- ▣ It is similar to TextInputFormat as it also treats each line of input as a separate record.
- ▣ While TextInputFormat treats entire line as the value, but the KeyValueTextInputFormat breaks the line itself into key and value by a tab character ('\t').
- ▣ Here Key is everything up to the tab character while the value is the remaining part of the line after tab character.

### **SequenceFileInputFormat**

- ▣ Hadoop SequenceFileInputFormat is an InputFormat which reads sequence files.
- ▣ Sequence files are binary files that stores sequences of binary key-value pairs.
- ▣ Sequence files block-compress and provide direct serialization and deserialization of several arbitrary data types (not just text).
- ▣ Here Key & Value both are user-defined.

### **SequenceFileAsTextInputFormat**

- ▣ Hadoop SequenceFileAsTextInputFormat is another form of SequenceFileInputFormat which converts the sequence file key values to Text objects.
- ▣ By calling 'toString()' conversion is performed on the keys and values.
- ▣ This InputFormat makes sequence files suitable input for streaming.

### **SequenceFileAsBinaryInputFormat**

- ▣ Hadoop SequenceFileAsBinaryInputFormat is a SequenceFileInputFormat using which the sequence file's keys and values can be extracted as an opaque binary object.

### **NLineInputFormat**

- ▣ Hadoop NLineInputFormat is another form of TextInputFormat where the keys are byte offset of the line and values are contents of the line.

- ▣ Each mapper receives a variable number of lines of input with TextInputFormat and KeyValueTextInputFormat and the number depends on the size of the split and the length of the lines.
- ▣ If the mapper has to receive a fixed number of lines of input, then NLineInputFormat is used.
- ▣ N is the number of lines of input that each mapper receives.
  - φ By default (N=1), each mapper receives exactly one line of input.
  - φ If N=2, then each split contains two lines.
  - φ One mapper will receive the first two Key-Value pairs and another mapper will receive the second two key-value pairs.

### **DBInputFormat**

- ▣ Hadoop DBInputFormat is an InputFormat that reads data from a relational database, using JDBC.
- ▣ It is best for loading relatively small datasets, perhaps for joining with large datasets from HDFS using MultipleInputs.
- ▣ Here Key is LongWritable while Value is DBWritable.

### **InputSplit**

- ▣ InputSplit in Hadoop MapReduce is the logical representation of data.
- ▣ It describes a unit of work that contains a single map task in a MapReduce program.
- ▣ Hadoop InputSplit represents the data which is processed by an individual Mapper.
- ▣ The split is divided into records.
  - φ Hence, the mapper processes each record (which is a key-value pair).
- ▣ MapReduce InputSplit length is measured in bytes and every InputSplit has storage locations (hostname strings).
- ▣ MapReduce system uses storage locations to place map tasks as close to split's data as possible.
- ▣ Map tasks are processed in the order of the size of the splits so that the largest one gets processed first (greedy approximation algorithm) and this is done to minimize the job runtime
- ▣ InputSplit in Hadoop is user defined.
- ▣ User can control split size according to the size of data in MapReduce program.
  - φ Thus, the number of map tasks is equal to the number of InputSplits.

### **RecordReader**

- ▣ The MapReduce RecordReader in Hadoop takes the byte-oriented view of input, provided by the InputSplit and presents as a record-oriented view for Mapper.

- ▣ It uses the data within the boundaries that were created by the InputSplit and creates Key-value pair.
- ▣ The RecordReader instance is defined by the InputFormat.
- ▣ By default, it uses TextInputFormat for converting data into a key-value pair.
- ▣ TextInputFormat provides two types of RecordReader
  - ⊕ LineRecordReader
    - ∞ Line RecordReader in Hadoop is the default RecordReader that TextInputFormat provides and it treats each line of the input file as the new value and associated key is byte offset.
    - ∞ LineRecordReader always skips the first line in the split (or part of it), if it is not the first split.
    - ∞ It reads one line after the boundary of the split in the end (if data is available, so it is not the last split).
  - ⊕ SequenceFileRecordReader
    - ∞ It reads data specified by the header of a sequence file.

