**UNIT –IV MAP REDUCE APPLICATIONS**

**MapReduce and its work flow**

MapReduce is a software framework for processing (large1) data sets in a distributed fashion over a several machines. The core idea behind MapReduce *mapping* your data set into a collection of <key, value> pairs, and then *reducing* overall pairs with the same key.

**A MapReduce Workflow**

When we start a map/reduce workflow, the framework will *split* the input into

segments, passing each segment to a different machine. Each machine then runs the

*map script* on the portion of data attributed to it.

The *map script* (which you write) takes some input data, and maps it to <key, value>

pairs according to your specifications. The purpose of the map script is to model the data into <key,value> pairs for the reducer to aggregate.

Emitted <key, value> pairs are then “shuffled, which basically means that pairs with the same key are groupedand passed to a single machine, which will then run the *reduce script* over them.

The *reduce script* (which you also write) takes a collection of <key, value> pairs and

“reduces” them according to the user‐specified reduce script.

1. In Map-Reduce Programming , Jobs(Applications) are spilt into a set of map tasks and reduce tasks .Then these tasks are executed in a different fashion on Hadoop cluster.
2. Each task processes small subset of data that has been assigned to it.This way Hadoop distributes the load across the cluster.MapReduce job takes a set of files that is stored in HDFS as input.MAP Task takes care of loading, parsing, transforming and filtering.
3. The responsibility of Reduce task is grouping and aggregating data that is produced by Map tasks to generate final output.Each MAP task is broken into following phases :

RecordReader

Mapper

Combiner

Partitioner

1. The output produced by Map task is known as intermediate keys and values.These intermediate keys and values are sent to reducer .The reduce tasks are broken into following phases:

Shuffle

Sort

Reducer

Output Format

1. Hadoop assigns map tasks to the Datanode where the actual data to be processed resides.This way , Hadoop ensures data locality. Data Locality means the data is not moved over network; only computational code is moved to process data which saves network bandwidth.

**Record Reader**- Converts a byte oriented view of the input into a record oriented view and presents it gto mapper tasks.It presents tasks with keys and values.

**MAP**- Map Functions works on the key value pair produced by RecordReader and generates zero or more intermediate key-value pairs.The MapReduce decides the key value pair based on the context.

**The MAP Side** –

When map function starts producing output,it is not simply written to the disk but it includes buffering writes and some presorting.Each map writes output to a circular memory buffer **(default size 100 MB)** assigned to it. When the contents of the buffer reaches a certain threshold size , a background thread will start to spill the contents to disk. Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete.Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to.  
  
Each time the memory buffer reaches the spill threshold, a new spill file is created, so after the map task has written its last output record, there could be several spill files. **Before the task is finished, the spill files are merged into a single partitioned and sorted output file**.

**Combiner**

It is an optional function but provides high performance in terms of network bandwidth and disk space.It takes intermediate key-value pair provided by mapper and applies user specific aggregate function to only that mapper.It is also known as local reducer.

**Partitioner**

The partitioner takes intermediate key-value pairs produced by the mapper, splits them into shard and sends the shard to the particular reducer as per the user-specific code.

**Reducer**

* Shuffle and Sort – This phase takes the output of all the partitioners and downloads them into the local machine where the reducer is running.Then these individual data pipes are sorted by keys .
* Reduce- Takes the grouped data , applies reduce function and processes one group at a time .The reducer function iterates all the values associated with that key.Reducer operation provides various operations such as aggregation, filtering and combining data.Once it is done , the output of reducer is sent to the output format.

**The Reduce Side**

The reduce task needs the map output for its particular partition from several map tasks across the cluster. The map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes. This is known as the copy phase of the reduce task. The reduce task has a small number of copier threads so that it can fetch map outputs in parallel. The default is five threads, but this number can be changed by setting the mapred.reduce.parallel.copies property.

* The map outputs are copied to the reduce task JVM’s memory if they are small enough (the buffer’s size is controlled by mapred.job.shuffle.input.buffer.percent, which specifies the proportion of the heap to use for this purpose); otherwise, they are copied to disk. When the in-memory buffer reaches a threshold size **(controlled by mapred.job.shuffle.merge.percent)** or reaches a threshold number of map outputs **(mapred.inmem.merge.threshold)**, it is merged and spilled to disk. If a combiner is specified, it will be run during the merge to reduce the amount of data written to disk. As the copies accumulate on disk, a background thread merges them into larger, sorted files. This saves some time merging later on. Note that any map outputs that were compressed (by the map task) have to be decompressed in memory in order to perform a merge on them.
* When all the map outputs have been copied, the reduce task moves into the sort phase **(which should properly be called the merge phase, as the sorting was carried out on the map side)**, which merges the map outputs, maintaining their sort ordering. This is done in rounds. For example, if there were 50 map outputs and the merge factor was 10 (the default, controlled by the **io.sort.factor** property, just like in the map’s merge), there would be five rounds. Each round would merge 10 files into one, so at the end there would be five intermediate files. Rather than have a final round that merges these five files into a single sorted file, the merge saves a trip to disk by directly feeding the reduce function in what is the last phase: the reduce phase. This final merge can come from a mixture of in-memory and on-disk segments.
* During the reduce phase, the reduce function is invoked for each key in the sorted output. **The output of this phase is written directly to the output filesystem, typically HDFS**. In the case of HDFS, because the [**tasktracker node**](http://dailyhadoopsoup.blogspot.in/2013/12/two-types-of-nodes-that-control-job-of.html) (or node manager) is also running a datanode, the first block replica will be written to the local disk.

