CS 523 – BDT
Big Data Technology

Lesson 3

MapReduce Paradigm

Spontaneous Fulfillment of Desires

Maharishi International University

WHOLENESS OF THE LESSON

MapReduce is a simple programming paradigm that seamlessly facilitates the analysis of big data stored in HDFS with its parallel computation model.

Science & Technology of Consciousness: TM is a simple, effortless mental technique that can be used by anyone, no matter what their lifestyle. It promotes spontaneous fulfillment of desires, by bringing the desires of the individual into accord with Natural Law, without the individual having to know the underlying mechanism.

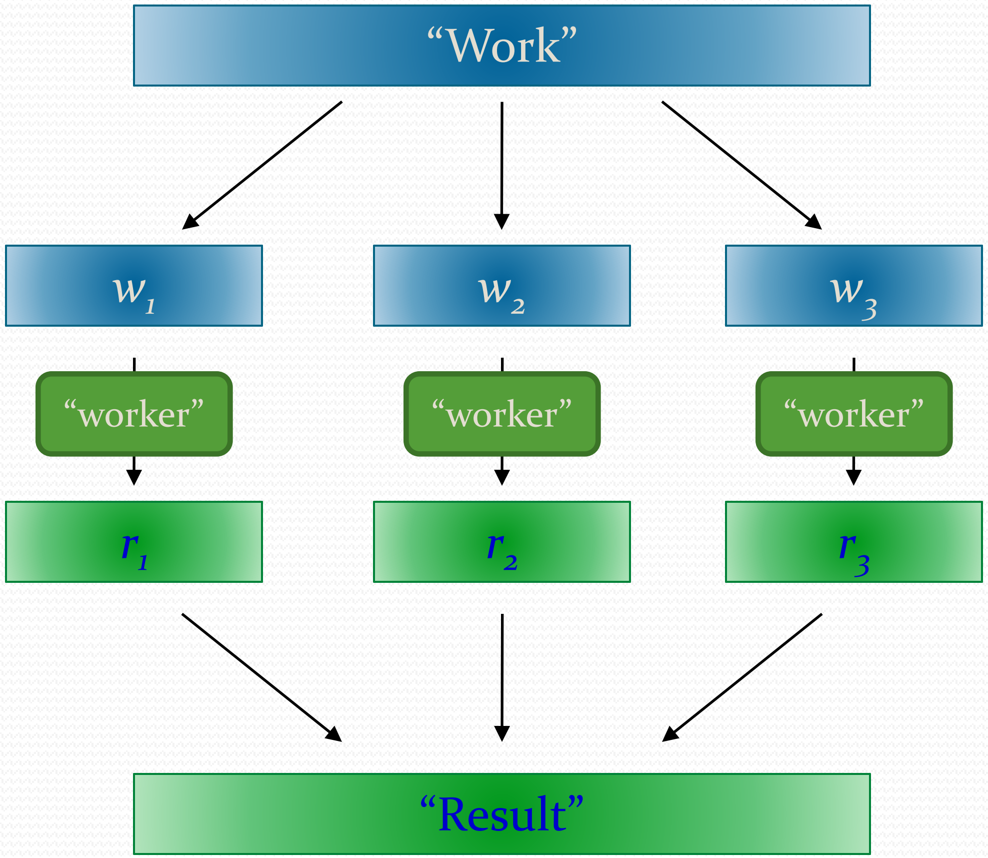
Components of Distributed Systems

- Distributed Storage
- Distributed Processing
  - Resource Management
  - Distributed Computation
- Network

- Hadoop 1.0
  - HDFS – Distributed Storage
  - MR – Distributed Processing
- Hadoop 2.0
  - HDFS – Distributed Storage
  - Distributed Processing
    - YARN – Resource Management
    - MR – Distributed Computation

How To Solve Large Data Problems?

- **Divide and Conquer** is the only feasible approach today to tackle large-data problems.



Complexities in Divide and Conquer Approach

- How to break up a large problem into smaller tasks? More specifically, how to decompose the problem so that the smaller tasks can be executed in parallel?
- How to assign tasks to workers distributed across a potentially large number of machines (while keeping in mind that some workers are better suited to run some tasks than others, e.g., due to available resources, locality constraints, etc.)?
- How do we ensure that the workers get the data they need?
- How do we coordinate synchronization among the different workers?
- How do we share partial results from one worker that is needed by another?
- How do we accomplish all of the above in the face of software errors and hardware faults?

MapReduce is the Answer

- MapReduce is a massively scalable, parallel processing framework for big data.
- MapReduce abstraction shields the programmer from having to explicitly worry about system-level issues such as synchronization, IPC and machine failures.
- The MR runtime automatically parallelizes the computation across large-scale clusters of machines. This frees the programmer to focus on solving the problem at hand.
- MapReduce assumes an architecture where processors and storage (disk) are co-located and it is flexible enough to work with unstructured data.
- MapReduce has its roots in functional programming (eg. ML, LISP)

MapReduce Inspiration

The name MapReduce comes from Functional programming.

- **map** is the name of the higher order function that applies a given function to each element of a list.