### **Partitioner**

- ▣ Partitioning of the keys of the intermediate map output is controlled by the Partitioner.
- ▣ By the use of hash function, a key (or a subset of the key) is used to derive the partition.
- ▣ According to the key-value each mapper output is partitioned and records having the same key value go into the same partition (within each mapper), and then each partition is sent to a reducer. Partition class determines which partition a given (key, value) pair will go. Partition phase takes place after map phase and before reduce phase.
- ▣ Partitioning specifies that all the values for each key are grouped together and make sure that all the values of a single key go to the same reducer, thus allows even distribution of the map output over the reducer.
- ▣ Partitioner in Hadoop MapReduce redirects the mapper output to the reducer by determining which reducer is responsible for the particular key.
- ▣ The total number of Partitioners that run in Hadoop is equal to the number of reducers i.e. Partitioner will divide the data according to the number of reducers which is set by `JobConf.setNumReduceTasks()` method.
- ▣ Data from single partitioner is processed by a single reducer.
- ▣ Partitioner is created only when there are multiple reducers.

### **Combiner**

- ▣ The combiner in MapReduce is also known as 'Mini-reducer'.

- ▣ The primary job of Combiner is to process the output data from the Mapper, before passing it to Reducer.
- ▣ It runs after the mapper and before the Reducer and its use is optional.
- ▣ Hadoop Combiner reduces the time taken for data transfer between mapper and reducer.
- ▣ It decreases the amount of data that needed to be processed by the reducer.

### **Shuffling and Sorting**

- ▣ Shuffle phase in Hadoop transfers the map output from Mapper to a Reducer in MapReduce.
- ▣ Sort phase in MapReduce covers the merging and sorting of map outputs.
- ▣ Data from the mapper are grouped by the key, split among reducers and sorted by the key.
- ▣ Every reducer obtains all values associated with the same key.
- ▣ Shuffle and sort phase in Hadoop occur simultaneously and are done by the MapReduce framework.
- ▣ Shuffling
  - φ The process of transferring data from the mappers to reducers is known as shuffling i.e. the process by which the system performs the sort and transfers the map output to the reducer as input.
  - φ MapReduce shuffle phase is necessary for the reducers, otherwise, they would not have any input (or input from every mapper).
  - φ Since shuffling can start even before the map phase has finished, this saves some time and completes the tasks in lesser time.
- ▣ Sorting
  - φ The keys generated by the mapper are automatically sorted by MapReduce Framework.
  - φ Before starting of reducer, all intermediate key-value pairs in MapReduce that are generated by mapper get sorted by key and not by value.
  - φ Values passed to each reducer are not sorted; they can be in any order.
  - φ Sorting in Hadoop helps reducer to easily distinguish when a new reduce task should start.
  - φ This saves time for the reducer.
  - φ Reducer starts a new reduce task when the next key in the sorted input data is different than the previous.
  - φ Each reduce task takes key-value pairs as input and generates key-value pair as output.

