MapReduce

Data and Analytics Training

Big Data – Self-Study Guide (203)

This study guide serves as a basis for your self-study on the basics of MapReduce. The study guide gives you an overview on the main topics and presents a number of internal and external resources you should use to get a deeper understanding of MapReduce, the related terms and technical details.

Please especially use the external resources provided. They are a main part of this study guide and are required to get a good understanding of the discussed topics.

Estimated Completion Time

4 hours (including time for external resources)

Prerequisites

We recommend completing the courses "*Big Data – 201 – HDFS*" and "*Big Data – 202 – Provisioning HDInsight*" to get a basic understanding of the underlying filesystem and the cluster structure, as well as completing the level 100 courses (especially the course "*Big Data – 101 – Getting started with Big Data*") before starting with this course.

Objectives

After completing this study guide, you will be able to understand

- The concept of MapReduce
- Executing a MapReduce job
- Developing custom MapReduce jobs in Java and C#
- Using HDInsight Emulator to test the application
- Using Azure Blob Storage in MapReduce jobs

Contents

Introduction

Hadoop MapReduce is a software framework for writing applications which process vast amounts of data. To simplify the complexities of analyzing unstructured data from various sources, the MapReduce programming model provides a core abstraction that underwrites closure for map and reduce operations. The MapReduce programming model views all of its jobs as computations over datasets consisting of key-value pairs. So both input and output files must contain datasets that consist only of key-value pairs. The primary takeaway from this constraint is that the MapReduce jobs are, as a result, composable.

The map function is written to accept a single line of incoming data and is expected to emit output data in the form of key and value combinations. A typical map function emits a single key and value derived from the incoming data, but it is possible that the map function generates zero or multiple key and value combinations in response to the incoming data.

Once the map tasks have completed their work, the data output from their various calls of the map function are sorted and shuffled so that all values associated with a given key are arrayed together. Reduce tasks are generated and keys and their values are handed over to the tasks for processing.

As part of the MapReduce job definition, a Reducer class was defined. This class contains a reduce function and this function is supplied a key value and its array of associated values for processing by the reduce task. The reduce function typically generates a summarized value derived across the array of values for the given key but as before the developer maintains quite a bit of flexibility in terms of what output is emitted and how it is derived.

The reduce function output for each reducer task is written to a file in the output folder identified as part of the MapReduce job's configuration. A single file is associated with each reduce task so that in our example with two reduce tasks there are two output files produced. These files can be accessed individually but more typically they are combined into a single output file using the getmerge command (at the command line) or similar functionality.

Not every job consists of both a Mapper and a Reducer class. At a minimum, a MapReduce job must have a Mapper class but if all the data processing work can be tackled through the map function, that will be the end of the road. In that case the output files align with the map tasks and will be found with similar names in the output folder identified with the job configuration.

In addition, MapReduce jobs can accept a Combiner class. The Combiner class is defined like the Reducer class (and has a reduce function) but it is run on the data associated with a single map task. The idea behind the Combiner class is that some data can be "reduced" (summarized) before it is sorted and shuffled. Sorting and shuffling data to get it over to the reduce tasks is an expensive operation and the Combiner class can serve as an optimizer to reduce the amount of data moving between tasks. Combiner classes are by no means required and you should consider using them when you absolutely must squeeze performance out of our MapReduce job.[1]

Map

Here's an example that really just shows the principle on which all of this is based. The ideas is that the map phase is where different chunks of the data is taken, it is spread across multiple data nodes, which are all accessing the same HDFS files store, take a chunk of the data, and do what's called the map function.

Depending on how the map function has been written, it parses through the source data and generates key-value pairs from that source data. Each of the nodes is doing that in parallel, which results in saving of time by having a whole rack of servers, if that initial phase is effeftively shared between them. Classically once the map is done the output is a set of key-value pairs.

When thinking about processing a large amount of data and e.g. trying to find the percentage of user base that was talking about games[2]:

First, we will identify the keywords which we are going to map from the data to conclude that its something related to games. Next, we will write a mapping function to identify such patterns in our data. For example, the keywords can be Gold medals, Bronze medals, Silver medals, Olympic football, basketball, cricket, etc.

Let us take the following chunks in a big data set and see how to process it.

"Hi, how are you"
"We love football"
"He is an awesome football player"
"Merry Christmas"

[1] http://blogs.msdn.com/b/data_otaku/archive/2013/09/04/hadoop-for-net-developers-understanding-mapreduce.aspx

[2] http://www.thegeekstuff.com/2014/05/map-reduce-algorithm/

"Olympics will be held in China"
"Records broken today in Olympics"
"Yes, we won 2 Gold medals"
"He qualified for Olympics"

So our map phase of our algorithm will be as follows:

1. Declare a function "Map"
2. Loop: For each words equal to "football"
3. Increment counter
4. Return key value "football"=>counter

In the same way, we can define a number of mapping functions for mapping various words: "Olympics", "Gold Medals", "cricket", etc.