  - E.g.

```
val numbers = List(2, 7, 9)
numbers.map(x => x * x) == List (4, 49, 81)
```

  - $\{2, 7, 9\} \xrightarrow{\texttt{map(f)}} \{4, 49, 81\}$

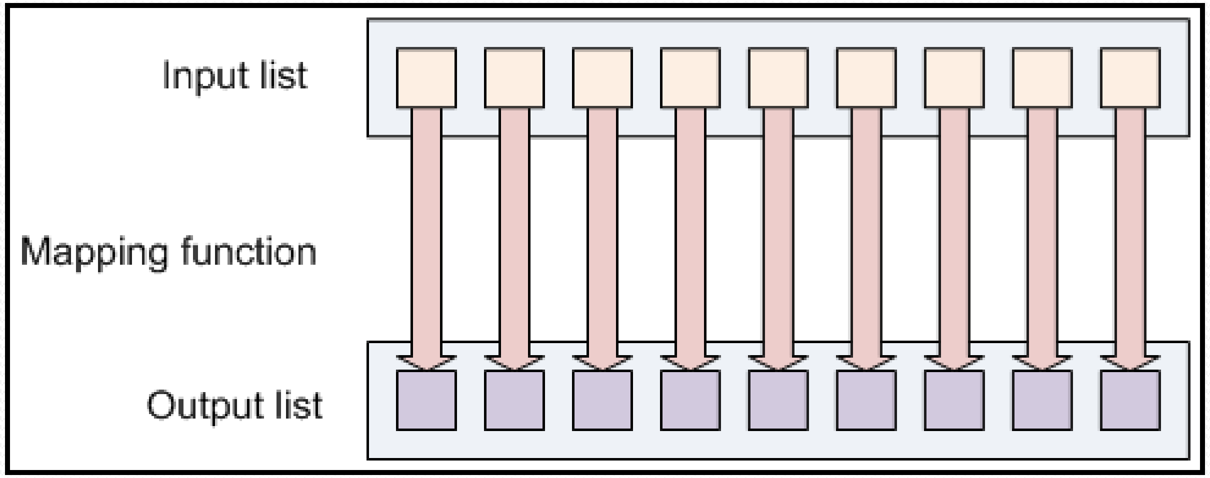
- **reduce** is the name of a higher order function that analyzes a recursive data structure and aggregates its constituent parts by using a given aggregation operation.

E.g.

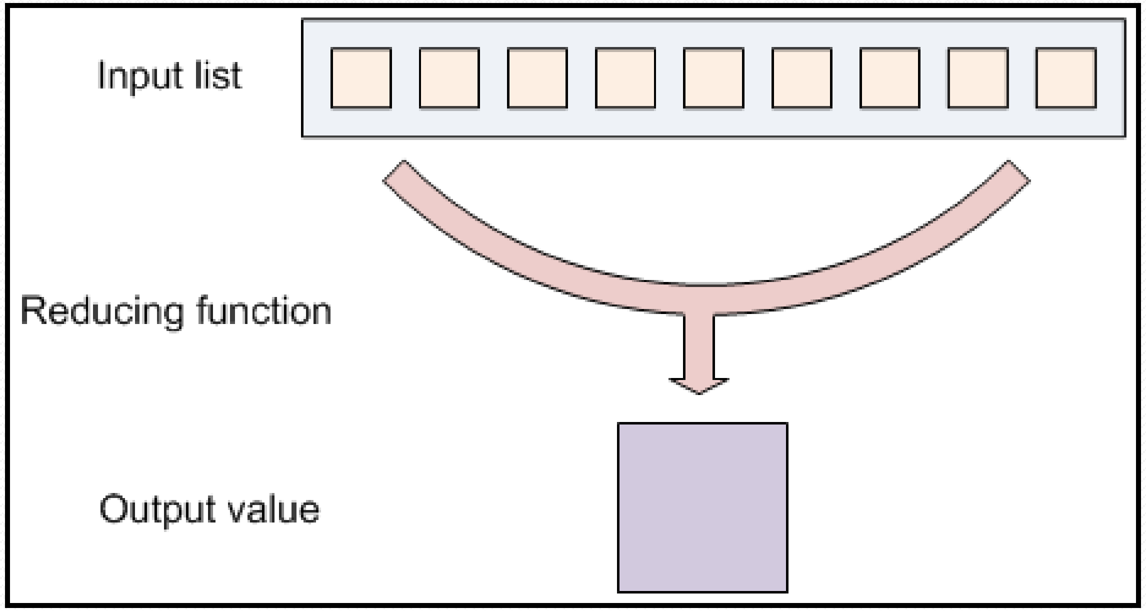
```
val numbers = List(4, 49, 81)
numbers.reduce(_ + _) == 134
```

$\{4, 49, 81\} \xrightarrow{\texttt{reduce(g, 0)}} 134$

Mapping creates a new output list by applying a function to individual elements of an input list



Reducing a list iterates over the input values to produce an aggregate value as output.



Observations

- We can view ***map*** as a concise way to represent the **transformation** of a dataset.
- We can view ***reduce*** as an **aggregation** operation.
- The application of *map* to each item in a list (or more generally, to elements in a large dataset) can be parallelized in a straightforward manner, since each functional application happens in isolation.
  - In a cluster, these operations can be distributed across many different machines.
- The *reduce* operation, on the other hand, has more restrictions on data locality. Elements in the list must be brought together before the function can be applied.