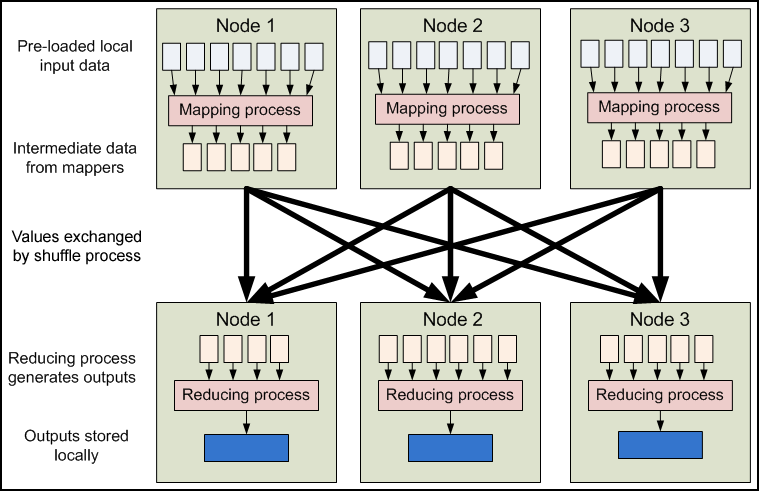
**Shuffle and Sort**

MapReduce makes the guarantee that the input to every reducer is sorted by key.The process by which the system performes the sort and transfers the map outputs to the reducers as inputs is known as shuffle.In many ways, the shuffle is the heart of MapReduce.

**Output Format**

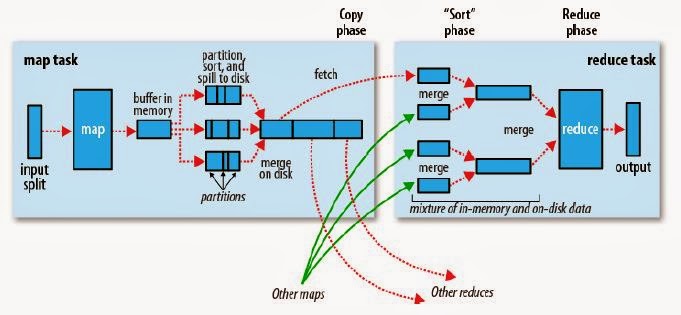
* The output format separates key-value pair with tab (default) and writes it out to a file using record writer.

MAP REDUCE : DATA FLOW



Shuffle Sort

MapReduce makes the guarantee that the input to every reducer is sorted by key.The process by which the system performes the sort and transfers the map outputs to the reducers as inputs is known as shuffle.In many ways, the shuffle is the heart of MapReduce.

[](http://1.bp.blogspot.com/-fipawFd9_Ts/Usrx42dJekI/AAAAAAAAAFI/Mp9b95K6nSI/s1600/has.JPG)

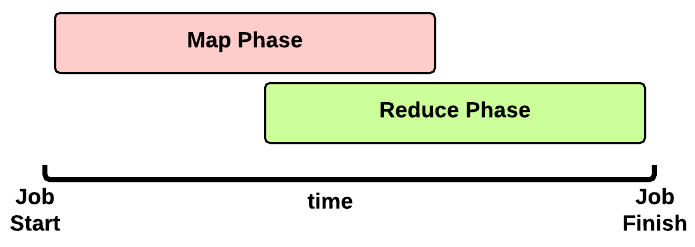
MAP REDUCE FEATURES

* Fine grained Map and Reduce tasks
  + Improved load balancing
  + Faster recovery from failed tasks
* Automatic re-execution on failure
  + In a large cluster, some nodes are always slow or flaky
  + Framework re-executes failed tasks
* Locality optimizations
  + With large data, bandwidth to data is a problem
  + Map-Reduce + HDFS is a very effective solution
  + Map-Reduce queries HDFS for locations of input data
  + Map tasks are scheduled close to the inputs when possible

# Anatomy of a MapReduce Job

In MapReduce, a YARN application is called a **Job**. The implementation of the Application Master provided by the MapReduce framework is called MRAppMaster.

### [Timeline of a MapReduce Job](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "timeline-of-a-mapreduce-job)

 This is the timeline of a MapReduce Job execution:

* **Map Phase**: several **Map Tasks** are executed
* **Reduce Phase**: several **Reduce Tasks** are executed

Notice that the Reduce Phase may start before the end of Map Phase. Hence, an interleaving between them is possible.

### [Map Phase](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "map-phase)

We now focus our discussion on the Map Phase. A key decision is how many MapTasks the Application Master needs to start for the current job.

#### [What does the user give us?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "what-does-the-user-give-us)

Let’s take a step back. When a client submits an application, several kinds of information are provided to the YARN infrastucture. In particular:

* a configuration: this may be partial (some parameters are not specified by the user) and in this case the default values are used for the job. Notice that these default values may be the ones chosen by a Hadoop provider like Amanzon.
* a JAR containing:
  + a map() implementation
  + a combiner implementation
  + a reduce() implementation
* input and output information:
  + input directory: is the input directory on HDFS? On S3? **How many files?**
  + output directory: where will we store the output? On HDFS? On S3?

