## What is the difference between TextInputFormat and KeyValueInputFormat class?

Answer: The TextInputFormat class converts every row of the source file into key/value types where the BytesWritable key represents the offset of the record and the Text value represents the entire record itself.

The KeyValueTextInputFormat is an extended version of TextInputFormat , which is useful when we have to fetch every source record as Text/Text pair where the key/value were populated from the record by splitting the record with a fixed delimiter.

Consider the below file contents,

AL#Alabama

AR#Arkansas

FL#Florida

If TextInputFormat is configured , you might see the key/value pairs as,

0 AL#Alabama

14 AR#Arkansas

23 FL#Florida

if KeyvalueTextInputFormat is configured with conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", "#") , you might see the results as,

AL Alabama

AR Arkansas

FL Florida

## How is the splitting of file invoked in Hadoop framework?

Answer: The InputFormat is responsible to provide the splits and take care of the split of the file.

**FileInputFormat** is an abstract class which defines how the input files are read and spilt up. FileInputFormat provides following functionalities:

1. Select files/objects that should be used as input,

2. Defines input splits that breaks a file into task.

The file at a given map node is split, based on the value we set for *InputFormat*. It is done in java using [setInputFormat()](https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/JobConf.html#setInputFormat(java.lang.Class)).

For an Example: conf.setInputFormat(TextInputFormat.class); Here, by passing TextInputFormat to the setInputFormat function, we are telling hadoop to treat each *line* of the input file at the map node as the input to the map function. Line feed or carriage-return is used to signal end of line.

For example: Keys are the position in the file, and values are the lines of text.

FileInputFormat.addInputPath(job, new Path(args[0]));

conf.setInputFormat(TextInputFormat.class);

**FileInputFormat** class and function **addInputPath**, **setInputFormat** take care of inputsplit, also this code defines the number of mappers get created.

All InputFormat classes are subclass of FileInputFormat, which take care of the split.

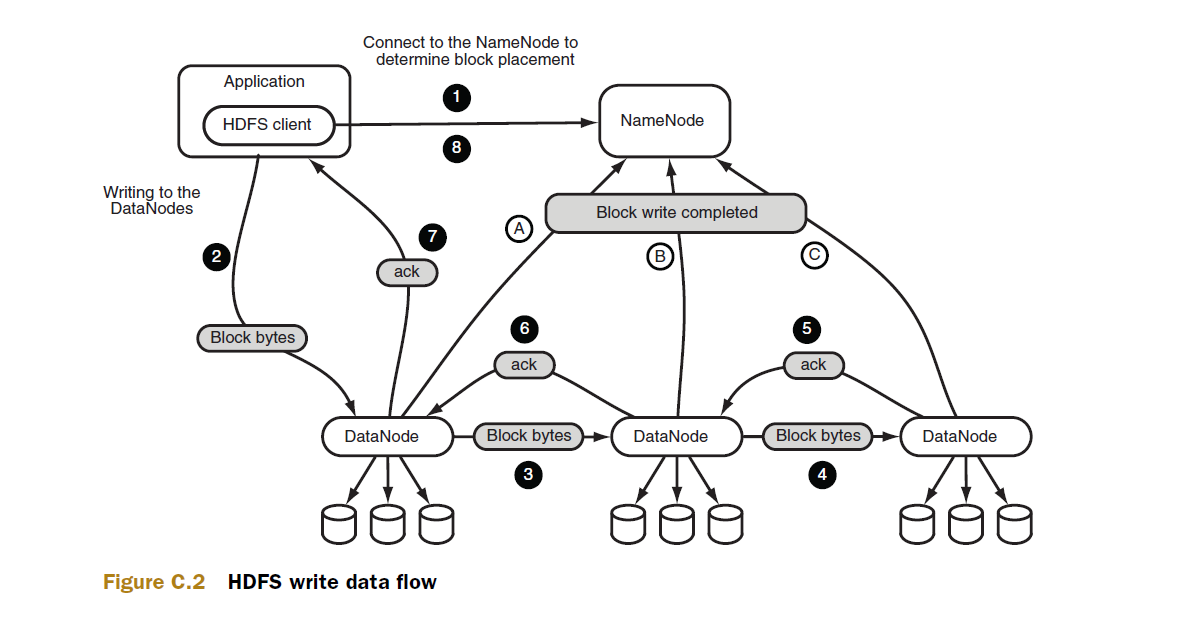
Specifically, getSplit function of FileInputFormat generate a list of InputSplit, from the list of files defined in JobContext. The split is based on the size of bytes, whose Min and Max could be defined arbitrarily in project xml file. We use FileInputFormat for large files and CombineFileInputFormat for smaller ones. We can also check whether the input can be split into blocks by issplittable method. When a Hadoop job runs, it splits input files into chunks and assigns each split to a mapper to process; this is called InputSplit Each block is then fed to a data node where a mapreduce job runs for further analysis. The size of a block would depend on the size that you have mentioned in **mapred.max.split.size** parameter.