Hadoop MapReduce

- In Hadoop MapReduce, we need to express our queries as a MapReduce job.
- A MapReduce job consists of two main phases: *Map*, and then *Reduce*
- Each phase has key-value pairs as i/p and o/p, the types of which may be chosen by the programmer.
- Programmer defines a mapper and a reducer with the following signatures:

map : (k1; v1) ⇒ list (k2; v2)

reduce : (k2; list (v2)) ⇒ list (k3; v3)

Mappers and Reducers

- Key-value pairs form the basic data structure in MapReduce.
- Keys and values may be primitives such as integers, floats, strings, and raw bytes, or they can be any complex structures (lists, tuples, maps, etc.).
- No value stands on its own. Every value has a key associated with it.
- Each Map task operates on a specific portion of the overall dataset.
- After all Maps are complete, the MapReduce system distributes the intermediate data to nodes which perform the Reduce phase.

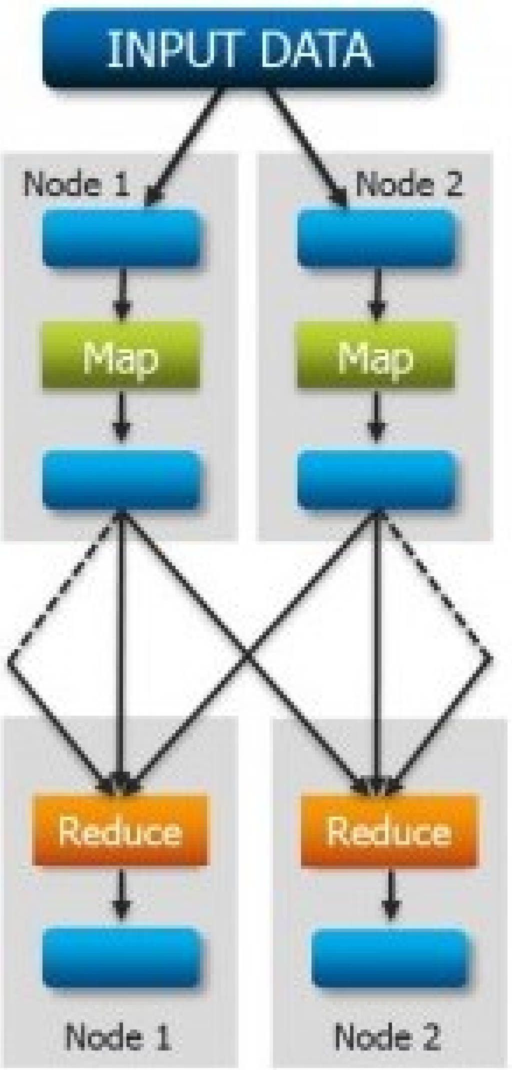
Mappers and Reducers

- Map function is just a data preparation phase, setting up the data in such a way that the reduce function can do it’s work on it.
- The input to a MapReduce job starts with the data stored on HDFS. The mapper is applied to every input key-value pair to generate an arbitrary number of intermediate key-value pairs.
- The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs.
- Implicit between the map and reduce phases is a distributed “group by" operation on intermediate keys.

Phases of MapReduce Job

- MapReduce program is executed in six phases.