The number of files inside the input directory is used for deciding the number of Map Tasks of a job.

#### [How many Map Tasks?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "how-many-map-tasks)

The Application Master will launch one MapTask for each map split. Typically, there is a map split for each input file. If the input file is too big (bigger than the HDFS block size) then we have two or more map splits associated to the same input file. This is the pseudocode used inside the method getSplits() of the FileInputFormat class:

num\_splits = 0

for each input file f:

remaining = f.length

while remaining / split\_size > split\_slope:

num\_splits += 1

remaining -= split\_size

where:

split\_slope = 1.1

split\_size =~ dfs.blocksize

Notice that the configuration parameter mapreduce.job.maps is ignored in MRv2 (in the past it was just an hint).

#### [MapTask Launch](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "maptask-launch)

The MapReduce Application Master asks to the Resource Manager for Containers needed by the Job: one MapTask container request for each MapTask (map split).

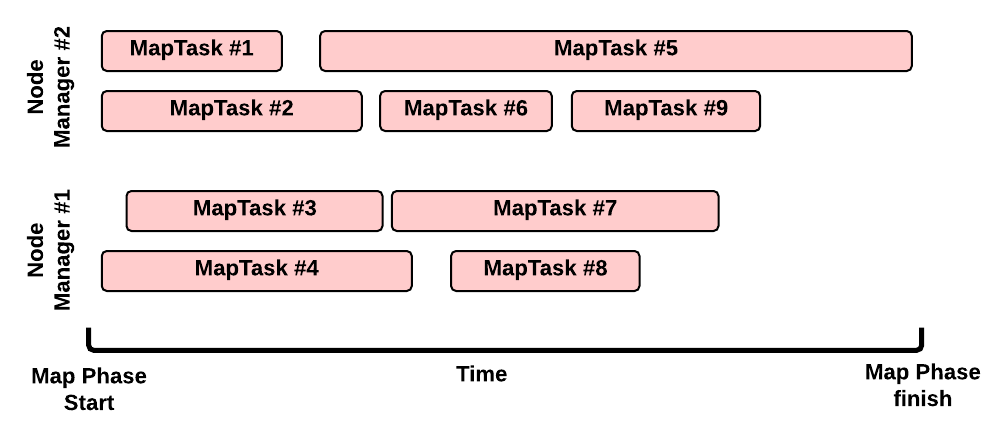
A container request for a MapTask tries to exploit data locality of the map split. The Application Master asks for:

* a container located on the same Node Manager where the map split is stored (a map split may be stored on multiple nodes due to the HDFS replication factor);
* otherwise, a container located on a Node Manager in the same rack where the the map split is stored;
* otherwise, a container on any other Node Manager of the cluster

This is just an hint to the Resource Scheduler. The Resource Scheduler is free to ignore data locality if the suggested assignment is in conflict with the Resouce Scheduler’s goal.

When a Container is assigned to the Application Master, the MapTask is launched.

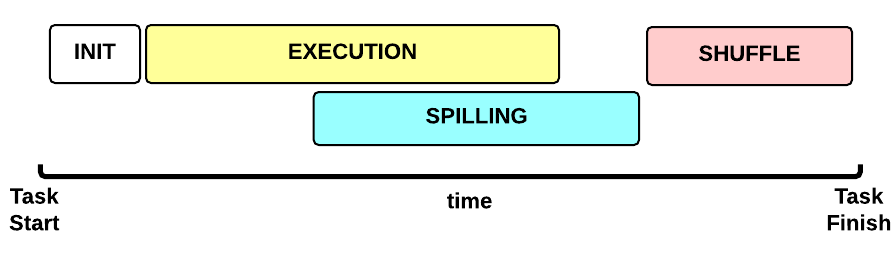
#### [Map Phase: example of an execution scenario](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "map-phase-example-of-an-execution-scenario)



This is a possible execution scenario of the Map Phase:

* there are two Node Managers: each Node Manager has 2GB of RAM (NM capacity) and each MapTask requires 1GB, we can run in parallel 2 containers on each Node Manager (this is the best scenario, the Resource Scheduler may decide differently)
* there are no other YARN applications running in the cluster
* our job has 8 map splits (e.g., there are 7 files inside the input directory, but only one of them is bigger than the HDFS block size so we split it into 2 map splits): we need to run 8 Map Tasks.

#### [Map Task Execution Timeline](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "map-task-execution-timeline)

 Let’s now focus on a single Map Task. This is the Map Task execution timeline:

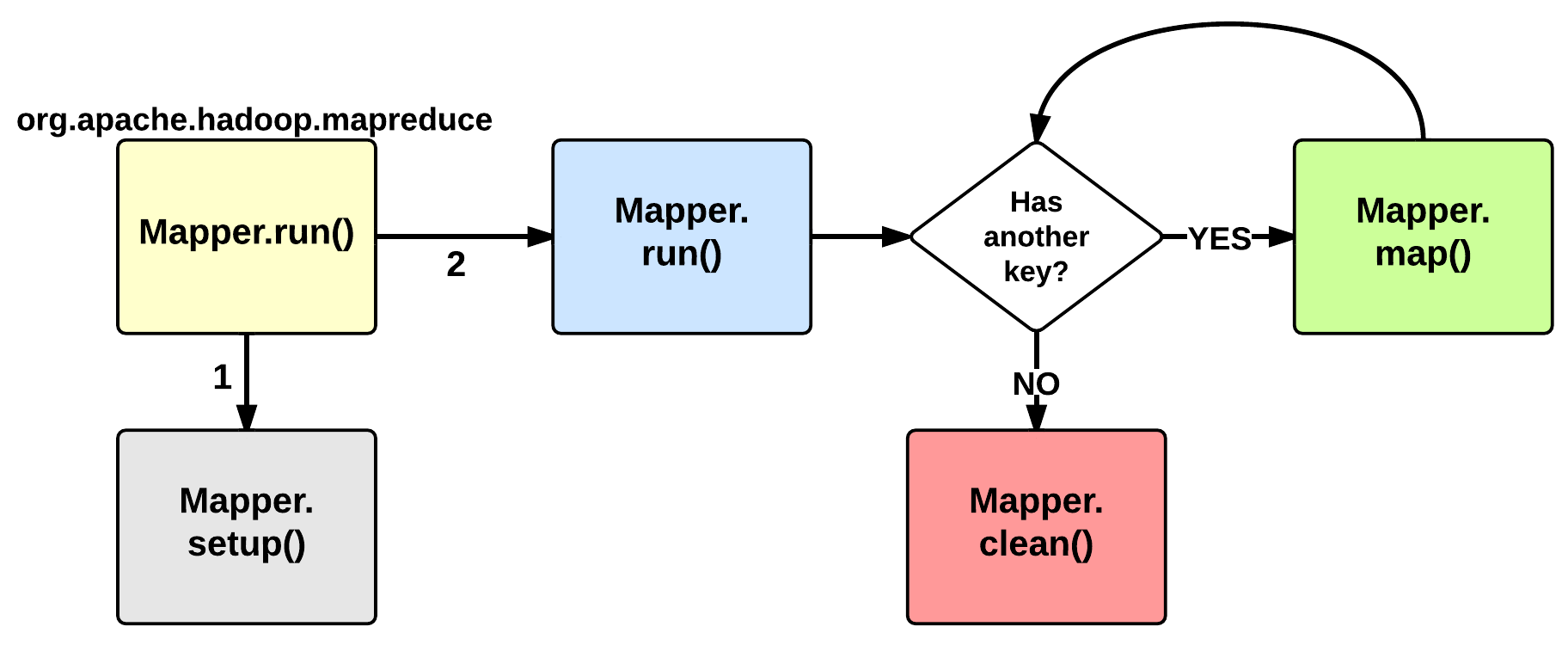
* **INIT** phase: we setup the Map Task
* **EXECUTION** phase: for each (key, value) tuple inside the map split we run the map() function
* **SPILLING** phase: the map output is stored in an in-memory buffer; when this buffer is almost full then we start (in parallel) the spilling phase in order to remove data from it
* **SHUFFLE** phase: at the end of the spilling phase, we merge all the map outputs and package them for the reduce phase

#### [MapTask: INIT](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "maptask-init)

During the INIT phase, we:

1. create a context (TaskAttemptContext.class)
2. create an instance of the user Mapper.class
3. setup the input (e.g., InputFormat.class, InputSplit.class, RecordReader.class)
4. setup the output (NewOutputCollector.class)
5. create a mapper context (MapContext.class, Mapper.Context.class)
6. initialize the input, e.g.:
7. create a SplitLineReader.class object
8. create a HdfsDataInputStream.class object

#### [MapTask: EXECUTION](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "maptask-execution)

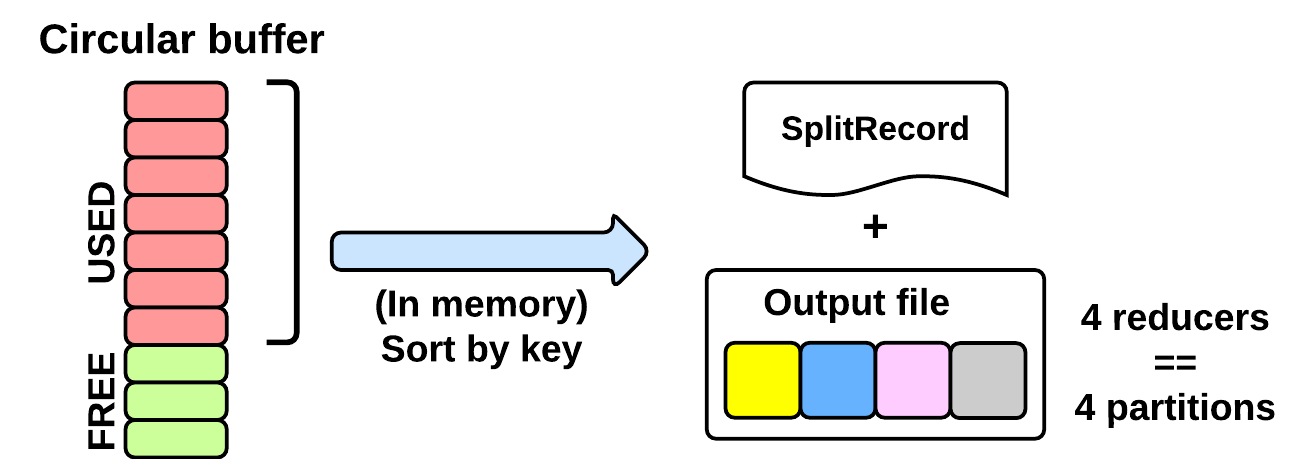


The EXECUTION phase is performed by the run method of the Mapper class. The user can override it, but by default it will start by calling the setup method: this function by default does not do anything useful but can be override by the user in order to setup the Task (e.g., initialize class variables). After the setup, for each <key, value> tuple contained in the map split, the map() is invoked. Therefore, map() receives: a key a value, and a mapper context. Using the context, a map stores its output to a buffer.