There are two distinct concepts at work here; each concept is handled by a different entity in the hadoop framework

Firstly --

1. **Dividing a file into blocks** -- The way HDFS has been set up, it breaks down very large files into large blocks (for example, measuring 128MB), and stores three copies of these blocks on different nodes in the cluster. HDFS has no awareness of the content of these files.

When a file is written into HDFS, HDFS split files into blocks based on the defined block size and takes care of its replication. This is done once (mostly), and then is available to all MR jobs running on the cluster. This is a cluster wide configuration



The second point in the diagram is probably where the split happens; note that this has nothing to do with running of a MR Job

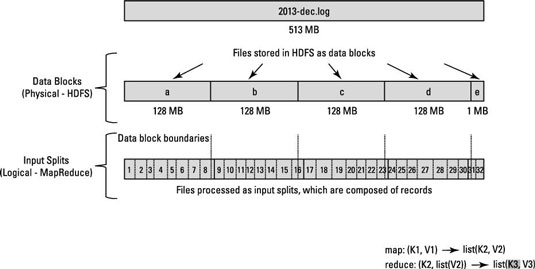
Secondly --

2) **Splitting a file into input splits** -- In Hadoop, files are composed of individual records, which are ultimately processed one-by-one by mapper tasks. One line represents one record. If each map task processes all records in a specific data block, what happens to those records that span block boundaries? File blocks are exactly 128MB (or whatever you set the block size to be), and because HDFS has no conception of what’s inside the file blocks, it can’t gauge when a record might spill over into another block.

To solve this problem, Hadoop uses a logical representation of the data stored in file blocks, known as input splits. When a MapReduce job client calculates the input splits, it figures out where the first whole record in a block begins and where the last record in the block ends.

In cases where the last record in a block is incomplete, the input split includes location information for the next block and the byte offset of the data needed to complete the record.