Reduce

The next steps is the feeding of these key-value pairs into the Reduce phase. In that phase a reducer function runs and takes the output from different map operations on the different nodes and combines them to create the collated results.

The reducing function accepts the input from all these mappers in form of key value pair and then processes those. The input to the reduce function will look like the following[3]:

reduce("football"=>2)

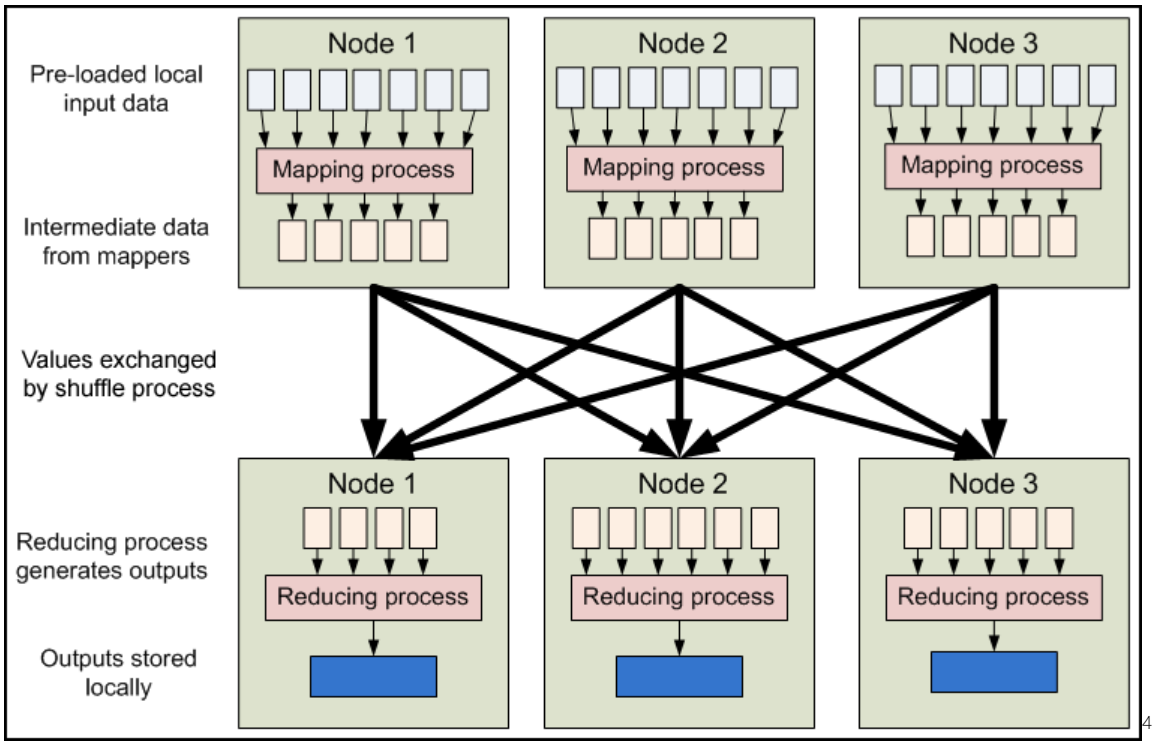
reduce("Olympics"=>3)

Our example algorithm will continue with the following steps:

5. Declare a function reduce to accept the values from map function.
6. Where for each key-value pair, add value to counter.
7. Return "games"=> counter.

At the end, we will get the output like "games"=>5.

[3] http://www.thegeekstuff.com/2014/05/map-reduce-algorithm/



[4]

MapReduce inputs typically come from input files loaded onto our processing cluster in HDFS. These files are evenly distributed across all our nodes. Running a MapReduce program involves running mapping tasks on many or all of the nodes in our cluster. Each of these mapping tasks is equivalent: no mappers have particular "identities" associated with them. Therefore, any mapper can process any input file. Each mapper loads the set of files local to that machine and processes them.

