Cloudera Certification Notes

Hadoop 1

MapReduce 5

MapReduce Process 5

Writing MapReduce Jobs 10

Combiners and Partitioners 15

Hadoop API 18

Advanced Topics 20

Algorithms 21

Hadoop Ecosystem 22

Sqoop 22

Pig 23

Hive 24

HBase 25

ZooKeeper 25

Hue 25

Impala 25

Oozie 25

# Hadoop

1. Distributed computing
   1. Motivation for distributed processing
      1. When handling extremely large files, not all the data will fit into memory
      2. Processing large amounts of data may take too long on a traditional system
      3. Handling huge amounts of data may not be cost effective
         1. Memory may not scale
         2. Disk storage systems may no longer be able to keep up at a reasonable cost
   2. Distributed systems
      1. Use multiple machines for single job
      2. Job is divided among the machines
      3. Job is moved to the data
      4. Pieces of the job are run in parallel
   3. Challenges
      1. Programming jobs can be complex
      2. Difficult to keep data and processes in synchronization
      3. Finite bandwidth sharing data and process info across nodes
      4. How does one deal with job failure or hardware failure?
2. The Hadoop framework solves the aforementioned challenges and hides the plumbing so that developers can focus on the analytics.
3. Radical new approach to distributed computing:
   1. Distribute data when the data is stored
   2. Run computation where the data is located
4. Originally based upon work done at Google
5. Open-source project overseen by Apache Software Foundation
6. Terminology
   1. Cluster - a group of computers working together
   2. Node – an individual computer in the cluster
      1. Master nodes control distribution of work and data to the slave nodes
      2. SR
   3. Daemon - a program running on a node
7. Core concepts
   1. Computation is based upon the Map-Reduce programming approach
   2. Nodes talk to each other as little as possible
   3. Data is distributed in advance when it is ingested
   4. Hadoop system scales to “Big Data” - Petabytes
   5. Hadoop system is fault-tolerant
      1. If job fails on one node, it is started again on another node
      2. One or more nodes fail → system not affected
8. Scalability
   1. Adding nodes adds capacity proportionally
   2. Increasing the load results in a graceful decline in performance, not failure of the entire system
9. Fault-tolerance
   1. What happens if a job on a node fails?
      1. The job manager assigns the job to run on a different node
      2. Original job is terminated
   2. What happens if one or more nodes fail due to a hardware/software problem?
      1. System continues to function
      2. Master re-assigns tasks to a different node
      3. Data replication ensures no loss of data
      4. Nodes which recover rejoin the cluster automatically
10. Use Cases
    1. Extremely large amounts of data
    2. Analysis or processing can be done in batch
    3. Analytic algorithm can be performed within the Map-Reduce framework
    4. All or most of the data must be scanned
    5. Data can be written one time and then read / analyzed multiple times
    6. Data typically unstructured - no set schema
11. Architecture of a Hadoop Cluster
    1. Hadoop Cluster: a group of machines working together to store and process data
    2. Two Master nodes
       1. NameNode
          1. Manages HDFS
          2. Knows where the data lives, on which node.
          3. The NameNode daemon has to be running for Hadoop to work.
             1. Must be running at all times.
             2. If the NameNode stops, then the cluster becomes inaccessible
          4. High Availability Mode
             1. Available in CDH4 and later
             2. Two NameNodes - one active and one on Standby
          5. Classic Mode
             1. One NameNode
             2. One “helper” node called the Secondary NameNode
       2. JobTracker
          1. Manages MapReduce
          2. Knows where the jobs are being run.
    3. Any number of Slave nodes
       1. Use HDFS to store data
       2. Run MapReduce jobs
12. HDFS Concepts
    1. HDFS is a filesystem written in Java
    2. Sits on top of native filesystem
    3. Provides redundant storage for massive amounts of data
    4. Files in HDFS are “write-once.” Files can be deleted but not updated.
    5. Ways to Access HDFS
       1. Shell: hadoop fs
       2. Java API
       3. EcoSystem Projects: Flume, Sqoop, Hue
    6. To work with data, you have to load it into HDFS.
       1. hadoop fs -put <source file> <destination file>
          1. This will copy the file to /user/<username>/<destination file>
          2. E.g., for the user pford, hadoop fs –put foo.txt foo-hdfs.txt copies foo.txt to /user/pford/foo-hdfs.txt
       2. SR
    7. More HDFS examples
       1. To get a directory listing of the user’s home directory in HDFS: hadoop fs –ls
       2. To get a directory listing of HDFS root: hadoop fs –ls /
       3. Display contents of HDFS file: hadoop fs -cat /user/username/bar.txt
       4. Copy a file to local disk from HDFS:  
          hadoop fs -get /user/<username>/<hdfs file> <local file>
       5. Create a directory called “input” in the user’s home directory:  
          hadoop fs -mkdir input
    8. HDFS performs best with a modest number of large files.
       1. Why? …. The NameNode stores file meta-data in memory.
       2. Thus, millions of files is better than billions.
       3. Files are typically 100MB or more in size.
    9. HDFS is optimized for large, streaming reads of files
    10. How HDFS stores files
        1. Data files are split into blocks and distributed at load time.
        2. Each block is replicated on multiple data nodes (default 3X).
        3. The NameNode stores metadata about the files in memory.  
           
