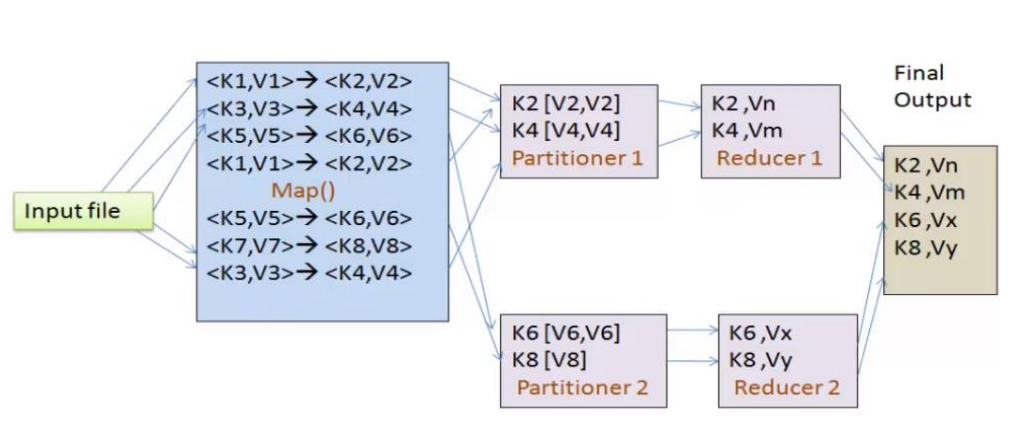


- Input Files
- InputFormat in Hadoop
- InputSplits
- RecordReader
- Mapper
- Combiner
- Partitioner
- Shuffling and Sorting
- Reducer
- RecordWriter and OutputFormat



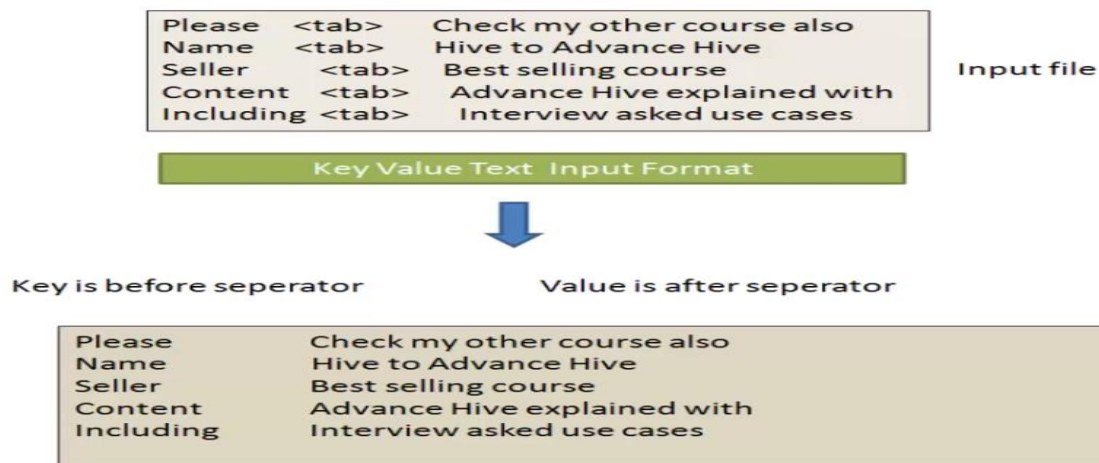
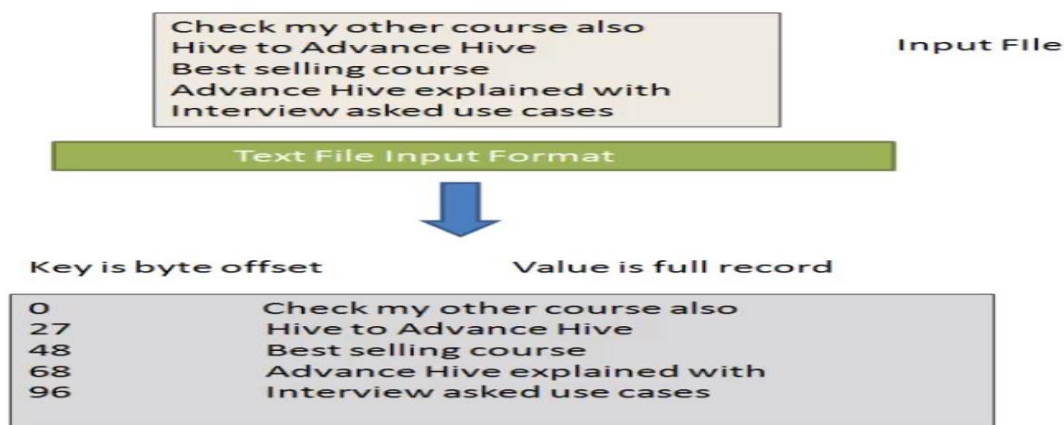
Map Function and Reduce Function.

- Every phase will get input as key value pair and output data in form of key value pair.

- First Step is Map phase which gets data in form of key value pair.
- Map function will apply logic and produce another key value pair to new set of key value pair.
- These are taken by partitioner and shuffling phase.
- Partitioner will take same keys and generate list of values from same pair of key.
- Which key value will go to which reducer?
- By default hadoop use hash partitioner. Sorting is also done on basis of key.
- No of partitions is always equal to no of reducers.
- Partitioning, shorting and shuffling are intermediate phase.
- Last one is reducer phase.

File Input Format:

- Text File input format
- Key value Text input format
- Fixed length input format
- N line input format
- Sequence File Input Format
 - Sequence file as text input format
 - Sequence file as binary input format



N Line: Keys are byteoffset of line and values are content of line. Each mapper will process the set of n number of lines.

- Map output is store intermediate as sequence file.

Sequence File Input Format

- ☐ Format Used to read Sequence files.

- ☐ Reads the data from Intermediate Map files.

- ☐ Stores the data in form of binary key value pairs

- ☐ Keys and Values are User defined.

- *Sequence File as Text Input Format:* Converts key value pair to text objects by calling `toString()` method.

- *Sequence File as Binary Input Format:* Retrieves the Sequence file's key and values as Binary objects.


Various Classes in MapReduce:

- Mapper Class
- Reducer Class
- Driver Class
- Partitioner Class
- Combiner Class
- FileInput Format class

Mapper, reducer and driver class are important classes whose method needs we need to overwrite. We need to overwrite as there will be no data processing.

Mapper Class:

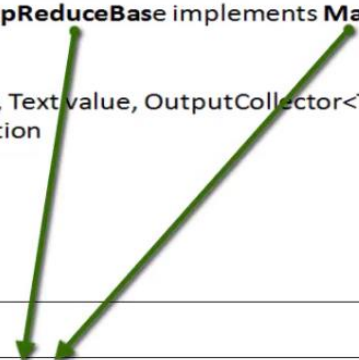
```
public class MyMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable>
{
    public void map (LongWritable key, Text value, OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException
    {
        <logic>
        output.collect(key,value);
    }
}
```



- Should implements mapper class and inherit MapReduceBase and overwrite method in both.
- Mapper interface expect four input, Input key input value,output key , output value.
- OutputCollector is responsible for writing intermediate data generated by the mapper


```
public class MyMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable>
{
    public void map (LongWritable key, Text value, OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException
    {
        <logic>
        output.collect(key,value);
    }
}
```

Hadoop 1



```
public class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    public void map (LongWritable key, Text value, Context obj) throws IOException
    {
        <logic>
        obj.write(key,value);
    }
}
```

Hadoop 2



- In MR1 we use OutputCollector and Reporter while in MR2 we are using Object of Context class and its write method to collect the output.
- Maps input key/value pairs to a set of intermediate key/value pairs.
- Maps are the individual tasks which transform input records into an intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs.

