MapReduce Exercise notes

Exercise 5: Write a wordcount program in C or Java and find the number of occurrences of each word in a file. It is called "helloworld" program in MR. Then, find the time complexity.

→

Wordcount program in MR is called "helloworld" program. These programs take the upper bound of $O(n^2)$.

Exercise 6: Write a wordcount MR job using HDT.

If a MR job is launched in Eclipse, it is submitted to LocalJobRunner (no HDFS and MR daemons running). It is very similar to setting mapred.job.tracker property to "local" for local mode. LocalJobRunner is a class that runs a complete MR environment locally in single JVM. MR environment itself applies MR algorithm to execute local job.

It can execute any number of mappers (dependent on number of IS) and at most one reducer. Therefore, to test and debug our MR job, it is enough to run MR job in Eclipse itself. If you want to experience with HDFS and MR concepts, you must at least use pseudo-distributed mode.

When you run Eclipse in Ubuntu, you need not add any extra program to support running MR job. But, running a MR job in Eclipse on windows need a program (WinLocalFileSystem.java) that allows local file system access permission. This program is included as a property in driver code of MR job.

Ex: create a WinLocalFileSystem.java program and include the following line in driver code.

```
job.getConfiguration().set("fs.file.impl", "WinLocalFileSystem")
```

This line should be included before file input format and file output format.

All in one: you can run this job as it includes mapper, reducer, and driver class in a single java file.

Separate files: mapper, reducer, and driver classes are written in different java file. So, you have to run driver program, which launches mapper and reducer.

Exercise 7: Launch a MR wordcount job on command line.

Write all the jobs in MRv2 hereafter. To compile and create jar: keep the following the programs in current directory (Ubuntu or Windows)

- `WCMRv2Driver.java`
- `WCMRv2Mapper.java`
- `WCMRv2Reducer.java`

$ javac -classpath location-of-jars *.java

```
$javac -classpath /usr/local/hadoop/share/hadoop/common/hadoop-common-
2.7.0.jar:/usr/local/hadoop/share/hadoop/hdfs/hadoop-hdfs-
2.7.0.jar:/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-
common-2.7.0.jar:/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-
client-core-2.7.0.jar:/usr/local/hadoop/share/hadoop/yarn/hadoop-yarn-
client-2.7.0.jar:/usr/local/hadoop/share/hadoop/yarn/hadoop-yarn-common-
2.7.0.jar:/usr/local/hadoop/share/hadoop/common/lib/commons-cli-1.2.jar
*.java
```

$ jar cvf Job.jar *.class

```
$ jar cvf Job.jar *.class
```

Or using shell file

$ vi compile.sh // compiling MR program and convert to jar

```
javac -classpath /usr/local/hadoop/share/hadoop/common/hadoop-common-
2.7.0.jar:/usr/local/hadoop/share/hadoop/hdfs/hadoop-hdfs-
2.7.0.jar:/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-
client-common-
2.7.0.jar:/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-
client-core-2.7.0.jar:/usr/local/hadoop/share/hadoop/yarn/hadoop-
yarn-client-2.7.0.jar:/usr/local/hadoop/share/hadoop/yarn/hadoop-
yarn-common-
2.7.0.jar:/usr/local/hadoop/share/hadoop/common/lib/commons-cli-
1.2.jar   *.java

jar cvf Job.jar *.class
```

$ sh compile.sh **or**

Copy all those supporting Hadoop 2.7.0 jars into a single folder "libfolder" and then give *.jar instead of long -classpath argument.

$ javac -classpath libfolder/*.jar *.java && jar cvf Job.jar *.class

Exercise 8: Hadoop Streaming and Pipes

Write a MR job in python and execute using Hadoop streaming: any language that reads and writes from standard input and output can be connected with MR.

In Hadoop streaming, there is no driver class.
Hadoop pipes is an interface for non-java programming languages to run MR program. It uses sockets for networking.

Exercise 9: Launch a MR wordcount job using maven using Eclipse.

In section 6.6.1, instead of using step 6 to add MR dependent jars manually, we are going to use maven that automatically downloads all MR dependent jars to our Eclipse project. There are no changes in all other steps. To create a maven project,

First create a java project → right click on the java project → new → maven project → give name: myproj, goupid:com, artefact id:new.

In myproj pom.xml: include repository location, where dependent jars exist. Syntax is:

```
<repositories>
      <repository>
      </repository>
</repositories>
<dependencies>
      <dependency>
            <groupId> </groupId>
            <artifactId> <//artifactId>
            <version> <version>
      </dependency>
</dependencies>
```

groupID artefactID version

```
for HDFS and other file systems, IO, security, utility functions…
```
org.apache.hadoop hadoop-common 2.7.0

```
for MR core functionalities
```
org.apache.hadoop hadoop-mapreduce-client-core 2.7.0

```
JHS, MRAppMaster… services
```
org.apache.hadoop hadoop-mapreduce-client-common 2.7.0

```
yarn related functionalities
```
org.apache.hadoop hadoop-yarn-common 2.7.0

Open pom.xml and include the following dependencies. At least, we need common and mapreduce packages. If we do not use yarn package, MR jobs are submitted to local job runner that does not communicate with HDFS and MR services. Ex:

```
<dependencies>

	<dependency>
		<groupId>org.apache.hadoop</groupId>
		<artifactId>hadoop-common</artifactId>
		<version>2.7.0</version>
	</dependency>

	<dependency>
		<groupId>org.apache.hadoop</groupId>
		<artifactId>hadoop-mapreduce-client-core</artifactId>
		<version>2.7.0</version>
	</dependency>

	<dependency>
		<groupId>org.apache.hadoop</groupId>
		<artifactId>hadoop-mapreduce-client-common</artifactId>
		<version>2.7.0</version>
	</dependency>

	<dependency>
		<groupId>org.apache.hadoop</groupId>
		<artifactId>hadoop-yarn-common</artifactId>
		<version>2.7.0</version>
	</dependency>

</dependencies>
```

Save and right click on pom.xml → maven clean → maven build. Once it is done, you can see MR dependencies downloaded to JRE system library. It automatically downloads all dependent jars from internet repository. Therefore, you need not add any dependencies manually.

Exercise 10: Experience with MapReduce wordcount job for small vs large files.

Setup single node or multi-node, and execute MR wordcount job with a small (few 100 MBs) and a large large (10 GB) dataset. Observe the running time in both MR and non-MR wordcount job.

Exercise 13: Browse Hadoop API libraries and grepcode.com

Visit grepcode.com and search MR classes/interfaces source code.
Google Hadoop MR API in https://hadoop.apache.org/docs/r2.7.0/api/

Exercise 14 to 17:

If datatype of map output key and map output value are same with reducer output key and value, enough setting as follows in driver class.

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);
```

If datatype of map output key and map output value are different with reducer output key and value, you have to set explicitly as follows in driver class.

```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);
```

Default mappers: `Mapper, ChainMapper, FieldSelectionMapper, InverseMapper, MultithreadedMapper, RegexMapper, TokenCounterMapper, ValueAggregatorMapper, WrapperMapper.`

Default Reducers: `Reducer, ChainReducer, FieldSelectionReducer, IntSumReducer, LongSumReducer, ValueAggregatorCombiner, ValueAggregatedReducer, WrapperReducer.`

You can implement wordcount job without writing any code manually using the following two lines in driver class:

```
job.setMapperClass(TokenCounterMapper.class);
job.setReducerClass(IntSumReducer.class);
```

Mapper: it is a default Mapper produced by Hadoop MR Framework and automatically picked when no mapper specified in driver class. Mapper class implements default map function that outputs whatever it receives (no datatype is changed). Default mapper in MRv1 is called IdentityMapper. By default, TextInputFformat is used as input format and TextOutputFformat is used as output format. Therefore, byte offset is the key and each line is a value. By default, there is no default combiner but there is a default partitioner (Hash partitioner).

```
public class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>{
      protected void map(KEYIN key, VALUEIN value, Context context) throws
      IOException, InterruptedException{
                  context.write(key,value)
      }
}
```

Identity/default Mapper/Reducer are just like the concept of Identity function in mathematics. Input is not changed and given output as it is. These default classes package are in org.apache.hadoop.mapred.IdentityMapper/IdentityReducer.

InverseMapper: this mapper class swaps its input (key, value) pairs into (value, key) pair.

TokenCounterMapper: tokenizes its input data (splitting into words) and writes each word with value 1 (word, 1) format. Its input key and value can be of any type. However, output key must be text and output value must be IntWritable.

RegexMapper: this mapper class extracts text matching with the given regular expression.

ChainMapper: it can be used to run multiple mappers in sequence. All mapper classes run like a chain such that output of first mapper is given to input of second mapper, and so on until

last mapper. No need to specify the output key/value classes for the ChainMapper, this is done by the addMapper() method for the last mapper in the chain. It is also defined in org.apache. hadoop.mapred.lib.map package.

Reducer: it is a default reducer provided by MR framework and automatically picked when no reducer is specified in driver class. Similar to default mapper, it does not perform any data processing and produces whatever it gets as input key and value as output key and value. Default reducer in MRv1 is called IdentityReducer.