    11. SR

# MapReduce

## MapReduce Process

1. The test is mostly about MapReduce.
2. MapReduce is a method for distributing a computational task across multiple nodes.
   1. MapReduce breaks complex tasks down into smaller elements which can run in parallel
   2. SR
3. Each node process the data stored on that node (when possible)
4. Consists of two major phases
   1. The “map” - maps keys to values
   2. The “reduce” - reduces all values mapped to the same key
5. Features
   1. Automatic parallelization and distribution of jobs across the cluster
   2. Fault-tolerance
   3. A clean abstraction for programmers
      1. Jobs are usually written in Java
      2. Jobs can also be written in scripting languages using *Hadoop Streaming*
   4. MapReduce abstracts all of the housekeeping away from the developer
6. Terminology
   1. **Job** - a complete execution of Mappers and Reducers over a dataset
   2. **Task** - the execution of a single Mapper or Reducer over a slice of data
   3. **Speculative execution** - when Hadoop schedules another copy of a mapper task in order to speed up a slow running mapper.
   4. **Task Attempt** - particular instance of attempt to execute a task
      1. There will be at least as many task attempts as there are tasks
      2. When a task attempt fails, another is started either by the JobTracker (v1) or by the ApplicationMaster (v2)
      3. Speculative execution can result in more task attempts than completed tasks
7. Stages
   1. Mapper
      1. Mapper takes as input a key/value pair and will output a list of zero or more “intermediate” key/value pairs.
      2. Each map task (typically) operates on a single HDFS block
      3. Map tasks (usually) run on the node where the block is stored
      4. Mapper task intermediate data is stored on local disk, not HDFS.
   2. Shuffle and Sort  
      
      1. Sorts and consolidates intermediate data from all the mappers
      2. Happens after all the map tasks are complete and before the Reduce starts
      3. Intermediate keys are grouped together
      4. Each key and value list is pulled by reducer from the intermediate data zone
         1. All values for intermediate key go to same reducer
         2. The intermediate keys/value lists are passed in sorted order
   3. Reducer
      1. Reducer outputs zero or more final key/value pairs for each input key - written to HDFS.
      2. Pulls shuffled/sorted intermediate data from Map tasks
      3. Operates on this data to reduce to one value per key.
      4. Produces final output, which it writes to HDFS.
      5. In the reduce phase, all items with the same key (from the various mappers) will be processed by the same reducer, guaranteed. Important to know for certain algorithms.
      6. Uses
         1. Summation
         2. Mat functions like max/min, avg.
         3. Collating
         4. Sorting
8. Mappers
   1. Mapper takes as input a key/value pair and will output a list of zero or more key value pairs.  
      
   2. The mapper does not have to use the input key.
      1. Use of the key is optional
      2. Often the mapper’s input key is irrelevant.
   3. If the mapper writes anything out, then the output must be in the form of key/value pairs
   4. Common Uses for Mappers
      1. Data Translation
         1. Convert input into upper case
         2. Convert text language
      2. Data “Explosion”
         1. Read in line of text but write out each instance of a word
         2. Useful in counting and sorting tasks
      3. Data Filtration
         1. Only output key/value pairs where some condition is true
         2. Useful for filtering type of data
         3. For example, getting temperature where it is greater than threshold.
      4. Changing Keyspaces - key mapped to lookup key
9. Reducers  
   