- The framework first calls **#setup(org.apache.hadoop.mapreduce.Mapper.Context)** followed by **#map(Object, Object, org.apache.hadoop.mapreduce.Mapper.Context)** for each key/value pair in the InputSplit and finally **#cleanup(org.apache.hadoop.mapreduce.Mapper.Context)**.
 - All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to a {@link **Reducer**} to determine the final output. Users can control the sorting and grouping by specifying two key {@link **RawComparator**} classes.
 - The Mapper outputs are partitioned per Reducer. Users can control which keys (and hence records) go to which Reducer by implementing a custom {@link **Partitioner**}.
 - Applications can specify if and how the intermediate outputs are to be compressed and which {@link CompressionCodec} are to be used via the `Configuration`.
- 1) setup: Called once at the beginning of the task.
 - 2) map: called once for each key/value pair on the input split. Most application should override this.
 - 3) cleanup: called once at the end of the task.
 - 4) run: expert user can override this method for more complete over the execution of the mapper.

Types of Mapper:

- **Identity Mapper:** This will be picked automatically when no mapper is specified in driver class. Implement the identity function which directly write all its input key value pairs into output.
- Inverse Mapper
- Regex Mapper
- Token Counter Mapper

Identity Mapper - Default mapper used to write the output same as input.
Implementation - Mapper < K, V, K, V >

Inverse Mapper- Used to reverse the Key Value pair.
Implementation - Mapper < K, V, V, K >

Regex Mapper - Provides a way to use regular expression in Map function.
Implementation - Mapper < K, Text, Text, Longwritable >

Token Counter Mapper – Used to generate token counts for key.
Implementation - Mapper < K, Text, Text, Longwritable >

REDUCER CLASS:

Mapper Output < K, V > → Partitioner output < K, List [V] > → Reducer

- Reducer class get input from mapper class.
- Basically we can say we get input from partitioner class.


```
public class MyReducer extends MapReduceBase implements Reducer<Text, IntWritable,
Text,IntWritable>
{
    public void reduce (Text key, Iterator<IntWritable> values, OutputCollector<Text,IntWritable>
output, Reporter reporter) throws IOException
    {
        <logic>
        output.collect(key,value);
    }
} }
```

Hadoop 1

```
public class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
    public void reduce (Text key, Iterator<IntWritable> values, Context Obj ) throws IOException
    {
        <logic>
        Obj.write(key,value);
    }
} }
```

Hadoop 2

Types of Reducer:

- Identity Reducer.
- Long Sum Reducer

- 1) Identity Reducer - Default reducer used to write the output same as input.
Implementation - Reducer < Key, List[values], Key , List[values] >
- 2) Long Sum Reducer – Used to determine sum of all values corresponding to a given key.
Implementation - Reducer < Key , Iterator<IntWritable> values, Key , Longwritable>



DRIVER CLASS:

Partitioner Class

It is a class which takes input as key value pair from mapper and for each key it creates a list of values and which in turn is send to reducer.

It also decide which key value pair will go to which reducer.

Default is Hash Partitioner Class.

```

public abstract class Partitioner <KEY,VALUE> extends Object
{
    public abstract int getPartition (KEY key, VALUE value, int numPartitions)
}

```

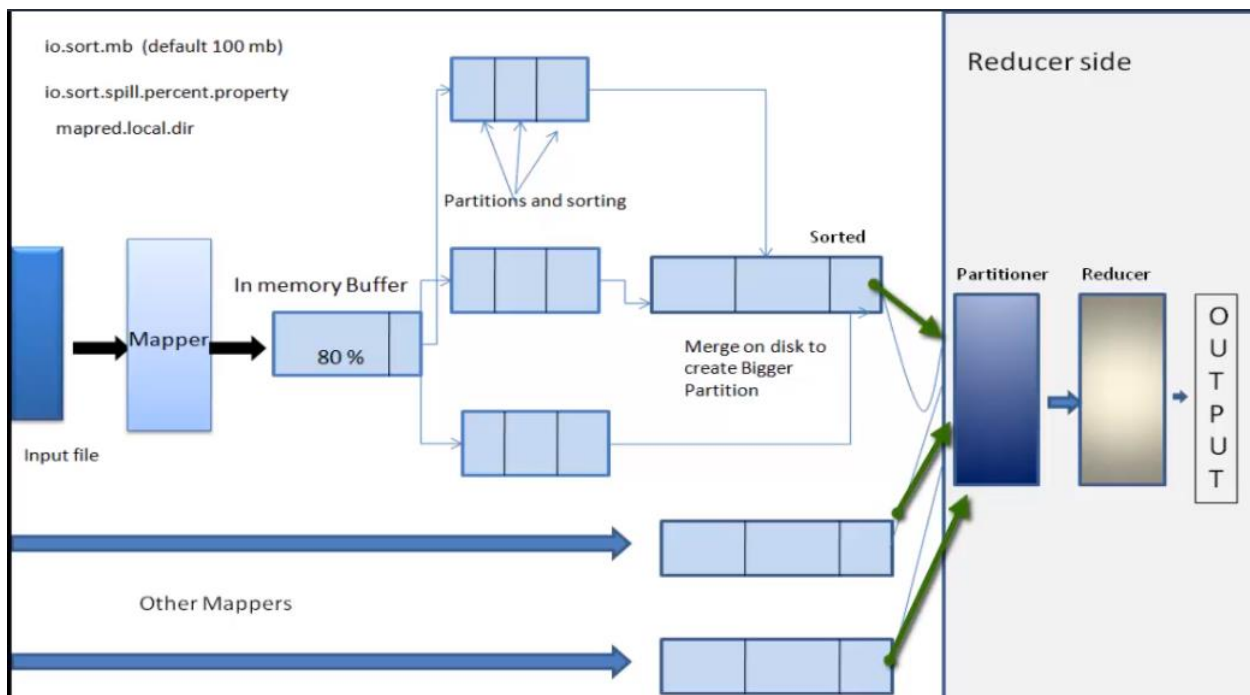
Default HashPartitioner Class

```

Public class HashPartitioner < K, V > extends Partitioner < K, V >
{
    public int getPartition (K key, V value, int numReduceTasks)
    {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}

```

Shuffling, Sorting & Partitioning



- Mapper doesn't directly write to disk rather it takes advantage of in memory buffer.
- Each mapper write its output to inmemory buffer whose by default value is 100mb, but it can be change by **io.sort.mb** . When buffer reaches the threshold a background thread get triggeres and writes data to local disk.
- This threshold can also be changed by io.sort.spill.percent.property

Coding:

- IntWritable instead of int , Text instead of String
- Mapreduce uses two interface: 1) Writable and other one is 2) writable comparable for the data type conversion

Writable in MapReduce:

- Writable is an interface in Hadoop
- It acts as wrapper to primitive data type in Java
- All the Mapreduce datatype must implement writable Interface.
- We can also create our own custom data type that implement writable or writable comparable interface.