### **Output Format**

- ▣ RecordWriter writes the output key-value pairs from the Reducer phase to output files.
- ▣ The way the output key-value pairs are written in output files by RecordWriter is determined by the Output Format.
- ▣ The Output Format and InputFormat functions are alike.
- ▣ OutputFormat instances provided by Hadoop are used to write to files on the HDFS or local disk.
- ▣ OutputFormat describes the output-specification for a Map-Reduce job.
- ▣ On the basis of output specification;
  - φ MapReduce job checks that the output directory does not already exist.
  - φ OutputFormat provides the RecordWriter implementation to be used to write the output files of the job.
- ▣ Output files are stored in a FileSystem.
- ▣ FileOutputFormat.setOutputPath() method is used to set the output directory.
- ▣ Every Reducer writes a separate file in a common output directory.
- ▣ The different Output Formats allowed are
  - φ TextOutputFormat
  - φ SequenceFileOutputFormat
  - φ SequenceFileAsBinaryOutputFormat
  - φ MapFileOutputFormat
  - φ MultipleOutputs
  - φ LazyOutputFormat
  - φ DBOutputFormat
- ▣ **TextOutputFormat**
  - φ MapReduce default Hadoop reducer Output Format is TextOutputFormat, which writes (key, value) pairs on individual lines of text files and its keys and values can be of any type since TextOutputFormat turns them to string by calling toString() on them.
  - φ Each key-value pair is separated by a tab character, which can be changed using MapReduce.output.textoutputformat.separator property.
  - φ KeyValueTextOutputFormat is used for reading these output text files since it breaks lines into key-value pairs based on a configurable separator.
- ▣ **SequenceFileOutputFormat**
  - φ It is an Output Format which writes sequences files for its output.
  - φ It is an intermediate format used between MapReduce jobs since these are compact and readily compressible.
  - φ Compression is controlled by the static methods on SequenceFileOutputFormat.
- ▣ **SequenceFileAsBinaryOutputFormat**

- ϕ It is another form of SequenceFileInputFormat which writes keys and values to sequence file in binary format.
- ▢ **MapFileOutputFormat**
  - ϕ It is another form of FileOutputFormat in Hadoop Output Format, which is used to write output as map files.
  - ϕ The key in a MapFile must be added in order.
  - ϕ Ensure that reducer emits keys in sorted order.
- ▢ **MultipleOutputs**
  - ϕ It allows writing data to files whose names are derived from the output keys and values, or in fact from an arbitrary string.
- ▢ **LazyOutputFormat**
  - ϕ Sometimes FileOutputFormat will create output files, even if they are empty.
  - ϕ LazyOutputFormat is a wrapper OutputFormat which ensures that the output file will be created only when the record is emitted for a given partition.
- ▢ **DBOutputFormat**
  - ϕ DBOutputFormat in Hadoop is an Output Format for writing to relational databases and HBase.
  - ϕ It sends the reduce output to a SQL table.
  - ϕ It accepts key-value pairs, where the key has a type extending DBWritable.
  - ϕ Returned RecordWriter writes only the key to the database with a batch SQL query.

### **Block vs. InputSplit**

- ▢ Block is a continuous location on the hard drive where data is stored.
- ▢ In general, FileSystem stores data as a collection of blocks.
- ▢ In the same way, HDFS stores each file as blocks.
- ▢ The Hadoop application is responsible for distributing the data block across multiple nodes.
- ▢ The data to be processed by an individual Mapper is represented by InputSplit.
- ▢ The split is divided into records and each record (which is a key-value pair) is processed by the map.
- ▢ The number of map tasks is equal to the number of InputSplits.
- ▢ Initially, the data for MapReduce task is stored in input files and input files typically reside in HDFS.
- ▢ InputFormat is used to define how these input files are split and read.
- ▢ InputFormat is responsible for creating InputSplit.

|   |
|---|
| <b>InputSplit vs Block Size in Hadoop</b> |
|---|



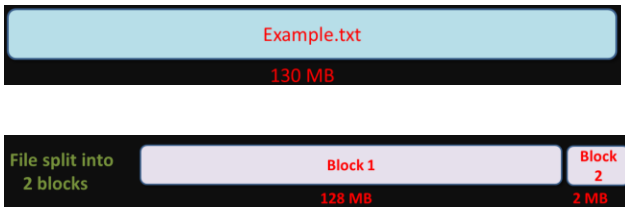
| Block   | InputSplit   |
|---|--|
| <ul style="list-style-type: none"> <li>▢ The default size of the HDFS block is 128 MB which is configured as per the requirements.</li> <li>▢ All blocks of the file are of the same size except the last block, which can be of same size or smaller.</li> <li>▢ The files are split into 128 MB blocks and then stored into Hadoop FileSystem.</li> </ul> | <ul style="list-style-type: none"> <li>▢ By default, split size is approximately equal to block size.</li> <li>▢ InputSplit is user defined and the user can control split size based on the size of data in MapReduce program.</li> </ul> |