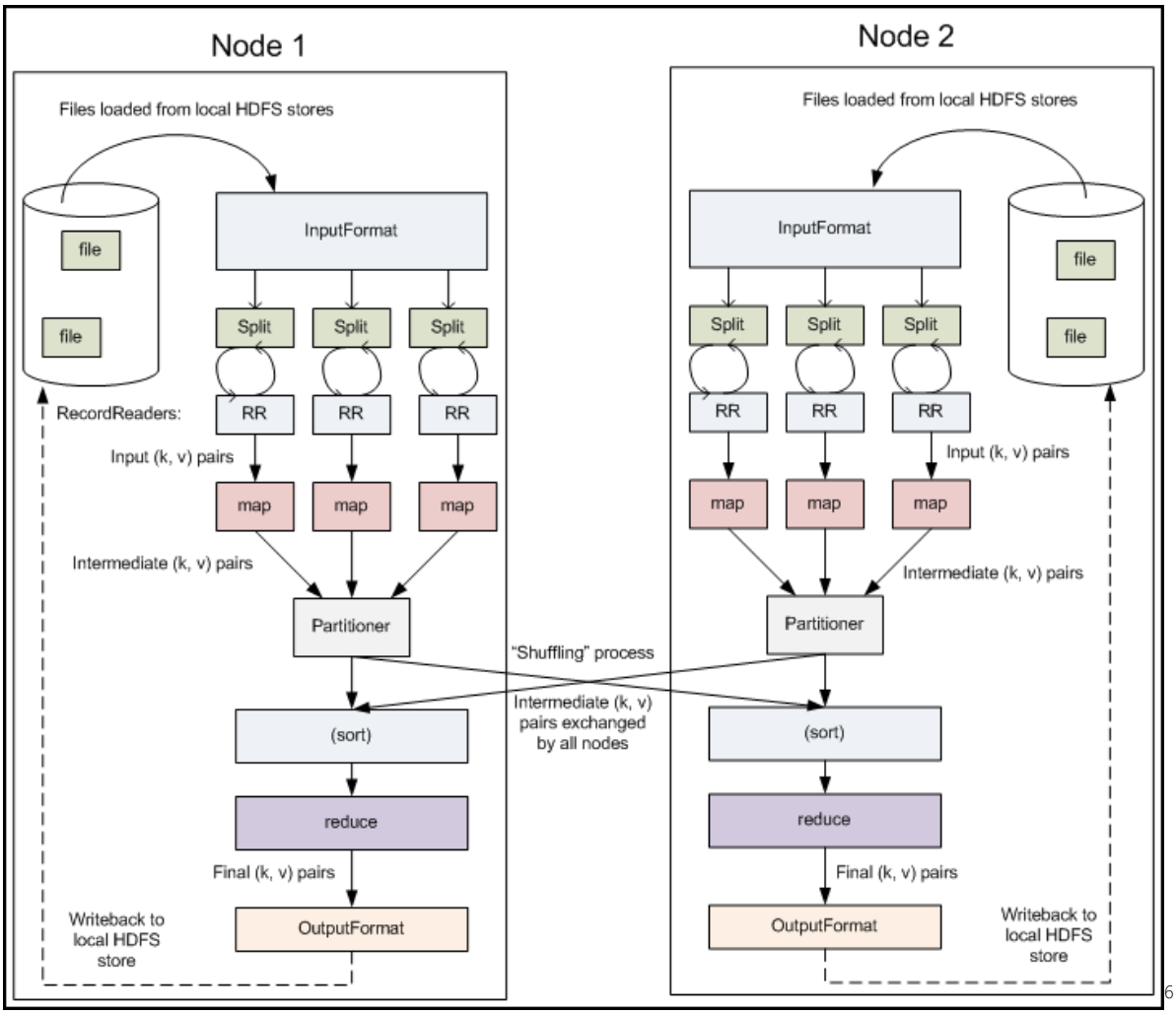
When the mapping phase has completed, the intermediate (key, value) pairs must be exchanged between machines to send all values with the same key to a single reducer. The reduce tasks are spread across the same nodes in the cluster as the mappers. This is the only communication step in MapReduce. Individual map tasks do not exchange information with one another, nor are they aware of one another's existence. Similarly, different reduce tasks do not communicate with one another. The user never explicitly marshals information from one machine to another; all data transfer is handled by the Hadoop MapReduce platform itself, guided implicitly by the different keys associated with values. This is a fundamental element of Hadoop MapReduce's reliability. If nodes in the cluster fail, tasks must be able to be restarted. If they have been performing *side-effects*, e.g., communicating

[4] https://developer.yahoo.com/hadoop/tutorial/module4.html#basics

with the outside world, then the shared state must be restored in a restarted task. By eliminating communication and side-effects, restarts can be handled more gracefully.[5]

Split, Record Reader, Shuffle

From this diagram, you can see where the mapper and reducer components of the Word Count application fit in, and how it achieves its objective. We will now examine this system in a bit closer detail.



[6]

The default InputFormat is the *TextInputFormat*. This treats each line of each input file as a separate record, and performs no parsing. This is useful for unformatted data or line-based records like log

[5] https://developer.yahoo.com/hadoop/tutorial/module4.html#basics

[6] https://developer.yahoo.com/hadoop/tutorial/module4.html#basics

files. A more interesting input format is the *KeyValueInputFormat*. This format also treats each line of input as a separate record. While the TextInputFormat treats the entire line as the value, the KeyValueInputFormat breaks the line itself into the key and value by searching for a tab character. This is particularly useful for reading the output of one MapReduce job as the input to another, as the default OutputFormat (described in more detail below) formats its results in this manner. Finally, the SequenceFileInputFormat reads special binary files that are specific to Hadoop. These files include many features designed to allow data to be rapidly read into Hadoop mappers. Sequence files are block-compressed and provide direct serialization and deserialization of several arbitrary data types (not just text). Sequence files can be generated as the output of other MapReduce tasks and are an efficient intermediate representation for data that is passing from one MapReduce job to anther.