```
public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>{
      protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context
      context) throws IOException, InterruptedException{
                  for(value:values)
                        context.write(key,value)
      }
}
```

IntSumReducer: it outputs sum of integer values associated with each reducer input key.

LongSumReducer: outputs sum of long values per reducer input key.

ChainReducer: permits to run a chain of reducer classes in sequence. Output of first reducer becomes the input of second reducer and so on. It is present in org.apache.hadoop.mapred. lib.lib package.

Exercise 18: Write a combiner for wordcount job and observe the number of input records in counters for reducer before and after using combiner.

In the output counters displayed in console after job execution, you can see the number of records carried over shuffle phase. You will observe the reduction in number of records carried in shuffle and number of records processed in reduce function. Find the difference in CPU time of job also. To enable combiner for wordcount job,

set in driver class

```
job.setCombinerClass(practice.WCMRv2Reducer.class);
```

or pass as argument at runtime

```
$ yarn jar *.jar classname -D mapreduce.job.combine.class=org.apache.hadoop.
mapreduce.lib.reduce.LongSumReduer /input /output
```

Exercise 19: Write a MR wordcount job with multiple reduce tasks and observe the output files and its input load in counters.

Set mapper and/or reducer in driver class.

case 1: setMapper() and no reducer is set. Now, you get the mapper output as sorted version as it passes through shuffle, sort, and group. Though it is grouped, the default reducer prints key,value for all records. Only one output file will be generated: part-r-00000 (with sorted version).

case 2: setReducer(Reducer.class) along with mapper. It means that user defined reducer has been setup but with default Reducer class. It also passes through shuffle sort and group to reducer. So, output is obviously sorted.

case 3: setMapper() and setReducer() but include job.setNumReduceTasks(0). It means that default reducer is also disable. So, no shuffle, sort, and group process performed. Output of mapper is the output of this job. Output folder name will be like part-m-00000, but not sorted. Instead of including job.setNumReduceTasks(0) in driver class, you can pass at runtime.

$ yarn jar WC.jar WC -D mapred.job.reduces=0 input output // in MRv2

Exercise 20: Runtime arguments passing via command-line. Ex: find words that occur 2 to 4 times.

To pass configuration properties via command line arguments, driver class should extend Configured class and implement Tool interface.

$ hadoop jobname.jar classnmae -D *property=value*

In Eclipse, you can pass property: run → run configurations → select appropriate driver program → enter -D mapred.reduce.tasks=0 input.txt output

If you run job in Ubuntu command line:

$ yarn jar job.jar WCDriverWithGOP -D mapred.reduce.tasks=0 /input.txt /output

In both cases, generic options is specified right before input and output file specification. Now, how to set/read property via driver code.

Setting Property

MRv1: in driver class

```
JobConf job = (JobConf) getConf();
job.set("test","123");
```

MRv2: in driver class

```
Configuration conf = new Configuration();
conf.set("test", "123");
Job job = new Job(conf);
```

or

```
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "word count");
job.getConfiguration().set("test", "123");
```

Getting Property

MRv1: Using the old API in the Mapper and Reducer.

```
private static Long N;
N = Long.parseLong(job.get("test"));
```

The variable N can then be used ine map and reduce functions.

MRv2: Using the new API in the Mapper and Reducer.

```
Configuration conf = context.getConfiguration();
String param = conf.get("test"); // property type is interpreted while reading
context.getConfiguration().get("property_name")
```

There are also typed versions of these methods: getBoolean, getInt, getFloat…
int i= getInt("file.size");

Do not confuse this with JVM configuration as follows:

$ hadoop jobname.jar classnmae -D*property=value*

Observe there is no space between -D and property. JVM properties are looked up using java.lang.System class, whereas Hadoop properties are looked up using Configuration object with GenericOptionsParser.

Exercise 22: Controlling IS size by suing NLineInputFormat (fixing number of lines per map task)

If you want your mapper to receive a fixed number of lines in IS, use NLineInputFormat. If N is set to 1 (by default), each mapper receives exactly one line in IS. mapreduce.input.lineinputformat.linespermap property controls the value of N. Ex:

`Input.txt` contains four lines

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

If, N is 2, each IS gets two lines. Therefore, two mappers are launched. First mapper receives as follows,

(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)

And another mapper will receive another two key-value pairs:

(57, But his face you could not see,)
(89, On account of his Beaver Hat.)

Please note that all mappers will not start key (byte offset from 0). Only first map task starts byte offset 0, all other map tasks offset is calculated from the original input file by input format. The syntax is,

```
job.setInputFormatClass(TextInputFormat.class);
job.setInputFormatClass(NLineInputFormat.class);
NLineInputFormat.setNumLinesPerSplit(job, 100);
```

Exercise 23: Processing a single file regardless of IS size by a mapper

Some applications do not want files to be split as it needs a single mapper to process each input file in its entirely. Ex: Most XML parsers operate on whole XML documents, so if a large XML document is made up of multiple IS, it is a challenge to parse them individually.

Override isSplitable() with false to set number of IS to be 1. Therefore, arbitrary file size itself is an IS regardless of number of blocks of a file. Create the following class along with your other MR job classes.

```
import org.apache.hadoop.fs.*;
import org.apache.hadoop.mapred.TextInputFormat;
public class NonSplittableTextInputFormat extends TextInputFormat {
            @Override
            protected boolean isSplitable(FileSystem fs, Path file) {
                        return false;
            }
}
```

Exercise 24: Writing user-defined counters and display them in the result

Print all the counters after the job done.

```
int resultCode = job.waitForCompletion(true) ? 0 : 1;
System.out.println("Job is complete! Printing Counters:");
Counters counters = job.getCounters();
for ( String groupName : counters.getGroupNames()){
      CounterGroup group = counters.getGroup(groupName);
      System.out.println(group.getDisplayName());
      for (Counter counter : group.getUnderlyingGroup()){
            System.out.println(" " + counter.getDisplayName() +"=" +
            counter.getValue());
      }
}
```

Exercise 25: Logging

```
import java.io.IOException;
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> {
      enum Temperature {
            OVER_100
      }
      private NcdcRecordParser parser = new NcdcRecordParser();

      @Override
      public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
        parser.parse(value);
            if (parser.isValidTemperature()) {
                  int airTemperature = parser.getAirTemperature();
            if (airTemperature > 1000) {
                  System.err.println("Temperature over 100 degrees for
                  input: " + value);
                  context.setStatus("Detected possibly corrupt record: see
                  logs.");
                  context.getCounter(Temperature.OVER_100).increment(1);
            }
      context.write(new Text(parser.getYear()), new
      IntWritable(airTemperature));
    }
  }
}
```

Exercise 26: Too many small files

Solution 1: You can modify the input split size to be bigger.

Solution 2: Combine all files manually and then upload onto HDFS

Solution 3: Many files to one IS using CombineTextInputFormat

1. Create a directory “dir” with many small files
2. Write a wordcount job.

In HDFS, for each input file, at least one block is created even if size is less than a block size.

Initially, pass "dir" as input and check the counters. The number of map tasks is equal to the number of small files present in dir directory.

After including CombineTextInputFormat, pass "dir" and check the counters. Only one map is launched as all small files are put into one IS.

Solution 4: Hadoop archive

Solution 5: SequenceFileInputFormat (check)

FileInputFormats

MapFile: It is actually MapDirectory. It contains index and data file, which are sequence files. Based on index range, we can divide data file into IS to give map tasks if index file contains 1 to 100 keys. If you want only 40 to 60 key to an IS, it is possible using mapFile. It allows random access. You have to set input file format as mapFile. This file format is mainly used for intermediate map task results. NLineInputformat gives n lines of a file as IS to map tasks.

Binary Input: Hadoop MapReduce has support for binary formats, too.

SequenceFile: Both key and value are represented using binary fashion. The biggest advantage is compression: record level compression, block level compression. Block level compression takes less disk space, less IO, splittale…. It is mainly used when output of one job is fed to another job.

SequenceFileInputFormat: Hadoop sequence file format stores sequences of binary key-value pairs. Sequence files are well suited as a format for MR data because they are splittable. To use data from sequence files as the input to MR, you can use SequenceFileInputFormat.

SequenceFileAsTextInputFormat: is a variant of SequenceFileInputFormat that converts the sequence file keys and values to Text objects. The conversion is performed by calling toString() on the keys and values. This format makes sequence files suitable input for Streaming.

SequenceFileAsBinaryInputFormat: is a variant of SequenceFileInputFormat that retrieves the sequence file keys and values as opaque binary objects. They are encapsulated as BytesWritable objects, and the application is free to interpret the underlying byte array as it pleases FixedLengthInputFormat, which is for reading fixed-width binary records from a file, when the records are not separated by delimiters. The record size must be set via fixedlengthinputformat.record.length.

Multiple Input formats: If there are 100 files of text, it is enough coding for a single common mapper to handle. What if a job consists of multiple input file format as input? Ex: one might be tab-separated plain text, and the other a binary sequence file… Even if they are in the same format, they may have different representations, and therefore need to be parsed differently.

These cases are handled elegantly by using the MultipleInputs class, which allows you to specify which InputFormat and Mapper to use on per-path basis. Ex: we have text input format in different fashion to process. So, replace FileInputFormat.addInputPath() and job.setMapperClass() with the following codes in driver code.

