MapReduce

Kaustuv Kunal
kaustuv.kunal@gmail.com
https://github.com/kaustuvkunal/Big-Data/

Outline

1. MapReduce Introduction
2. MapReduce Flow
3. Shuffle-Sort-Merge
4. Word-Count Program
5. Execution Modes
6. Partitioner
7. Combiner
8. Writables
9. Counters
10. Input Format
11. Output Format
12. Setup & Cleanup Methods
13. Side Data Distribution (Distributed Cache)
14. Sorting Large Datasets
15. Input Sampler
16. Total-Order-Partitioner
17. Map-Side & Reduce-Side Joins
18. Compression
19. Speculative Execution
20. MR Unit
21. Use Cases

MapReduce Intro

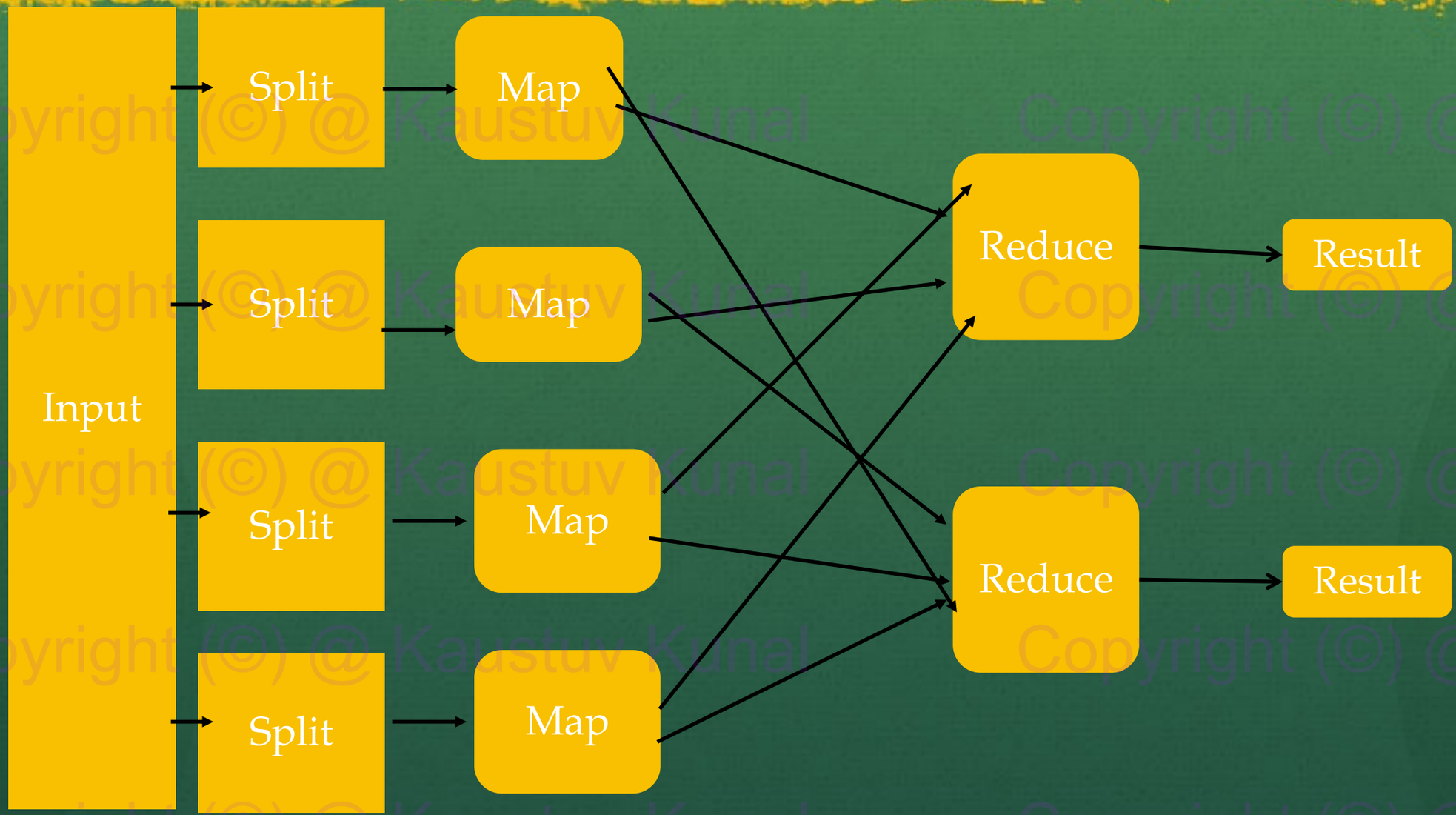
- ✓ A functional programming model developed by Google
- ✓ Processes data in parallel
- ✓ Works on Divide & Conquer principle
- ✓ Instead of taking data to code, takes code to data
- ✓ Processing framework of Hadoop

MapReduce Basics

- ✓ MapReduce divides data into Input splits before processing*
- ✓ Input split is processed in two sequential phases, first Map and then Reduce
- ✓ Input & output of each phases is key, value pairs
- ✓ Map method takes a key, value pair and generates intermediate key, value pair *Map : (k1,v1) -> k2,v2*
- ✓ Reducer method takes Intermediate key and all the values associated with this key as list and outputs key,value pair ***Reduce(k1, <v1, v2,…,vn>) -> k2,v2***
- ✓ Reducer starts only when all mapper finish execution