An InputSplit describes a unit of work that comprises a single map task in a MapReduce program. A MapReduce program applied to a data set, collectively referred to as a Job, is made up of several (possibly several hundred) tasks. Map tasks may involve reading a whole file; they often involve reading only part of a file. By default, the FileInputFormat and its descendants break a file up into 64 MB chunks (the same size as blocks in HDFS). You can control this value by setting the mapred.min.split.size parameter in hadoop-site.xml, or by overriding the parameter in the JobConf object used to submit a particular MapReduce job.

Record Reader

The InputSplit has defined a slice of work, but does not describe how to access it. The *RecordReader* class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper. The RecordReader instance is defined by the InputFormat. The default InputFormat, *TextInputFormat*, provides a *LineRecordReader*, which treats each line of the input file as a new value. The key associated with each line is its byte offset in the file. The RecordReader is invoke repeatedly on the input until the entire InputSplit has been consumed. Each invocation of the RecordReader leads to another call to the `map()` method of the Mapper.

Split

By processing a file in chunks, we allow several map tasks to operate on a single file in parallel. If the file is very large, this can improve performance significantly through parallelism. Even more importantly, since the various blocks that make up the file may be spread across several different nodes in the cluster, it allows tasks to be scheduled on each of these different nodes; the individual blocks are thus all processed locally, instead of needing to be transferred from one node to another. Of course, while log files can be processed in this piece-wise fashion, some file formats are not amenable to chunked processing. By writing a custom InputFormat, you can control how the file is broken up (or is not broken up) into splits.

Shuffling and Paritioning

After the first map tasks have completed, the nodes may still be performing several more map tasks each. But they also begin exchanging the intermediate outputs from the map tasks to where they are required by the reducers. This process of moving map outputs to the reducers is known as *shuffling*. A different subset of the intermediate key space is assigned to each reduce node; these subsets (known as "partitions") are the inputs to the reduce tasks. Each map task may emit (key, value) pairs to any partition; all values for the same key are always reduced together regardless of which mapper is its origin. Therefore, the map nodes must all agree on where to send the different pieces of the intermediate data.

The *Partitioner* class determines which partition a given (key, value) pair will go to. The default partitioner computes a hash value for the key and assigns the partition based on this result.

To understand more details on the MapReduce Data Flow and Chaining, we recommend reading this article:

https://developer.yahoo.com/hadoop/tutorial/module4.html

Combiner

When the map operation outputs its pairs they are already available in memory. For efficiency reasons, sometimes it makes sense to take advantage of this fact by supplying a combiner class to perform a reduce-type function. If a combiner is used then the map key-value pairs are not immediately written to the output. Instead they will be collected in lists, one list per each key value. When a certain number of key-value pairs have been written, this buffer is flushed by passing all the values of each key to the combiner's reduce method and outputting the key-value pairs of the combine operation as if they were created by the original map operation.

For example, a word count MapReduce application whose map operation outputs (word , 1) pairs as words are encountered in the input can use a combiner to speed up processing. A combine operation will start gathering the output in in-memory lists (instead of on disk), one list per word. Once a certain number of pairs is output, the combine operation will be called once per unique word with the list available as an iterator. The combiner then emits (word , count-in-this-part-of-the-input) pairs. From the viewpoint of the Reduce operation this contains the same information as the original Map output, but there should be far fewer pairs output to disk and read from disk.[7]

Combiners are used to increase the efficiency of a MapReduce program. They are used to aggregate intermediate map output locally on individual mapper outputs. Combiners can help you reduce the

[7] http://wiki.apache.org/hadoop/HadoopMapReduce

amount of data that needs to be transferred across to the reducers. You can use your reducer code as a combiner if the operation performed is commutative and associative. The execution of combiner is not guaranteed, Hadoop may or may not execute a combiner. Also, if required it may execute it more then 1 times. Therefore your MapReduce jobs should not depend on the combiners execution.[8]

You can find more details on the Combiner here:

http://hadooptutorial.info/combiner-in-mapreduce/

Running a MapReduce job

In this section, we will use Azure PowerShell from our workstation to submit a MapReduce program that counts word occurrences in a text to an HDInsight cluster. The word counting program is written in Java and the program comes with the HDInsight cluster. The word counting program already exists as an example on the provisoned server, please note that we are not newly creating the Map and/or Reduce function in this example, but are really only executing the job on the cluster.

To run this example you should have completed the "*Big Data – 202 – Provisioning HDInsight*" module, where details on the sewtup of a cluster in Azure is explained.

Running a MapReduce job requires the following elements:

- A MapReduce program. In this section, you will use the word counting sample that comes with HDInsight clusters so you don't need to write your own. It is located on */example/jars/hadoop-mapreduce-examples.jar*.
- An input file. You will use */example/data/gutenberg/davinci.txt* as the input file.
- An output file folder. You will use */example/data/WordCountOutput* as the output file folder. The MapReduce job will fail if the folder exists. If you want to run the MapReduce job for the second time, make sure to delete the output folder or specify another output folder.