Notice that the map split is fetched chuck by chunk (e.g., 64KB) and each chunk is split in several (key, value) tuples (e.g., using SplitLineReader.class). This is done inside the Mapper.Context.nextKeyValue method.

When the map split has been completely processed, the run function calls the clean method: by default, no action is performed but the user may decide to override it.

#### [MapTask: SPILLING](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "maptask-spilling)



As seen in the EXECUTING phase, the map will write (using Mapper.Context.write()) its output into a circular in-memory buffer (MapTask.MapOutputBuffer). The size of this buffer is fixed and determined by the configuration parameter mapreduce.task.io.sort.mb (default: 100MB).

Whenever this circular buffer is almost full (mapreduce.map. sort.spill.percent: 80% by default), the SPILLING phase is performed (in parallel using a separate thread). Notice that if the splilling thread is too slow and the buffer is 100% full, then the map() cannot be executed and thus it has to wait.

The SPILLING thread performs the following actions:

1. it creates a SpillRecord and FSOutputStream (local filesystem)
2. in-memory sorts the used chunk of the buffer: the output tuples are sorted by (partitionIdx, key) using a quicksort algorithm.
3. the sorted output is split into partitions: one partition for each ReduceTask of the job (see later).
4. Partitions are sequentially written into the local file.

##### [How Many Reduce Tasks?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "how-many-reduce-tasks)

The number of ReduceTasks for the job is decided by the configuration parameter mapreduce.job.reduces.

#### [What is the partitionIdx associated to an output tuple?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "what-is-the-partitionidx-associated-to-an-output-tuple)

The paritionIdx of an output tuple is the index of a partition. It is decided inside the Mapper.Context.write():

partitionIdx = (key.hashCode() & Integer.MAX\_VALUE) % numReducers

It is stored as metadata in the circular buffer alongside the output tuple. The user can customize the partitioner by setting the configuration parameter mapreduce.job.partitioner.class.

#### [When do we apply the combiner?](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "when-do-we-apply-the-combiner)

If the user specifies a combiner then the SPILLING thread, before writing the tuples to the file (4), executes the combiner on the tuples contained in each partition. Basically, we:

1. create an instance of the user Reducer.class (the one specified for the combiner!)
2. create a Reducer.Context: the output will be stored on the local filesystem
3. execute Reduce.run(): see Reduce Task description

The combiner typically use the same implementation of the standard reduce() function and thus can be seen as a local reducer.

### [MapTask: end of EXECUTION](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "maptask-end-of-execution)

At the end of the EXECUTION phase, the SPILLING thread is triggered for the last time. In more detail, we:

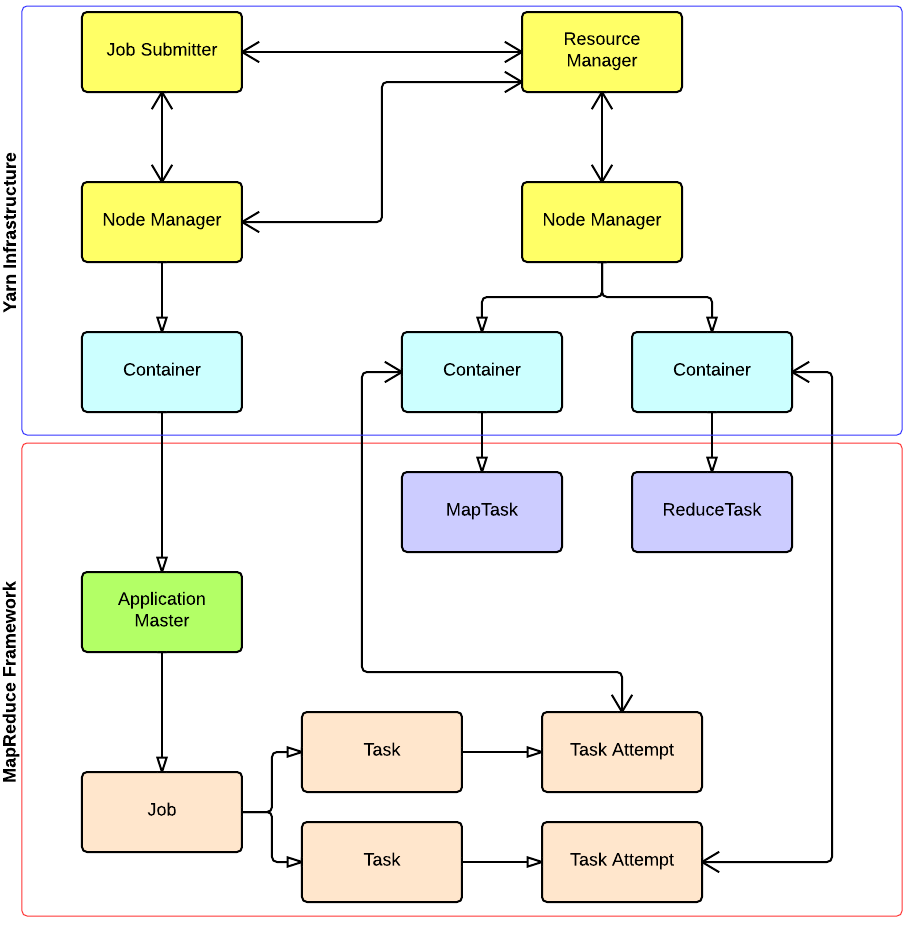
1. sort and spill the remaining unspilled tuples
2. start the SHUFFLE phase