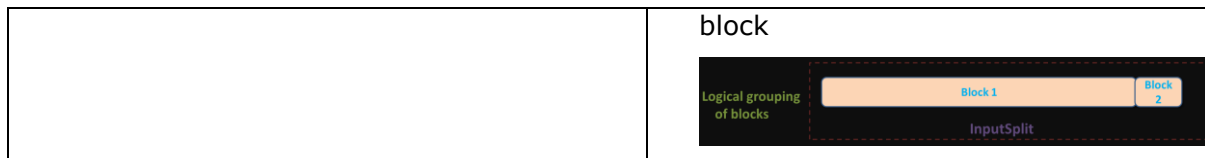
### Data Representation in Hadoop Blocks vs InputSplit

| Block   | InputSplit   |
|---|--|
| <ul style="list-style-type: none"> <li>▢ It is the physical representation of data.</li> <li>▢ It contains a minimum amount of data that can be read or write.</li> </ul> | <ul style="list-style-type: none"> <li>▢ It is the logical representation of data present in the block.</li> <li>▢ It is used during data processing in MapReduce program or other processing techniques.</li> <li>▢ InputSplit doesn't contain actual data, but a reference to the data.</li> </ul> |

- ▢ HDFS stores files as blocks.
- ▢ Block is the smallest unit of data that can be stored or retrieved from the disk and the default size of the block is 128MB.
- ▢ HDFS break files into blocks and stores these blocks on different nodes in the cluster.

### Example

| Block  | InputSplit  |
|--|---|
| <ul style="list-style-type: none"> <li>▢ Suppose the file is of 130 MB, so HDFS will break this file into 2 blocks.</li> </ul>  <p>The diagram illustrates a file named 'Example.txt' with a size of 130 MB. This file is split into two blocks: 'Block 1' which is 128 MB and 'Block 2' which is 2 MB. The text 'File split into 2 blocks' is shown next to the block representations.</p> | <ul style="list-style-type: none"> <li>▢ To perform a MapReduce operation on the blocks will not work because the 2<sup>nd</sup> block is incomplete.</li> <li>▢ This problem is solved by InputSplit.</li> <li>▢ InputSplit will form a logical grouping of blocks as a single block, because the InputSplit include a location for the next block and the byte offset of the data needed to complete the</li> </ul> |