Using PowerShell

1. Open Azure PowerShell.
2. Set the two variables in the following commands, and then run them:

[8] http://www.fromdev.com/2010/12/interview-questions-hadoop-mapreduce.html

```
$subscriptionName = "<SubscriptionName>"    # Azure subscription name
$clusterName = "<ClusterName>"              # HDInsight cluster name
```

3. Run the following command and provide your Azure account information:

```
Add-AzureAccount
```

4. Run the following commands to create a MapReduce job definition:

```
# Define the MapReduce job
$wordCountJobDefinition = New-AzureHDInsightMapReduceJobDefinition `
       -JarFile "wasb:///example/jars/hadoop-mapreduce-examples.jar" `
       -ClassName "wordcount" `
       -Arguments "wasb:///example/data/gutenberg/davinci.txt", "wasb:///example/data/WordCountOutput"
```

The hadoop-mapreduce-examples.jar file comes with the HDInsight cluster distribution. There are two arguments for the MapReduce job. The first one is the source file name, and the second is the output file path. The source file comes with the HDInsight cluster distribution, and the output file path will be created at the run-time.

5. Run the following command to submit the MapReduce job:

```
# Submit the job
Select-AzureSubscription $subscriptionName
$wordCountJob = Start-AzureHDInsightJob `
                    -Cluster $clusterName `
                    -JobDefinition $wordCountJobDefinition
```

In addition to the MapReduce job definition, you also provide the HDInsight cluster name where you want to run the MapReduce job, and the credentials. The Start-AzureHDInsightJob is an asynchronized call. To check the completion of the job, use the Wait-AzureHDInsightJob cmdlet.

6. Run the following command to check the completion of the MapReduce job:

```
Wait-AzureHDInsightJob -Job $wordCountJob -WaitTimeoutInSeconds 3600
```

7. Run the following command to check any errors with running the MapReduce job:

```
# Get the job output
Get-AzureHDInsightJobOutput -Cluster $clusterName `
                            -JobId $wordCountJob.JobId `
                            -StandardError
```

Retrieve Results

The output of the MapReduce job is a set of key-value pairs. The key is a string that specifies a word and the value is an integer that specifies the total number of occurrences of that word in the text. This is done in two stages:

The mapper takes each line from the input text as an input and breaks it into words. It emits a key/value pair each time a work occurs of the word followed by a 1. The output will be sorted before sending to reducer.

The reducer then sums these individual counts for each word and emits a single key/value pair containing the word followed by the sum of its occurrences.

1. Open Azure PowerShell
2. Run the following command to change directory to c:\ root:

```
cd \
```

The default Azure Powershell directory is C:\Windows\System32\WindowsPowerShell\v1.0. By default, you don't have the write permission on this folder. You must change directory to either the C:\ root directory or a folder where you have write permission.

```
$subscriptionName = "<SubscriptionName>"        # Azure subscription name

$storageAccountName = "<StorageAccountName>"    # Azure storage account name
$containerName = "<ContainerName>"              # Blob storage container name
```

3. Set the three variables in the following commands, and then run them:

   The Azure Storage account is the one you created earlier in the tutorial. The storage account is used to host the Blob container that is used as the default HDInsight cluster file system. The Blob storage container name usually share the same name as the HDInsight cluster unless you specify a different name when you provision the cluster.

4. Run the following commands to create an Azure storage context object:

```
# Select the current subscription
Select-AzureSubscription $subscriptionName

# Create the storage account context object
$storageAccountKey = Get-AzureStorageKey $storageAccountName | %{ $_.Primary }
$storageContext = New-AzureStorageContext `
                    -StorageAccountName $storageAccountName `
                    -StorageAccountKey $storageAccountKey
```

The *Select-AzureSubscription* is used to set the current subscription in case you have multiple subscriptions, and the default subscription is not the one to use

5. Run the following command to download the MapReduce job output from the Blob container to the workstation:

```
# Download the job output to the workstation
Get-AzureStorageBlobContent -Container $ContainerName `
                            -Blob example/data/WordCountOutput/part-r-00000 `
                            -Context $storageContext `
                            -Force
```

The */example/data/WordCountOutput* folder is the output folder specified when you run the MapReduce job. part-r-00000 is the default file name for MapReduce job output. The file will be downloaded to the same folder structure on the local folder. For example, if the current folder is the C root folder. The file will be downloaded to the *C:\example\data\WordCountOutput folder*.

6. Run the following command to print the MapReduce job output file:

```
cat ./example/data/WordCountOutput/part-r-00000 | findstr "there"
```

The MapReduce job produces a file named *part-r-00000* with the words and the counts. The script uses the findstr command to list all of the words that contains "*there*".

Note that the output files of a MapReduce job are immutable. So if you rerun this sample you will need to change the name of the output file.