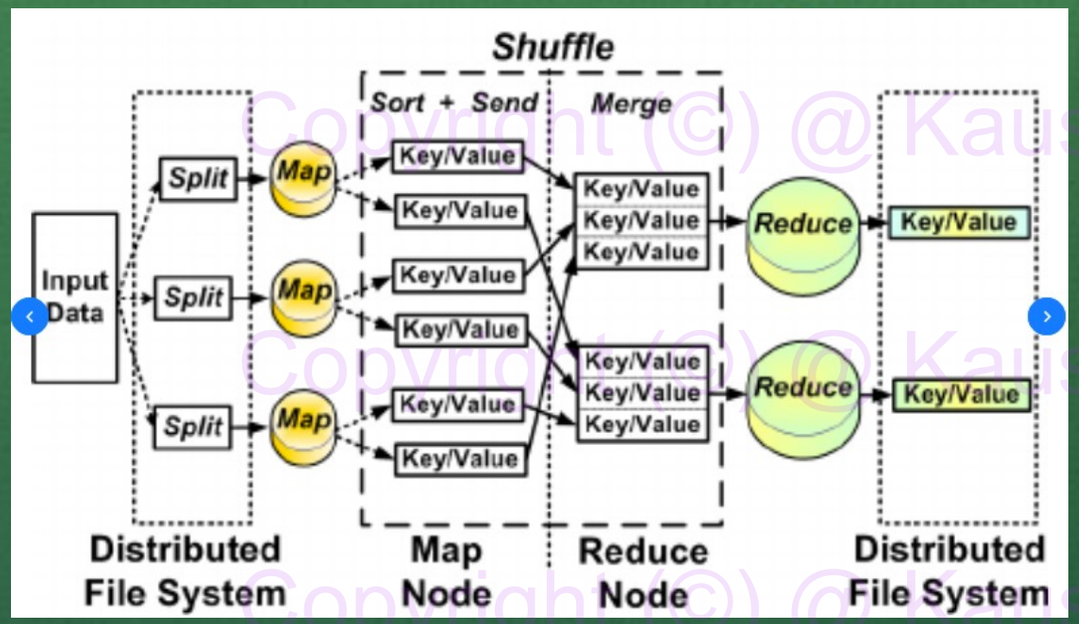
*In HDFS, size of input split is ideally equals to block size