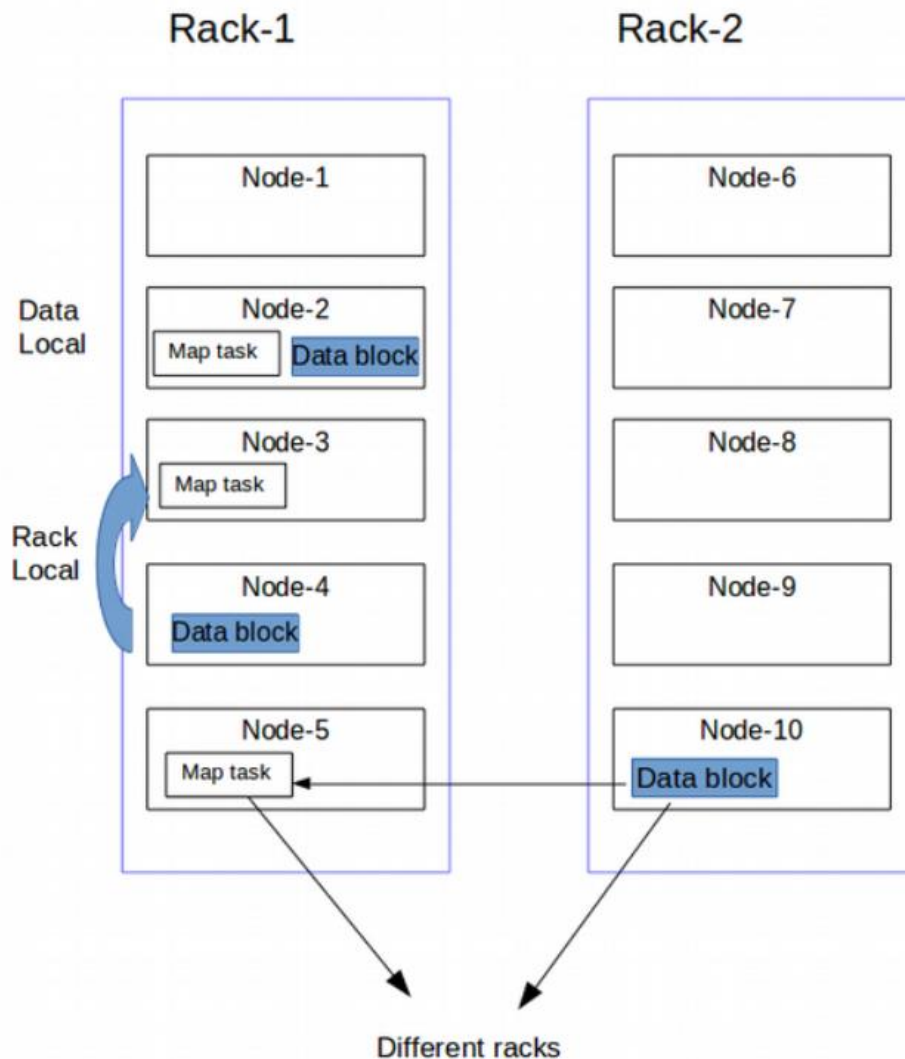
### Map - Only Job

- ▣ Map-Only job is the process in which mapper does all task, no task is done by the reducer and mapper's output is the final output.
- ▣ Setting `job.setNumreduceTasks(0)` in the configuration in a driver will remove a reducer from the picture.
- ▣ This will make a number of reducers as 0 and thus the only mapper will be doing the complete task
- ▣ In between map and reduces phases there is key, sort and shuffle phase.
- ▣ Sort and shuffle are responsible for sorting the keys in ascending order and then grouping values based on same keys.
- ▣ This phase is very expensive and if reduce phase is not required it should be avoided.
  - ϕ Avoiding reduce phase would eliminate sort and shuffle phase as well.
  - ϕ This also saves network congestion as in shuffling, an output of mapper travels to reducer and when data size is huge, large data needs to travel to the reducer.
- ▣ The output of mapper is written to local disk before sending to reducer but in map only job, this output is directly written to HDFS.
- ▣ This further save time and reduces cost as well.

### Categories of Data Locality in Hadoop

- ▣ When the data is located on the same node as the mapper working on the data it is known as data local data locality.
- ▣ In this case, the proximity of data is very near to computation.
  - ϕ Data local data locality in Hadoop
    - ∞ This is the most preferred scenario.
  - ϕ Intra-Rack data locality in Hadoop
    - ∞ It is not always possible to execute the mapper on the same datanode due to resource constraints.
    - ∞ In such case, it is preferred to run the mapper on the different node but on the same rack.
  - ϕ Inter-Rack data locality in Hadoop
    - ∞ Sometimes it is not possible to execute mapper on a different node in the same rack due to resource constraints.

- ∞ In such a case, the mapper will be executed on the nodes on different racks.
- ∞ This is the least preferred scenario.



### Advantages of Hadoop Data locality

- ▣ There are two benefits of data Locality in MapReduce.
  - ϕ Faster Execution
    - ∞ In data locality, the program is moved to the node where data resides instead of moving large data to the node, this makes Hadoop faster.
    - ∞ The size of the program is always lesser than the size of data, so moving data is a bottleneck of network transfer.
  - ϕ High Throughput
    - ∞ Data locality increases the overall throughput of the system.