Number of Maps and Reduces

Picking the appropriate size for the tasks for your job can radically change the performance of Hadoop. Increasing the number of tasks increases the framework overhead, but increases load balancing and lowers the cost of failures. At one extreme is the 1 map/1 reduce case where nothing is distributed. The other extreme is to have 1,000,000 maps/ 1,000,000 reduces where the framework runs out of resources for the overhead.[9]

[9] http://wiki.apache.org/hadoop/HowManyMapsAndReduces

Number of Maps

The number of maps is usually driven by the number of DFS blocks in the input files. Although that causes people to adjust their DFS block size to adjust the number of maps. The right level of parallelism for maps seems to be around 10-100 maps/node, although we have taken it up to 300 or so for very cpu-light map tasks. Task setup takes awhile, so it is best if the maps take at least a minute to execute.

Actually controlling the number of maps is subtle. The mapred.map.tasks parameter is just a hint to the InputFormat for the number of maps. The default InputFormat behavior is to split the total number of bytes into the right number of fragments. However, in the default case the DFS block size of the input files is treated as an upper bound for input splits. A lower bound on the split size can be set via mapred.min.split.size. Thus, if you expect 10TB of input data and have 128MB DFS blocks, you'll end up with 82k maps, unless your mapred.map.tasks is even larger. Ultimately the InputFormat determines the number of maps.

The number of map tasks can also be increased manually using the JobConf's conf.setNumMapTasks(int num). This can be used to increase the number of map tasks, but will not set the number below that which Hadoop determines via splitting the input data.

Number of Reduces

The ideal reducers should be the optimal value that gets them closest to:

- A multiple of the block size
- A task time between 5 and 15 minutes
- Creates the fewest files possible

Anything other than that means there is a good chance your reducers are less than great. There is a tremendous tendency for users to use a REALLY high value ("More parallelism means faster!") or a REALLY low value ("I don't want to blow my namespace quota!"). Both are equally dangerous, resulting in one or more of:

Terrible performance on the next phase of the workflow; Terrible performance due to the shuffle; Terrible overall performance because you've overloaded the namenode with objects that are ultimately useless; Destroying disk IO for no really sane reason; Lots of network transfers due to dealing with crazy amounts of CFIF/MFIF work

Now, there are always exceptions and special cases. One particular special case is that if following that advice makes the next step in the workflow do ridiculous things, then we need to likely 'be an exception' in the above general rules of thumb.

Currently the number of reduces is limited to roughly 1000 by the buffer size for the output files (io.buffer.size * 2 * numReduces << heapSize). This will be fixed at some point, but until it is it provides a pretty firm upper bound.

The number of reduce tasks can also be increased in the same way as the map tasks, via JobConf's conf.setNumReduceTasks(int num).[10]

Setting the number of reducers to zero is a valid configuration in Hadoop. When you set the reducers to zero no reducers will be executed, and the output of each mapper will be stored to a separate file on HDFS. [This is different from the condition when reducers are set to a number greater than zero and the Mappers output (intermediate data) is written to the Local file system(NOT HDFS) of each mappter slave node.][11]

Creating a custom MapReduce job

You can write MapReduce jobs by coding them in Java, but also by using other languages like C# by using the Hadoop streaming utility.

Develop MapReduce job in Java

Let's start by creating a MapReduce job in Java, which is the language Hadoop was originally designed in.

This exercise is using Apache Maven. Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.[12]

Maven gives you the ability to easily create the folder structure and basic files that are required to develop a MapReduce job.

You can download Maven here into your development environment:
http://maven.apache.org/download.cgi#Installation

Afterwards you can follow this step by step tutorial to create and deploy a MapReduce job to an HDInsight cluster or on the HDInsight emulator:

---

[10] http://wiki.apache.org/hadoop/HowManyMapsAndReduces

[11] http://www.fromdev.com/2010/12/interview-questions-hadoop-mapreduce.html

[12] http://maven.apache.org/

http://azure.microsoft.com/en-us/documentation/articles/hdinsight-develop-deploy-java-mapreduce/

Implementing a simple MapReduce job in C#

In this exercise, we will write and execute a very simple MapReduce job using C# and the .NET SDK. The purpose of this exercise is to illustrate the most basic concepts behind MapReduce.[13]

The map function we will write will accept a line (a single integer), determine whether the value is even or odd, and emit the integer value with a key of "even" or "odd", accordingly. The reduce function we will write will accept all of integer values associated with a given key, ***i.e.*** "even" or "odd", and then count and sum those values. The reduce function will then emit results in a key-value format with "even" or "odd" again serving as the key and the associated count and sum separated by a simple tab as the value.

You can find the sample file inters.txt that is used in this tutorial here:

http://blogs.msdn.com/cfs-file.ashx/__key/communityserver-components-postattachments/00-10-44-20-50/samples.zip

After downloading this file to your development environment you can go ahead by following the steos defined here:
http://blogs.msdn.com/b/data_otaku/archive/2013/09/07/hadoop-for-net-developers-implementing-a-simple-mapreduce-job.aspx

Develop Hadoop streaming program in C#

Hadoop Streaming is a utility which allows users to create and run jobs as executables (e.g. shell utilities) as the mapper and/or the reducer. In other words, it enables you to write map and reduce functions [14]:

In the example, both the mapper and the reducer are executables that read the input from stdin (line by line) and emit the output to stdout. The program counts all of the words in the text.