MapReduce Flow

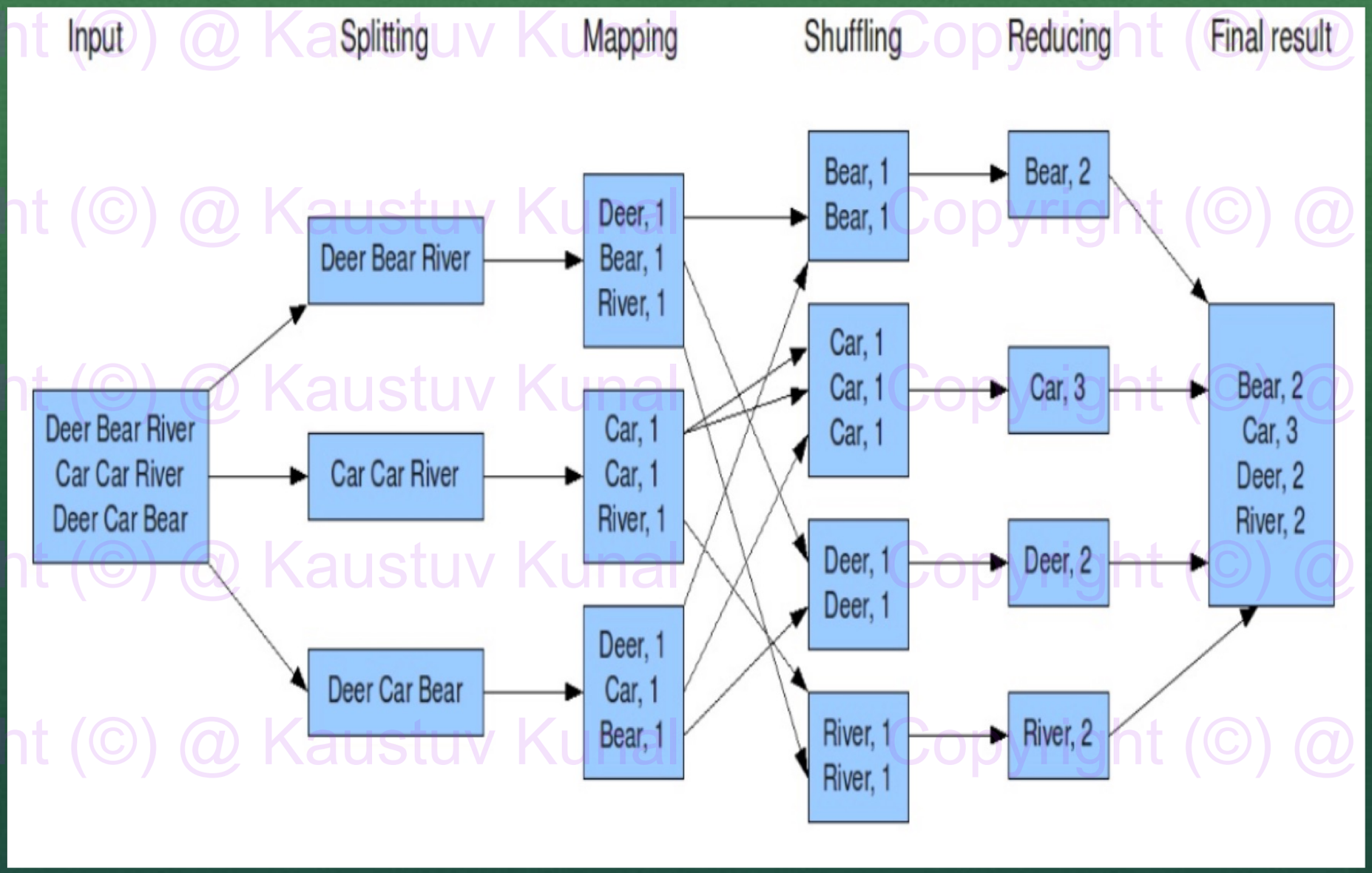


Shuffle-Sort-Merge

- ✓ Before reducer, output of all mappers are merged and sorted key wise
- ✓ No particular ordering on key's value list



MapReduce–WordCount



WordCount Mapper Class

```java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WCMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{

    private final static IntWritable one  = new IntWritable(1);
    private Text                     word = new Text();

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException
    {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens())
        {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```

WordCount Reducer Class

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

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

    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);
    }
}
```

WordCount Driver class

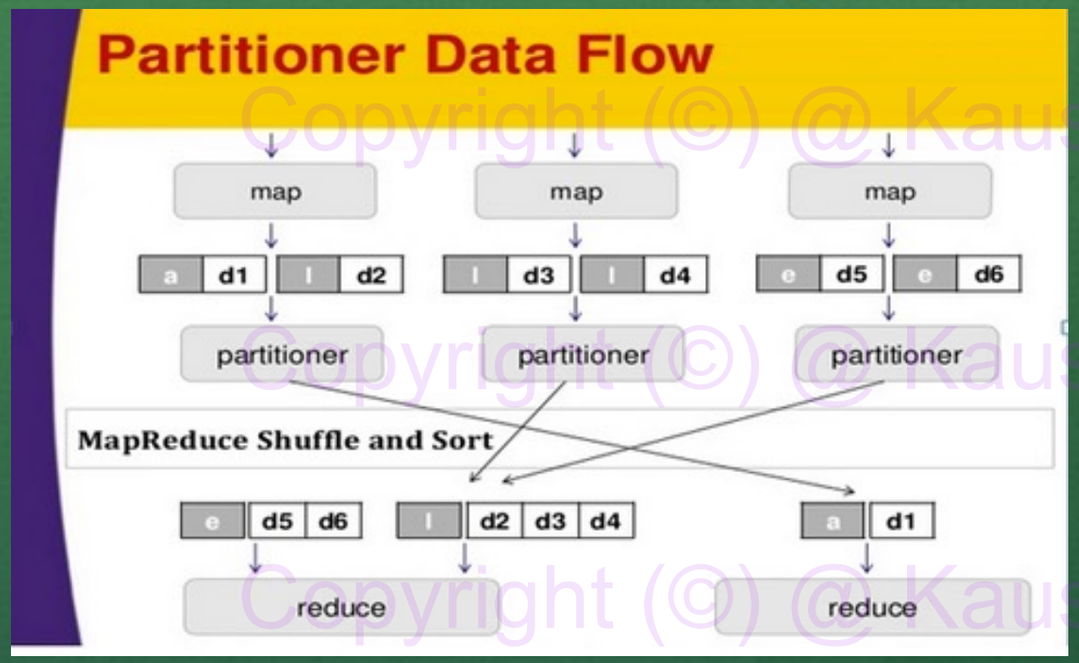
```java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class WCDriver
{
    public static void main(String[] args) throws Exception
    {

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WCDriver.class);
        job.setMapperClass(WCMapper.class);
        job.setCombinerClass(WCReducer.class);
        job.setReducerClass(WCReducer.class);
        job.setNumReduceTasks(1);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        Path inputPath = new Path(args[0]);
        Path outputPath = new Path(args[1]);

        FileInputFormat.addInputPath(job, inputPath);
        FileOutputFormat.setOutputPath(job, outputPath);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

MapReduce Execution Modes

- ✓ **Local mode** : Execute in an IDE locally using hadoop library and single JVM
- ✓ **Psudo distribution mode** : All hadoop daemons are in same machine, daemons use separate JVM
- ✓ **Distribution mode**: Daemons run on different machines and separate JVM

Partitioner

- ✓ Partitioner class partitions the keys of intermediate Map output
- ✓ Ensure identical keys go to same reducer
- ✓ Total number of partitions equal to number of Reducer
- ✓ Default partition is hash function



Custom Partitioner

To implement custom partitioner,

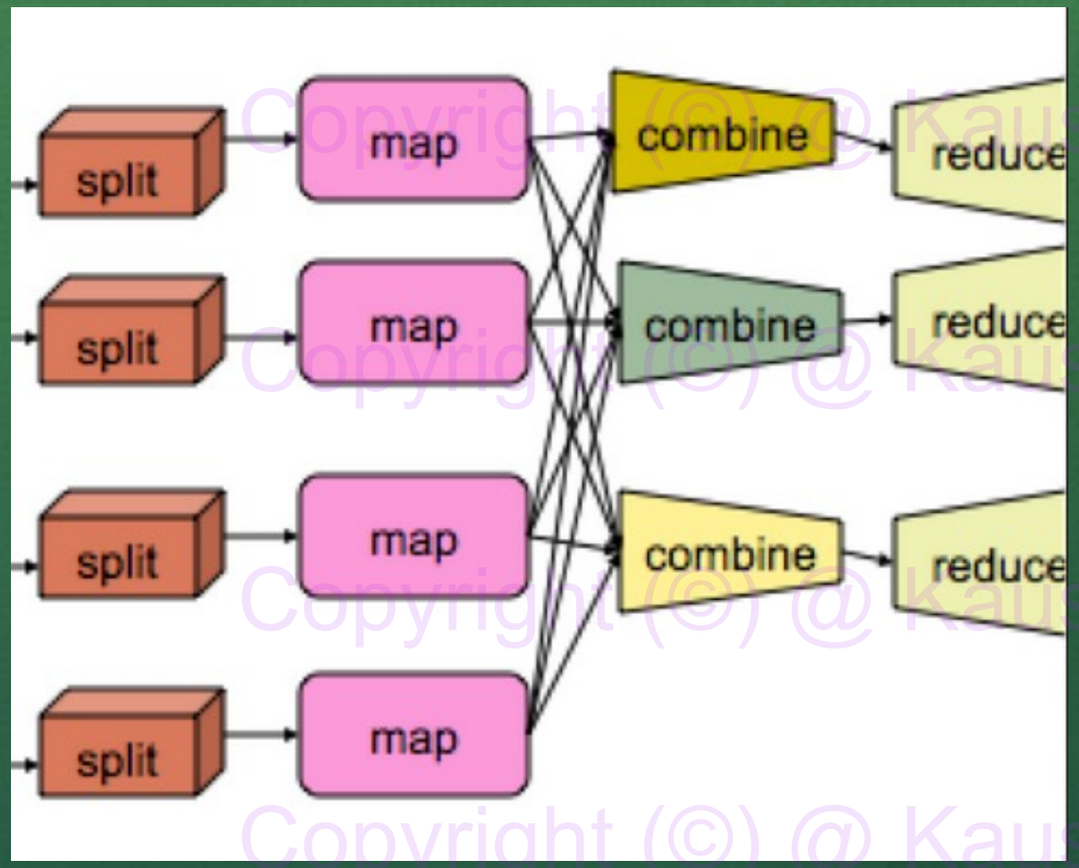
- ✓ Extend partitioner class and implement its *getPartion()* method
- ✓ Specify custom partitioner class inside driver class

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/partitioner