```
MultipleInputs.addInputPath(job, ncdcInputPath, TextInputFormat.class,  Mapper1.class);
MultipleInputs.addInputPath(job, textInputPath, TextInputFormat.class, Mapper2.class);
```

The important thing is that the map outputs have the same types, since the reducers see the aggregated map outputs and are not aware of the different mappers used to produce them.

WholeFileInputFormat: When you assign whole file as input, then consider the heap size for that task. Because, if file size is 10 GB, heap size should be more than 10 GB.

Database Input: DBInputFormat is an input format for reading data from a relational database, using JDBC. HBase TableInputFormat is designed to allow a MR program to operate on data stored in an HBase table. TableOutputFormat is for writing MapReduce outputs into an HBase table.

FileOutputFormats

Different fileoutput formats are: DBOutputFormat, FileOutputFormat, FilterOutputFormat, LazyOutputFormat, MapFileOutputFormat, MultipleOutputFormat, MultipleSequenceFileOutput Format, MultipleTextOutputFormat, NullOutputFormat, SequenceFileAsBinaryOutputFormat, SequenceFileOutputFormat, TextOutputFormat

FileOutputFormat
- TextOutputFormat
- SequenceFieOutputFormat
  - SequenceFileAsBinaryOutuptFormat

NullOutputFormat
DBOutputFormat
FilterOutputFormat ← LazyOutputFormat

Text Output: The default output format, TextOutputFormat, writes records as lines of text. Its keys and values may be of any type, since TextOutputFormat turns them to strings by calling toString() on them. Each key-value pair is separated by a tab character, although that may be changed using the mapreduce.output.textoutputformat.separator property.

The counterpart to TextOutputFormat for reading in this case is KeyValueTextInputFormat, since it breaks lines into key-value pairs based on a configurable separator.

You can suppress the key or the value from the output (or both, making this output format equivalent to NullOutputFormat, which emits nothing) using a NullWritable type. This also causes no separator to be written, which makes the output suitable for reading in using TextInputFormat.

Binary Output: SequenceFileOutputFormat: SequenceFileAsBinaryOutputFormat write sequence files for its output.

MapFileOutputFormat: MapFileOutputFormat writes map files as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order. Input of reduce task is in sorted order without using sorting step in magical phase.

Lazy Output: FileOutputFormat subclasses will create output (part-r-nnnnn) files, even if they are empty. Some applications prefer empty files not be created, which is where LazyOutput Format helps. It is a wrapper output format that ensures that the output file is created only when the first record is emitted for a given partition. To set, include the following in driver code

```
setOutputFormatClass(LazyOutputFormat.class)
```

Exercise 27: Understanding hash partitioner and writing custom partitioner.

In eclipse, by default, only one reducer is launched. If you set more than one reducer in Eclipse, exception is thrown. So, run these jobs in Hadoop cluster. Use card dataset and assign one colour to each reducer.

What if you create more number of partitions than the number of reducers? produces exception.

What if you create less number of partitions than the number of reducers? produces one empty file.

Hadoop uses HashPartitioner as the default Partitioner implementation to calculate the distribution of the intermediate data to the reducers. HashPartitioner requires the hashCode() method of the key objects to satisfy the following two properties:

- Provide the same hash value across different JVM instances
- Provide a uniform distribution of hash values

Hence, you must implement a stable hashCode() method for your custom Hadoop key types satisfying the above mentioned two requirements. Ex:

```
public int hashCode(){}
```

Using hash-partitioner, every individual output file is sorted locally. Combining all files one after other we can get global sorted output. But, this extra manual effort is hectic.

Default partitioners: BinaryPartitioner, HashPartitioner, KeyFieldBasedPartitioner, TotalOrderPartitioner

You can change the partitioning logic in this order

- Evaluate all the pre-defined partitioners
- Play with hashValue (for custom keys)
- Implement custom partitioner: Range partitioner, rule-based partitioner…

You should try to address the skew as much as possible while using existing partitioners or developing new ones.

Exercise 29: Inverted index

Input

one.txt → hi how are you
two.txt → where are you
three.txt → fine dude

Output

are	three.txt, one.txt
dude	two.txt
fine	two.txt
hi	one.txt
how	one.txt
where	three.txt
you	three.txt, one.txt

Exercise 30: Multiple Output files from map/reduce task

FileOutputFormat and its subclasses generate a set of files in the output directory. There is one file per reducer, and files are named by the partition number: part-r-00000, part-r-00001… Sometimes, there is a need to have more control over the naming of the files or to produce multiple files per reducer. MapReduce comes with the MultipleOutputs class to help you.

HashPartitioner works so well with any number of partitions and ensures each partition has a good mix of keys, leading to more evenly sized partitions.

MultipleOutputs allows you to write data to files whose names are derived from the output keys and values, or in fact from an arbitrary string. This allows each reducer (or mapper in a map-only job) to create more than a single file. Filenames are of the form name-m-nnnnn for map outputs and name-r-nnnnn for reduce outputs, where name is an arbitrary name that is set by the program and nnnnn is an integer designating the part number, starting from 00000. The part number ensures that outputs written from different

The base path specified in the write() method of MultipleOutputs is interpreted relative to the output directory, and because it may contain file path separator characters (/), it's possible to create subdirectories of arbitrary depth. Ex: 029070-99999/1901/part-r-00000:

```
protected void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {

      for (Text value : values) {
            parser.parse(value);
            String basePath = String.format("%s/%s/part",
            parser.getStationId(), parser.getYear());
            multipleOutputs.write(NullWritable.get(), value, basePath);
      }
}
```

Furthermore, the mapper or reducer (or both) may write to multiple output files for each record processed.

Exercise 31 and 32: Develop custom key/value data type in MR

If you want to add more fields in the map output key, you can do convert all attributes as text and concatenate together with MR Text datatype. After receiving at reducer side, you can tokenize the text. Despite there are more attributes included in map output key, it is not possible to sort using one variable, group with another variable, etc., as all variables are concatenated with one text. In order to do more actions with many variables included in map output key, we go for creating custom key and custom value.

Ex: say a log file records look like:

```
SI No  Name            IP          Description       Date
   1   raja       10.10.10.10       failed           12-4-5
```

SI No: IntWritable, Name: Text, IP : IntWritable, Description: Text, Date: IntWritable

Custom key is a composite version that contain more than one variables/fields/attributes. Each attribute can be of any MR datatype. You have a job that shuffles and sorts based on IP. Now, if you want to sort and group with date, you have to use a separate map task to write date as key and entire line as value. Alternatively, you can create custom key that comprises five-inbuilt variable, which can be used at any time as key or value without using any extra mappers.

To create custom key data type, a class need to implement WritableComparable

- Implement readFields() and write() from Writable interface
- Implement compareTo() from java.lang.Comparable
- Override following functions from java base class object: hashCode, equals, toString

To create custom value data type, a class need to implement Writable

- need not implement Comparable as values are not sorted and grouped.
- override only readFields(), write() of Writable interface.

Exercise 33: Find out the number of people and their sex who died and survived.

Titaninc data set description:

- Column 1: PassengerId
- Column 2: Survived (survived=0 & died=1)
- Column 3: Pclass
- Column 4: Name
- Column 5: Sex
- Column 6: Age
- Column 7: SibSp
- Column 8: Parch
- Column 9: Ticket
- Column 10: Fare
- Column 11: Cabin
- Column 12: Embarked

Create custom InputFormat: Here, we need to implement a custom key which is a combination of two columns, that is, 2nd column, which consists of the dead or the survivors and the 5th column, which contains the gender of the person. So, let us prepare a custom key by combining both these columns and sort them using the gender column.

Understand type of file you want to process with MR, then you have to create RR to change input key and value to feed mapper. Hadoop supports processing of many different formats and types of data through InputFormat. The InputFormat of a Hadoop MR computation generates the key-value pair inputs for the mappers by parsing the input data. InputFormat also performs the splitting of the input data into logical partitions, essentially determining the number of Map tasks of a MR computation and indirectly deciding the execution location of the map tasks. Hadoop generates a map task for each logical data partition and invokes the respective mappers with the key-value pairs of the logical splits as the input.

1. override the following

```
public abstract class InputFormat<K, V> {
      public abstract List<InputSplit> getSplits(JobContext context)
      throws IOException, InterruptedException;

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

Calculate splits by calling InputFormat.getSplits

2. For each split, schedule a map task.

3. For each Mapper instance, a reader is retrieved by InputFormat.createRecordReader() that takes InputSplit instance as a parameter.