Java primitive

Boolean

Byte

Short

Int

Float

Long

Double

Writable implementation

BooleanWritable

ByteWritable

ShortWritable

IntWritable

FloatWritable

LongWritable

DoubleWritable

Java class

String

Object

Writable implementation

Text

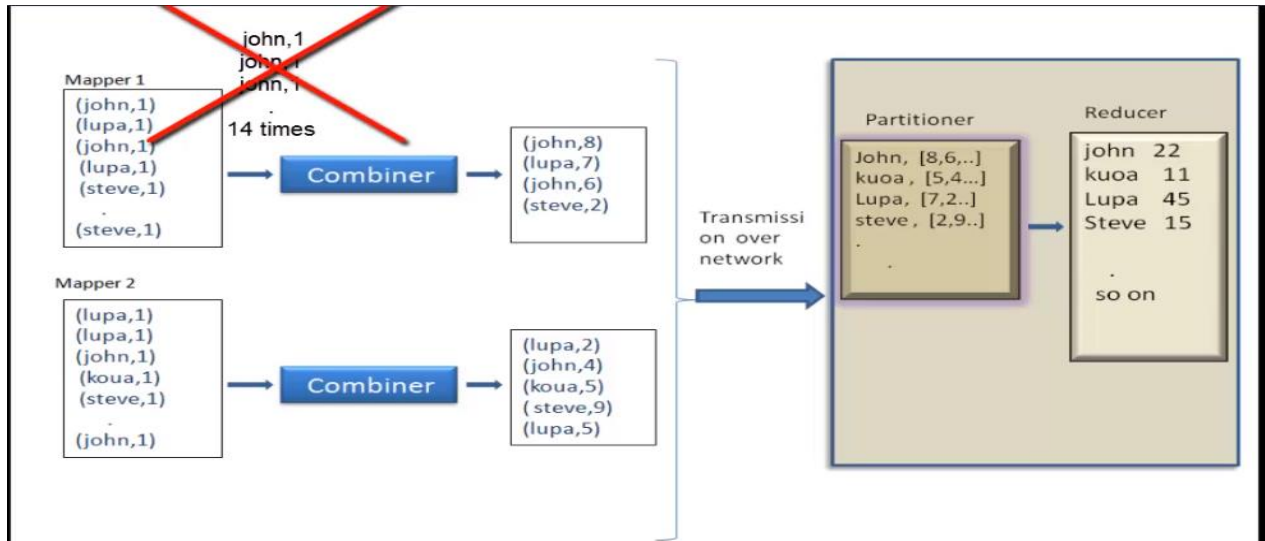
ObjectWritable

Combiner

Q) What is combiner in Mapreduce?

Ans: The combiner is to minimize no of key value pair that is too be transmitted over the network.

- Combiner does this is combining the same in mapper phase only
- It is also called mini reducer. Combiner and reducer uses the same code. Combiner will aggregate the result locally for an individual mapper.
- Combiner can be run any no of times in mapper output.
- We cannot use combiner in every code.



Combiner Key Points

- ☐ Combiner is a optimization technique to fasten the job execution.
- ☐ Combiner reduces the number of Key Value pairs to be transmitted to reducer.
- ☐ It runs locally on each mapper.
- ☐ It can run 0 , 1 or n number of tims on a mapper.
- ☐ Also known as 'Mini Reducer'.
- ☐ It should be used where the nature of problem is Associative and Commutative.

Q) Which of the following will run first: the partitioner or combiner?

Ans: Combiner as it is a mini reducer.

Creating your own writable class: Writable and WritableComparable in Hadoop

After reading this blog you will get a clear understanding of:

- 1) What are Writables?
- 2) Importance of Writables in Hadoop?
- 3) Why Hadoop is not using java serialization as well as java datatype?
- 4) Why are Writables introduced in Hadoop?
- 5) What if Writables were not there in Hadoop?
- 6) How can Writable and WritableComparable be implemented in Hadoop?
- 7) How to create custom writables?
- 8) What is serialization in java?
- 9) What is wrapper class?
- 10) What is rawComparator?
- 11) What is RPC? Remote processor call: Light weight protocol which carry small information.

Serialization: A process of converting object state to byte state.

Object State: object state are instance variable in object. Ex: id, name, salary

Object behavior: method ex: getsalary()

Serializable interface: It is a marker interface.

Marker Interface: Interface that doesn't contain any method. Empty interface.

All the object are by default not serializable.

When we do java serialization it stores a lot of information like class name, variable etc. In Hadoop serialization we can be choosy. We can choose that we want to serialize only id and name only not salary. But in java we have to serialize complete object.

Hadoop has two interface:

- 1) **Writable**
- 2) **WritableComparable**: It don't have a method of its own. It takes two method from writable interface (**write** method and **readFields** method) and one method from comparable (**compareTo** method).
public interface WritableComparable<T> extends Writable, Comparable<T> {}
We will use comparable for sorting purpose. Sorting will happen on basis of Key. Key will be writableComparable

Writable:

```
public interface Writable {  
    void readFields(DataInput in);  
    void write(DataOutput out);
```

}

Hadoop is using its own serializable called writable interface. Writable interface is not a marker interface. It has its own method:

- 1) **Write** method : For serializing
- 2) **ReadFields** method: DE serializing. reads the data from network and write will write the data into local disk

Key data will be writableComparable. Sorting is done with quick sort algorithm. Key must be writableComparable.

Writable is an interface in Hadoop. Writable in Hadoop acts as a wrapper class to almost all the primitive data type of Java. That is how int of java has become IntWritable in Hadoop and String of Java has become Text in Hadoop.

Now the question is whether Writables are necessary for Hadoop. Hadoop framework definitely needs Writable type of interface in order to perform the following tasks:

- Implement serialization
- Transfer data between clusters and networks
- Store the deserialized data in the local disk of the system

writableComparable has 3 method which is mandatory to implement. We need to overwrite these methods

- 1) **Write**: how to write value to byte stream.
- 2) **readFields**: How to read values to byte stream.
- 3) **compareTo**: It is overwritten according to our use.

All these primitive writable wrappers have get() and set() methods to read or write the wrapped value. Below is the list of primitive writable data types available in Hadoop.

- BooleanWritable
- ByteWritable
- IntWritable
- VIntWritable
- FloatWritable
- LongWritable
- VLongWritable
- DoubleWritable

In the above list VIntWritable and VLongWritable are used for variable length Integer types and variable length long types respectively

Distributed Cache:

Distributed cache is a mechanism supported by hadoop mapreduce framework where we can broadcast small or moderate sized files (read only) to all the worker nodes where the map/reduce tasks are running for a given job.

Each worker node that runs the tasks of a given job will have one copy of the file(s) sent via Distributed cache. It is possible to control the size of distributed cache with cache size property in mapred-site.xml

After successful run of the job, the distributed cache files (these are temporary files) will be deleted from worker nodes.

It is facility provided by hadoop mapreduce to access small files needed by application during its execution. These files are small as they are in kbs and mbs. Types of files are mainly text file, archive and jar files.

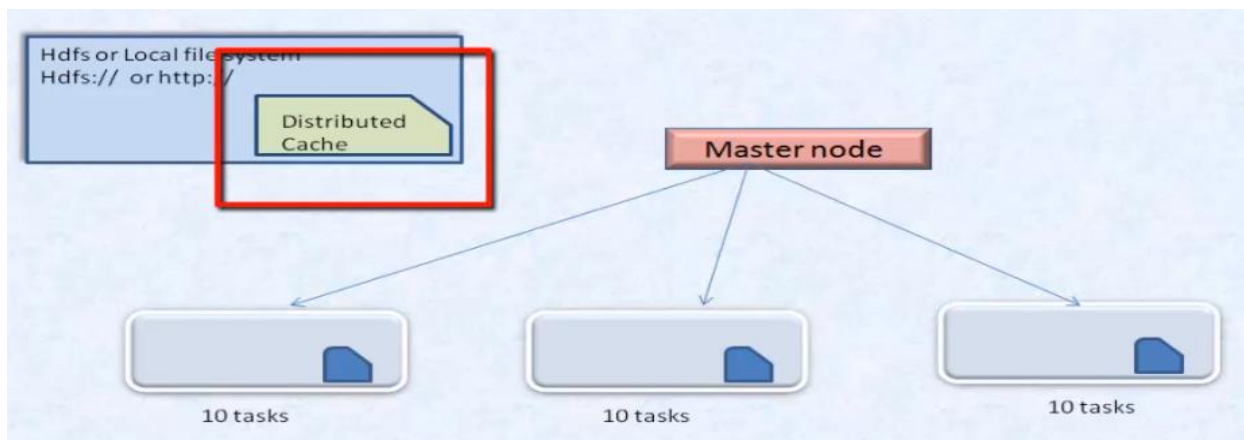
- It is of read only file.
- By default hadoop distributed cache is of 10GB, but can be changed with cache size property in mapred-site.xml.
- Cache is much faster and it copies the data to each slave nodes.