```java
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class MyCustomPartitioner extends Partitioner<Text, Text>
{
    public int getPartition(Text key, Text value, int numReduceTasks)
    {
        if (numReduceTasks == 0)
            return 0;
        if (key.equals(new Text("Male")))
            return 0;
        if (key.equals(new Text("Female")))
            return 1;

        return numReduceTasks;
    }
}
```

Combiner

- ✓ Combiner is Reducer for a single Map task
- ✓ It optimises processing by minimising the amount of data being flown from one node to another.
- ✓ It's input and output key & value type should be same
- ✓ Ideally used if reducer operation is commutative& associative
- ✓ Specify combiner inside driver as *job.setCombinerClass*



Writables

- ✓ In distributed systems, data spend lots of time doing inter node transfer hence undergoes frequent serialisation & de-serialisation
- ✓ Standard java data type are not suitable for this
- ✓ To overcome, hadoop defines their own datatype known as writable
- ✓ WritableCompareble is a writable which is also comparable
- ✓ All MapReduce keys are instance of WritableComparable and all values are instance of Writable.
- ✓ Examples : *IntWritable, FloatWritable,Text etc*

Custom Writable

- ✓ User can write their own writable type by implementing writable interface*
- ✓ Writable interface defines two methods write & readFields
- ✓ WritableComparable interface is a sub interface of the Writable and java.lang.Comparable interfaces.

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/writables

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

public interface WritableComparable
{
public void readFields(DataInput in);
public void write(DataOutput out);
public int compareTo( WritableComparable Obj);
}
```

Counters

- ✓ Counter are facility for MapReduce applications to report its statistics
- ✓ It is useful in problem diagnosis and validation
- ✓ Counter can be either built-in or user defined
- ✓ Figure shows some default built-in counters which mapreduce produces after execution

```
Counters: 17
  Map-Reduce Framework
    Spilled Records=248
    Map output materialized bytes=1489
    Reduce input records=124
    Map input records=72
    SPLIT_RAW_BYTES=92
    Map output bytes=1592
    Reduce shuffle bytes=0
    Reduce input groups=124
    Combine output records=124
    Reduce output records=124
    Map output records=167
    Combine input records=167
    Total committed heap usage (bytes)=321912832
  File Input Format Counters
    Bytes Read=1093
```

User Define Counters

Implement custom counter in below three steps,

1. **Defined** custom counter as java enum type

2. **Process**(increment/decrement) counter inside mapper or reduce.