Notice that for each time the buffer was almost full, we get one spill file (SpillReciord + output file). Each Spill file contains several partitions (segments).

#### [MapTask: SHUFFLE](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "maptask-shuffle)

### [Reduce Phase](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "reduce-phase)

[…]

[YARN and MapReduce interaction](http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html" \l "yarn-and-mapreduce-interaction)

Scaling a simple program manually

* Count the number of times each word occurs in a set of documents. In this example , we have a set of documents having only one document with only one sentence:

Do as I say, not as I do.

as 2

Do 2

I 2

Not 1

Say 1

define wordCount as Multiset;

for each document in documentSet {

T = tokenize(document);

for each token in T {

wordCount [token]++;

}

}

Display(wordCount);

For each document, the words are extracted one by one using a tokenization process.A multiset is a set where each element also has a count.This program works fine until the set of documents you want to process becomes large.e.g., you want to build a spam filter to know the words frequently used in the millions of spam emails you have received.Looping through all the documents using a single computer will be extremely time consuming.

Issues with program

* Another flaw with this program is that wordCount are stored in memory.When processing large document sets, the number of unique words can exceed the RAM storage of machine.Looking into growing complexities of wordCount program , we need to add functionalities:

- Store files over many processing machines

- write a disk-based hash table permitting processing without being limited by RAM capacity

- Partition the intermediate data

- Shuffle the partitions to the appropriate machines in phase 2

Scaling the same program with Map-Reduce

* MapReduce programs are executed in two phases: mapping and Reducing.Each phase is defined by a data processing function and these functions are called mapper and reducer.
* Mapper-MapReduce takes input dataand feeds each data element to the mapper.
* Reducer-processes all the outputs from the mapper and arrives at a final result.
* Mapper is meant to filter and transform the input into something that the reducer can aggregate over.
* Partitioning and shuffling are common design patterns that go along with mapping and reducing.

List Key-Value pairs

* In Map Reduce framework you write applicationsby specifying the mapper and reducer.

1. The input to your applications must be structured as a list of (key/value) pairs, list(<k1,v1>).The input format for processing multiple files is usually list(<String filename,String file\_content>).The input format for processing one large file, such as a log file is , list(<integer line\_number,String log\_event>).
2. The list of (key/value) pairs is broken up and each individual (key/value) pair <k1,v1>, is processed by calling the map function of the mapper.
3. The output of all mappers are aggregated into one giant list of <k2,v2> pairs.All pairs sharing the same k2 are grouped together into a new (key/value) pair , <k2,list(v2)>.The framework asks reducer to process each one of these aggregated pairs individually.

Word count Data Flow



Word Count Mapper

*public static class Map extends MapReduceBase implements Mapper<LongWritable,Text,Text,IntWritable> {*

*private static final IntWritable one = new IntWritable(1);*

*private Text word = new Text();*

*public static void map(LongWritable key, Text value, OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException {*

*String line = value.toString();*

*StringTokenizer = new StringTokenizer(line);*

*while(tokenizer.hasNext()) {*

*word.set(tokenizer.nextToken());*

*output.collect(word,one);*

*}*

*}*

*}*

Word Count Reducer

*public static class Reduce extends MapReduceBase implements Reducer<Text,IntWritable,Text,IntWritable> {*

*public static void map(Text key, Iterator<IntWritable> values, OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException {*

*int sum = 0;*

*while(values.hasNext()) {*

*sum += values.next().get();*

*}*

*output.collect(key, new IntWritable(sum));*

*}*

*}*

# Map Reduce Types and Formats

## MapReduce types

* map: (k1, v1) -> list(k2, v2)
* combine: (k2, list(v2)) -> list(k2, v2)
* reduce: (k2, list(v2)) -> list(k3, v3)

Context objects are used for emitting key-value pairs. While it's good to match map output with reduce input, it's not enforced by the Java compiler

The partition operates on the intermediate key and value types (k2 and v2) and returns the partition index.

Input types are set by the input format, the other types are set in the Job. If not set, the intermediates types default to the final output types, so if k2 and k3 are the same there is no need to call **setMapOutputKeyClass()**. Some must be set because some in some aspects the type can be checked at compile time.