Input phase
Split phase
Map
Shuffle/Sort
Reduce
Output phase



Phases of MapReduce

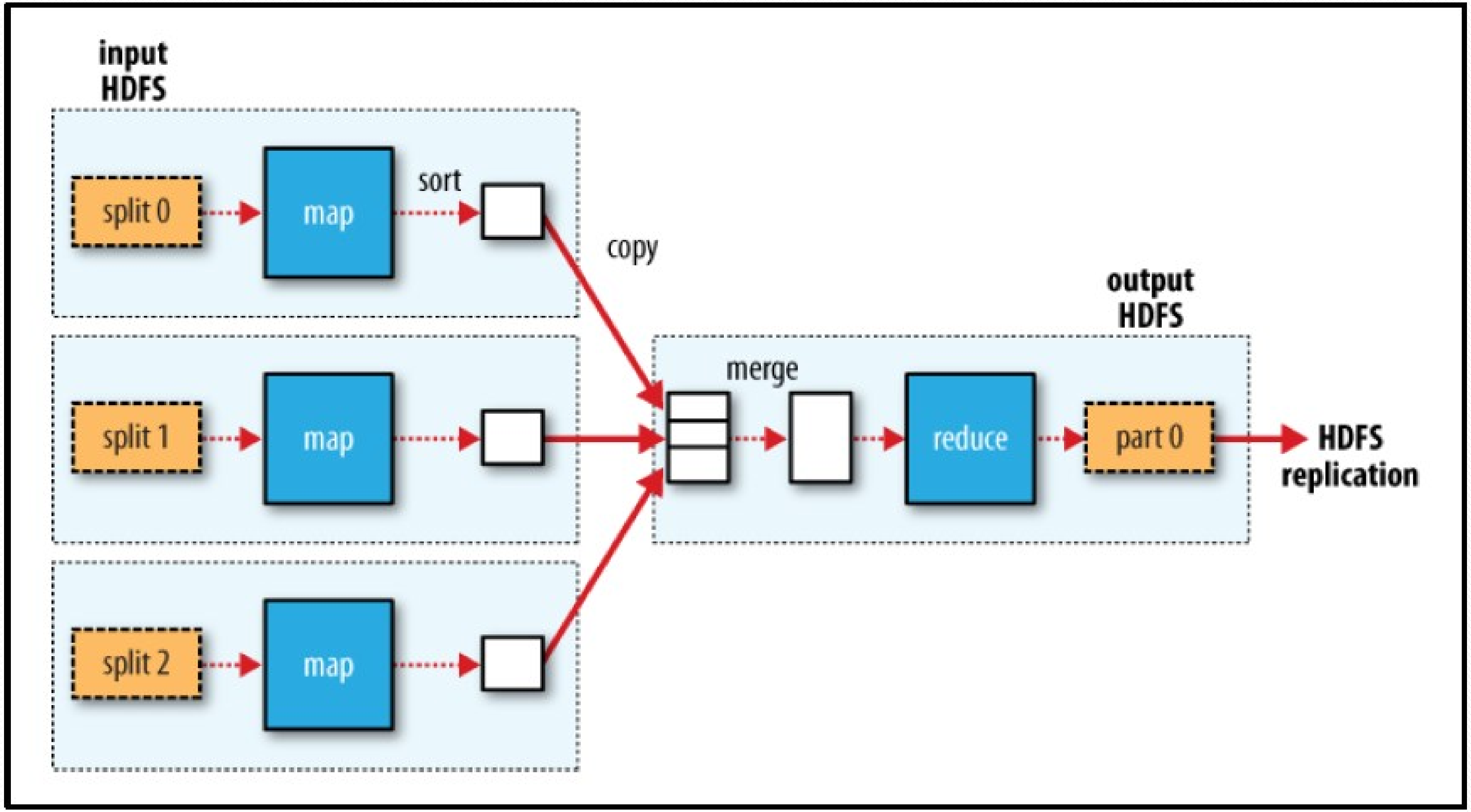
- **Input & Split Phase :** Hadoop divides the input file stored on HDFS into splits (typically of the size of an HDFS block) and assigns every split to a different mapper, trying to assign every split to the mapper where the split physically resides.
- **Mapper Phase :**
  - Locally, Hadoop reads the split of the mapper line by line.
  - Hadoop calls the method map() of the mapper for every line (record) passing it as the key/value parameters.
  - The mapper computes its application logic and emits other key/value pairs.

Phases of MapReduce contd..

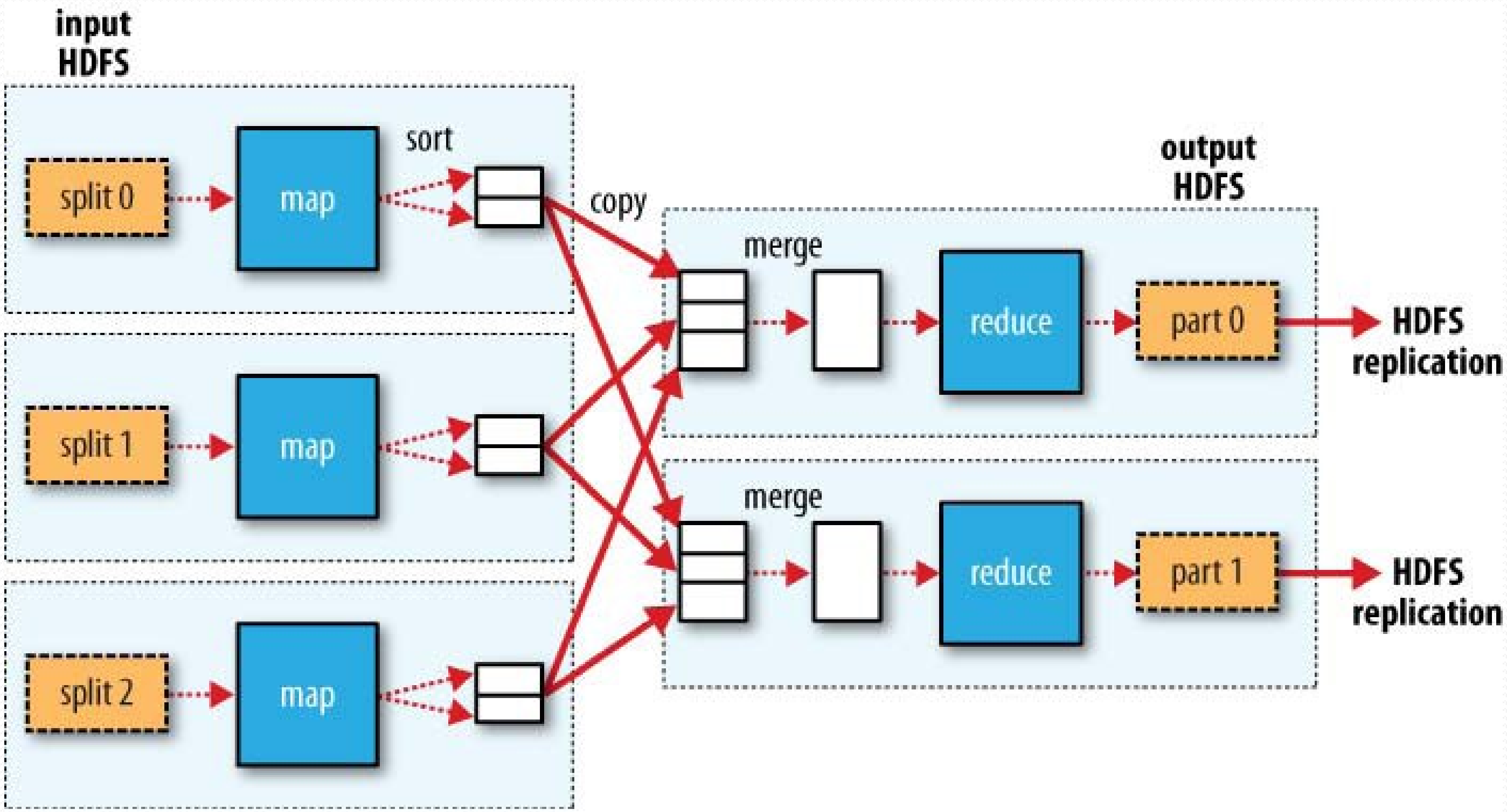
- **Shuffle & Sort Phase :**
  - Map output is then sorted by the framework. Sorting happens only on the key and not on the value.
  - This sorted and grouped (key, value) pairs are then shuffled to the machine where reduce task is going to run.
  - "Group-by" operation happens based on key followed by another sorting on key.