When an executable is specified for mappers, each mapper task launches the executable as a separate process when the mapper is initialized. As the mapper task runs, it converts its inputs into lines and feeds the lines to the stdin of the process. In the meantime, the mapper collects the line-

[13] http://blogs.msdn.com/b/data_otaku/archive/2013/09/07/hadoop-for-net-developers-implementing-a-simple-mapreduce-job.aspx

[14] http://azure.microsoft.com/en-us/documentation/articles/hdinsight-sample-csharp-streaming/

oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper.

To create the Mapper program: As specified above this is a simple ConsoleApplication. After building the project, you should see one executable for each of the below classes in the respective build folder.

You can find details on how to create a console application here:
http://msdn.microsoft.com/en-us/library/452fz12a%28v=vs.90%29.aspx

```
class WordCountMapper
{
    static void Main(string[] args)
    {
        if (args.Length > 0)
            Console.SetIn(new StreamReader(args[0]));


        string line;
        string[] words;

        while ((line = Console.ReadLine()) != null)
        {
            words = line.Split(' ');

            foreach (string word in words)
                Console.WriteLine(word.ToLower());
        }
    }
}
```

When an executable is specified for reducers, each reducer task launches the executable as a separate process when the reducer is initialized. As the reducer task runs, it converts its input key/values pairs into lines and feeds the lines to the stdin of the process. In the meantime, the reducer collects the line-oriented outputs from the stdout of the process, converts each line into a key/value pair, which is collected as the output of the reducer. By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) is the value.

To create the Reducer program: Follow the same steps as specified for the Mapper program and use the following code.

```
class WordCountReducer
{
    static void Main(string[] args)
    {
        string word, lastWord = null;
        int count = 0;

        if (args.Length > 0)
            Console.SetIn(new StreamReader(args[0]));

        while ((word = Console.ReadLine()) != null)
            if (word != lastWord)
            {
                if (lastWord != null)
                    Console.WriteLine("{0}[{1}]", lastWord, count);

                count = 1;
                lastWord = word;
            }
            else
                count += 1;

        Console.WriteLine(count);
    }
}
```

After building the code or running the debugger, the executables should be located in a folder similar to the below.
Lets assume the Mapper and the Reducer executables are located at:

- C:\Tutorials\WordCount\WordCountMapper\bin\Debug\WordCountMapper.exe
- C:\Tutorials\WordCount\WordCountReducer\bin\Debug\WordCountReducer.exe

If not done yet, you can use the HDInsight emulator in your development environment to test the program. To do so, please follow the steps defined here:
http://azure.microsoft.com/en-us/documentation/articles/hdinsight-get-started-emulator/

This tutorial uses the following folder structure:

Folder	Note
\WordCount	The root folder for the word count project.
\WordCount\Apps	The folder for the mapper and reducer executables.

\WordCount\Input	The MapReduce source file folder.
\WordCount\Output	The MapReduce output file folder.
\WordCount\MRStatusOutput	The job output folder.

This tutorial uses the .txt files located in the %hadoop_home% directory.

NOTE: The Hadoop HDFS commands are case sensitive.

From the Hadoop command line window, run the following command to make a directory for the input files:

```
hadoop fs -mkdir /WordCount/
hadoop fs -mkdir /WordCount/Input
```

Run the following command to copy some text files to the input folder on HDFS:

```
hadoop fs -copyFromLocal %hadoop_home%\share\doc\hadoop\common\*.txt \WordCount\Input
```

Run the following command to list the uploaded files:

```
hadoop fs -ls \WordCount\Input
```

Upload the Mapper and Reducer applications to the Emulator HDFS

Open Hadoop command line from your desktop and create the /Apps folder in HDFS

```
hadoop fs -mkdir /WordCount/Apps
```

Run the following commands:

```
hadoop fs -copyFromLocal
C:\Tutorials\WordCount\WordCountMapper\bin\Debug\WordCountMapper.exe
/WordCount/Apps/WordCountMapper.exe

hadoop fs -copyFromLocal
C:\Tutorials\WordCount\WordCountReducer\bin\Debug\WordCountReducer.exe
/WordCount/Apps/WordCountReducer.exe
```

Run the following command to list the uploaded files:

```
hadoop fs -ls /WordCount/Apps
```

You shall see the two .exe files.

Submit a word count MapReduce job

1. Open Azure PowerShell
2. Run the following commands to set variables:

```
$clusterName = "http://localhost:50111"

$mrMapper = "WordCountMapper.exe"
$mrReducer = "WordCountReducer.exe"
$mrMapperFile = "/WordCount/Apps/WordCountMapper.exe"
$mrReducerFile = "/WordCount/Apps/WordCountReducer.exe"
$mrInput = "/WordCount/Input/"
$mrOutput = "/WordCount/Output"
$mrStatusOutput = "/WordCount/MRStatusOutput"
```

   The HDInsight emulator cluster name is "http://localhost:50111".