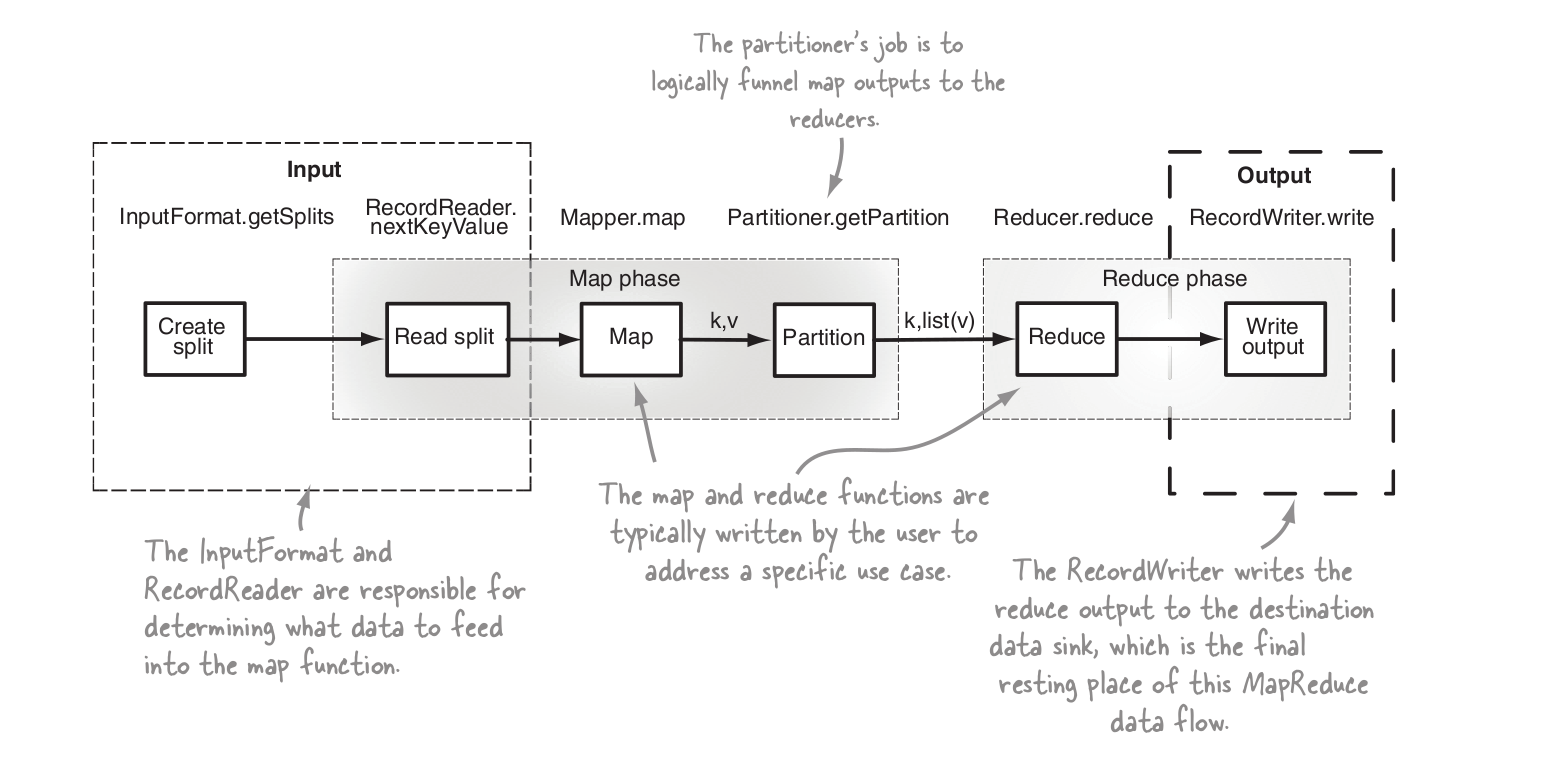
The figure shows this relationship between data blocks and input splits.



MapReduce data processing is driven by this concept of input splits. The number of input splits that are calculated for a specific application determines the number of mapper tasks. Each of these mapper tasks is assigned, where possible, to a slave node where the input split is stored. The Resource Manager (or JobTracker, if you’re in Hadoop 1) does its best to ensure that input splits are processed locally. Each of these mapper tasks is assigned, where possible, to a slave node where the input split is stored. The Resource Manager (or JobTracker, if you’re in Hadoop 1) does its best to ensure that input splits are processed locally.

When input path is passed into a MR job, the MR job uses the path along with the input format configured to divide the files specified in the input path into splits, each split is processed by a map task. Calculation of input splits is done by the input format each time a job is executed. Each input format has its own logic how files can be split into part for the independent processing by different mappers. Compression, usually is a foe of the splitting, so we employ block compression technique to enable splitting of the compressed data. It means that each logical part of the file (block) is compressed independently. If isSplitable returns false - file will be processed by one mapper, regardless of the number of blocks.

The second point in the diagram is probably where the split happens; note that this has nothing to do with running of a MR Job Now have a look at the execution steps of a MR job



Here the first step is the calculation of the input splits via the inputformat configured for the job.

## Consider case scenario: In M/R system, - HDFS block size is 64 MB

## - Input format is FileInputFormat – We have 3 files of size 64K, 65Mb and 127Mb

## How many input splits will be made by Hadoop framework for each file?

Answer: For file of size 64K– 1 input split

For file of size 65 MB – depends, it could be 1 split or could be 2 splits

For file of size 127 MB – 2 input slits

## After the Map phase finishes, the Hadoop framework performs “Partitioning, Shuffle and sort”. Explain each event in brief.