   1. Reducer outputs zero or more final key/value pairs for each intermediate input key - written to HDFS.
   2. Takes an intermediate key, with all of the intermediate values with that key, as input.
   3. Common uses for Reducers
      1. Summation
      2. Math functions: max/min, avg.
      3. Identity – dump out each value with the key
      4. Collating – group values by key
      5. Sorting
   4. How many reducers are needed?
      1. It is important to consider how many reducers should be specified when configuring the job.
      2. The default is a single reducer.
      3. With a single Reducer, one task receives all the keys in sorted order.
         1. This can be advantageous if the output must be in sorted order.
         2. However, this can cause performance issues if there is a large amount of intermediate data.
            1. In that case, the node running the reducer may not have sufficient disk space
            2. Thus, the job may take a long time to run.
      4. Some jobs may require a specific number of reducers, for example:
         1. Batching output by month [0..11] - specify 12 reducers.
         2. Batching output by day or week [0..6] - specify 7 reducers.
         3. Batching messages by hour of the day [0..23] - specify 24 reducers.
      5. Many jobs can be run with a variable number of reducers.
         1. Each reducer should get a reasonable amount of intermediate data, but not too much.
         2. The developer must decide how many to specify by:
            1. Testing the job first with a relatively small test data set,
            2. Then extrapolating to calculate the amount of intermediate data expected from the ‘real’ input data.
            3. Then using that result to calculate the number of reducers that should be specified.
      6. Note: you should take into account the number of Reduce slots likely to be available on the cluster.
         1. If your job requires one more Reduce slot than there are available, then a second wave of Reducers will have to run.
         2. In this way, while increasing the number of Reducers may cut down the time spent for each Reducer, the second wave may add more overall time to the job.
10. MapReduce 1.x vs. 2.x
    1. MapReduce V1
       1. Uses a JobTracker / TaskTracker architecture
       2. One JobTracker per cluster - limits cluster size to about 4000 nodes
       3. Slots on slave nodes designated for Map or Reduce tasks
    2. MapReduce V2
       1. Build on top of YARN
       2. Uses ResourceManager / NodeManager architecture - scalability
       3. Node resources can be used for any type of task
       4. Improves cluster utilization
       5. Support for non-MapReduce jobs
    3. MapReduce v1 Architecture  
       
       1. One JobTacker per cluster that manages MR jobs and distributes tasks to TaskTrackers
       2. TaskTracker - one per slave node that starts and monitors Map or Reduce tasks
    4. Running a Job in a MapReduce v1 Cluster  
       
    5. MapReduce v2 architecture  
       
       1. **ResourceManager** - 1 per cluster, starts ApplicatonMasters and allocates resources on slave nodes
       2. **ApplicationMaster** - 1 per job, requests resources, manages MR tasks
       3. **NodeManager** - 1 per slave node, manages resources on slave
       4. **JobHistory** - 1 per cluster, archives jobs’ metrics and metadata
    6. Running a Job in a MapReduce v2 Cluster  
       
11. Job data
    1. Mapper Data Locality Preference
       1. When possible, Hadoop will attempt to run map tasks on the node where the block of data is stored locally.
       2. If this is not possible, the map task will transfer the data across the network as it processes that data.
    2. Where does the job data live?
       1. Mapper task intermediate data is stored on the local disk (not HDFS)
       2. What about reducers?
          1. There is no concept of data locality for reducers.
          2. Reducers pull the mapper intermediate data across the network.
          3. Reducers write their output to HDFS.
          4. The reduce method won’t start until all intermediate data has been transferred.
          5. However, Hadoop will start to allow reducers to pull intermediate data as soon as the mappers finish their work.
12. Speculative Execution
    1. It is possible for one map task to run more slowly than others:
       1. Faulty hardware
       2. Slow machine
    2. This would create a bottleneck since reduce method won’t start until all mappers have completed.
    3. Hadoop uses ***speculative execution*** to mitigate this condition.
       1. If a mapper appears to be running much more slowly than others, a new instance of the mapper will be started on another machine, operating on the same data (new task attempt for same task)
       2. The results of the first mapper to finish will be used.
       3. Hadoop will kill off the mapper that is still running.

## Writing MapReduce Jobs

1. Components of a MapReduce job
   1. Write a **mapper**.
   2. Write a **reducer**.
   3. Write a **driver** program.
      1. Contains the main method which invokes the mapper(s) and reducer(s).
      2. Specifies job configuration and the type of components to use in the job.
   4. Other possible tasks to write[[1]](#footnote-1)
      1. Combiner
      2. Partitioner
2. The Mapper must extend the abstract Mapper class

|  |
| --- |
| // Example of a concrete Mapper 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 MaxTemperatureMapper |
| extends Mapper<LongWritable, Text, Text, IntWritable> { |
|  |
| @Override |
| public void map(LongWritable key, Text value, Context context) |
| throws IOException, InterruptedException {  .... |
| } |
| } |

* 1. The Mapper class is a generic type with 4 parameters: KEYIN, VALUEIN, KEYOUT, VALUEOT.
  2. Override the map method to do the mapping.
     1. The map method takes the parameters of KEYIN and VALUEIN type, as well as a Context object.
     2. The result, if any, is written to the Context object.
     3. The parameter types must match the parameter types of declared in the concrete class’ parameters—the KEYIN and VALUEIN types.