3. **Print** counter

```
//Declare  Counter

public enum GENDER_COUNTER {
MALE_COUNT,
FEMALE_COUNT};

//Process Counter

if( sex.contains("MALE") ){
context.getCounter(GENDER_COUNTER.MALE_COUNT).increment(1);
}
if(sex.contains("FEMALE") ){
context.getCounter(GENDER_COUNTER.FEMALE_COUNT).increment(1);
}


//Print Counter

Counter cn=job.getCounters();
Counter c1=cn.findCounter(GENDER_COUNTER.MALE_COUNT);
System.out.println(c1.getDisplayName()+":"+c1.getValue());
Counter c2=cn.findCounter(GENDER_COUNTER.FEMALE_COUNT);
System.out.println(c2.getDisplayName()+":"+c2.getValue());
```

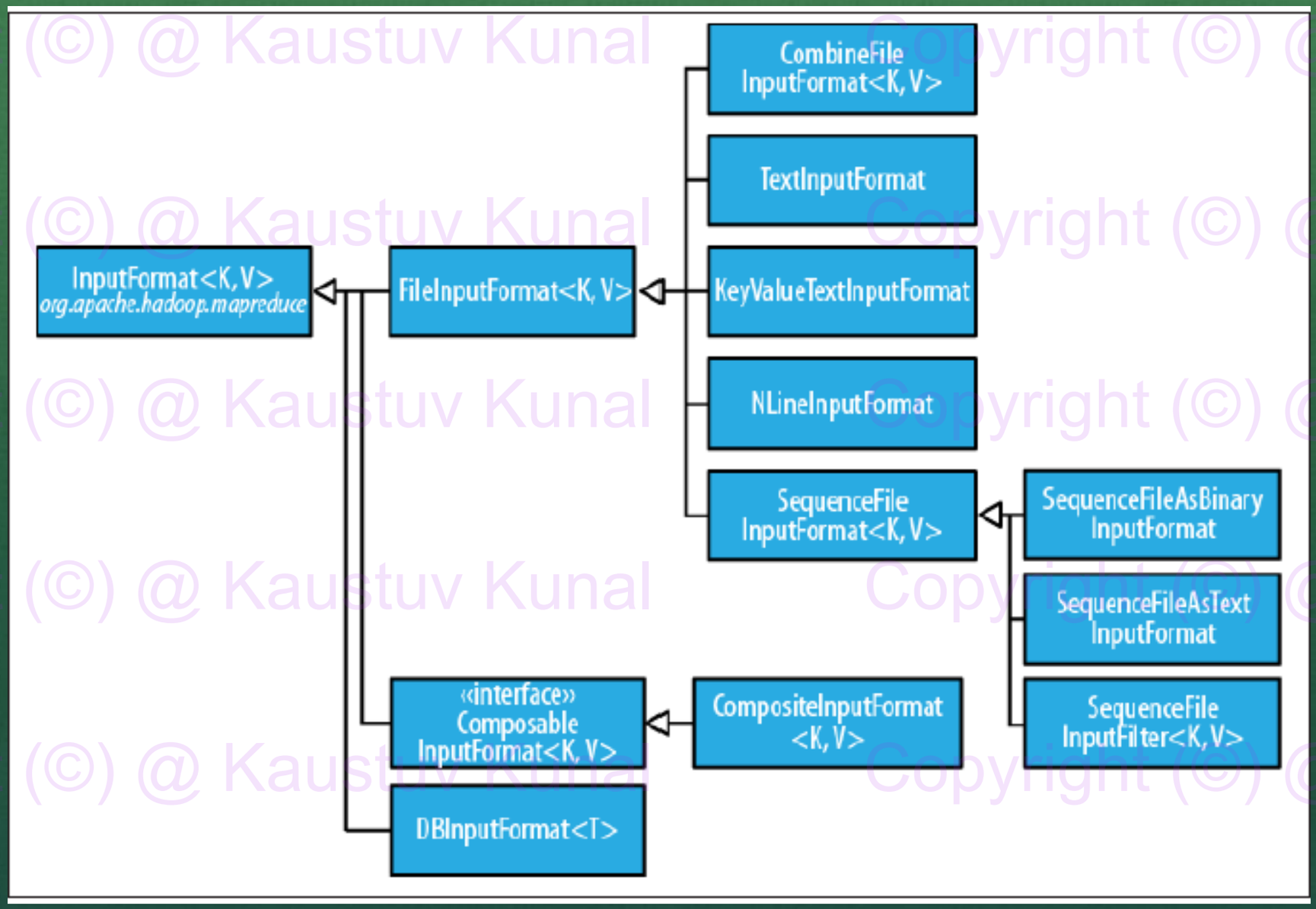
Input Format

- ✓ Input format class converts input into key value pairs
- ✓ Some often used Input format are TextInputFormat, KeyValueInputformat, SequenceFileInputFormat