Answer: **Partitioning:**

The mechanism of sending specific key-value pairs to specific reducers is called partitioning. There are two purposes for it. First, partitioning has a direct impact on the overall performance of job we want to run. Second, it maybe sometimes required to control the key/value pairs (emitted from mapper) partitioning over the reducers. For example lets sat the job.setNumReduceTasks(2); two partitions would be created but if size of the partition in too small, chances are there that only one output file would be created . So to delibrately tell the compiler to create two output files for each partition this line must be used. Partition the map outputs based on the key and it keeps the record of same key into the same partitions.

A partitioner works like a condition in processing an input dataset. The partition phase takes place after the Map phase and before the Reduce phase. The default partitioner is HashPartitioner, which hashes a record’s key to determine which partition (and thus which reducer) the record belongs in. The number of partition is then equal to the number of reduce tasks for the job. That means a partitioner will divide the data according to the number of reducers. Therefore, the data passed from a single partitioner is processed by a single Reducer. A partitioner partitions the key-value pairs of intermediate Map-outputs. It partitions the data using a user-defined condition, which works like a hash function.

Hadoop decides it at the time when the map reduce job starts that how may partitions will be there which is controlled by the JobConf.setNumReduceTasks()) method, suppose if decide 20 reduce tasks, the 20 partitions will be there and must be filled.

By default the partitioner implementation is called HashPartitioner. It uses the hashCode() method of the key objects modulo the number of partitions total to determine which partition to send a given (key, value) pair to.

Partitioner provides the getPartition() method that you can implement yourself if you want to declare the custom partition for your job.

The getPartition() method receives a key and a value and the number of partitions to split the data, a number in the range [0, numPartitions) must be returned by this method, indicating which partition to send the key and value to. For any two keys k1 and k2, k1.equals(k2) implies getPartition(k1, \*, n) == getPartition(k2, \*, n).

public int getPartition(K key, V value, int numReduceTasks) {

return (key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks;

}

Poor partitioning:

Suppose you know that one of the key in your data input will appear more than any other key so you may want to send all your key (large number) to one partition and then distribute the other keys over all other partition by their hasCode(). So now if you have two mechanism of sending data to partitions

First,the key appearing more will be send to one partition

Second, all other keys will be send to partitions according to their hashCode().

Now suppose if your hashCode() method does not uniformly distribute other keys data over partitions range. So the data is not evenly distributed in partitions as well as reducers.Since each partition is equivalent to a reducer.So here some reducers will have more data than other reducers.So other reducers will wait for one reducer(one with user defined keys) due to the work load it shares.

So here we should take an approach that its work load may be shared across many different reducers.

Shuffling

After the first map tasks have completed, the nodes may still be performing several more map tasks each. But they also begin exchanging the intermediate outputs from the map tasks to where they are required by the reducers. This process of moving map outputs to the reducers is known as shuffling. Shuffling can start even before the map phase has finished, to save some time. That's why you can see a reduce status greater than 0% (but less than 33%) when the map status is not yet 100%.

Sort: The set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer. MapReduce makes the guarantee that the input to every reducer is sorted by key. Sorting saves time for the reducer, helping it easily distinguish when a new reduce task should start. It simply starts a new reduce task, when the next key in the sorted input data is different than the previous, to put it simply. Each reduce task takes a list of key-value pairs, but it has to call the reduce() method which takes a key-list(value) input, so it has to group values by key. It's easy to do so, if input data is pre-sorted (locally) in the map phase and simply merge-sorted in the reduce phase (since the reducers get data from many mappers).

## What is a Combiner?

A Combiner is also known as a semi-reducer, is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class. The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input. The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Combiner reduces network congestion and optimized network bandwidth to a great extent. It also helps in improving the data processing speed.

**Combiner should be used only when the function is both commutative and associative.**

## What is Hadoop streaming?

Hadoop Streaming is a generic API which allows that allows programs written in virtually any language to be used as Hadoop Mapper and Reducer implementations. Hadoop streaming is a utility that comes with the Hadoop distribution. This utility allows you to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. Hadoop Streaming allows you to use arbitrary programs for the Mapper and Reducer phases of a MapReduce job. Input and output are always represented textually in Streaming. Streaming programs write their output to stdout in the same format: key \t value \n. MapReduce’s streaming feature allows programmers to use languages other than Java such as any Linux program or tool can be used or scripts in bash, python, perl, or another language to write MapReduce programs, provided that the necessary interpreter is present on all nodes in your cluster.