4. RecordReader reads records as key-value pairs.

5. map() method is called for each key-value pair.

job.setOutputFormat(NullOutputFormat.class); setting this, map and reduce tasks will be launched but no output directory will be created. Therefore, no output you will get.

NullWritable is a special type of Writable, as it has a zero-length after serialization. No bytes are written to, or read from, the stream. It is used as a placeholder; for example, in MapReduce, a key or a value can be declared as a `NullWritable` when you do not need to use that position. It effectively stores a constant empty value. NullWritable can also be useful as a key in `SequenceFile` when you want to store a list of values, as opposed to key-value pairs. It is an immutable singleton: the instance can be retrieved by calling `NullWritable.get()`

Exercise 34: Chaining mappers.

Job1: wordcount program

job2: counting how many words that start with same letter from the output of Job1.

There are 4 ways:

1. Cascading jobs: Using multiple job specifications in single driver class:

   `JobChainDriverMethod1.java in ChainingMRjobs package`

   or

   you can break this Driver into two and run manually one by one by giving job 1 output directory to input of job2 as given below. but, this is the simplest method.

   ```
   $ hadoop jar chain.jar Job1 /shakespeare /output
   $ hadoop jar chain.jar Job2 /output/part-r-0000 output2
   ```

2. Using JobControl:

   `JobChainDriverMethod2.java in ChainingMRjobs package`

3. Using ChainMapper and ChainReducer
4. Using oozie/falcon workflow scheduling library: it is not only chaining mappers in DAG and also schedules.

Exercise 35: Distributed cache for wordcount job to skip stop words

If you run this job in Eclipse, it results to permission error. So, run in Hadoop cluster with MRv2 libraries. We can pass files to distributed cache via two ways.

map() vs setup()

map() is invoked for every record.

setup() is invoked only once when JVM is initialized before map task is launched. Therefore, some variables can be commonly assigned for all mappers.

cleanup() is executed at last after all map/reduce tasks completed before writing output files. Therefore, you can maintain a global variable, calculate something and you can write into same output file using context.write().

We are going to give two text files as input for word counting and one file as pattern input (symbols to exclude from word counting). So, third file is given to all mappers via setup() that are going to perform counting. So, symbols like. , / are not counted.

file1.txt:

```
hi , how are you !
How is your studies .
```

file2.txt:

```
hi did ! you give file to your friend ,
how is your bike !
```

patterns.txt:

```
\.
\,
\!
to
```

$ hdfs dfs -mkdir /input

$ hdfs dfs -put file1.txt file2.txt patterns.txt /input

$ yarn jar job.jar DCWCwithStopWordsDriver /input/file1.txt /input/file2.txt -skip /input/patterns.txt /output

$ hdfs dfs -cat /output/par*

```
How     1
are     1
bike    1
did     1
file    1
friend  1
give    1
hi      2
how     2
is      2
studies 1
you     2
your    3
```

After files are passed to distributed cache, it is loaded in location as follows

$ ls /usr/local/hadoop/tmp/nm-local-dir/

filecache nmPrivate usercache

$ ls /usr/local/hadoop/tmp/nm-local-dir/usercache/itadmin/filecache/10/

companylist_noheader.csv

Exercise 36: JUnit and MRUnit testing

JUnit

Unit: Small portion of the program/data.

Unit Test: Verifies the correctness of the functionality of that unit of code and should take less than a second.

Why to use? To test a portion of huge job than testing your full code

- It is a java-testing framework for writing and running automated test. Say, you had written a logic, now you have modified something in the logic.

- Now, you want to make sure that logic is true for all test cases without running the entire application.
- So, specific functions in an application can be tested without running the huge application using MR unit test.
- Junit uses assert test.
- Basic value checker checks the experimental value against actual value.
- passes if the values are the same and fails in any other condition

JUnit Basics:

- @Test: Java annotation., Indicates that this method is a test which JUnit should execute.
- @Before: Java annotation, Tells JUnit to call this method before every @Test method.
- Two @Test methods would result in the @Before method being called twice.

JUnit test methods:

- assertEquals(), assertNotNull()…
- Fail if the conditions of the statement are not met
- fail(msg) - Fails the test with the given error message.

The assertEquals is one of the JUnit methods from the Assert object that can be used to check if a specific value match to an expected one. It primarily accepts two parameters. First one if the actual value and the second is a value expected. It will then try to compare this two and returns a boolean result if it's a match or not.

```
Text t = new Text("hadoop");
t.set("pig");
assertEquals(key.getLength(), 3);
assertEquals(t.getBytes().length, 3);
```

MR unit testing

Add supporting jars: MRv2 lib jars, add MRv2 common, hdfs, mapreduce, yarn jars, add run with mrunit-1.0.0-hadoop2.jar and MRv2 jars.

- MR job will run over hours. At last, if any error occurs, it is waste of time and resource. You can test either mapper or reducer or whole job.
- MRUnit is a testing framework based on Junit for unit testing mappers, reducers, combiners and the combination of the three.
- The MapDriver allows one input and output per test. You can call withInput and withOutput multiple times if you want, but the MapDriver will overwrite the existing values with the new ones, so you will only ever be testing with one input/output at any time.
- It is used to test whether anything wrong in the input record or code logic or output. Therefore, we write mapper and reducer, testmapper and testreducer, and testmapreduce job.

JUnit cannot be used directly to test Mappers and Reducers because Unit tests require mocking up classes like IS, OutputCollector, Reporter… of MR framework.

MRUnit is built on top of Junit and works with the mockito framework to provide required mock objects. So, rather than testing entire MR job, you can check with whatever component you want to test.

MR unit provides a mock IS and other classes. It can test

- Mapper – Supported by MapDriver Class.
- Reducer – Supported by ReduceDriver Class.
- Full MapReduce flow – Supported by MapReducerDriver Class.

Five steps to write the Test :

- Instantiate the Driver class (MapDriver or ReduceDriver or MapReduceDriver) parametrized exactly as the mapper under test.
- Add an instance of the mapper/reducer you are testing by using the withMapper call.
- The withInput call enables you to pass in a desired key and input value.
- The expected output is specified using the withOutput call.
- The last call, run test, feeds the specified input values into the mapper/reducer and compare the actual output against the expected output set in withOutput method.

Exercise 37: Switching between local, pseudo-distributed and cluster mode of Hadoop

It is a way of testing MR framework using MiniDFSCluster, MiniMRCluster, MiniYARN Cluster. You may have more than one version Hadoop cluster like local mode, distributed mode, pseudo distributed mode… You can create three xml files in a conf directory as follows to mention which Hadoop flavour you are using:

```
$ vi hadoop-local.xml
<?xml version="1.0"?>
<configuration>
      <property>
            <name>fs.defaultFS</name>
            <value>file:///</value>
      </property>
      <property>
      <name>mapreduce.framework.name</name>
            <value>local</value>
      </property>
</configuration>
$ hdfs dfs  -conf hadoop-local.xml -ls /
$ yarn  jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.0.jar wordcount  -conf hadoop-local.xml input.txt output
$ vi hadoop-localhost.xml
<?xml version="1.0"?>
<configuration>
      <property>
            <name>fs.defaultFS</name>
            <value>hdfs://localhost/</value>
      </property>
      <property>
            <name>mapreduce.framework.name</name>
            <value>yarn</value>
      </property>
      <property>
            <name>yarn.resourcemanager.address</name>
            <value>localhost:8032</value>
            </property>
</configuration>
```

$ hdfs dfs -conf hadoop-localhost.xml -ls /

$ yarn jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.0.jar wordcount -conf hadoop-localhost.xml input.txt output

$ vi hadoop-cluster.xml

```
<?xml version="1.0"?>
<configuration>
      <property>
            <name>fs.defaultFS</name>
            <value>hdfs://namenode/</value>
      </property>
      <property>
            <name>mapreduce.framework.name</name>
            <value>yarn</value>
      </property>
      <property>
            <name>yarn.resourcemanager.address</name>
            <value>resourcemanager:8032</value>
      </property>
</configuration>
```

$ hdfs dfs -conf hadoop- cluster.xml -ls /

$ yarn jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.0.jar wordcount -conf hadoop- cluster.xml input.txt output

With this setup, it is easy to use any Hadoop environment. It overrides all the property setup in /etc/hadoop/*-site.xml files. If you omit the -conf option, you pick up the Hadoop configuration in the *etc/hadoop* under $HADOOP_HOME or if HADOOP_CONF_DIR is set, Hadoop configuration files will be read from that location.