- **Reducer and Output Phase :**
  - Reads the sorted input which is stored locally on Reducer machine and calls the reduce() method for every line of the input (for every (key and list of value) pairs).
  - The reducer computes it's application logic and emits other key/value pairs which is written to HDFS as output.

MapReduce Data Flow with a Single Reducer



MapReduce Data Flow with Multiple Reducers

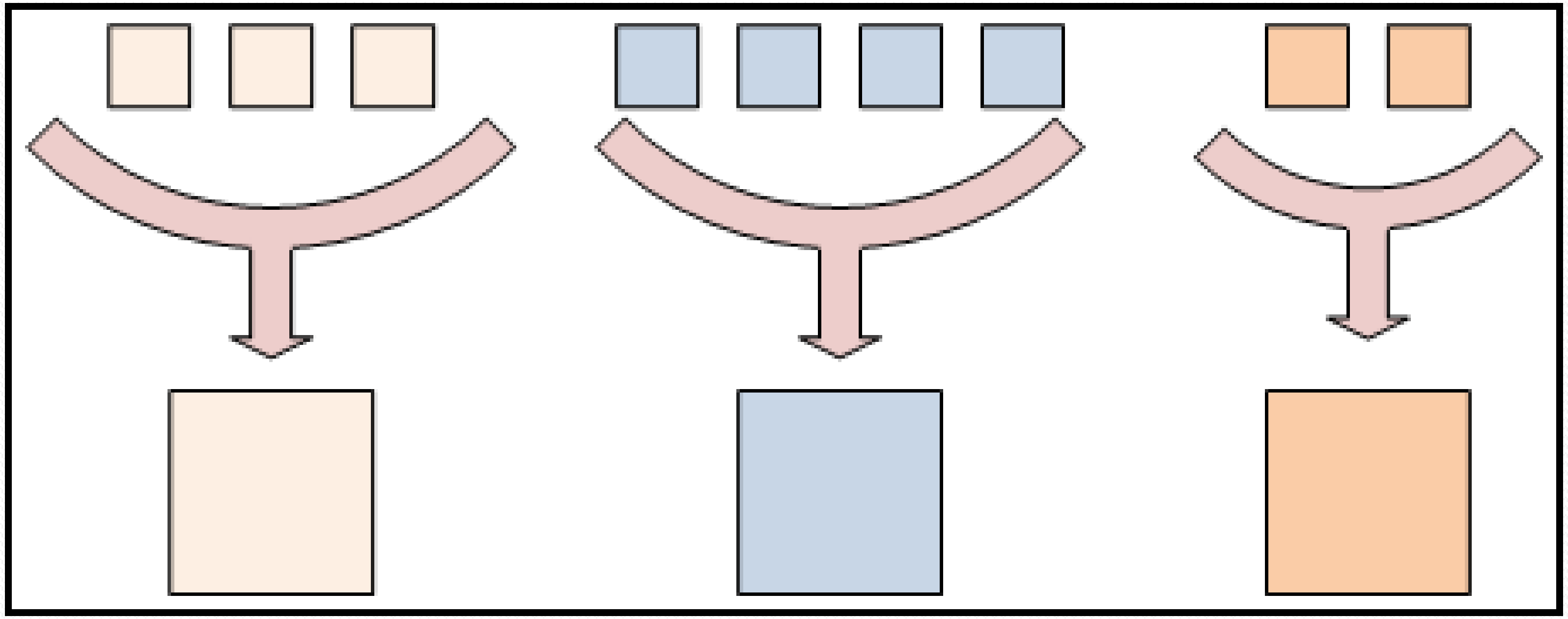


Reducer I/p and O/p

- Intermediate data arrive at each reducer in order, **sorted by the key.**
- **However, no ordering relationship is guaranteed for keys across different reducers.**
- Output key-value pairs from each reducer are written persistently back onto HDFS (whereas intermediate key-value pairs are transient and not preserved).
- The output ends up in r files on the distributed file system, where r is the number of reducers.
- For the most part, there is no need to consolidate reducer output, since the r files often serve as input to yet another MapReduce job.