1. The Reducer must extend the abstract Reducer class.

//Example of a concrete Reducer  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.mapreduce.Reducer;

|  |
| --- |
| public class MaxTemperatureReducer extends Reducer<Text, IntWritable, Text, IntWritable> {    @Override |
| public void reduce(Text key, Iterable<IntWritable> values, Context context)  throws IOException, InterruptedException {  ...... |
| } |

}

* 1. It is also a generic, taking the KEYIN, VALUEIN, KEYOUT, VALUEOUT parameters.
     1. The KEYIN and VALUEIN types must match the KEYOUT and VALUEOUT types of the concrete Mapper class.[[2]](#footnote-2)
     2. SR
  2. Override the reduce method to perform your reduction.
     1. The method takes a KEYOUT, VALUEOUT, and Context object as input.
     2. The parameter types must be the same as those declared in the class’ parameters—the KEYOUT and VALUEOUT.
     3. Any result is written to the Context object.

1. The third essential piece of a MapReduce job is a driver program that runs the job.

// Example of a driver program to run the MR job

public class MaxTemperature {

public static void main(String[] args) throws Exception {  
 ....  
 Job job = new Job();  
 job.setJarByClass(MaxTemperature.class);  
 job.setJobName("Max temperature");  
  
 # Setting input and output paths  
 FileInputFormat.addInputPath(job, new Path(args[0]));  
 FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
 # Setting Mapper and Reducer classes  
 job.setMapperClass(MaxTemperatureMapper.class);  
 job.setReducerClass(MaxTemperatureReducer.class);  
 job.setOutputKeyClass(Text.class);  
 job.setOutputValueClass(IntWritable.class);  
 System.exit(job.waitForCompletion(true) ? 0 : 1);  
 }  
}

* 1. The driver should create a Job object to control the job.
  2. The input and output paths for the job are command line arguments that are added to an InputFormat, like FileInputFormat above.
  3. Set the mapper class and reducer class in the Job object.

1. Package the job in a JAR.
   1. Hadoop will distribute that JAR around the cluster.
   2. When you run the job, you will use the hadoop command line task, specifying the name of the JAR containing the job, followed by the input file as the next parameter and the output file as the last parameter.
2. The data passed to the mapper is specified by an ***InputFormat*** object.
   1. Specified in the driver code
   2. Defines the location of the input data - typically a file or a directory
   3. Determines how to split the input data into ***input splits***
      1. Each mapper deals with a single input split
      2. SR
   4. Creates a ***RecordReader*** object
      1. The RecordReader parses the input data into key/value pairs to pass to the mapper
      2. SR
   5. The default InputFormat is ***TextInputFormat***.  
      
      1. Creates a group of ***LineRecordReader*** objects.
      2. Treats each \n-terminated line of a file as a value.
      3. The key is the byte offset of that line in the file.
   6. Other standard *InputFormat* classes
      1. ***FileInputFormat***: abstract base class used for all file-based InputFormats
      2. ***KeyValueTextInputFormat***: maps \n terminated lines as key[separator] value (tab is default)
      3. ***SequenceFileInputFormat***: binary file of (key,value) pairs plus meta-data
      4. ***SequenceFileAsTextInputFormat***: binary file of (key,value) but maps (key.toString(), value.toString())
3. Hadoop Wrapper Classes
   1. Keys and Values in Hadoop are wrappers objects around primitives
      1. Keys and Values are NOT primitives.
      2. This is done to support serialization for HDFS
      3. Examples:
         1. IntWritable -> int
         2. LongWritable -> long
         3. FloatWritable -> float
         4. DoubleWritable -> doubles
         5. Text -> String
   2. Values are objects which implement the ***Writable*** Interface.
   3. Keys are objects which implement ***WritableComparable*** Interface.
      1. A *WritableComparable* is a Writable which is also Comparable
      2. Two *WritableComparable* objects can be compared to determine their sort order.
      3. Keys must be *WritableComparable* so that they can be passed to the Reducer in sorted order.
      4. Hadoop box classes implement both *Writable* and *Comparable*.
         1. Ex: *IntWritable* is actually a *WritableComparable*.
         2. SR
4. ToolRunner
   1. The job configuration can be set in the job driver; however, doing so reduces portability.
   2. By implementing the Tool interface and extending the Configured class, we can leverage Hadoop’s *Configuration* object via the *GenericOptionsParser*.
      1. Hence, we can specify configuration at run-time via command line.
      2. hadoop jar uber.jar mypackage.MyDriver **-D mapred.reduce.tasks=1** input output
   3. How to use command-line parameters with the –D option[[3]](#footnote-3)
      1. Extend Configured and implement Tool interface
      2. Anything specified in the option -D at command line becomes part of the Configuration:  
         hadoop jar myjob.jar package.MyDriver -D color=yellow inputDir outputDir
      3. Set the values onto the configuration object in main of the driver.