### Speculative execution

- ▣ A key feature of Hadoop that improves job efficiency.

- ▣ In Hadoop, MapReduce breaks jobs into tasks and these tasks run parallel rather than sequential, thus reduces overall execution time.
- ▣ This model of execution is sensitive to slow tasks (even if they are few in numbers) as they slow down the overall execution of a job.
- ▣ There may be various reasons for the slowdown of tasks
  - ⊕ Hardware degradation
  - ⊕ Software misconfiguration
    - ∞ It may be difficult to detect causes since the tasks still complete successfully, although more time is taken than the expected time.
- ▣ Hadoop doesn't try to diagnose and fix slow running tasks, instead, it tries to detect them and runs backup tasks for them.
- ▣ This is called speculative execution in Hadoop.
- ▣ These backup tasks are called Speculative tasks in Hadoop.

### **When is speculative task started?**

- ▣ Once the map tasks or reduce tasks are started and monitored for some time Hadoop framework can determine which map task or reduce task is not making as much progress as the other running tasks of the same type.
- ▣ Only after this monitoring for some time and determining which tasks are slower Hadoop starts speculative execution of the tasks.
- ▣ Since the speculative task in MapReduce and the original task both are working on the same set of data, output of which ever task finishes first successfully is used and the other one is killed.

### **How to configure speculative execution in Hadoop**

- ▣ Speculative execution is enabled by default for both map and reduce tasks.
- ▣ Properties for speculative execution are set in mapred-site.xml file.
- ▣ `mapreduce.map.speculative`
  - ⊕ If set to true then speculative execution of map task is enabled.
  - ⊕ Default is true.
- ▣ `mapreduce.reduce.speculative`
  - ⊕ If set to true then speculative execution of reduce task is enabled.
  - ⊕ Default is true.
- ▣ `mapreduce.job.speculative.speculative-cap-running-tasks`
  - ⊕ The max percent (0-1) of running tasks that can be speculatively re-executed at any time.
  - ⊕ Default value is 0.1.
- ▣ `mapreduce.job.speculative.speculative-cap-total-tasks`
  - ⊕ The max percent (0-1) of all tasks that can be speculatively re-executed at any time.

- ☐ Default value is 0.01.

### **Consideration for turning off speculative execution**

- ▣ Since speculative execution of task means running duplicate tasks, it increases the cluster load.
- ▣ If there is a very busy cluster or a cluster with limited resources then it may be turning off the speculative execution.
- ▣ Another thing to consider is that reduce task gets its input from more than one map tasks running on different nodes so there is data transfer in case of reduce tasks.
- ▣ Running a duplicate reduce task means same data transfer happens more than once thus increasing load on network.

### **Counters**

- ▣ Hadoop Counters provides a way to measure the progress or the number of operations that occur within map/reduce job.
- ▣ Counters in Hadoop MapReduce are a useful channel for gathering statistics about the MapReduce job: for quality control or for application-level.
- ▣ They are also useful for problem diagnosis.
- ▣ Counters represent Hadoop global counters, defined either by the MapReduce framework or applications.
- ▣ Each Hadoop counter is named by an "Enum" and has a long for the value.
- ▣ Counters are bunched into groups, each comprising of counters from a particular Enum class.
- ▣ Hadoop Counters validate that:
  - ☐ The correct number of bytes was read and written.
  - ☐ The correct number of tasks was launched and successfully ran.
  - ☐ The amount of CPU and memory consumed is appropriate for the job and cluster nodes.
- ▣ Types of Hadoop MapReduce Counters
  - ☐ Built-In Counters in MapReduce
    - ∞ Hadoop maintains some built-in Hadoop counters for every job and these report various metrics, like, there are counters for the number of bytes and records, which allow us to confirm that the expected amount of input is consumed and the expected amount of output is produced.
    - ∞ Hadoop Counters are divided into groups and there are several groups for the built-in counters.
    - ∞ Each group either contains task counters (which are updated as task progress) or job counter (which are updated as a job progress).