Using Multiple Reducers

Different colors represent different keys. All values with the same key are presented to a single reduce task.



Mappers and Reducers

- Mappers and Reducers are objects that implement the Map and Reduce methods, respectively.
- In Hadoop, a mapper object is initialized for each map task (associated with a particular sequence of key-value pairs called as input split) and the Map method is called on each key-value pair by the execution framework.
  - In a MapReduce job, the programmer provides a hint on the number of map tasks to run, but the execution framework makes the final determination based on the physical layout of the data.
- A reducer object is initialized for each reduce task, and the Reduce method is called once per intermediate key.
  - In contrast with the number of map tasks, the programmer can precisely specify the number of reduce tasks.

Word Count Example

- WordCount example reads text files and counts how often words occur.
- The input is text file and the output is text file, each line of which contains a word and the count of how often it occurred, separated by a tab.
- Each mapper takes a line as input and breaks it into words. It then emits a key/value pair of the form *<word , 1>*.
- Each reducer sums the counts for each word and emits a single key/value of the form *<word, sum>*.
- This example serves as a first step in building a unigram language model (i.e., probability distribution over words in a collection).
  - Sample application: analyze web server logs to find popular URLs

Word Count Example

For example, if we had a file:

foo.txt: This is the foo file
This is the test file

We would expect the output to be:

This	2
is	2
the	2
foo	1
test	1
file	2

Word Count Algorithm

```
Map (long lineOffset, String line):
     for each word w in line:
          Emit(w, 1)

Reduce (String word, Iterator<Int> values):
     int sum = 0
     for each v in values:
          sum += v
     Emit(word, sum)
```

Word Count Algorithm

- In the map function, the keys are the line offsets within the file, which we ignore in our map function.
- The mapper takes an input key-value pair, tokenizes the value (finds words), and emits an intermediate key-value pair for every word: the word itself serves as the key, and the *integer one* serves as the value (denoting that we've seen the word once).
- The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Therefore, in our word count algorithm, we simply need to sum up all counts (ones) associated with each word.
- The reducer does exactly this and emits final key-value pairs with the word as the key, and the count as the value.
- Final output is written to HDFS, one file per reducer.
- Words within each output file will be sorted by alphabetical order, and each file will contain roughly the same number of words.

MapReduce Job

- A complete MapReduce job consists of code for the mapper, reducer, along with job configuration parameters.
- The execution framework handles everything else.
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults detects worker failures and restarts

- You don’t know:
  - Where mappers and reducers run
  - When a particular mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

Java MapReduce API

- Let’s write the Word Count algorithm in Java.
- We need three things: a *map* function, a *reduce* function, and some code to run the job.
- Rather than using built-in Java types, Hadoop provides its own set of basic types that are optimized for network serialization. These are found in the *org.apache.hadoop.io* package.
  - In Word Count, we use *LongWritable*, which corresponds to a Java Long, *Text* (like Java String), and *IntWritable* (like Java Integer).

Java MapReduce API link

Hadoop Writables

- In MapReduce, 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.
- A *WritableComparable* is a *Writable*, which is also *Comparable*. Two *WritableComparables* can be compared against each other to determine their 'order'.
- Keys must be *WritableComparable* because they are passed to the Reducer in sorted order.
- Note that despite their names, all Hadoop box classes implement both *Writable* and *WritableComparable*, for example, *Intwritable* is actually a *WritableComparable.*

Writable Interface (org.apache.hadoop.io)

- This is the interface in Hadoop which provides methods for serialization and deserialization.

S. No.	Methods and Description
1	**void readFields(DataInput in)** This method is used to deserialize the fields of the given object.
2	**void write(DataOutput out)** This method is used to serialize the fields of the given object.

WritableComparable Interface

- This interface inherits `Writable` interface of Hadoop as well as `Comparable` interface of Java.
- Therefore, it provides methods for data serialization, deserialization, and comparison.

S. No.	Methods and Description
1	**int compareTo(class obj)** This method compares current object with the given object obj.

- In addition to these classes, Hadoop supports number of wrapper classes that implement `WritableComparable` interface.
  - Each class wraps a Java primitive type.