Configuration conf = new Configuration();  
conf.setInt(“favcolor”, “blue”); // blue is default color  
Job job = new Job(conf);

* + 1. Get the parameters within the mapper or the reducer.

Configuration conf = context.getConfiguration();

String myColor = conf.get(“favcolor”, “blue”);

* 1. ToolRunner command-line options[[4]](#footnote-4)
     1. -D flag will override any default or site properties in the configuration but will not override those set in the driver code.
     2. -D options must appear before any additional program arguments:  
        hadoop jar myjob.jar MyDriver -D mapred.reduce.tasks=10 indir outdir

1. Mapper/Reducer initialization[[5]](#footnote-5)
   1. If you want to perform some initialization prior to Map or Reduce, you can use the setup method.
   2. Some tasks that you may want to perform in setup:
      1. Initialize data structures
      2. Read from external file
      3. Grab parameters from the Configuration
   3. The setup method is run prior to the map or reduce method being called for the first time:  
      public void setup(Context context)
2. Mapper/Reducer teardown.
   1. Use the cleanup method to perform some action after all the records have been processed by your Mapper or Reducer.
   2. The cleanup method is called before the Mapper or Reducer terminates:
   3. Signature:  
      public void cleanup(Context context) throws IOException, InterruptedException
3. Side data distribution
   1. Often you will want to use some side data along with the processing going on in your Mapper or Reducer. For Instance:
      1. Lookup Tables
      2. Dictionaries
      3. Configuration values
   2. Options for accessing side data
      1. Use the Configuration
         1. In the setup method, use setter/getter methods on Configuration to access configuration properties.
         2. This works well for small amounts of data, but won’t scale because:
            1. Job configuration is read by JobTracker, TaskTracker, and child JVM.
            2. Every time configuration is read, all entries read into memory
      2. Use the Distributed Cache
         1. Distributed Cache
            1. A service for copying files and archives to task nodes.
            2. Files are copied in time for tasks to use them when they run.
            3. Files are normally copied to a node once per job.
            4. Files are automatically deleted from slave nodes when the job finishes.
         2. Files in Distributed Cache are read-only.
         3. Distributed Cache is the better method long-term. Very important.
         4. Distributed Cache is the only way to do a join between separate sets of data in Hadoop.
         5. Use ToolRunner to add files to Distributed Cache from command line.
            1. Don’t need to copy files to HDFS first.
            2. Use the -files option to add files:  
               hadoop jar myjob.jar MyDriver -files file1, file2, …, file\_n
         6. The -archives flag adds archived files, and automatically unarchives them on the destination machines.
         7. The -libjars flag adds jar files to the classpath.
         8. Files added to the Distributed Cache are made available in your task’s local working directory.
            1. These files would be accessed from your mapper or reducer the same way that you would read any ordinary local file:  
               File f = new File(“file\_loaded\_into\_cache”);[[6]](#footnote-6)
            2. SR

## Combiners and Partitioners

1. Combiners
   1. Use them to help reduce large amounts of intermediate data.
   2. Mappers often produce large amounts of intermediate data.
      1. That data must be passed to the reducers.
      2. This can result in a lot of network traffic.
   3. One solution is to specify a Combiner
      1. This acts like a ‘mini-Reducer’
      2. The combiner runs locally on a single Mapper’s output.
      3. Output from the Combiner is sent to the reducers.
   4. Combiner and Reducer code are often identical.
      1. For example, in Tom White’s example of a job with Combiner, in Ch. 2, the combiner and reducer are the exact same class—using the same class in job.setCombinerClass() and job.setReducerClass().
      2. This is possible if the operation performed is commutative and associative.