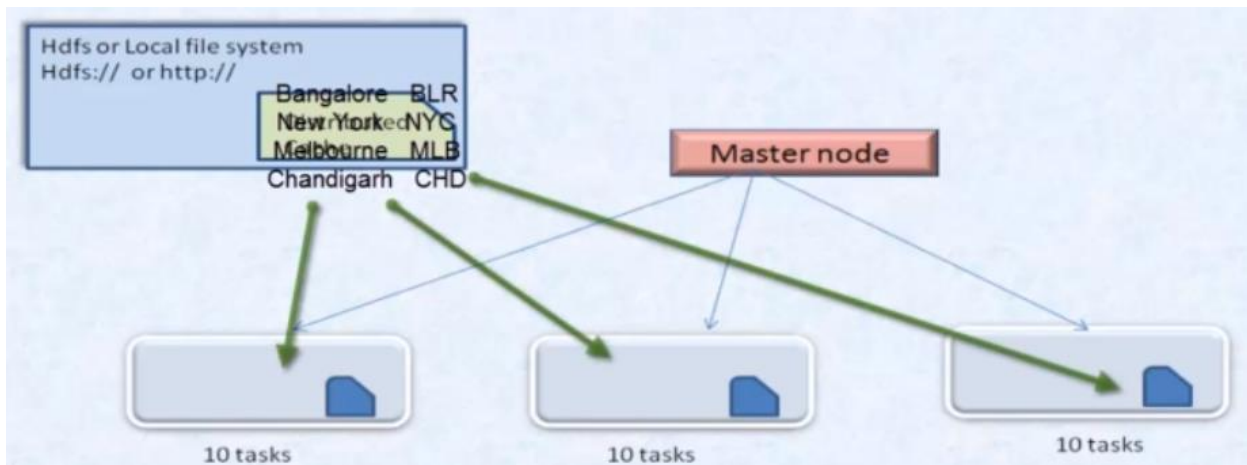
Package we need to use for distributed cache:

```
import org.apache.hadoop.filecache.DistributedCache;
```

```
import java.net.URI;
```

```
import java.net.URISyntaxException;
```





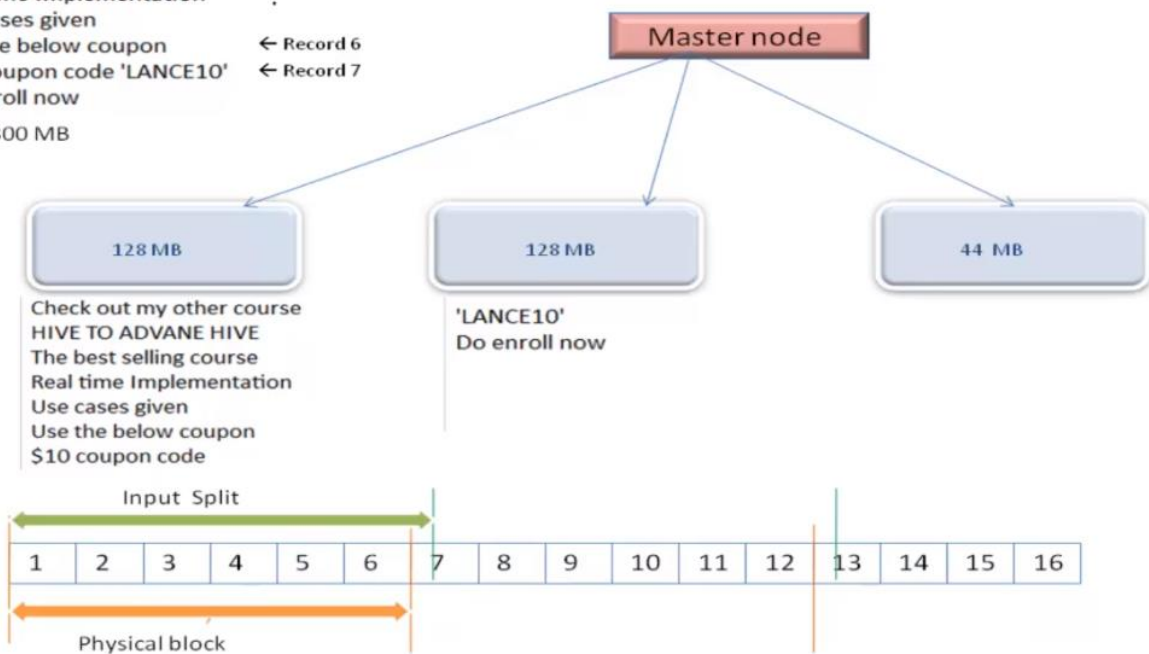
Distributed Cache Key Points

- ☐ Distributed Cache is facility provided by Mapreduce to cache files when needed by application.
- ☐ Files that can be cached includes read-only files, archive and jar files.
- ☐ The cached files are copied to each slave node before execution.
- ☐ Distributed Cache reduce the number of reads from HDFS location.
- ☐ Implemented by setting it up in the Job Conf Driver Class.

Input Split:

- When mapreduce job client calculate the input split it figures out where the first whole record in a block begins and where the last whole records in the block ends. In case where the last record is incomplete the input split stores the location information for the next block and the byteoffset of the data which is needed to complete the record.
- It is called logical representation
- By default inputsplit size = hadoop block size but we can adjust it by making some changes in some parameters.
- It ensures that a record is completely processed by a single mapper.

Check out my other course ← Record 1
 HIVE TO ADVANE HIVE ← Record 2
 The best selling course
 Real time Implementation
 Use cases given
 Use the below coupon ← Record 6
 \$10 coupon code 'LANCE10' ← Record 7
 Do enroll now
 300 MB



Input Split size = $\text{Max}(\text{minimumSize}, \min(\text{maximumSize}, \text{blockSize}))$

Name	Type	Default value
Block size	long	128 MB
Maximum split size	long	9223372036854775807 MB
Minimum split size	int	1

Default case	<p>Min size < block size < Max size</p> <p>Input size = block size</p>	<p>Case2</p> <p>Minimum size = 256 Maximum size = 92233.....07 Block size = 128</p> <p>Input split size = 256 mb</p>
Case1	<p>Minimum size = 1 Maximum size = 92233.....07 Block size = 256 $\text{Max}(\text{minSize}, \min(\text{maxSize}, \text{blockSize}))$</p> <p>Input split size = 256 mb</p>	<p>Case3</p> <p>Minimum size = 1 Maximum size = 64 Block size = 128</p> <p>Input split size = 64 mb</p>

- InputSplit is an abstract class which has two methods **getLength()** and **getLocations()** method. Both these methods are abstract methods. Any class that is going to inherit InputSplit class must provide the implementations to these two methods.

- **getLength()** is used to get the size of the split so that input split can be sorted by size and then the processing can start from bigger split first, so that the larger split get processed first to minimize the job run time. This methods returns the no of bytes in a particular split.
- **getLocation()** method is used to get list of nodes where the physical data for a split would be local because a map task can be started on any node weather it contains the data or not. By using getLocation() method we create a list of nodes where the required data is physically present so that framework can place the map task locally to that node and hence the data can be used locally, this results in faster execution of our job.

MULTIPLE INPUTS CLASS

A real-time project can consist of n no. of input files with different structure and therefore should read by different input formats, this class helps the job in reading and processing the multiple input files.