You can do even job specific in driver class rather than launching via command line

```
Configuration conf = new Configuration();
conf.set("fs.defaultFS", "file:///");
conf.set("mapreduce.framework.name", "local");
conf.setInt("mapreduce.task.io.sort.mb", 1);
```

Exercise 39: Compression algorithms

Data – compress and serialize – store in HDFS – reserialize and decompress – process

Job output compression

```
job.setOutputFormatClass(SequenceFileOutputFormat.class);
SequenceFileOutputFormat.setCompressOutput(job, true);
SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
SequenceFileOutputFormat.setOutputCompressionType(job,CompressionType.BLOCK
);
```

Map output compression

```
job.setMapperClass(CleanerMapper.class);
job.setOutputKeyClass(IntWritable.class);
job.setOutputValueClass(Text.class);
job.setNumReduceTasks(0);
job.setOutputFormatClass(SequenceFileOutputFormat.class);
```

```
SequenceFileOutputFormat.setCompressOutput(job, true);
SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
SequenceFileOutputFormat.setOutputCompressionType(job,CompressionType.BLOCK
);
```

Scheduling

$ hadoop job -set-priority <jobid> priority

You can add a job to any specific pool as in capacity scheduler while launching job

yarn.resourcemanager.scheduler.class property is used to set the desired scheduler.

Case Study: Application Development for NYSE dataset

Dataset: New York Stock Exchange, a trading company records all trades in year wise in .csv file as follows. Ex:

- nyse_2009.csv
- nyse_2010.csv
- nyse_2011.csv
- nyse_2012.csv
- nyse_2013.csv
- nyse_2014.csv

Upload these files onto HDFS. Each file has the following fields/attributes delimited by comma:

stock ticker, trading date, open price, high, low, closing price, volume

A,01-Jan-2009,15.63,15.63,15.63,15.63,0
AA,01-Jan-2009,11.26,11.26,11.26,11.26,0
AAP,01-Jan-2009,33.65,33.65,33.65,33.65,0
AAV,01-Jan-2009,4.21,4.21,4.21,4.21,0
AB,01-Jan-2009,20.79,20.79,20.79,20.79,0
ABB,01-Jan-2009,15.01,15.01,15.01,15.01,0
ABC,01-Jan-2009,17.83,17.83,17.83,17.83,0
ABG,01-Jan-2009,4.57,4.57,4.57,4.57,0
ABM,01-Jan-2009,19.05,19.05,19.05,19.05,0
…..

Exercise 1: Get average volume traded for each stock per month from NYSE dataset

1. There is no date datatype in MR. So, you have to parse the data attribute using POJO and convert to our desired type. We can parse in map method itself by tokenizing. But, why should we create separate parser? Because, for simple logic, it is enough to do in map method itself. For complex logic involving different applications dealing with same dataset, you need not repeat parsing in every map function. So, just write a common parsing logic.

`NYSEParser.java` in `NYSE.Parsers` package

Right click in the code file → source → generate getters and setters to create POJO

2. Identify input and output formats.

TextInputFormat and TextOutputFormat as we are dealing with text data.

3. Develop user-defined key and value data types.

If possible, create user-defined key-value pairs. Without creating user-defined data types, you can append in Text type, but it would be meaningful to sort based on different fields in the same key by creating user-defined data types.

Therefore, develop custom key with two fields (Text datatype) and custom value with two fields (LongWrtiable datatype).

`TextPair.java` in `NYSE.CustomKeyValue` package is for custom key type, where first is trade date and second is stock ticker name

`LongPair.java` `NYSE.CustomKeyValue` package is for custom value type, where first is sum and second is total

These classes implement `WritableComparable` (from `org.apache.hadoop.util`) interface and override the following methods.

> `readFields()` and `write()` from `Writable` interface
>
> `compareTo()` from `Comaprable` from `java.lang` (not required for classes used for values only). Go to `org.apache.hadoop.io.TextWritable` in `grepcode` to see implemented methods for a MR data type. Learn this logic

Override following function from java API base class Object

> `hashCode()` – I implement this to address intermediate skew
>
> `equals()` – I implement this to compare objects
>
> `toString()` – I implement this to return any object as string

Right click in the code file → source → generate hashcode, equals, toString automatically instead of typing.

Relation between `WritableComparable` and `WritableComparator`

> By default, `compareTo()` from `java.lang` uses object level comparison and it requires serialization and deserialization, which can impact performance.
>
> MR API provide `RawComparator` (sub interface for `java.lang.Comparator`) interface and hierarchy of interfaces `compareTo()` to compare data byte level without serialization.

4. Develop mapper

`AvgStockVolMonthMapper` in `NYSE.AvgStockVolPerMonth` package.

Input: ABC,01-Jan-2009,17.83,17.83,17.83,17.83,0
Output: 2009-01 A 0 1

- Input key – line offset.
- Input value – each line in NYSE data.
- Output key – custom key calendar month (YYYY-MM) and stock ticker name, hence use custom key taking 2 text values.
- Output value – custom value with sum and total.

5. Develop combiner

`AvgStockVolMonthCombiner.java` in `NYSE.AvgStockVolPerMonth` package.

Finding average stock volume is not suitable to use Reducer itself as Combiner. Therefore, in combiner, we just total the trade, and in reducer find average.

- Input key – same as map output key.
- Input value – same as map output value.
- Output key – custom key calendar month (YYYY-MM) and stock ticker name, hence use custom key taking 2 text values.
- Output value – custom value with sum and total.

6. Develop reducer

`AvgStockVolMonthReducer.java` in `NYSE.AvgStockVolPerMonth` package

Key and value type are same as combiner

Note: combiner and reducer both extend Reducer class. Key and value should match, but processing logic can differ.

7. Driver code

`AvgStockVolMonthDriver.java` in `NYSE.AvgStockVolPerMonth` package

Exercise 2: Working with file system (org.apache.hadoop.fs) APIs

Run file system related jobs in real Hadoop environment. Because, in Eclipse there is no HDFS running.

$ yarn jar Job.jar FileSystemAPI.FileDisplay

Practice with FileSystem API to display URL while filtering files.

- `ListFiles.java` in `FileSystemAPI` package. "input" is a HDFS directory here.

  $ yarn jar job.jar FileSystemAPI.GetFiles /input

  hdfs://node4:10001/input/nyse_2010.csv

  hdfs://node4:10001/input/nyse_2011.csv
- `FilePattern.java` in `FileSystemAPI` package. Similar to previous job, but provides more customization.

  $ yarn jar job.jar FileSystemAPI.FilePattern /input
- `CopyMerge.java` in `FileSystemAPI` package (all small files in a directory are concatenated to a single new file). I created few files in a directory in HDFS.

  $ yarn jar job.jar FileSystemAPI.CopyMerge /input /file_name

  $ hdfs dfs -cat /file_name
- `URLCat.java`
- `FileSystemCat.java`
- `FileSystemDoubleCat.java`
- `FileCopyWithProgress.java`
- `ListStatus.java`
- `FileDisplay.java`
- `FileDisplay1.java`
- `FileDelete.java`

Every time providing new output directory location is tiresome and will leave us delete lot of files later. Therefore, include the following lines in driver code

```
Path outpath = new Path(otherArgs[otherArgs.length - 1]);
outpath.getFileSystem(conf).delete(outpath);
FileOutputFormat.setOutputPath(job,outpath);
```

Exercise 3: Filtering input files and input records

1. Filtering input files in driver code itself

- `AvgStockVolMonthDriver.java` in `FileSystemAPI` package.

2. Filtering input records in map()

Displaying only BAC stock ticker trade per month in map method itself

- `AvgStockMapperWithProperties.java` in `Filtering` package
- `AvgStockMapperWithSetUp.java` in `Filtering` package
- `AvgStockVolMonthDriver.java` in `Filtering` package

`In AvgStockMapperWithProperties.java`

```
public class AvgStockMapperWithProperties extends Mapper<LongWritable,Text,
TextPair,LongPair>{

      private static Parser parser=new Parser();
      private static TextPair mapoutputkey=new TextPair();
      private static LongPair mapoutputvalue=new LongPair();

      public void map(LongWritable key, Text record, Context context)throws
      IOException, InterruptedException{

            parser.parse(record.toString());

            if (parser.getStockTicker().equals("BAC")){

                  mapoutputkey.setFirst(new Text(parser.getTradeMonth()));
                  mapoutputkey.setSecond(new
                  Text(parser.getStockTicker()));

                  mapoutputvalue.setFirst(new
                  LongWritable(parser.getVolume()));
                  mapoutputvalue.setSecond(new LongWritable(1));

                  context.write(mapoutputkey,mapoutputvalue);
            }
      }
}
```

$ yarn jar job.jar `Filtering.AvgStockMapperWithProperties` /input /output

$ hdfs dfs -cat /output/part-r-00000 // it displays the stock ticker record of only BAC

3. Filtering input records in map() via command line arguments

However, if you want to change condition from BAC bank to AEO stock ticker, then again you have to hardcode, convert into jar and then run. So, rather than doing in this painful way you can pass stock ticker name via command line arguments as a property in the name of `filter.by.stockTicker`, accordingly we can filter in map method.