Associative and Communicative operations

* **Associative** - order of operations doesn’t matter as long as the sequence of operators is not changed.
  + addition and multiplication
  + min / max of a set
  + string concatenation
* **Commutative** - order of operands does not change the result
  + addition and multiplication
  + scalar multiplication of vectors
  + certain binary truth functions
  + intersection / union of sets
    1. Either way, the input and output data types for the Combiner/Reducer must be identical.
  1. The Combiner uses the same signature as the Reducer
     1. Input: intermediate key and a list of values
     2. Output: zero or more key/value pairs
     3. The actual method called is the reduce method in the class:  
        reduce(intermediate\_key, [v1,v2, ...vn]) -> (result\_key, result\_value)
  2. Specify the Combiner class to be used in the Job’s Driver routine:

job.setMapperClass(WordMapper.class);  
job.setReducerClass(SumReducer.class);  
**job.setCombinerClass(SumReducer.class);**

* 1. Again, input and output data types for Combiner and Reducer must be identical
  2. The combiner may run more than once on the output from any given Mapper. Therefore, do not put code in the Combiner that could influence your results in case the Combiner runs more than once.

1. Partitioners
   1. The job of the partitioner is to determine to which reducer each intermediate key and its associated value(s) will be sent.  
      
   2. Unless you specify a specific partitioner in the job configuration, then the default partitioner will be HashPartitioner.
      1. Guarantees that all pairs with the same key will go to the same Reducer
      2. Uses the hashcode() method of the key, and modulo with the number of partitions, in order to determine which partition to send a given key/value pair to.
      3. For most randomly-distributed data, this should result in all partitions being of roughly equal size.
   3. To specify a partitioner:  
      job.setPartitionerClass(Class); // instance method
   4. The partitioners feed data into the shuffle and sort.
   5. The reducers pull data from the intermediate results that came from the partitioners.
   6. **The type of partitioner and the configuration of the job determine how many reducers will be present in the job.** [[7]](#footnote-7)
   7. Getting sorted output
      1. **If a job needs to output a file containing all the keys in sorted order, then use a single reducer.**
      2. An alternative is to use the ***TotalOrderPartitioner***.
         1. This partitioner uses an externally generated file which contains information about intermediate key distribution.
         2. It partitions data such that all keys which go to the first reducer are smaller than any which go to the second, and so on…
         3. Therefore, the final output concatenated has the output files in a totally ordered list.
   8. Custom Partitioners
      1. To create your own partitioner custom class, implement the ***Partitioner*** interface

public interface Partitioner<K, V> extends JobConfigurable {  
 int getPartition(K key, V value, int numPartitions);  
}

* + - 1. You want a Partitioner which evenly distributes your map output.
      2. The getPartition method must return a number between 0 (inclusive) and the number of Reducers (exclusive).
      3. Make sure that the function you use to calculate the partition number does not return a negative number (wrap with Math.abs()).
    1. To use the customer Partitioner, configure your partitioner in the driver:  
         
       job.setPartitionerClass(MyCustomPartitioner.class);
    2. If you need to use configuration variables in the Partitioner, then the Partitioner should implement ***Configurable***.

class CustomPartitioner extends Partitioner<K,V> implements Configurable {

private Configuration configuration;

/ / Define your own variables here

@Override

public void setConf(Configuration configuration) {

this.configuration = configuration;

// set up your variables here

}

@Override

public Configuration getConf() {

return configuration;

}

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

// use your variables here

}

….

}

* + - 1. Any Hadoop object that implements *Configurable* allows its setConf() method to be called one time during instantiation.
      2. Thus you can set up variables in the setConf() method which your getPartition() method will then be able to access.
    1. Use cases
       1. The keys for your data are not evenly distributed but skewed in some way.
       2. You have a key that is a custom WritableCompable which contains a pair of values (A, B).
          1. You may need all keys with same value for A to go to same Reducer.
          2. Default partitioner won’t work in this case.
       3. You are performing a secondary sort.
    2. Native Hadoop Partitioners
       1. ***BinaryPartitioner***
          1. Used for when you have keys that are ***BinaryComparable***.
          2. The partition based upon the configurable part of the bytes array returned by BinaryComparable.getBytes().
       2. ***HashPartitioner***
          1. The default Partitioner that uses the key’s hashcode() method
          2. SR
       3. ***KeyFieldBasedPartitioner***
          1. Allows for using only parts of the key for comparison
          2. Programmer would configure the separator that separates the parts of the key.
       4. ***TotalOrderPartitioner***
          1. Used in conjunction with *InputSampler* or some other sampled index.
          2. *TotalOrderPartitioner* uses an external sample index to set the partition.