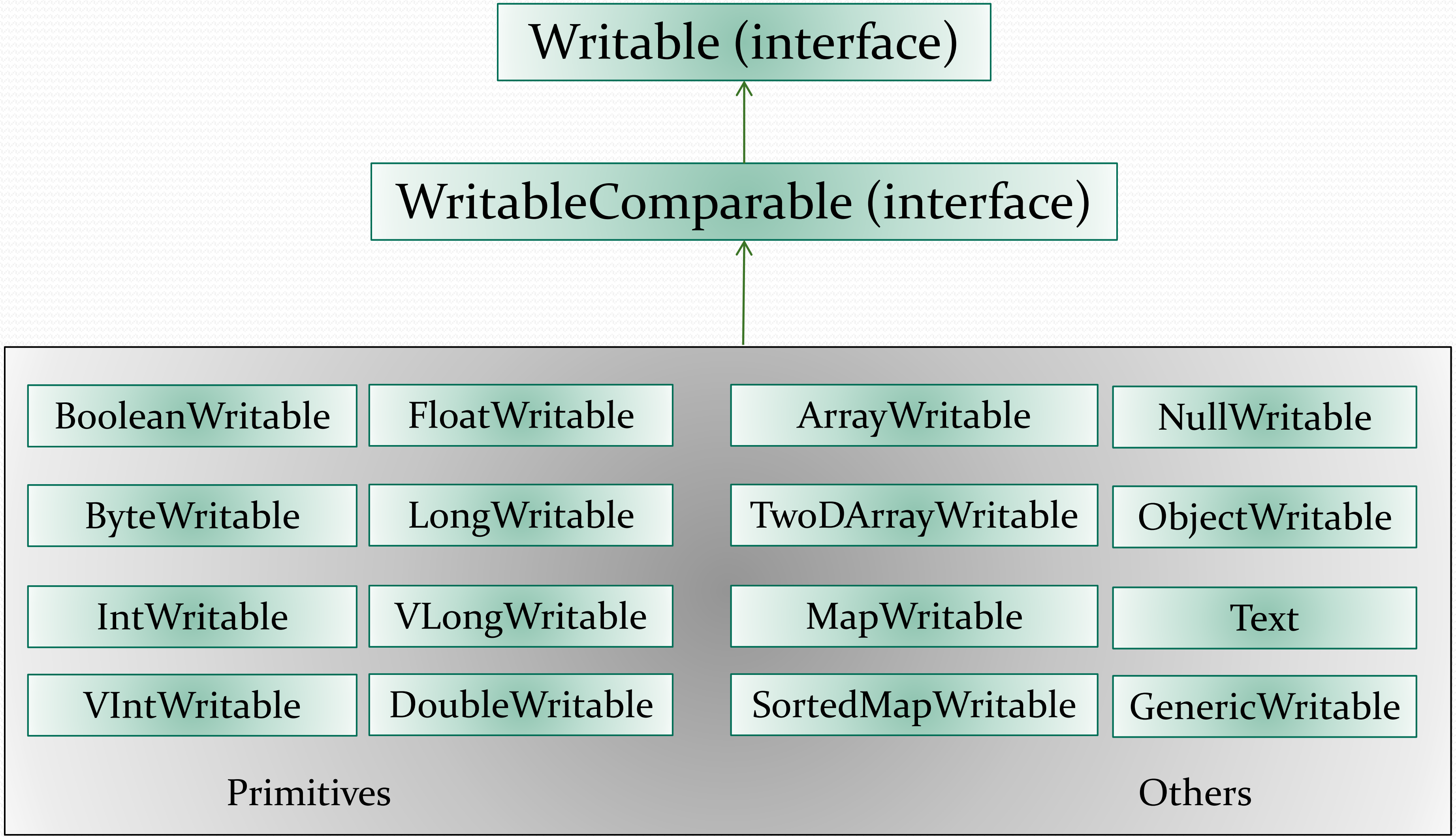
Writable Wrappers

Java primitive	Writable implementation
boolean	BooleanWritable
byte	ByteWritable
short	ShortWritable
int	IntWritable VIntWritable
float	FloatWritable
long	LongWritable VLongWritable
double	DoubleWritable

Java class	Writable implementation
String	Text
byte[]	BytesWritable
Object	ObjectWritable
null	NullWritable

Java collection	Writable implementation
array	ArrayWritable ArrayPrimitiveWritable TwoDArrayWritable
Map	MapWritable
SortedMap	SortedMapWritable
enum	EnumSetWritable

Class Hierarchy of Hadoop Serialization



NullWritables

- Use NullWritable to avoid unnecessary serialization overhead.
- If your MapReduce job does not require both the key and the value to be emitted, using NullWritable will save the framework the trouble of having to serialize unnecessary objects out to the disk.
- In many scenarios, it is often cleaner and more readable than using blank placeholder values or static singleton instances for output.

Requirements of Applications using MapReduce

- Specify the Job configuration
  - Specify input/output locations
  - Supply map and reduce functions via implementations of appropriate interfaces and/or abstract classes
- Job client then submits the job (jar/executables etc.) and the configuration to the ResourceManager.

Word Count (Map)

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

    @Override
    public void map(LongWritable key, Text value, Context context)
                                        throws IOException, InterruptedException
    {
        for (String token: value.toString().split("\\s+"))  {
                word.set(token);
                context.write(word, one);    }
    }
}
```

Word Count Mapper

- The map function is represented by the inbuilt **Mapper** class, which declares an abstract `map()` method. So our mapper implementation will extend **Mapper** and override `map()`.
- The Mapper class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function.
- Main Mapper engine: Mapper.run()
  - `setup()`
  - `map()` for each input record
  - `cleanup()`

Word Count Mapper

- Map function gets a key, value, and context
  - key – “byte offset of the line”
  - value - the current line
- In the while loop, each token is a "word" from the current line
- It emits a key-value pair of *<word, 1>*, written to the Context object.
- Note that the class-variable word is reused each time the mapper outputs another <word, 1> pairing; this saves time by not allocating a new variable for each output.

Context Object Details

- Recall Mapper code:

```
for (String token:value.toString().split("\\s+")) {
        word.set(token);
        context.write(word, one);    }
```

- Context object allows the Mapper to interact with the rest of the Hadoop system.
- Includes configuration data for the job as well as interfaces which allow it to emit output.
- Applications can use the Context
  - to report progress
  - to set application-level status messages
  - indicate that they are alive

Word Count (Reduce)