It defines two methods

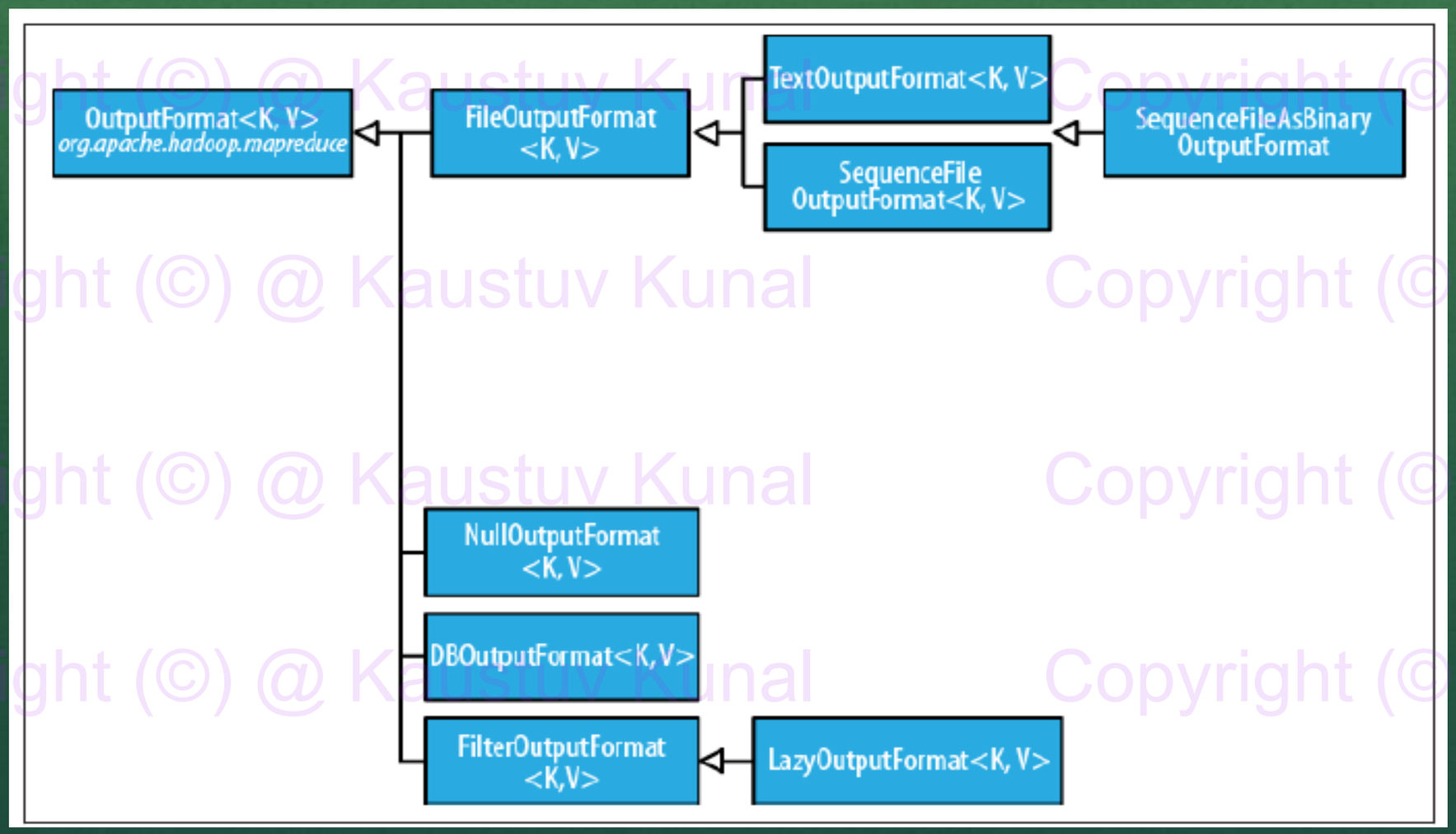
- ✓ *getSplits()* to split the input into records
- ✓ *RecordReader()* to read record as key, value pair