Q) We have to perform word count on two files in a single job where one file is to be read by text input format and second file is to be read by key value input format.

```
//adding the parameter for first input file
MultipleInputs.addInputPath(job,inputPath1,TextInputFormat.class,MI_mapper1.class);

//adding the parameter for second input file
MultipleInputs.addInputPath(job,inputPath2,KeyValueTextInputFormat.class,MI_mapper2.class);

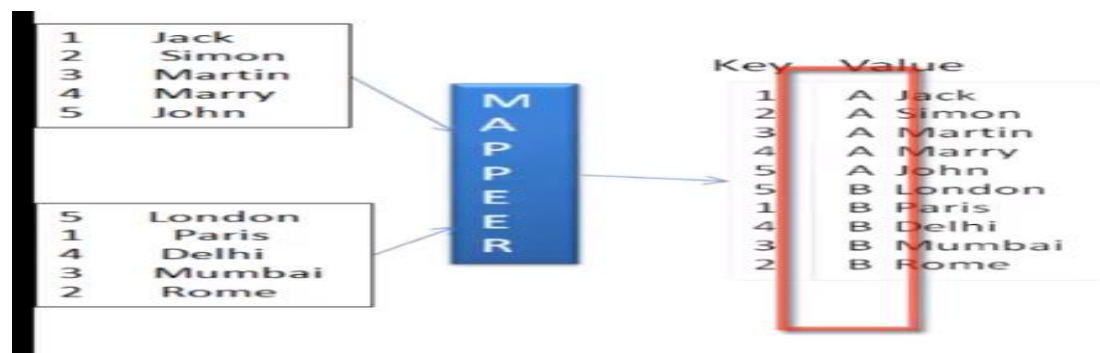
FileOutputFormat.setOutputPath(job,output_dir);
output_dir.getFileSystem(job.getConfiguration()).delete(output_dir,true);
```

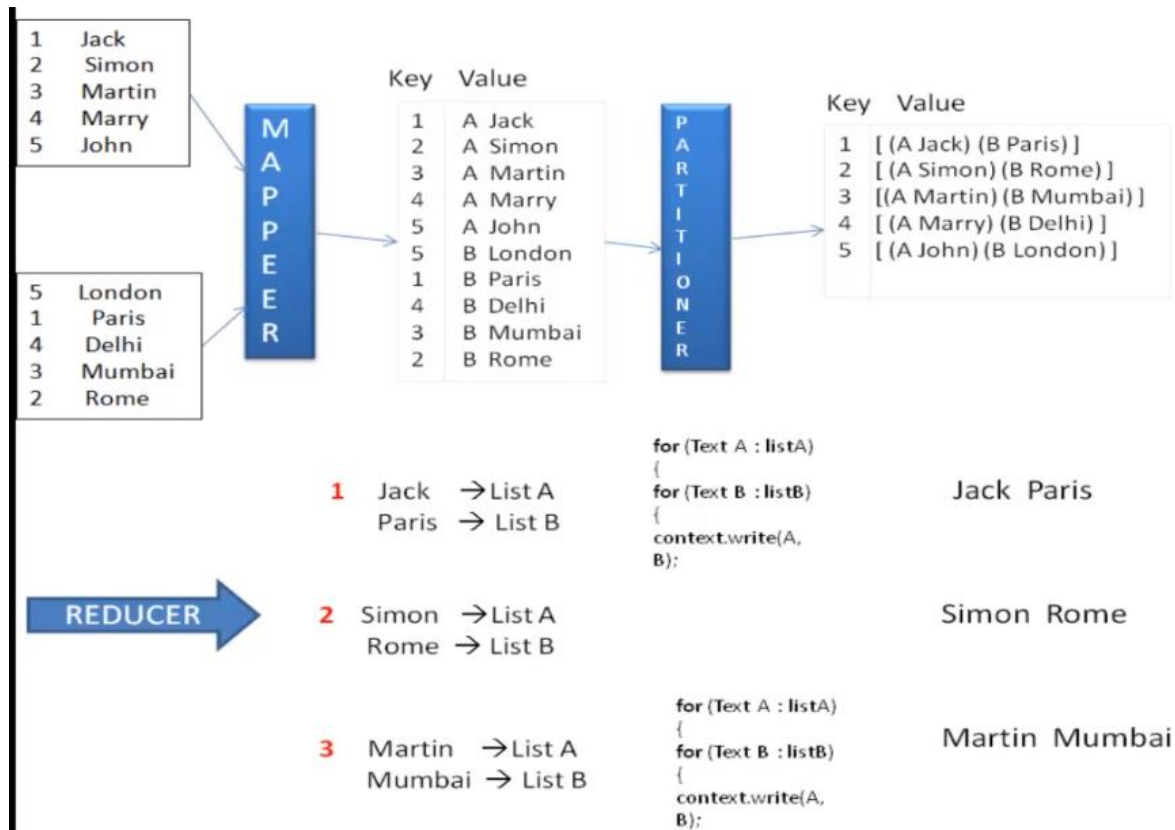
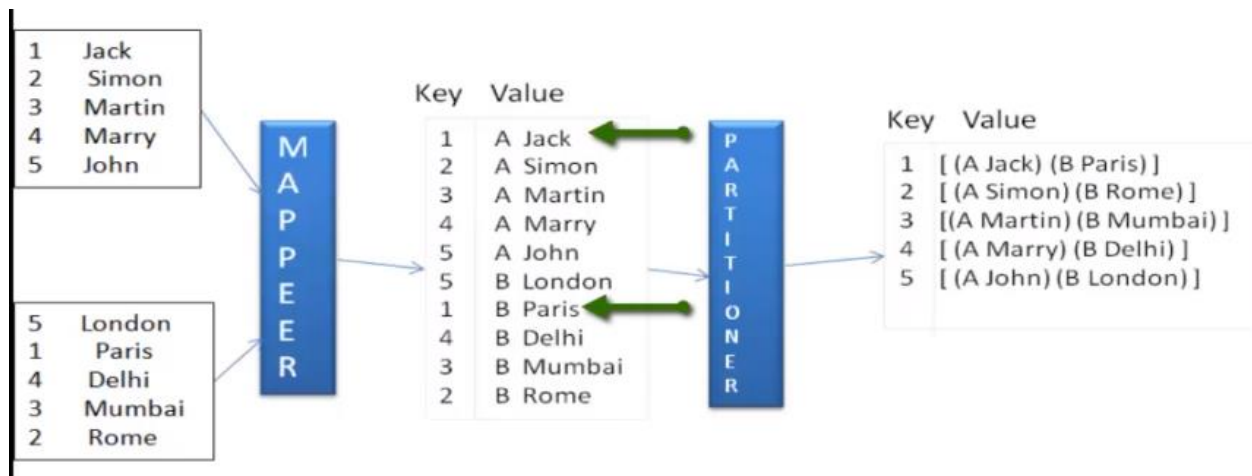
MULTIPLE OUTPUT CLASS

- To store the output in multiple class. Earlier we used to store in a single file.

JOINS IN MAPREDUCE

- Reduce side join and Map side join.
- In reduce side join joining is done at reducer phase and in map side join the joining is done at mapper phase.





- Inner Join:
- Outer Join
- Left Outer Join

- Right Outer Join
- Full Outer Join

Map Joins: Map joins is join performed at mapper phase only rather than going to reducer phase.

Map Joins

- ❑ Join is performed on Mapper side.
- ❑ Is used where files to be joined have varied data structure and not 1 common column key.
- ❑ Is used where files/tables are small enough to be fit into Mappers memory.
- ❑ Used as optimization technique.
 - Atleast 1 dataset should be small
 - Dataset set should be sorted
- ❑ Benefits of Distributed cache in Mapjoins -
 - All the files are read in a single shot and make data available for map function.
 - Fast join can be performed using Mapper local memory.