`In AvgStockMapperWithProperties.java` in `Filtering` package

```
public class AvgStockMapperWithProperties extends Mapper<LongWritable,Text,
TextPair,LongPair>{

      private static Parser parser=new Parser();
      private static TextPair mapoutputkey=new TextPair();
      private static LongPair mapoutputvalue=new LongPair();

      public void map(LongWritable key, Text record, Context context)throws
      IOException, InterruptedException{

            parser.parse(record.toString());

            if(parser.getStockTicker().equals(context.getConfiguration().ge
            t("filter.by.stockTicker"))){

            mapoutputkey.setFirst(new Text(parser.getTradeMonth()));
            mapoutputkey.setSecond(new Text(parser.getStockTicker()));

            mapoutputvalue.setFirst(new LongWritable(parser.getVolume()));
            mapoutputvalue.setSecond(new LongWritable(1));

            context.write(mapoutputkey,mapoutputvalue);
            }
      }
}
```

Then pass filter.by.stockTicker in command line arguments while launching job. Example:

$ yarn jar job.jar Filtering.AvgStockMapperWithProperties -Dfilter.by.stockTicker=AEO /input /output

$ hdfs dfs -cat /output/part-r-00000 // it displays the stock ticker record of only AEO

This method is not very effective. Because, getConfiguration() returns 1000's of properties and all are checked for the desire one.

When you use this property method, then use GenericOptionsParser to read property from command line.

4. Filtering input records in setup()

- setup() method is invoked once per JVM unlike map/reduce function that is invoked for every record.
- setup() method is used to share files, database connections…. in map/reduce tasks.
- cleanup() method is used for closing network/database, connections, files…
- filtering in reduce method is not recommended as reduce() is not meant for filtering.

In `AvgStockMapperWithSetUp.java` in `Filtering` package

```
public class AvgStockMapperWithSetUp extends Mapper<LongWritable,Text,
TextPair,LongPair>{

      private static Parser parser=new Parser();
      private static TextPair mapoutputkey=new TextPair();
      private static LongPair mapoutputvalue=new LongPair();
      Private String stockTicker=new String();

      Protected void setup(Context context)throws IOException,
      InterruptedException{

            stockTicker=context.getConfiguration().get("filter.by.stockTick
            er");
```

```
        }

        public void map(LongWritable key, Text record, Context context)throws
        IOException, InterruptedException{

              parser.parse(record.toString());

              if(parser.getStockTicker().equals(stockTicker))){

              mapoutputkey.setFirst(new Text(parser.getTradeMonth()));
              mapoutputkey.setSecond(new Text(parser.getStockTicker()));

              mapoutputvalue.setFirst(new LongWritable(parser.getVolume()));
              mapoutputvalue.setSecond(new LongWritable(1));

              context.write(mapoutputkey,mapoutputvalue);
              }
        }
}
```

Then pass filter.by.stockticker in command line arguments.

$ yarn jar job.jar Filtering.AvgStockMapperWithSetUp -Dfilter.by.stockTicker=BAC /input /output

$ hdfs dfs -cat /output/part-r-00000 it displays the stock ticker record of only BAC

So, now getConfiguration() is called only once for JVM using setup() method.

Similarly, you can pass multiple parameters like this, but you have to parse in setup() and assign to global variable of map task. You can use collections to store multiple properties, it is easy to lookup.

In AvgStockMapperWithSetUp.java

```
public class AvgStockMapperWithSetUp extends Mapper<LongWritable,Text,
TextPair, LongPair >{

        private static Parser parser=new Parser();
        private static TextPair mapoutputkey=new TextPair();
        private static LongPair mapoutputvalue=new LongPair();
        private static String stockTicker;
        private Set<String> stockTickers=new HashSet<String>();
        // Set is easy to lookup

        public void setup(Context context)throws IOException,
        InterruptedException{
              String[] tickers=null;
              stockTicker=context.getConfiguration().get("filter.by.stockTick
              er")  ;
              if(stockTicker!=null){
                    tickers=stockTicker.split(",");
                          //-Dfilter.by.stockTicker=BAC,AEO
              }
              for(String tick:tickers){
                    stockTickers.add(tick);
              }
        }
        public void map(LongWritable key, Text record, Context context)throws
        IOException, InterruptedException{

              parser.parse(record.toString());
```

```
                if(stockTickers.isEmpty()||
                stockTickers.contains(parser.getStockTicker())){

                mapoutputkey.setFirst(new Text(parser.getTradeMonth()));
                mapoutputkey.setSecond(new Text(parser.getStockTicker()));
                mapoutputvalue.setFirst(new LongWritable(parser.getVolume()));
                mapoutputvalue.setSecond(new LongWritable(1));
                context.write(mapoutputkey,mapoutputvalue);
                }
        }
}
```

$ yarn jar job.jar Filtering.AvgStockMapperWithSetUp -Dfilter.by.stockTicker=BAC,AEO /input /output

$ hdfs dfs -cat /output/part-r-00000 it displays both AEO, and BAC

Exercise 4: user-defined partitioner

In the previous application, we have got average stock volume per month, but the data is all scattered. Different reduce tasks have processed data related to same stock ticker (Ex: BAC data is processed by all reducers). But, in some use cases, getting all records related to particular stock together makes more sense.

Make sure data is partitioned by stock ticker name so that all records related to particular stock are written to one file. That is one stock ticker name will not appear in many output files. Moreover, partitioning and sorting are done based on the second key (stock ticker name) in custom key.

Advantage of using user-defined key with more than one field is, you can partition based on any field in the custom key. Whereas, if you append all fields in one Text key, then it is not possible to partition based on the specific field in the text.

Hashing converts any datatype into text and then decides what to do with that string. Therefore, writing numeric as text for map output key may make sense some time.

Run the job with four reducers and show in grepcode.com org.apache.hadoop.mapreduce.lib.partition.HashPartitioner and explain how getPartition() gives hashCode() of keys.

- `TextPair.java` in `NYSE.partitioner` package
  `//result = prime * result + ((first == null) ? 0 : first.hashCode());`
- `AvgStockVolMonthMapper.java` in `NYSE.partitioner` package
- `AvgStockVolMonthCombiner.java` in `NYSE.partitioner` package
- `AvgStockVolMonthReducer.java` in `NYSE.partitioner` package
- `AvgStockMonthPartitionerDriver.java` in `NYSE.partitioner` package
  `job.setPartitionerClass(HashPartitioner.class);`

So, partitioning is done based on the hash code of stock ticker name alone and not with (trade month +stock ticker name). See the number of records (lines in an output file) and size of each output file of mapper to checkout skewness.

Upload nyse_data/*.csv files

$ hdfs dfs -put nyse_data /

$ yarn jar Job.jar NYSE.Partitioner.AvgStockMonthPartitionerDriver -D filter.by.stockTicker=BAC,AEO /dataset/nyse_data/*.* /output

All AEO goes to one reducer and all BAC goes to another reducer. Therefore, corresponding output files are

```
$ hdfs dfs -du /output
0     /output/_SUCCESS
0     /output/part-r-00000
1448  /output/part-r-00001
1568  /output/part-r-00002
0     /output/part-r-00003
```

Exercise 5: Developing custom partitioner

To make a particular stock ticker name to go to specific reducer, comment first field in key in hashCode(), or if you want to get specific trade date to go to specific reducer, then comment second line in the hashCode().

Here, rather than commenting hashCode of second field in the key, we are going to create our hash partitioner that takes only second field of the custom key.

- `SecondKeyTextPairPartitioner.java` in `NYSE.partitioner` package
- `AvgStockMonthCustomPartitionerDriver` in `NYSE.partitioner` package

You can do range-based partitioning, rule-based partitioning…

Exercise 6: Partition decision based on runtime arguments

To choose the way you want partition based on either first field in the custom key or second field in the custom key

- `AvgStockMonthCustomPartitionerOptionsDriver.java` in `NYSE.partitioner` package
- `FirstKeyTextPairPartitioner.java` in `NYSE.partitioner` package
- `SecondKeyTextPairPartitioner.java` in `NYSE.partitioner` package

We can choose the right field that should be used to partition at run time

```
$ yarn jar job.jar NYSE.Partitioner.AvgStockMonthPartitionerDriver -D
partition.by=stockticker /dataset/nyse_data/*.* /newout13
$ yarn jar job.jar NYSE.Partitioner.AvgStockMonthPartitionerDriver -D
partition.by=trademonth /dataset/nyse_data/*.* /newout13
```

You can see some skew in the above output. Check output directory with following command.

```
$ hdfs dfs -cat /output/part-r-00000 | wc -l
$ hdfs dfs -cat /output/part-r-00001 | wc -l
$ hdfs dfs -cat /output/part-r-00002 | wc -l
$ hdfs dfs -cat /output/part-r-00003 | wc -l
```

You can see the skew in the number of records processed by all reducers. Therefore, we have to partition evenly across all reducers. Instead of using hashcode of first field in the custom key, we find numeric value in the first field of custom key to apply mod (%) and distribute evenly. Therefore, we are going to partition based on YYYYMMDD by removing "-" from the middle of date as numeric instead of string and then find hashcode for that string.