## Hadoop API

1. Mapper and Reducer are abstract classes.
   1. They both have 4 formal parameters: KEYIN, VALUEIN, KEYOUT, VALUEOUT.
   2. Mapper contains a map method, which you override to create the mapper code.
   3. Reducer contains a reduce method, which you override to create the reducer code.
2. Other important classes in the Hadoop API
   1. ***InverseMapper*** - swaps keys and values
   2. ***RegexMapper*** - extracts text based on a regular expression
   3. ***IntSumReducer*** - adds up all Integer values for a key
   4. ***LongSumReducer*** - adds up all Long values for a key
3. Old vs. New Java APIs
   1. The new API, sometimes referred to as “Context Objects,” was designed to make the API easier to evolve in the future.[[8]](#footnote-8)
   2. The new API is type-incompatible with the old, however, so applications need to be rewritten to take advantage of it.
   3. Differences
      1. The new API favors abstract classes over interfaces, since these are easier to evolve.
         1. This means that you can add a method (with a default implementation) to an abstract class without breaking old implementations of the class.
         2. For example, the Mapper and Reducer interfaces in the old API are abstract classes in the new API.
      2. The new API is in the org.apache.hadoop.mapreduce package (and subpackages). The old API can still be found in org.apache.hadoop.mapred.
      3. The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system.
         1. The new Context, for example, essentially unifies the role of the JobConf, the OutputCollector, and the Reporter from the old API.
         2. SR
      4. In both APIs, key-value record pairs are pushed to the mapper and reducer, but in addition, the new API allows both mappers and reducers to control the execution flow by overriding the run() method.
         1. For example, records can be processed in batches, or the execution can be terminated before all the records have been processed.
         2. In the old API this is possible for mappers by writing a MapRunnable, but no equivalent exists for reducers.
      5. Job control is performed through the Job class in the new API, rather than the old JobClient, which no longer exists in the new API.
      6. Configuration has been unified.
         1. The old API has a special JobConf object for job configuration, which is an extension of Hadoop’s vanilla Configuration object (used for configuring daemons).
         2. In the new API, job configuration is done through a Configuration, possibly via some of the helper methods on Job.
      7. Output files are named slightly differently: in the old API both map and reduce outputs are named part-nnnnn, whereas in the new API map outputs are named part-m-nnnnn, and reduce outputs are named part-r-nnnnn (where nnnnn is an integer designating the part number, starting from zero).
      8. User-overridable methods in the new API are declared to throw java.lang.InterruptedException.
         1. This means that you can write your code to be responsive to interrupts so that the framework can cancel long-running operations gracefully.
         2. SR
      9. In the new API, the reduce() method is passed values as a java.lang.Iterable, rather than as a java.lang.Iterator (as the old API does). This change makes it easier to iterate over the values using Java’s for-each loop construct:  
           
         for (VALUEIN value : values) { ... }

## Advanced Topics

1. IdentityReducer
   1. Doesn’t reduce, but you would use it to make sure that shuffle-and-sort still happens:
      1. Shuffle and sort doesn’t happen without a reducer.
      2. SR
   2. SR
2. Unit testing
3. Counters
4. Sequence files
5. File Compression
   1. Important to know in Hadoop because Hadoop is distributed and compressed files cannot be split in all places, and not all compressed files are splittable.
   2. SR