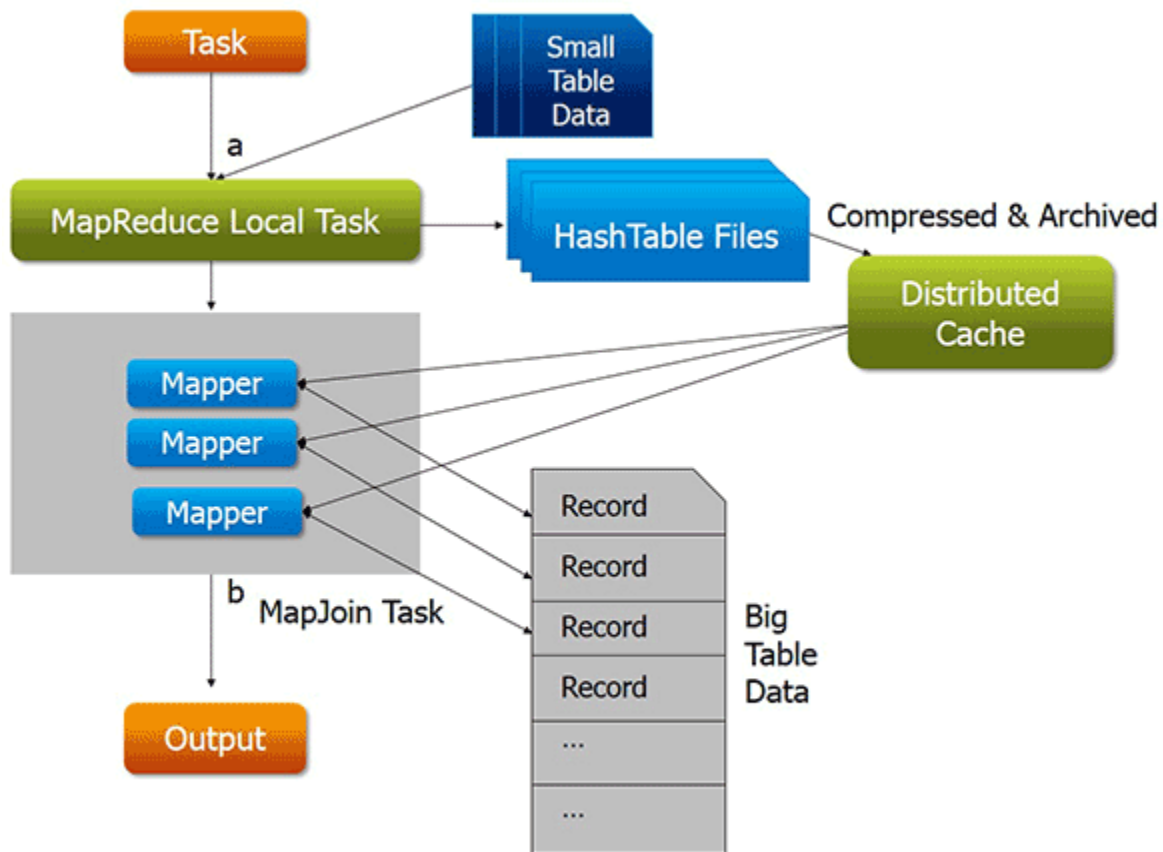
Use Map joins when:

-
- 1) When we have to join various file having varied data type and not a single common column among all the file.
 - 2) Is used where files/tables are small enough to be fit into Mappers memory.
 - 3) Map join can also be used as optimization technique as lots of time saved by not doing shuffling and partitioning as there is no need to go through those phases as our join is performed in mapper phase only.
 - 4) Input file should be sorted by same key.
 - 5) It is mandatory that the input to each map is partitioned and sorted according to the keys.

How will the map-side join optimize the task?

Assume that we have two tables of which one of them is a small table. When we submit a map reduce task, a Map Reduce local task will be created before the original join Map Reduce task which will read data of the small table from HDFS and store it into an in-memory hash table. After reading, it serializes the in-memory hash table into a hash table file.

In the next stage, when the original join Map Reduce task is running, it moves the data in the hash table file to the Hadoop distributed cache, which populates these files to each mapper's local disk. So all the mappers can load this persistent hash table file back into the memory and do the join work as before. The execution flow of the optimized map join is shown in the figure below. After optimization, the small table needs to be read just once. Also if multiple mappers are running on the same machine, the distributed cache only needs to push one copy of the hash table file to this machine.

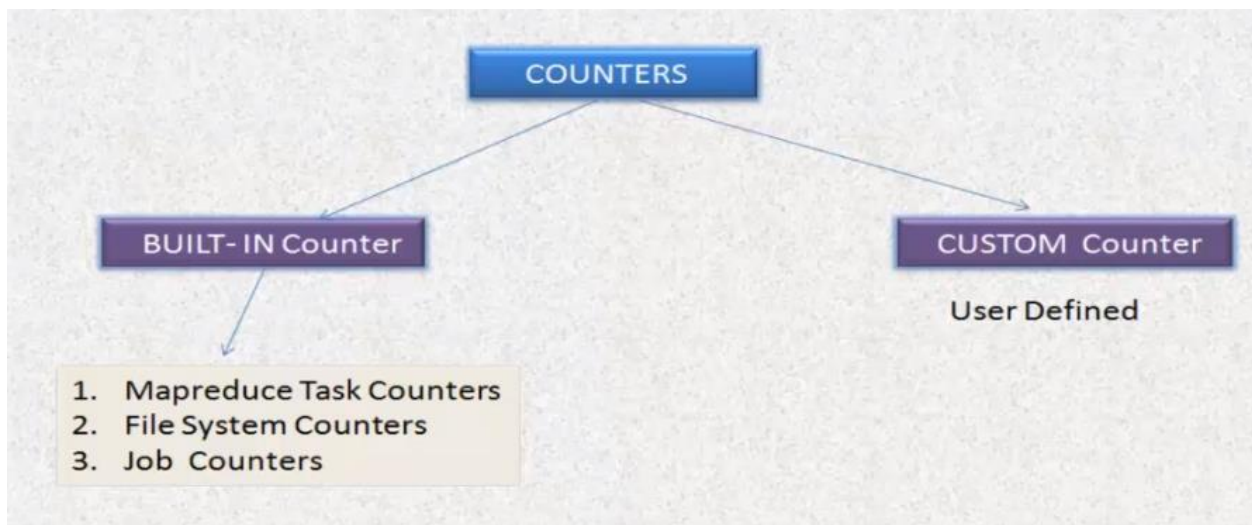


COUNTERS

COUNTERS

- ❑ Counters is a way to calculate the number of operations within a Mapreduce Job.
- ❑ Useful for gathering statistics about the Mapreduce Jobs. Also used for problem diagnosis.
- ❑ Whenever a Mapreduce job starts execution Hadoop initiates Counters to collect Job statistics.
- ❑ Some of the typical operations of Hadoop Counters are to get the stats about following :
 - Number of mappers and reducers launched .
 - Number of bytes read and written.
 - Number of tasks launched and ran.
 - Number of bytes shuffled across nodes.
 - Amount of CPU and memory consumed for the job.

-
- Counters are of two types: built-in counters and custom counter. By default map reduce provide built-in counter. It also provide us to create our own counters.



Counter	Description
Map input records (MAP_INPUT_RECORDS)	The number of input records consumed by all the maps in the job. Incremented for every RecordReader read.
Map input bytes (MAP_INPUT_BYTES)	The number of bytes of uncompressed input consumed by all the maps in the job. Incremented for every RecordReader read.
Map output records (MAP_OUTPUT_RECORDS)	The number of map output records produced by all the maps in the job. Incremented every time the collect() method is called on a map's OutputCollector.
Reduce input records (REDUCE_INPUT_RECORDS)	The number of input records consumed by all the reducers in the job. Incremented every time a value is read from the reducer's iterator over value.
Reduce output records (REDUCE_OUTPUT_RECORDS)	The number of reduce output records produced by all the reduces in the job. Incremented every time the collect() method is called on a reducer's OutputCollector.
Spilled records (SPILLED_RECORDS)	The number of records spilled to disk in all map and reduce tasks in the job.
CPU milliseconds (CPU_MILLISECONDS)	The cumulative CPU time for a task in milliseconds, as reported by /proc/cpuinfo.
Physical memory bytes (PHYSICAL_MEMORY_BYTES)	The physical memory being used by a task in bytes, as reported by /proc/meminfo.
Virtual memory bytes (VIRTUAL_MEMORY_BYTES)	The virtual memory being used by a task in bytes as reported by /proc/meminfo.
Committed heap bytes (COMMITTED_HEAP_BYTES)	The total amount of memory available in the JVM in bytes, as reported by Runtime.getRuntime().totalmemory().

Counter	Description
Filesystem Bytes read (BYTES_READ)	The number of bytes read by filesystem by map and reduce tasks. Filesystem may be local, HDFS, S3 etc.
Filesystem bytes written (BYTES_WRITTEN)	The number of bytes written by each filesystem by map and reduce tasks.

- **Uber task:** uber task are small task in a job for which application master runs in the same JVM in which application master is itself-running because it sense that allocating new container and running task in them is an overhead than if we run in its own JVM so rather than allocating new containers application master runs the task in its own JVM. These type of task are called uber task. This counter will capture all the uber task which is launched.