### The default MapReduce Job

public int run(String[] args) throws Exception {

Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);

job.setInputFormatClass(TextInputFormat.class);

job.setMapperClass(Mapper.class);

job.setMapOutputKeyClass(LongWritable.class);

job.setMapOutputValueClass(Text.class);

job.setPartitionerClass(HashPartitioner.class);

job.setNumReduceTasks(1);

job.setReducerClass(Reducer.class);

job.setOutputKeyClass(LongWritable.class);

job.setOutputValueClass(Text.class);

job.setOutputFormatClass(TextOutputFormat.class);

return job.waitForCompletion(true) ? 0 : 1;

}

We set the input format as **TextInputFormat** which produces **LongWritable** (current line in file) and **Text** values. The integer in the final output is actually the line number. The map is the default **Mapper** that writes the same input key and value, by default **LongWritable** as input and **Text** as output.

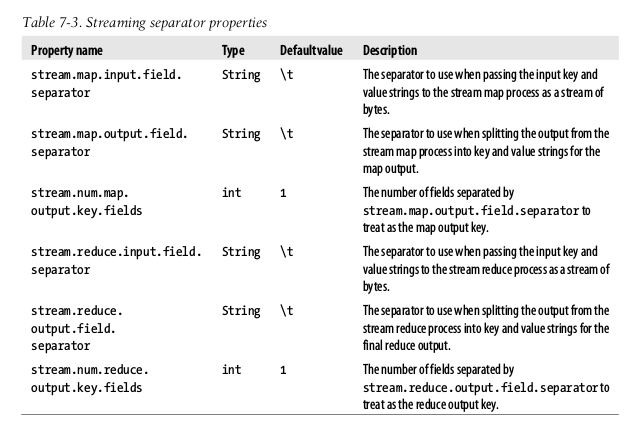
The partitioner is **HashPartitioner** that hashes the key to determine which partition belongs in. There are as many partitions as reducers and we have only one in the default but if not all records will be evenly allocated across reduce tasks and they will share the same key.

The number of map tasks is equal to the number of splits that the input is turned into. The number of reducers will be equal to the number of nodes multiplied by the slots per node mapred.tasktracker.reduce.tasks.maximum. It it's good to have slightly fewer reducers than total slots.

The output key is **LongWritable** and the output value is **Text**. Records are sorted before the reducer.

#### Keys and Values in Streaming

A Streaming app can control the separator used when a key-value pair is turned into a series of bytes (defaults is tab char)



## 

## Input formats

### InputSplits and Records

Each map processes a split that is divided into records. A client running a job calls **InputFormat.getSplits()** to retrieve the **InputSplit** list that is sent to the **JobTracker** to scheduling. On a **TaskTracker** the map passes the split to **InputFormat.createRecordReader()** to obtain the **RecordReader** (that is a iterator). The Map **run()** method:

public void run(Context context) throws IOException, InterruptedException {

setup(context);

while (context.nextKeyValue()) {

map(context.getCurrentKey(), context. getCurrentValue(), context);

}

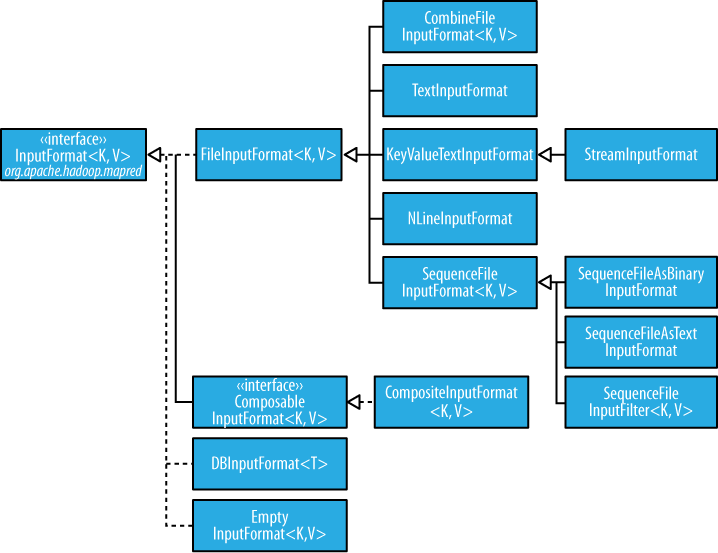
cleanup(context);

}

**run()** method is public and may be customized.

#### FileInputFormat

Base class of **InputFormat** for files. The input is a collection of input paths (file, directory or both)



To exlude certain files from the input, you can use **setInputPathFilter()**, by default it excludes hidden files.

##### FileInputFormat input splits

FileInputFormat splits files over the HDFS block size. You could also set a min and max size.

##### Small files and CombineFileInputFormat

**CombineFileInputFormat** packs many small files into each split. It also takes into account the node and rack locality.

You could also use a **SequenceFile**: keys are filenames and values file contents.

##### Preventing splitting

Some apps don't want files to be split, to avoid it you can increase the min split size to be larger than the largest file or subclass **FileInputFormat** to override the **isSplittable()** to return false.

##### File information in the mapper

A mapper processign a file input split can find information about the split by calling the **getInputSplit()** on Mapper's **Context** object

##### Processing a whole file as a record

First, avoid file splitting and to have a **RecordReader** that delivers the file contents as the value of the record.

### Text Input

#### TextInputFormat

Each record is a line of input. The key, a **LongWritable** is the byte offset (not line number) within the file of the beginning of the line.

