1: 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

2: Splitting of file is invoked by using the method getInputSplit() method of input format class like FileInputFormat defined by user.

3: Since HDFS block size is 64 MB and we have 3 files of size 64K, 65Mb and 127Mb

In this case there will be 5 input splits in order to read the whole file.

One input split for 64K file,

2 input split for 65 MB file where 64 MB will be one split and 1 MB will be other,

2 input splits for 127 MB files where 1 for 64 MB and 63 MB for other.

4: **Partitioning**

Partitioning is the process of determining which reducer instance will receive which intermediate keys and values.

Each mapper must determine for all of its output (key, value) pairs which reducer will receive them. It is necessary that for any key, regardless of which mapper instance generated it, the destination partition is the same

**Shuffle**

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.

**Sort**

Each reduce task is responsible for reducing the values associated with several intermediate keys. The set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer

5: The Combiner is a "mini-reduce" process which operates only on data generated by a mapper. The Combiner will receive as input all data emitted by the Mapper instances on a given node. The output from the Combiner is then sent to the Reducers, instead of the output from the Mappers.

6: Hadoop Streaming is a generic API which allows writing Mappers and Reduces in any language. But the basic concept remains the same. Mappers and Reducers receive their input and output on stdin and stdout as (key, value) pairs.

Apache Hadoop uses streams as per UNIX standard between your application and Hadoop system. Streaming is the best fit for text processing. The data view is line oriented and processed as a key/value pair separated by 'tab' character. The program reads each line and processes it as per the requirement.

In any MapReduce job, we have input and output as key/value pairs. The same concept is true for Streaming API. In Streaming, input and output are always represented as text. The 'tab' character is used to separate key and value. The Streaming program uses the 'tab' character to split a line into key/value pair. The same procedure is followed for output. The Streaming program writes its output on stdout following the same format as mentioned below.

key1 \t value1 \n

key2 \t value2 \n

key3 \t value3 \n

In this process, each line contains only one key/value pair. So the input to the reducer is sorted so that all the same keys are placed adjacent to one another. Any program or tool can be used as Mapper and Reducer if it is capable of handling input in text format as described above.

Other scripts like Perl, Python or Bash can also be used for this purpose, provided all the nodes have an interpreter to understand the language.

7: These are the most common input formats defined in Hadoop:

TextInputFormat: It reads lines of text files and provides the offset of the line as key to the Mapper and actual line as Value to the mapper

KeyValueInputFormat: Reads text file and parses lines into key, val pairs. Everything up to the first tab character is sent as key to the Mapper and the remainder of the line is sent as value to the mapper.

SequenceFileInputFormat: In addition to text format, hadoop also have support for binary files. One of binary file format is sequence files that stores serialized key/value pairs. Basically three types of sequence files are Uncompressed format, Record compressed format and block compressed format.

8: Distrbuted cache is a facility which MapReduce framework provides to access small files [kilobytes or few megabytes in size] ,mainly used as Meta files, needed by application during its execution. File can be text,archive,jar and so on. MapReduce copies all the cache files in the local file system of all the slave nodes before any task for the job starts on that node. It saves lots of task and I/O operations e.g sometimes it is necessary for every Mapper to read a single file.

9: TextInputFormat is the default InputFormat . Each record is a line of input. The key, a LongWritable , is the byte offset within the file of the beginning of the line. The value is the contents of the line, excluding any line terminators (e.g., newline or carriage return), and is packaged as a Text object. So a file containing the following text:

On the top of the Crumpetty Tree

The Quangle Wangle sat,

But his face you could not see,

On account of his Beaver Hat.

is divided into one split of four records. The records are interpreted as the following key-value pairs:

(0, On the top of the Crumpetty Tree)

(33, The Quangle Wangle sat,)

(57, But his face you could not see,)

(89, On account of his Beaver Hat.)

Clearly, the keys are not line numbers. This would be impossible to implement in general, in that a file is broken into splits at byte, not line, boundaries.

Splits are processed independently. Line numbers are really a sequential notion. You have to keep a count of lines as you consume them, so knowing the line number within a split would be possible, but not within the file.

However, the offset within the file of each line is known by each split independently of the other splits, since each split knows the size of the preceding splits and just adds this onto the offsets within the split to produce a global file offset.

The offset is usually sufficient for applications that need a unique identifier for each line. Combined with the file’s name, it is unique within the filesystem. Of course, if all the lines are a fixed width, calculating the line number is simply a matter of dividing the offset by the width.