Include the fowling in `FirstKeyTextPairPartitioner.java`

```
partitionValue=new Integer(key.getFirst().toString().replace("-",
"")).intValue()% numReducers;
```

Now, launch the job and check out the results whether all reducers processed and produced equal number of records.

Exercise 7: partitioning and sorting based on the first field in custom key using comparator

MR applications can be developed using many programming languages and one can implement business logic such as filtering, transformations, aggregations, joins, sorting… anywhere in the life cycle. But, one need to be aware of why MR framework architects have come up with different types of phases in the execution,

User-defined,

Map - filtering, row level transformations, field projections…
Reduce - aggregations, joins…

MR framework handles

Sort - using sorting comparator
Group - using grouping comparator

Sorting techniques: global sort, local sort, secondary sort

To work with sorting effectively, programmers need to have idea about partitioner, comparator, multiple output formats…

We need good understanding on comparators to work with sorting. There are different sorting methods:

- Local sort – each reduce task output is a sorted one.
- Global sort – keys are unique and sorted across all reduce output files. If you use one reduce task, then you will get sorted output obviously. But, we can also achieve this by using range partitioning, rule-based partitioning, multiple output file.
- Secondary sort – it is possible only when you create custom key with more than one fields. Sorting is done based any field present in the custom key.

Sorting and grouping are taken care by framework itself and done by comparator. By default, compareTo() is used for comparison. You can override this. As we are using custom key with more than one field, you can decide which field in the key to take part in sort and group. You should not write any logic for sorting, grouping… in map/reduce tasks as it is based on compareTo().

Usually, sorted keys are in ascending order. In order to sort in reverse (descending order), just negate the compareTo() return value. To partition and sort based on the first field in custom key using comparator,

- `AvgStockMonthComparatorDriver.java` in `NYSE.Comparator` package
- `AvgStockVolMonthMapper.java` in `NYSE.Comparator` package
- `AvgStockVolMonthCombiner.java` in `NYSE.Comparator` package
- `AvgStockVolMonthReducer.java` in `NYSE.Comparator` package
- `TextPair.java` in `NYSE.Comparator` package

In `TextPair.java of` compareTo()

```
public int compareTo(TextPair o) {
```

```
		int comp=this.first.compareTo(o.first);
		if(comp !=0)
			return comp;
		return -this.second.compareTo(o.second);
	}
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		//result = prime * result + ((first == null) ? 0 :
		first.hashCode());
		result = prime * result + ((second == null) ? 0 :
		second.hashCode());
		return result;
	}
```

Increase number of reduce tasks and validate the output. If you used multiple reducers, you can observe that keys have been distributed as a sorted manner across all reducers, and each reducer has sorted locally also.

MR data types such as IntWritable… compares key at byte level (no serialization/de-serialization). But, when we write custom key, comparison is done based on objects (after serialization/de-serialization) as we implement WritableComparable interface. Therefore, we need to implement RawComparator in order perform byte level comparison to improve performance.

Exercise 8: To get top 3 traded stocks per day from NYSE datasets

Design:

- Mapper performs filtering and row level transformation.
- Reducer will have logic to get top 3 stocks.
- Using a prtitioner, sorting comparator and grouping comparator we will get data as
  Key - Day, volume in descending order
  Value - Other details

Implementation plan

- InputFormat (CombineTextInputFormt if there are lot of small files)
- Mapper
  - Input key and value – line offset and each line
  - Output key – Custom key (date in YYYYMMDD format and Volume). We convert trade to numeric (long) using getTradeNumeric in `NYSEParser.java`.
  - Output value – Text (each record will be passed as output value as we want to see all the details of top 3 stocks per day by volume)
- CustomKey and CustomValue type
- Partititoner: we need to sort in descending by volume
- Sorting comparator – within each date we need to sort in descending by volume
- Grouping comparator – even though map output key is composite with date and volume, we need to invoke reducer only once per date (not once per date and volume)
- Reducer – reducer will receive data partitioned by date and then descending by volume. But, the grouping will be done based on date alone. There will be records for all the stocks in array for each key group. We just need to process first 3 records to get top 3 stocks by volume each day.

Global sorting can be done by using range partitioning, rule-based partitioning, multiple output file. Comparator can be used for sorting and grouping. Rather than creating objects for first and second field in CustomKey, we are going to use primitive type that lets byte level comparison using raw comparator.

Initially try with default reducer so that you can observe the difference by referring counters

- `TopThreeStocksByVolumePerDayDriver.java` in `NYSE.Top3Stocks` package
- `TopThreeStocksByVolumePerDayMapper.java` in `NYSE.Top3Stocks` package
- `FirstKeyLongPairPartitioner.java` in `NYSE.Partitioner` package
- `LongPairPrimitive.java` in `NYSE.CustomKeyValue` package

In `FirstKeyLongPairPartitioner.java`, using the following code and hash partitioner number of records for each reducer may be balanced. but, it does not help for global sorting.

```
public int getPartition(LongPairPrimitive key, Text value, int
numPartitions) {
      long partValue = key.getFirst();
      return (int) partValue % numPartitions;
}
```

Using the range partitioning, you can sort but, load may not be balanced

```
public int getPartition(LongPairPrimitive key, Text value, int
numPartitions) {
            long partValue = key.getFirst();
            if(partValue >=20120101 && partValue <= 20120630)
                  return 0;
            else if(partValue >=20120701 && partValue <= 20121231)
                  return 1;
            else if(partValue >=20130101 && partValue <= 20130630)
                  return 2;
            else if(partValue >=20130701 && partValue <= 20131231)
                  return 3;
            return 0;
}
```

Custom comparator steps

- Develop class which extends relevant comparator or implements raw comparator.
- Provide functionality for compare.
- To override compreTo() method register the new comparator.
- Also, one can use comparator for grouping or sorting to tweak their behaviour.

Grouping comparator

- While sorting comparator controls to change the sorting behaviour, grouping comparator will change the behaviour of how many times reducer should be invoked
- By default, sorting and grouping comparators will be same

Sorting data as Text object do not sort lexicographically if data is signed. instead you can write map outputs as sequence file to sort with bytes.

If you want to use data type as key data type, then class must implement compareTo() and hashCode(). Because, these are required to compare, partition and sorting purpose. Value data type do not require these things. However, key also can be used as value but these methods

won't be called. enough implementing Writable interface alone for value data type unlike key data type implements WritableComparable interface.

compareTo() method performs comparison based on objects and requires serialization and deserialization before and after comparing objects to convert to and from hadoop data types. So, we need to use RawComparator that performs byte level comparison.

Comparator can be used for sorting and grouping. byte level comparison using raw comparator. In case of non-primitive data types such as Text, *Writable… require serialization and de-serialization. Moreover, it is difficult to implement raw comparator for non-primitive data types.

Relation between WritableComparable and WritableComparator

- By default compareTo() from WritableComparable interface uses object level of comparison and it requires serialization and deserialization. Because, MR datatypes are IntWritable, FloatWritable... that performs comparison based on its types but not like objects.
- Serialization and deserialization can impact performance
- So, MR provide RawComparator (sub interface for java.lang.Comparator) interface and hierarchy of interfaces and classes.

If you set key of mapper/reducer as NullWritable, then you have to include out.write(NullWritable.get(), value). It ensures that we need not print key at all. You have to mention this in driver program too.

Hadoop data types are used throughout the MapReduce computational flow, starting with reading the input data, transferring intermediate data between Map and Reduce tasks, and finally, when writing the output data. Choosing the appropriate Writable data types for your input, intermediate, and output data can have a large effect on the performance and the programmability of your MapReduce programs. The Writable interface defines how Hadoop should serialize and de-serialize the values when transmitting and storing the data.

The sort order for keys is controlled by a RawComparator, which is found as follows:

1. If the property `mapreduce.job.output.key.comparator.class` is set, either explicitly or by calling `setSortComparatorClass()` on `Job`, then an instance of that class is used. (In the old API, the equivalent method is `setOutputKeyComparatorClass()` on `JobConf`.)

2. Otherwise, keys must be a subclass of `WritableComparable`, and the registered comparator for the key class is used.

3. If there is no registered comparator, then a `RawComparator` is used. The `RawComparator` deserializes the byte streams being compared into objects and delegates to the WritableComparable's `compareTo()` method.

These rules reinforce the importance of registering optimized versions of `RawComparators` for your own custom.

Development cycle:

- Develop parser

- Develop custom key/value classes
- Develop mapper
- Develop partitioner (with global ordering using multiple reducers)
- Develop comparators (sorting and grouping)
- Develop reducer
- Develop driver

Try1: Range partitioning to achieve global sorting

Hash partitioner (default partitioner) will distribute records to different reducers based on the hash value of the keys.