##### KeyValueTextInputFormat

To interpret text files where the key needs to be something different than the line offset. For example, a file with key-values separated by commas:

a,2

a,3

b,1

You can specify the separator (the comma) via the **mapreduce.input.keyvaluelinerecordreader.key.value.separator**

##### NLineInputFormat

To send a fixed number of lines to the mapper use NLineInputFormat

##### XML

**StreamXMLRecordReader** is used, setting the input format to **StreamInputFormat** and the **stream.recordreader.class** property to **org.apache.hadoop.streaming.StreamXmlRecordReader**

### Binary Input

#### SequenceFileInputFormat

The common way is to call **SequenceFileInputFormat** that will delivers the corresponding key-values, like IntWritable:Text but a casting is possible that it delivers only Text:Text with **SequenceFileAsTextInputFormat** or BytesWritable objects with **SequenceFileAsBinaryInputFormat**

### Multiple Inputs

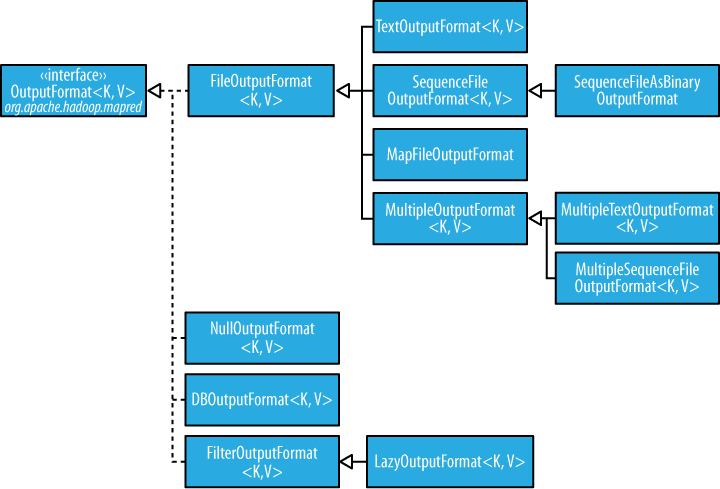
All of the inputs is interpreted by a single **InputFormat** and a single **Mapper**. But when you have different inputs you must use the **MultipleInputs** class that allows to use the **InputFormat** and **Mapper** on a per-path basis.

### Database Input (and output)

**DbInputFormat** is for reading RDBS using JDBC. **DbOutputFormat** is for dumping the results to a DDBB.

**TableInputFormat** is used to read from **HBase**.

## Output Formats



### Text Output

The default **TextOutputFormat** writes records as lines of text tab-separated (that can be changed using mapreduce.output.textoutputformat.separator) calling to the **toString()** method on each record.

### Binary Output

#### SequenceFileOutputFormat, SequenceFileAsBinaryOutputFormat and MapFileOutputFormat

The two classes are to create **SequenceFile** files, with and without compression. **MapFileOutputFormat** is to write MapFiles.

### Multiple Outputs

For example, We will consult data from weather stations we would like to have a file for each station. To do this we need **MultipleOutputs** to write data to files whose names are derived from the output keys and values calling the **MultipleOutputs.write()** in the reducer instead of the Context.

### LazyOutput

**FileOutputFormat** will create output (part-r-nnnnn) files even if they are empty. **LazyOutput** is used to avoid this.

# Unit Test MapReduce using MRUnit

In order to make sure that your code is correct, you need to Unit test your code first. And like you unit test your Java code using JUnit testing framework, the same can be done using MRUnit to test MapReduce Jobs.

MRUnit is built on top of JUnit framework. So we will use the JUnit classes to implement unit test code for MapReduce. If you are familiar with JUnits then you will find unit testing for MapReduce jobs also follows the same pattern.

I will now discuss the template that can be used for writing any unit test for MapReduce job.

To Unit test MapReduce jobs:

* Create a new test class to the existing project
* Add the mrunit jar file to build path
* Declare the drivers
* Write a method for initializations & environment setup
* Write a method to test mapper
* Write a method to test reducer
* Write a method to test the whole MapReduce job
* Run the test

#### **Declare the drivers**

Instead of running the actual driver class, for unit testing we will declare drivers to test mapper, reducer and the whole MapReduce job.

|  |  |
| --- | --- |
| 1  2  3 | MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;  ReduceDriver<Text, IntWritable, Text, IntWritable> reduceDriver;  MapReduceDriver<LongWritable, Text, Text, IntWritable, Text, IntWritable> mapReduceDriver; |

Note that you need to import the following:

|  |  |
| --- | --- |
| 1  2  3 | import org.apache.hadoop.mrunit.mapreduce.MapDriver;  import org.apache.hadoop.mrunit.mapreduce.MapReduceDriver;  import org.apache.hadoop.mrunit.mapreduce.ReduceDriver; |

and not

|  |  |
| --- | --- |
| 1  2  3 | import org.apache.hadoop.mrunit.MapDriver;  import org.apache.hadoop.mrunit.MapReduceDriver;  import org.apache.hadoop.mrunit.ReduceDriver; |

As the word count mapper takes LongWritable offset and Text of line, we have given the same as the generic parameters of the mapDriver. Same is the case with the reduceDriver and mapReduceDriver.