Counter	Description
Launch Map Tasks (TOTAL_LAUNCHED_MAPS)	The number of map tasks that were launched.
Launched reduce tasks (TOTAL_LAUNCHED_REDUCES)	The number of reduce tasks that were launched
Launched uber tasks (TOTAL_LAUNCHED_UBERTASKS)	The number of Uber tasks that were launched.
Maps in uber tasks (NUM_UBER_SUBMAPS)	The number of maps in Uber tasks.
Reducers in uber tasks (NUM_UBER_SUBREDUCES)	The number of reduces in Uber tasks.
Failed map tasks (NUM_FAILED_MAPS)	The number of map tasks that failed.
Failed reduce tasks (NUM_FAILED_REDUCES)	The number of reduce tasks that failed.
Failed uber tasks (NUM_FAILED_UBERTASKS)	The number of Uber tasks that failed.
Data-local map tasks (DATA_LOCAL_MAPS)	The number of map tasks that ran on the same node as their input data.
Rack-local map tasks (RACK_LOCAL_MAPS)	The number of map tasks that ran on a node in the same rack as their input data, but that are not data-local.

USER DEFINED COUNTERS

- Created by user according to their needs.
- Can be static or dynamic. Static counter are defined by java enumeration. We can define n number of enums and each enum with n number of fields. Name of enum is group name and enum fields are counter name.
- Dynamic counter can be used simply anywhere in the program.
- For static counter we need to create it under enum and for dynamic counter we can create it any time anywhere.

```

18/03/12 19:59:34 INFO mapred.JobClient: File Output Format Counters
18/03/12 19:59:34 INFO mapred.JobClient:   Bytes Written=46
18/03/12 19:59:34 INFO mapred.JobClient: com.hadoop.counters.LOCATION
18/03/12 19:59:34 INFO mapred.JobClient:   BANGALORE=40
18/03/12 19:59:34 INFO mapred.JobClient:   CHENNAI=54
18/03/12 19:59:34 INFO mapred.JobClient:   HYDERABAD=57
18/03/12 19:59:34 INFO mapred.JobClient:   TOTAL=151
18/03/12 19:59:34 INFO mapred.JobClient: FileSystemCounters
18/03/12 19:59:34 INFO mapred.JobClient:   FILE_BYTES_READ=2530
18/03/12 19:59:34 INFO mapred.JobClient:   HDFS_BYTES_READ=5095
18/03/12 19:59:34 INFO mapred.JobClient:   FILE_BYTES_WRITTEN=11553

```

Dynamic Counter

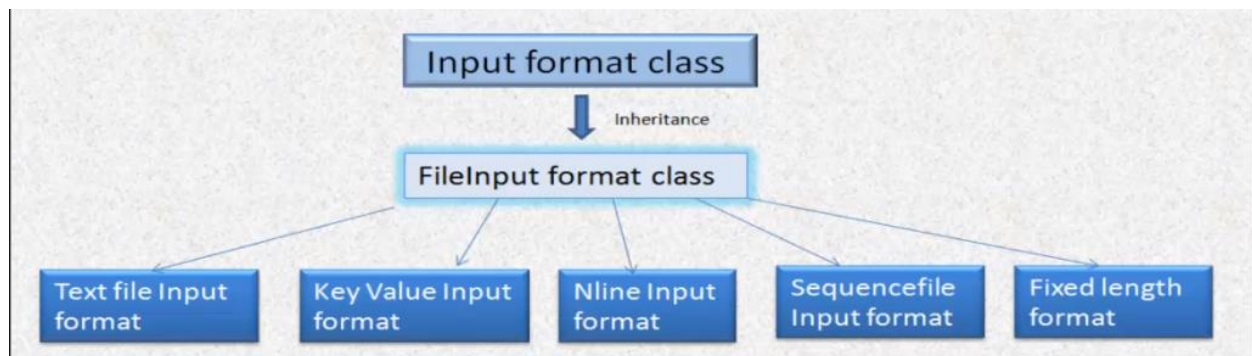

```

18/03/12 19:59:34 INFO mapred.JobClient: Bytes Read=4989
18/03/12 19:59:34 INFO mapred.JobClient: SALES
18/03/12 19:59:34 INFO mapred.JobClient: HIGH_REVENUE=41
18/03/12 19:59:34 INFO mapred.JobClient: LOW_SALES=70
18/03/12 19:59:34 INFO mapred.JobClient: Map-Reduce Framework
18/03/12 19:59:34 INFO mapred.JobClient: Map output materialized bytes=2530
18/03/12 19:59:34 INFO mapred.JobClient: Map input records=151

```

FILE INPUT FORMAT CLASS

- Input format is a class which decide how to read data from a file. As we know input to a mapper is in key value pair, there should be some criteria to create those key value pair from that file.
- Map reduce framework has provided us input format class to serve the purpose for creating those key value pair and then providing it to mapper.



Input Format Class:

- This class is an abstract class which contain two method. Get Splits() and createRecordReader()
- Both methods are of abstract type, they should be defined in the class which will inherit this input format class.
- **Getsplit()** return list of input split. Hadoop process the data in the form of input split, so this method helps in creating those splits out of a big input file and return a list of input split. This method is responsible for creating the splits and will generate list of those input splits.
- **createRecordReader()** iterates over the data in the input split and returns the key value pair from it. This method will run for each input split present in the list that was created by the getsplit() method and its return type is object of RecordReader<k,v> class in form of key value pair. This method internally calls the constructor of record reader class and that is the class that actually generates the key value pair and then return it to this method.

1) get Splits()

public abstract List <InputSplit> **getSplits** (JobContext context) throws IOException, InterruptedException

```
{  
    <logic>  
}
```

2) createRecordReader()

public abstract RecordReader<K,V> **createRecordReader** (InputSplit split, TaskAttemptContext context) throws IOException, InterruptedException

```
{  
    <logic>  
}
```

FileInput Format Class Methods

- ❑ static void **addInputpath** (**Job** job, **Path** path) - Add a **Path** to the list of inputs for the map-reduce job.
- ❑ static void **addInputPaths** (**Job** job, **String** commaSeparatedPaths) - Add the given comma separated paths to the list of inputs for the map-reduce job
- ❑ protected boolean **isSplittable** (**JobContext** context, **Path** filename) - Decides if the file is to be split or not.
- ❑ static void **setMaxInputSplitSize** (**Job** job, **long** size) - Set the maximum split size
- ❑ static void **setMinInputSplitSize** (**Job** job, **long** size) - Set the minimum input split size

Different Types of Files in Hadoop

Different Types of Files in Hadoop

File should :

- Get Read fast.
- Get Written fast.
- Be Splittable i.e. multiple tasks can run parallel on parts of file.
- Support Schema evolution, allowing us to change schema of file.
- Support advanced compression through various available compression codecs (Bzip2,LZO,Snappy etc).

Text Files (Csv,Tsv)

Behaviour - Each line is a record/data, and lines are terminated by a newline character (\n).

Read/Write - Good write performance but slow reads.

Compression - Do not support Block compression.

Splittable - Text-files are inherently splittable on \n character.

Schema Evolution - Limited Schema evolution (New fields can only be appended to existing fields while old fields can never be deleted).

Sequence File

Behaviour - Each record is stored as a key value pair in binary format.

Read/Write - Good write performance than Text files.

Compression - Support Block compression.

Splittable - Sequence files are splittable.

Schema Evolution - Limited Schema evolution (New fields can only be appended to existing fields while old fields can never be deleted).

Avro File

Behaviour - It is a file format plus a serialization and deserialization framework. Avro uses JSON for defining data types and serializes data in a compact binary format.

Read/Write - Average read/write performance.

Compression - Support Block compression.

Splittable - Avro files are splittable.

Schema Evolution - Was mainly designed for Schema evolution. Fields can renamed, added, deleted while old files can still be read with the new schema.

Columnar File Formats

- In columnar file format instead of just storing rows of data adjacent to one another we also store column values adjacent to each other.
- So datasets are partitioned both horizontally and vertically.