- ∞ There are several groups for the Hadoop built-in Counters:
  - ⌘ MapReduce Task Counter in Hadoop
- ∞ Hadoop Task counter collects specific information like number of records read and written about tasks during its execution time. Hadoop Task counters are maintained by each task attempt and periodically sent to the application master so they can be globally aggregated.
  - ⌘ FileSystem Counters
    - δ Hadoop FileSystem\_Counters in Hadoop MapReduce gather information like a number of bytes read and written by the file system. Below are the name and description of the file system counters:
    - δ FileSystem bytes read– The number of bytes read by the filesystem by map and reduce tasks.
    - δ FileSystem bytes written– The number of bytes written to the filesystem by map and reduce tasks.
  - ⌘ FileInputFormat Counters in Hadoop
    - δ FileInputFormat Counters in Hadoop MapReduce gather information of a number of bytes read by map tasks via FileInputFormat.
  - ⌘ FileOutputFormat counters in MapReduce
    - δ FileOutputFormat counters in Hadoop MapReduce gathers information of a number of bytes written by map tasks (for map-only jobs) or reduce tasks via FileOutputFormat.
  - ⌘ MapReduce Job Counters
    - δ MapReduce Job counter measures the job-level statistics, not values that change while a task is running.
    - δ For example, TOTAL\_LAUNCHED\_MAPS, count the number of map tasks that were launched over the course of a job (including tasks that failed).
    - δ Application master maintains MapReduce Job counters, so these Hadoop Counters don't need to be sent across the network, unlike all other counters, including user-defined ones.
- ϕ User-Defined Counters/Custom counters in MapReduce

## Hadoop Streaming

- ▢ Hadoop streaming is a utility that comes with the Hadoop distribution.
- ▢ It is a generic API.
- ▢ This utility allows user to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer.
- ▢ It permits any program that uses standard input and output to be used for map tasks and reduce tasks.

- ▣ With this utility Map/Reduce jobs with any executable or script as the mapper and/or the reducer can be created and run.

### How Streaming Works

- ▣ Input is read from standard input and the output is emitted to standard output by Mapper and the Reducer.
- ▣ Utility creates a Map/Reduce job, submits the job to an appropriate cluster, and monitors the progress of the job until completion.
- ▣ Every mapper task will launch the script as a separate process when the mapper is initialized after a script is specified for mappers.
- ▣ Mapper task inputs are converted into lines and fed to the standard input and Line oriented outputs are collected from the standard output of the procedure Mapper and every line is changed into a key, value pair which is collected as the outcome of the mapper.
- ▣ Each reducer task will launch the script as a separate process and then the reducer is initialized after a script is specified for reducers.
- ▣ As the reducer task runs, reducer task input key/values pairs are converted into lines and fed to the standard input (STDIN) of the process.
- ▣ Each line of the line-oriented outputs is converted into a key/value pair after it is collected from the standard output (STDOUT) of the process, which is then collected as the output of the reducer.

### Commands

| Parameters                                     | Description   |
|--|---|
| -input directory/file-name                     | Input location for mapper. (Required)   |
| -output directory-name                         | Output location for reducer. (Required)   |
| -mapper executable or script or JavaClassName  | Mapper executable. (Required)   |
| -reducer executable or script or JavaClassName | Reducer executable. (Required)  |
| -file file-name                                | Create the mapper, reducer or combiner executable available locally on the compute nodes. |