```
public static class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context    context)
                                                    throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

Word Count Reducer

- The reduce function is represented by the inbuilt `Reducer` class, which declares an abstract `reduce()` method. So our reducer implementation will extend `Reducer` and override `reduce()`.
- The Reducer class is a generic type, with four formal type parameters that specify the i/p and o/p types.
- **The input types of the reduce function must match the output types of the map function.**
- Reduce engine: `Reducer.run()`
  - `setup()`
  - `reduce()` per key associated with reduce task
  - `cleanup()`

Word Count Reducer

- `Reducer.reduce()`
  - Called once per key
  - Passed in an Iterable which returns all values associated with that key
  - Emits output with `Context.write()`
- Number of reducers for the job set by user via **`job.setNumReduceTasks(int)`**
- Reduce engine
  - receives a Context containing job’s configuration as well as interfacing methods that return data back to the framework

Word Count (Driver)

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    FileInputFormat.addInputPath(job, new Path("input"));
    FileOutputFormat.setOutputPath(job, new Path("output"));
    job.setMapperClass(WordCountMapper.class);
    job.setReducerClass(WordCountReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Driver Code Details

- A **`Job`** object forms the specification of the job and gives you control over how the job is run.
- When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster).
- Rather than explicitly specifying the name of the JAR file, we can pass a class in the Job's **`setJarByClass()`** method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.
- Having constructed a Job object, we specify the input and output paths. An input path is specified by calling the static **`addInputPath()`** method on **`FileInputFormat`**, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern. As the name suggests, `addInputPath()` can be called more than once to use input from multiple paths.

Driver Code Details

- The output path (of which there is only one) is specified by the static **`setOutputPath()`** method on **`FileOutputFormat`**. It specifies a directory where the output files from the reduce function are written.
- The directory shouldn't exist before running the job because Hadoop will complain and not run the job. This precaution is to prevent data loss (it can be very annoying to accidentally overwrite the output of a long job with that of another).
- Specify the map and reduce types to use with **`setMapperClass()`** and **`setReducerClass()`** methods.
- The **`setOutputKeyClass()`** and **`setOutputValueClass()`** methods control the output types for the reduce function and must match what the Reduce class produces.
- The map o/p types default to the same types, so they do not need to be set if the mapper produces the same types as the reducer.
- However, if they are different, the map output types must be set using **`setMapOutputKeyClass()`** and **`setMapOutputValueClass()`**.

Driver Code Details

- The input types are controlled via the input format, which we have not explicitly set because we are using the default **`TextInputFormat`**.
- After setting the classes that define the map and reduce functions, we are ready to run the job.
- The **`waitForCompletion()`** method on `Job` submits the job and waits for it to finish.
  - The single argument to the method is a flag indicating whether verbose output is generated. When true, the job writes information about its progress to the console.
  - The return value of the `waitForCompletion()` method is a Boolean indicating success (true) or failure (false), which we translate into the program's exit code of 0 or 1.

Input Format

- How the input files are split up and read is defined by the **`InputFormat`**. It is a class that provides the following functionality:
  - Selects the files or other objects that should be used for input
  - Defines the **`InputSplits`** that break a file into tasks
  - Provides a factory for **`RecordReader`** objects that read the file
- Several InputFormats are provided with Hadoop.
- An abstract type is called **`FileInputFormat`**; all InputFormats that operate on files inherit functionality and properties from this class.
- When starting a Hadoop job, **`FileInputFormat`** is provided with a path containing files to read. It will read all files in this directory.
- It then divides these files into one or more **`InputSplits`** each. You can choose which **`InputFormat`** to apply to your input files for a job by calling the *`setInputFormat()`* method of the Job object that defines the job.

Input Formats

InputFormat:	Description:	Key:	Value:
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueTextInputFormat	Used for plain text files where the files are broken into lines. Parses lines into key, val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format used for reading files in sequence	user-defined	user-defined

`InputFormat` is responsible for creating the input splits and dividing them into records.

Hadoop Running Modes

Hadoop can run in three different modes-

- **Local/Standalone Mode**
  - This is the single process mode of Hadoop, which is the default mode, wherein no daemons are running.
  - This mode is useful for testing and debugging.
- **Pseudo-Distributed Mode**
  - This mode is a simulation of fully distributed mode but on single machine. This means that all the daemons of Hadoop will run as separate processes on a single machine.
  - This mode is useful for development.
- **Fully Distributed Mode**
  - This mode requires two or more systems as cluster.
  - Name Node, Data Node and all the processes run on different machines in the cluster.
  - This mode is useful for the production environment.

Special cases of MapReduce Job

Case 1. MapReduce programs can contain no reducers, in which case mapper output is directly written to disk (one file per mapper). Example problems: parse a large text collection or independently analyze a large number of images.

Case 2. MapReduce program with no mappers is not possible, although in some cases it is useful for the mapper to implement the identity function and simply pass input key-value pairs to the reducers. This has the effect of sorting and regrouping the input for reduce-side processing.

Case 3. Similarly, in some cases it is useful for the reducer to implement the identity function, in which case the program simply sorts and groups mapper output.

Case 4. Running identity mappers and reducers has the effect of regrouping and resorting the input data (which is sometimes useful).