Input Format Class Hierarchy

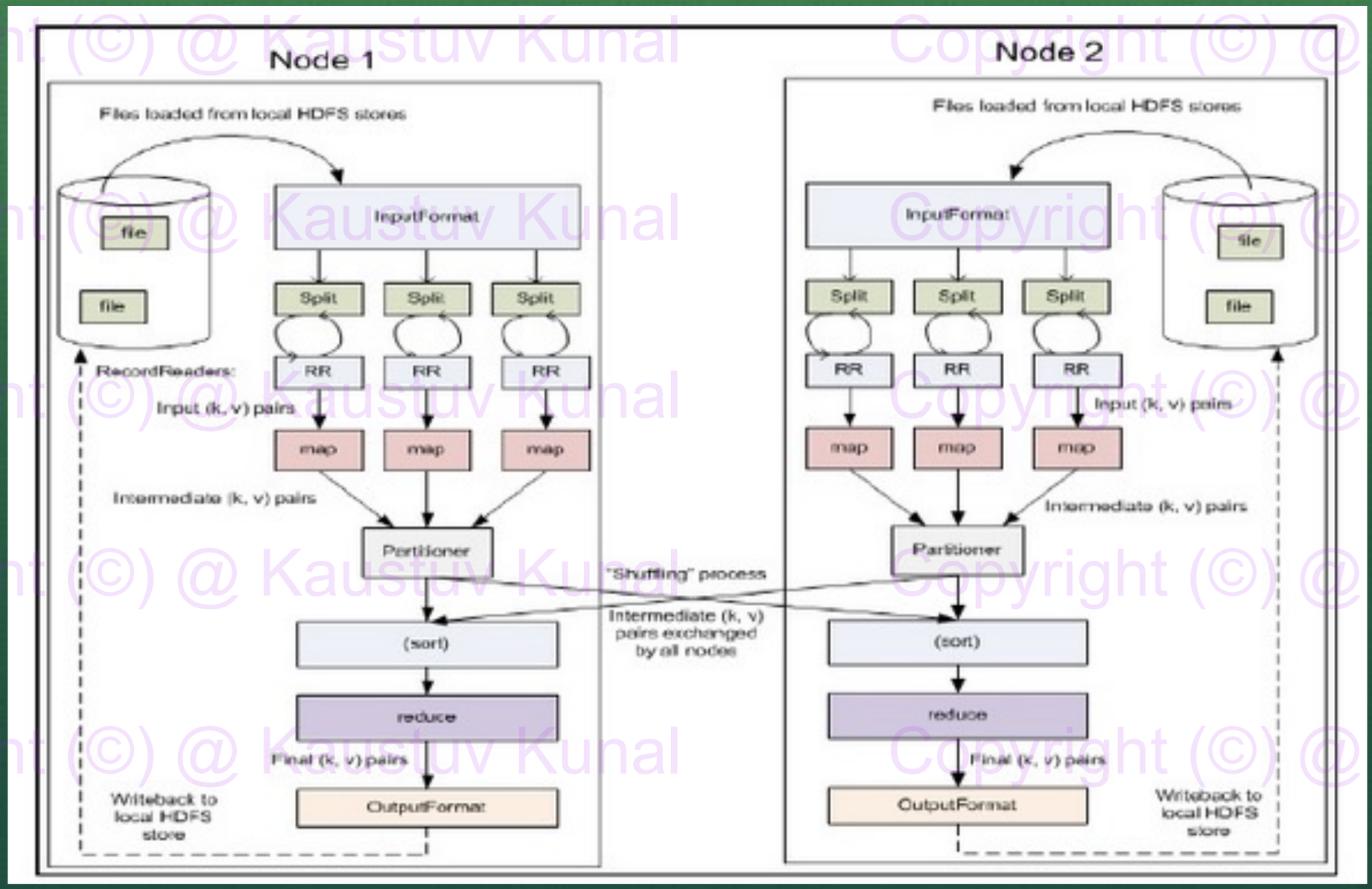


Output Format

- ✓ Output format class writes MapReduce output into a particular format



Better Picture ?



Setup & Cleanup Methods

- ✓ Each Mapper and reducer contains a setup and a cleanup method
- ✓ Mapper's setup method runs before map method is called for first time
- ✓ Reducer's setup method runs before reduce method is called for first time
- ✓ Setup method is useful in initializing data structure, reading data from external file, setting parameters etc.
- ✓ Mapper's clean up method is called after processing of all records by mapper but before termination of mapper
- ✓ Similarly Reducer's cleanup is called after processing of all records by reducer but before termination of reducer

Side Data Distribution

- ✓ Read only data needed by a job in order to process main data is known as side data

- ✓ MapReduce job accesses side data by below two ways,

1. **Job Configuration** : It serializes the data, put all the data inside memory and accessed using context's get configuration method. Use it when side data size is in under few kilobytes

2. **Distributed cache** : Distributed cache provide service to copy side data to the task node. Files are copied to node one per job. File path is specified in driver class as : Job.addCacheArchive(URI) , use for larger side data

MapReduce & Sorting

- ✓ Remember, keys are passed to reducer in sorted order
- ✓ Due to this feature, MapReduce is idle for sorting large data sets
- ✓ Sorting can effectively test hadop systems I/O as well

How to Sort Large Datasets ?

- ✓ Option1: Use single reducer
  - Inefficient for large files
  - Too much load on one node
- ✓ Option2: Partition key space based on insertion order using custom partitioner (e.g. 1-25, 25-50, 50-75, 75-100)
  - Data might be partition uneven
  - Uneven load on nodes
- ✓ Option3: Sample key space to approximate on key distribution then partition the key space

Hadoop comes with auto Samplers & TotalOrderPartioner

Input Sampler

- ✓ Samples keys across all input splits and sorts them using the job's Sort-Comparator
- ✓ It writes a 'partition file' a sequence file* in HDFS to delimit the different partition boundaries based on the sorted samples. For example, if number of reducer is 3 the partition file will have 2 boundary entries
- ✓ Partition file is shared with the tasks running on the cluster as side data
- ✓ Each map output is sorted & partitioned based on these boundaries

Sequence file is a flat file consist of binary key/value pairs

Input Sampler Types

✓ Random sampler: samples randomly based on a given frequency
Random Sampler(freq, numSamples, maxSplitsSampled)

✓ Interval Sampler: samples at every fixed interval
IntervalSampler(freq, maxSplitsSampled)

✓ Split Sampler: takes the first n samples from each split
SplitSampler(numSamples, maxSplitsSampled)

Here,

- **Freq,** is probability that key will be picked from input
- **numSamples,** is the number of samples extracted from input and
- **maxSplitsSample,** is maximum number of input splits that will be read to extract the samples

TotalOrderPartitioner

- ✓ TotalOrderPartitioner class is packed with Hadoop distribution
- ✓ Key objective of TotalOrderPartitioner class is to partition key space based on partition file ranges

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/totalordersort

Secondary Sort

What if our use case require us to sort values also ?