## Algorithms

1. Things to know
   1. You get a separate JVM for each mapper.
   2. SR
2. Sorting
3. Searching
4. Indexing

# Hadoop Ecosystem

## Sqoop

1. Covered heavily on the exam
2. Imports data from RDBMS to HDFS
   1. It can also export from HDFS to RDBMS
   2. Uses MapReduce to import the data.
      1. No reducer is used, however.
      2. Throttles the number of mappers to avoid overloading the cluster.
   3. Can import just one table, all tables in a database, or just part of a table (i.e., supports the WHERE clause).
   4. Can be used for incremental imports
      1. First import retrieves all rows in a table.
      2. Subsequent imports retrieve all rows created since the last import.
   5. Generates a class file which can encapsulate a row of the imported data
      1. Useful for serializing and de-serializing data in subsequent map-reduce jobs.
      2. SR
3. Basically a mapper job without a reducer.
4. By default, uses 4 mappers.
   1. This number is configurable, however.
   2. SR
5. Uses a JDBC interface
   1. Custom connectors exist for various RDBMSs.
      1. These use the RDBMS’s native protocols instead of JDBC.
      2. And that yields better performance.
   2. Custom connectors are not open source, but are free, and can be found at Cloudera’s site.
6. Syntax[[9]](#footnote-9)
   1. Sqoop <tool-name> [tool-options]
   2. Tools include:
      1. import
      2. import-all-tables
      3. list-tables
      4. export
      5. create-hive-table
   3. Options include:
      1. --connect
      2. --username
      3. --password
   4. Examples
      1. Import a table called employees from a database called personnel in a MySQL RDBMS:

$ sqoop import --username gina --password muff3n! \

--connect jdbc:mysql://database.example.com/personnel \

--table employees

* + 1. Import MySQL DB table employees from DB personnel with ID > 1000:

$ sqoop import --username gina --password muff3n! \

-- connect jdbc:mysql://database.example.com/personnel \

--table employees --where “id > 1000”

* + 1. Export files from HDFS /user/gina/data using 5 mappers and inject their contents into the student1 table in the testDb database:[[10]](#footnote-10)

$ sqoop export --connect jdbc:mysql://localhost/testDb \

--table student1 -m 5 --export-dir /uer/gina/data

## Pig

1. Apache Pig is a tool for data analysis and processing on Hadoop.
2. Pig offers an alternative to writing MapReduce code directly.
   1. You write in “Pig Latin Script”—a data flow language.
   2. Hadoop Pig converts the Pig Latin into one or more MapReduce jobs.
3. Pig has an interactive shell (called Grunt) where you can type statements.
   1. Pig interprets each Pig Latin statement as you type it.
   2. Execution is delayed until output is required.
   3. Useful for ad-hoc data inspection and debugging Pig Latin Scripts.
   4. To start Pig, type pig at the command line.
   5. Useful commands
      1. $ pig -help
      2. $ pig -version
      3. $ pig -execute
      4. $ pig *script.pig*
4. Pig supports many features which allow developers to perform data analysis without having to write Java Map Reduce code.
   1. Joining data sets
   2. Grouping data
   3. Loading non-delimited data
   4. Creation of user-defined functions one can write in Java.
5. Key Concepts
   1. In relational databases, we are used to dealing with relational data (rows, columns, fields).
   2. Pig uses similar concepts, but with different names:
      1. field - single element
      2. tuple
         1. Collection of values
         2. Analogous to a row in RDBMS
      3. bag
         1. Collection of tuples
         2. Analogous to a group of rows in RDBMS (though not analogous a table).  
            

Figure 1: Example of a bag, multiple rows in a table.

1. Pig Latin Scripts
   1. Pig Latin is not like a traditional programming language
   2. Pig Latin is a data flow language - like a data workflow
      1. The flow of data is expressed as a sequence of statements.
      2. SR
   3. The typical Pig Latin script starts by loading one or more datasets into bags, and then creates new bags by modifying those it already has.

## Hive

1. No transactions
2. Don’t study Hadoop streaming for the test.

## HBase

## ZooKeeper

## Hue

## Impala

## Oozie

1. Defines workflows

1. Combiners and partitioners would be used after the mappers and before the shuffle-and-sort stage in the execution flow, and combiners come before partitioners in the flow. [↑](#footnote-ref-1)
2. Tom White, Hadoop: The Definitive Guide, 3rd Edition, O’Reilly, Ch. 2 “Analyzing Data with Hadoop” [↑](#footnote-ref-2)
3. To learn more about configuration, see *Definitive Guide*, Chapter 5. [↑](#footnote-ref-3)
4. Example using configuration - ToolRunner project, AvgWordLength.java [↑](#footnote-ref-4)
5. See the setup method and how it grabs parameter from Configuration in LetterMapper.java [↑](#footnote-ref-5)
6. Hadoop offers a complete API for working with HDFS. See *Definitive Guide* Chapter 3 - The Java Interface. [↑](#footnote-ref-6)
7. See IX.D for more information about how many reducers to use in a job. [↑](#footnote-ref-7)
8. *Hadoop: The Definitive Guide*, Ch.2 [↑](#footnote-ref-8)
9. Knowing Sqoop syntax is very important on the exam. [↑](#footnote-ref-9)
10. Note that the target table must already exist in the database. [↑](#footnote-ref-10)