- `FirstKeyLongPairPartitioner.java` in `NYSE.Partitioner` package

Try2: To group keys based on the desired field in the composite key

Reduce number of groups/clusters in grouping. Check the result in reduce input group counters after running in Hadoop environment. Do with custom grouping and general grouping and then compare. Reducing number of groups is important as number of times reduce() is called depending on it. However, number of reduce output records will be same.

- `LongPairPrimitiveGroupingComparator.java in NYSEComparator package`

```
job.setGroupingComparatorClass(LongPairPrimitiveGroupingComparator.class);
```

It reduces the number of group, so the number of times reduce function called is reduced, but not the number of records to be processed in both condition (using group comparator and without using group comparator) by reduce tasks. See the counters after job execution.

To sort keys based on the desired field in the composite key

- `LongPairPrimitiveSortingComparator.java in NYSEComparator package`

```
job.setSortComparatorClass(LongPairPrimitiveSortingComparator.class);
```

Try3: Display top 3 records in each day based on volume in the composite key with user-defined reducer

`TopThreeStocksByVolumePerDayReducer.java` in `NYSE.Top3Stocks` package

Refer counter for every try for output variance

If you want not to display key or value in output of reducer, then use NullWritable data type

```
extends Reducer<LongPairPrimitive, Text, NullWritable, Text> {
      context.write(NullWritable.get(), record);
}
```

Try4: Global sorting using multiple output file

You can create user-defined output file names for both mapper and reducer.

There will be lot of small files in the output folder if you execute this reducer. You can cat all files and see the date whether sorted or not. Multiple output file can be created in map and reduce. File names would be like 20120101-r-00001. Only “part” in the file name name is renamed.

Using copyMerge, all small files are put into one big file. There may be empty files but you can ignore them. There is an option to change the output file name of reducer itself.

Set in `FirstKeyLongPairPartitioner.java`, with following code and hash partitioner number of records for each reducer may be balanced. but, it does not help for global sorting.

```
public int getPartition(LongPairPrimitive key, Text value, int
numPartitions) {
      long partValue = key.getFirst();
      return (int) partValue % numPartitions;
}
```

Then write the following reducer

`TopThreeStocksGlobalSortingMultipleFiles.java` in `NYSE.Top3Stocks` package

Exercise 9: Compression

To check whether native libraries are available for specific compression algorithms:

```
$ hadoop checknative -a
```

```
Native library checking:
hadoop:  true /usr/local/hadoop/lib/native/libhadoop.so.1.0.0
zlib:    true /lib/x86_64-linux-gnu/libz.so.1
snappy:  false
lz4:     true revision:99
bzip2:   false
openssl: true /usr/lib/x86_64-linux-gnu/libcrypto.so
17/05/20 11:20:30 INFO util.ExitUtil: Exiting with status 1
```

Based on the above output, snappy, bzip2 are not available as part of Hadoop native libraries. Therefore, while converting Hadoop source code to Hadoop binaries, you have to make sure the right compression gets installed. Snappy, LZO, LZ4 do not have java implementation. They have native implementation such as in C. Check the counters for result discussion.

Way 1: Specify compression configuration in core-site.xml and mapred-site.xml

To specify what compression algorithms to use, edit in core-site.xml:

```
<property>
      <name>io.compression.codecs</name>
      <value>org.apache.hadoop.io.compress.GzipCodec,org.apache.hadoop.ioco
      mpress.DefaultCodec,org.apache.hadoop.io.compress.SnappyCodec</value>
</property>
```

To configure what to compress, edit mapred-site.xml

```
<property>
      <name>mapreduce.compress.map.output</name>
      <value>true</value>
</property>

<property>
      <name>mapreduce.output.fileoutputformat.compress</name>
      <value>true</value>
</property>

<property>
      <name>mapreduce.output.fileoutputformat.compress.type</name>
      <value> BLOCK </value>
</property>
```

Way 2: Specify compression parameters in driver code

`AvgStockVolMonthCompressionDriver.java` in `NYSE.Compression` package

Run the program for dataset without compression

$ yarn jar job.jar NYSE.AvgStockVolMonthDriver /input /output

$ hdfs dfs -ls /output

```
Found 2 items
-rw-r--r--   3 itadmin supergroup     0 2017-05-20 11:46 /output/_SUCCESS
-rw-r--r--3 itadmin supergroup 1500726 2017-05-20 11:46 /output/part-r-00000
```

Run the program with compression

$ yarn jar job.jar NYSE.AvgStockVolMonthCompressionDriver /input /output

Verify the output file part-r-00000 which is appended by .snappy to denote that it has been compressed.

$ hdfs dfs -ls /output

```
Found 2 items
-rw-r--r--    3 itadmin supergroup        0 2017-05-20 11:46 /output/_SUCCESS
-rw-r--r--    3 itadmin supergroup    1200726 2017-05-20 11:46 /output/part-
r-00000.snappy
```

Moreover, for both cases (compression, without compression) notice the size of output and compare the counters in shuffle and input of reducers to see the difference.

Exercise 10: Counters

You can access counters using three ways: command line, counters API, custom counters (user-defined counters and dynamic counters).

Way 1: via command line

The general command line syntax is

$ hadoop command [genericOptions] [commandOptions]

$ mapred

$ mapred job

$ mapred job -counter jobid groupname countername

For group name, counter name – browse https://hadoop.apache.org/docs/r2.7.0/api/

$ mapred job -counter job_1494830873386_0040
org.apache.hadoop.mapreduce.JobCounter DATA_LOCAL_MAPS

When you execute the above command in Hadoop 2.7.0, JHS must be running if job is already done. If job is currently being executed then you need not run JHS to get counters.

Way 2: via counters API in simple java program

If you want to extract by program using API from completed job then

`CounterDriver.java` in `NYSE.Counters` package

$ yarn jar job.jar NYSE.Counters.CounterDriver job_1494830873386_0043 org.apache.hadoop.mapreduce.JobCounter DATA_LOCAL_MAPS

Way 3: custom counters

1. construct user defined counter using enum to find untraded days (where volume is zero). Enum is defined at compile time itself. Best place to create user defined counter for this is reduce task or after job completion statement in main method.

- `TopThreeStocksByVolumePerDayReducer.java` in `NYSE.Counters` package
- `NoTradeDaysCounterDriver.java` in `NYSE.Counters` package
- `TradeDaysEnum.java` in `NYSE.Counters` package

Once job is done look at the counters. You will see the user defined counter name. Ex:

```
NYSE.Counters.NoTradeDays
        NO_TRADE_DAYS=1
```

2. To create counters at runtime, use dynamic counter to find untraded days (where volume is zero)

- `TopThreeStocksByVolumePerDayReducer.java` in `NYSE.Counters` package

```
context.getCounter("TradeDaysEnum","NO_TRADE").increment(1);
```

Exercise 11: Joining trade dataset with company header list dataset

In driver code, if you use job.addCacheFiles() in MRv2 or DistributedCache.addCacheFile() in MRv1, then you need not pass files via -files option. It is mutually exclusive. But, always use -files option, because it gives you more control.

- `CompanyParser.java` in `NYSE.Parsers` package
- `StockCompanyDistCacheJoinMapper.java` in `NYSE.DistributedCache` package
- `StockCompanyJoinDistCacheDriver.java` in `NYSE.DistributedCache` package

Pass NYSE dataset onto HDFS

Have companylist_noheader.csv dataset in home folder itself

$ yarn jar job.jar NYSEDistributedCache.AvgStockVolMonthDriver -files companylist_noheader.csv /input /out101

$ hdfs dfs -get /output/part-r-00000 output-part-1 // here I am renaming while downloading

You can set path to cache files, so, you will see filecache, usercache, nmprivate folders in HDFS containing the files we passed.

1. Copy different file types onto HDFS.

$ hdfs dfs -put lookup.dat map.zip mylib.jar mytargz.tar.gz /data

2. Add local files to distributed cache

```
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "word count");
DistributedCache.addCacheFile(new URI("/data/lookup.dat"),job);
```

```
DistributedCache.addCacheArchive(new URI("/data/map.zip", job);
DistributedCache.addFileToClassPath(new Path("/data/mylib.jar"), job);
DistributedCache.addCacheArchive(new URI("/data/mytargz.tar.gz", job);
```

3. Use the cached files in mapper/reducer

```
public static class MapClass extends Mapper<K, V, K, V> {
   private Path[] localArchives;
   private URI[] localFiles;

   protected void setup(Context context) throws IOException,
                                                    InterruptedException{
           Configuration conf = context.getConfiguration();
           localArchives = context.getLocalCacheArchives(job);
           localFiles = context.getLocalCacheFiles();
           BufferedReader br = new BufferedReader(new
                         FileReader(localFiles[0].toString()));
 }
 public void map(K key, V value, Context context)throws IOException {

           // Use data from the cached archives/files here

           output.write(k, v);
  }
}
```