- ✓ Option1 : Sort values inside reducer, faster but memory inefficient
- ✓ Option 2 : Define new key as combination of key &value, perform sorting & grouping of new key in specific order as per business requirement

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/secondarysort

Map-Side & Reduce-Side Joins

- ✓ MapReduce can be use to join two dataset
- ✓ When the join operation is performed in map phase it is Map-Side join and when it is performed in reduce phase it is Reduce-Side join.
- ✓ Map-Side join is faster as data is not going through sort and shuffle phase but requires precondition like data should be pre sorted & equally partitioned. It is ideal for small tables or when one table is side data
- ✓ Reduce-Side join is ideal for joining two large size tables

Compression

- ✓ Compression reduces space and speedup execution
- ✓ Facilities are available to compress intermediate Map output and final Reducer output
- ✓ Hadoop comes with many compression codec classes
- ✓ Compression codec classes are available for Gzip, BBZIP2, LLZO ,LZ4, Snappy type compression

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/maxtemp

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

Speculative Execution

- ✓ Remember, reducer task wait until all mapper finish execution
- ✓ Failed or slow mapper node delays the whole MapReduce job
- ✓ Speculative excution, a job level property, if set then job ensures that after certain time, task start executing at different node on same data set
- ✓ The task which completes first is taken and another one is discarded (killed)
- ✓ Speculative excution can be set for both mapper and reducer task
  conf.set("mapreduce.map.speculative", "true");
  conf.set("mapreduce.reduce.speculative", "true")

Job Chaining

- ✓ An MR job output can be input to another MR job

```
public class  ChainJob
{
    public static void main(String[] args) throws Exception
    {
        Configuration conf = new Configuration();
        Job job1 = Job.getInstance(conf," Job1 ");
         . . .

        int code = job1.waitForCompletion(true) ? 1 : 0;
        if (code == 1)
        {

            Job job2 = Job.getInstance(conf,
                    " Job2");
            ...
            System.exit(job2.waitForCompletion(true) ? 0 : 2);
        }

    }
}
```

MR-Unit

- ✓ Apache MRUnit is a Java library that helps developers unit test Apache Hadoop map reduce jobs

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/test/java/com/kk/test/mapreduce

MapReduce Use Cases

1. Find Top-N-Elements from a large data set
2. Sort item from large datatest (using TotalOrderPartitioner and Input sampler)
3. Find common (facebook) friends
4. Processing large number of small file in hadoop using combine file input format
5. Finding yearly maximum temperature using secondary sort

1. Finding Top-N-Elements

MapReduce is highly useful in when we need to fetch Top-N-elements from large data set, for example

- ✓ top 10 salaries or top 50 highest paid employee from orgnisations dataset,
- ✓ Top 10000 highest tax payers of the country,
- ✓ Top hash-tag tweets of the month,
- ✓ Top 5 tweeted candidate in an election etc..

https://github.com/kaustuvkunal/Big-Data/edit/master/map-reduce/src/main/java/com/kk/mapreduce/topnproblem/

2. Sorting Large Dataset

- ✓ MapReduce is an ideal framework for sorting large dataset
- ✓ Hadoop distributions is packaged with input sampler and TotalOrderPartitioner specially for sorting tasks.
- ✓ We can sort dataset using custom partitioner (when key frequency is know to us) or using TotalOrderPartitioner
- ✓ Example of both methods, where we have fetched sorted country names from a geographical data can be found

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/totalordersort

3. Common Friends

- ✓ One use case of MapReduce is in finding common Facebook friends, given a data set of person and her friends list.

- ✓ The sample solution is provide here

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/commonfriends#reducer-input

4. Processing Small Files in Hadoop

- ✓ Hadoop is suitable for processing large files. What if we have many text files of small sizes ? With Text-Input-Format, input split will process one small file which is not efficient
- ✓ Solution is to define a FileWritable to take file name along with its offset as key and use Combine-File-Input-Format which will pack multiple files into the same split

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/maxtempusingcombineinputformat

5. Max Temp using secondary sort

A famous MapReduce application program is finding yearly maximum temperature using secondary sort, which beautifully demonstrates usage of ,

- ✓ Custom Writable
- ✓ Key comparator
- ✓ Group Comparator

https://github.com/kaustuvkunal/Big-Data/tree/master/map-reduce/src/main/java/com/kk/mapreduce/secondarysort

References

- Apache MapReducehttps://hadoop.apache.org/docs/r2.7.5/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html
- Google Fie Systems https://ai.google/research/pubs/pub51
- Hadoop-The Definitive Guide -4th Edition