## What are the most commonly defined input formats in Hadoop and explain each in brief.

Answer: **CombineFileInputFormat**: CombineFileInputFormat packs many files into each split so that each mapper has more to process.

**CompositeInputFormat**: To run a map-side join. The input source and join type (inner or outer) for CompositeInputFormat are configured through a join expression that is written according to a simple grammar.

**DBInputFormat**: to read from databases

**FileInputFormat**: Base class for all file-based InputFormats

**FixedLengthInputFormat**:FixedLengthInputFormat is for reading fixed-width binary records from a file, when the records are not separated by delimiters. The record size must be set via fixed lengthinputformat.record.length.

**KeyValueTextInputFormat**: An InputFormat for plain text files. Files are broken into lines. Either line feed or carriage-return are used to signal end of line. Each line is divided into key and value parts by a separator byte. If no such a byte exists, the key will be the entire line and value will be empty. First value before delimiter is key and rest is value

**NLineInputFormat**: NLineInputFormat which splits N lines of input as one split. In many "pleasantly" parallel applications, each process/mapper processes the same input file (s), but with computations are controlled by different parameters. N number of lines is considered one value/record.

**SequenceFileAsBinaryOutputFormat**: SequenceFileAsBinaryOutputFormat—the counterpart to SequenceFileAsBinaryIn putFormat—writes keys and values in raw binary format into a sequence file container.

**SequenceFileAsBinaryInputFormat**: SequenceFileAsBinaryInputFormat is a variant of SequenceFileInputFormat that retrieves the sequence file’s keys and values as opaque binary objects. They are encapsulated as BytesWritable objects, and the application is free to interpret the underlying

byte array as it pleases. In combination with a process that creates sequence files with

SequenceFile.Writer’s appendRaw() method or SequenceFileAsBinaryOutputFormat, this provides a way to use any binary data type with MapReduce (packaged as a sequence file), although plugging into Hadoop’s serialization mechanism is normally a cleaner alternative

**SequenceFileAsTextInputFormat**: SequenceFileAsTextInputFormat is a variant of SequenceFileInputFormat that converts the sequence file’s keys and values to Text objects. The conversion is performed by calling toString() on the keys and values. This format makes sequence files suitable input for Streaming.

**SequenceFileInputFormat**: An InputFormat for SequenceFiles. For binary files.

**TextInputFormat**: An InputFormat for plain text files. Files are broken into lines. Either linefeed or carriage-return are used to signal end of line. Keys are the position in the file, and values are the line of text. Each line will be treated as value

**MapFileOutputFormat**: MapFileOutputFormat writes map files as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order.

## Explain what is distributed Cache in MapReduce Framework ?

DistributedCache is a facility provided by the Map-Reduce framework to cache files needed by applications. Applications specify the files, via urls (hdfs:// or http://) to be cached via the JobConf. The DistributedCache assumes that the files specified via urls are already present on the FileSystem at the path specified by the url and are accessible by every machine in the cluster. The framework will copy the necessary files on to the slave node before any tasks for the job are executed on that node. Once you cache a file for your job, hadoop framework will make it available on each and every data nodes (in file system, not in memory) where you map/reduce tasks are running. Then you can access the cache file as local file in your Mapper Or Reducer job. Now you can easily read the cache file and populate some collection (e.g Array, Hashmap etc.) in your code.

## Explain what happens in textinputformat ?

In textinputformat, Files are broken into lines. Each line in the text file is a record.

1. Key is the byte offset of the line, and
2. Value is the content of the line.

For instance, Key: longWritable, value: text.

## 10. Explain what is Sequencefileinputformat?

A Hadoop specific high performance binary format. Sequencefileinputformat is used for reading files in sequence. It is a specific compressed binary file format which is optimized for passing data between the output of one MapReduce job to the input of some other MapReduce job. The SequenceFileInputFormat reads special binary files that are specific to Hadoop. These files include many features designed to allow data to be rapidly read into Hadoop mappers. Sequence files are block-compressed and provide direct serialization and deserialization of several arbitrary data types (not just text). Sequence files can be generated as the output of other MapReduce tasks and are an efficient intermediate representation for data that is passing from one MapReduce job to another.