RC File

Behaviour - These are flat files consisting of binary key/value pairs, and it shares much similarity with Sequence File.

Read/Write - Was developed for faster reads but with a compromise with write performance.

Compression - Provides significant Block compression, can be compressed with high compression ratios.

Splittable - RC files are splittable.

Schema Evolution - Was mainly designed for Faster reads so NO schema evolution.

ORC File

Behaviour - A better version of RC file.

Read/Write - Was developed for faster reads but with a compromise with write performance
(Better than RC file).

Compression - Provides significant Block compression, can be compressed with high compression ratios
(Better than RC file).

Splittable - ORC files are splittable at stripe level.

Schema Evolution - Was mainly designed for Faster reads so NO schema evolution.

Parquet File

Behaviour - It is a columnar file format, similar to RC and ORC. Parquet stores nested data structures in a flat columnar format.

Read/Write - Faster reads with slow writes.

Compression - Support compression mostly with snappy algorithm.

Splittable - Parquet files are conditionally splittable.

Schema Evolution - Limited Schema evolution (New fields can only be appended to existing fields while old fields can never be deleted).

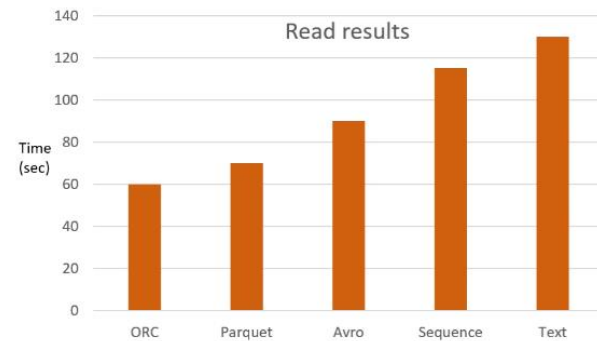
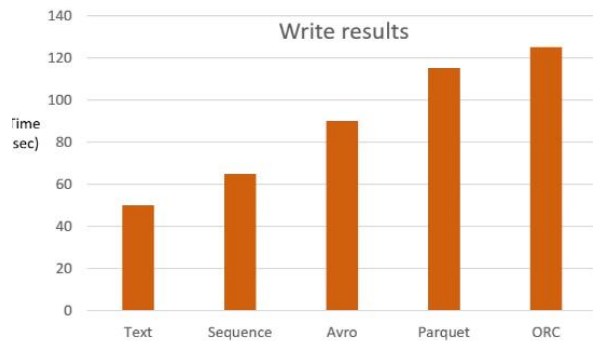
Read/Write Test results of various File Formats

Benchmark :

- 700 columns
- 2.5 million rows
- Hive queries

Read query : *select * from test_table where month_col = 'October';*

Write query : *insert into table test_table select * from source_table;*



Which File Format to choose?

Answer : Depends on your Use case and environment.

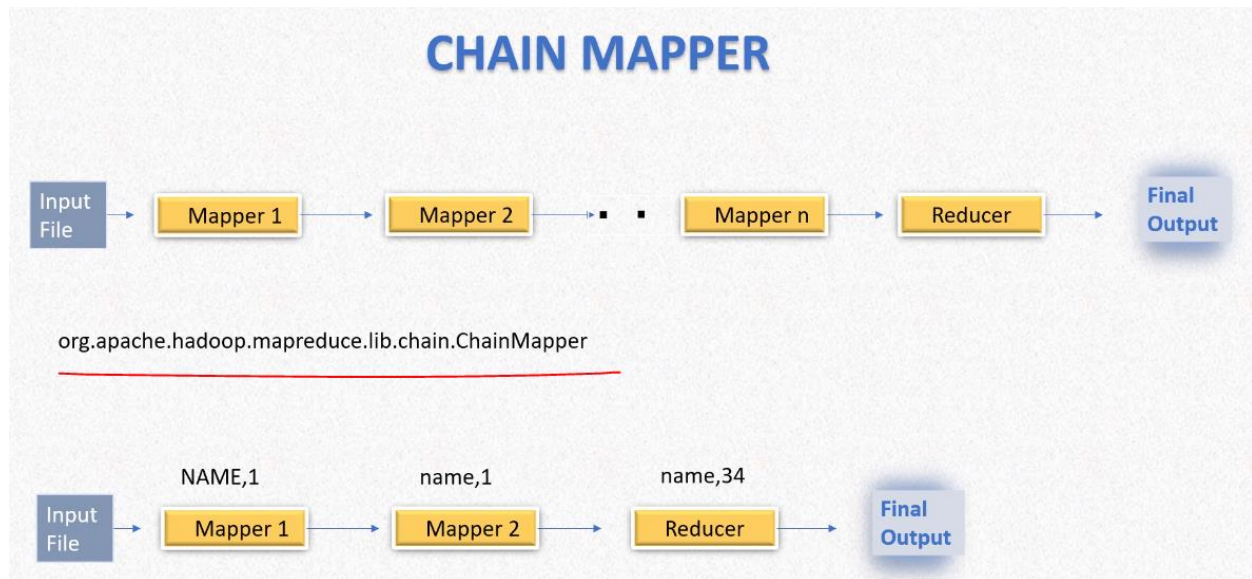
Schema going to change → Avro file

Dumping data from Hdfs → Text file

Reading data → Columnar files

Storing intermediate data → Sequence Files

- Map reduce give us feature to chain multiple mapper. Output of 1st mapper goes to 2nd mapper and output of third goes to next. In this way mapper become reusable each mapper has a particular function



Important Links:

- 1) <https://acadgild.com/blog/hdfs-interview-questions-and-answers-2019>
- 2) <https://acadgild.com/blog/hadoop-interview-questions-answers-2019-part-1>
- 3)

Interview Questions:

- 1) How to process different files by different mappers?
- 2) What is distributed cache in Hadoop map reduce?
- 3) What is Map side join?
- 4) When should you use Map side join?
- 5) What is the role of Writable and WritableComparable?
- 6) What should be the output key of map (Writable/WritableComparable)?
- 7) What is input split? What is the need of Input Split? Where Input split is created?
- 8) What is the role of RecordReader?
- 9) What are the tasks of InputSplit?
- 10) What is small file problem in Hadoop?
- 11) What are the advantages of sequence files?
- 12) What is HDFS namenode federation?
- 13) What is name node high availability?
- 14) How can you control block size and replication factor at file level?
- 15) What are the roles of Mapper, Combiner, Partitioner, Shuffle & Sort and Reducer classes?
- 16) How to control the number of reducers in a map reduce program?
- 17) How does Hadoop know how many mappers has to be started?
- 18) What is the difference between HDFS block and an InputSplit, and explain how the input split is prepared in Hadoop?
- 19) What are counters in MapReduce?
- 20) How do you copy files from one cluster to another cluster?

Ans. With the help of DistCp command, we can copy files from one cluster.

The most common invocation of DistCp is an inter-cluster copy:

- `bash$ Hadoop DistCp`
- `hdfs://nn1:8020/foo/bar`
- `hdfs://nn2:8020/bar/foo`

- 21) What details are present in FSIMAGE?
- 22) What is the purpose of Record Reader in Hadoop?
- 23) What are the different methods in Mapper class and order of their invocation?
- 24) Explain about Top-k Map-Reduce design pattern.
- 25) What is Mapreduce?
- 26) What is YARN?
- 27) What is data serialization?
- 28) What is deserialization of data?
- 29) What are the Key/Value Pairs in Mapreduce framework?
- 30) What are the constraints to Key and Value classes in Mapreduce?
- 31) What are the main components of Mapreduce Job?
- 32) What are the Main configuration parameters that user need to specify to run Mapreduce Job?
- 33) What are the main components of Job flow in YARN architecture?

- 34) What is the role of Application Master in YARN architecture?
- 35) What is identity Mapper?
- 36) What is identity Reducer?
- 37) What is chain Mapper?
- 38) What is chain reducer?
- 39) How can we mention multiple mappers and reducer classes in Chain Mapper or Chain Reducer classes?