|   |   |
|---|---|
| -inputformat JavaClassName                      | Class you offer should return key, value pairs of Text class.<br>If not specified TextInputFormat is used as the default.   |
| -outputformat JavaClassName                     | Class you offer should take key, value pairs of Text class.<br>If not specified TextOutputFormat is used as the default.  |
| -partitioner JavaClassName                      | Class that determines which reduce a key is sent to.  |
| -combiner streaming<br>Command or JavaClassName | Combiner executable for map output.   |
| -inputreader                                    | For backwards compatibility: specifies a record reader class instead of an input format class.  |
| -verbose  | Verbose output.   |
| -lazyOutput                                     | Creates output lazily. For example if the output format is based on FileOutputFormat, the output file is created only on the first call to output.collect or Context.write. |
| -numReduceTasks                                 | Specifies the number of reducers.   |
| -mapdebug                                       | Script to call when map task fails.   |
| -reducededbug                                   | Script to call when reduction makes the task failure  |
| -cmdenv name=value                              | Passes the environment variable to streaming commands.  |

### **JAVA MapReduce API vs. HADOOP Streaming**

- ▣ The difference is mainly in the processing implementation.
- ▣ In the standard Java API, the mechanism is to process each record, one at a time.
- ▣ So the framework is implemented to call the map () method (on your Mapper) for each record.
- ▣ With the Streaming API, the map program can control the processing of input.
- ▣ It can also read and process multiple lines at a time as it can control the reading mechanism.
- ▣ In Java, the same can be implemented but with the help of some other mechanism such as using instance variables to accumulate multiple lines and then process it.



## MapReduce in Python

### mapper.py

```
#!/usr/bin/env python3.4

import sys

#input from STDIN

for line in sys.stdin:

    #remove all trailing and leading whitespaces

    line = line.strip()

    #split line into words

    words = line.split()

    #loop for all the words in the words list

    for word in words:

        #output to STDOUT, the word and its count

        #we give count = 1 to every word here

        #this output is input to reducer

        #trivially tab delimited!

        print('{0}\t{1}'.format(word, 1))
```

### Expected Input and Output

Hello. How are you? I am fine. How are you?

Hello. 1

How 1

are 1

you? 1

I 1

am 1

fine. 1

How 1

are 1

you? 1

### **reducer.py**

```
#!/usr/bin/env python3.4
```

```
from operator import itemgetter
```

```
import sys
```

```
current_word = None
```

```
current_count = 0
```

```
word = None
```

```
#mappers output and reducers input
```

```
for line in sys.stdin:
```

```
    #remove all the leading and trailing whitespace
```

```
    line = line.strip()
```

```
    #split the input to get the word and its count
```

```
    #as given by mapper
```

```
    word, count = line.split('\t', 1)
```

```
    #typecast string count to integer
```

```
    try:
```

```
        count = int(count)
```

```
except ValueError:

    #discard if it wasn't really a number

    continue

#Hadoop MapReduce Framework has sorted the output

#sum up total count of the word

if current_word == word:

    current_count += count

else:

    if current_word:

        #output to STDOUT

        print('{0}\t{1}'.format(current_word, current_count))

        current_count = count

        current_word = word

#IMPORTANT Its the last word which too needs to be printed

if current_word == word:

    print('{0}\t{1}'.format(current_word, current_count))
```

### **Expected Input and Output**

Hello 1

How 1

**Hello 1**

How 1

are 1

**How 2**

are 1

you? 1

**are 2**

you? 1

**you? 1**

I 1

**you? 1**

am 1

**I 1**

fine. 1

**am 1**

**fine 1**

### To run it in Hadoop

1. In command prompt type start-all.cmd
2. Create a directory in local HDFS
3. Create a file called Sample.txt in the above directory.
4. Type the command

```
hadoop jar /C:/BigData/hadoop-2.9.1/share/hadoop/tools/sources/hadoop-  
*streaming*.jar -file /test/py_mapper.py -mapper /test/py_mapper.py -file  
/test/py_reducer.py -reducer /test/py_reducer.py -input /test/Sample.txt -output  
/test/wcount-out
```