3. Run the following commands to define the streaming job:

```
$mrJobDef = New-AzureHDInsightStreamingMapReduceJobDefinition `
                -JobName mrWordCountStreamingJob `
                -StatusFolder $mrStatusOutput `
                -Mapper $mrMapper `
                -Reducer $mrReducer `
                -InputPath $mrInput `
                -OutputPath $mrOutput

$mrJobDef.Files.Add($mrMapperFile)
$mrJobDef.Files.Add($mrReducerFile)
```

4. Run the following command to create a credential object:

```
$creds = Get-Credential -Message "Enter password" -UserName "hadoop"
```

   You will get a prompt to enter the password. The password can be any string. The username must be "hadoop".

5. Run the following commands to submit the MapReduce job and wait for the job to complete:

```
$mrJob = Start-AzureHDInsightJob `
            -Cluster $clusterName `
            -Credential $creds `
            -JobDefinition $mrJobDef

Wait-AzureHDInsightJob `
    -Credential $creds `
    -job $mrJob `
    -WaitTimeoutInSeconds 3600
```

When completed, you will get an output similar to the following:

```
StatusDirectory : /WordCount/MRStatusOutput
ExitCode        :
Name            : mrWordCountStreamingJob
Query           :
State           : Completed
SubmissionTime  : 11/15/2013 7:18:16 PM
Cluster         : http://localhost:50111
PercentComplete : map 100%  reduce 100%
JobId           : job_201311132317_0034
```

You can see the job ID in the output, for example, job-201311132317-0034.

Check the job status

1. From the desktop, click Hadoop YARN Status, or browse to http://localhost:50030/jobtracker.jsp.

2. Find the job using the job id either under RUNNING or FINISHED category.

3. If a job failed, you can find it under the FAILED category. You can also open the job details and find some helpful information for debugging.

Retrieve the job results

1. Open Hadoop command line
2. Run the following commands to display the output:

```
hadoop fs -ls /WordCount/Output/
hadoop fs -cat /WordCount/Output/part-00000
```

You can append "|more" at the end of the command to get the page view.

```
hadoop fs -cat /WordCount/Output/part-00000 | more
```

Using Azure Blob Storage

In the above example we have only executed the jobs on small sample files. In rfeal world scenarios it's of course more common that you have to execute your jobs on large amounts of data. In the

Azure environment it may also be very common that these are stored on the HDFS implementation of Azure: The Azure blob storage.

You can find details on the HDFS in the "*Big Data – 201 – HDFS*" module.

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. Following are differences between HDFS and NAS:[15]

- In HDFS Data Blocks are distributed across local drives of all machines in a cluster. Whereas in NAS data is stored on dedicated hardware.
- HDFS is designed to work with MapReduce System, since computation are moved to data. NAS is not suitable for MapReduce since data is stored seperately from the computations.
- HDFS runs on a cluster of machines and provides redundancy usinga replication protocal. Whereas NAS is provided by a single machine therefore does not provide data redundancy.

Azure HDInsight uses Azure Blob storage as the default file system. In this section, you will create a storage account and upload the data files to the Blob storage. The data files are the .txt files in the *%hadoop_home%\share\doc\hadoop\common* directory

Create a Blob storage and a container

1. Open Azure PowerShell.
2. Set the variables, and then run the commands:

```
$subscriptionName = "<AzureSubscriptionName>"
$storageAccountName = "<AzureStorageAccountName>"
$containerName = "<ContainerName>"
$location = "<MicrosoftDataCenter>"  # For example, "East US"
```

[15] http://www.fromdev.com/2010/12/interview-questions-hadoop-mapreduce.html

3. Run the following command to create a storage account and a Blob storage container on the account:

```
# Select Azure subscription
Select-AzureSubscription $subscriptionName

# Create a storage account
New-AzureStorageAccount `
    -StorageAccountName $storageAccountName `
    -location $location

# Create a Blob storage container
$storageAccountKey = Get-AzureStorageKey $storageAccountName | %{ $_.Primary }

$destContext = New-AzureStorageContext `
                  -StorageAccountName $storageAccountName `
                  -StorageAccountKey $storageAccountKey

New-AzureStorageContainer `
    -Name $containerName `
    -Context $destContext
```

4. Run the following commands to verify the storage account and the container:

```
Get-AzureStorageAccount -StorageAccountName $storageAccountName

Get-AzureStorageContainer -Context $destContext
```

Upload the data files

1. In the Azure PowerShell window, set the values for the local folder and destination folders.

```
$localFolder = "C:\hdp\hadoop-2.4.0.2.1.3.0-1981\share\doc\hadoop\common"

$destFolder = "WordCount/Input"
```

Notice the local source file folder is C:\hdp\hadoop-2.4.0.2.1.3.0-1981\share\doc\hadoop\common, and the destination folder is WordCount/Input. The source location is the location of the .txt files on the HDInsight Emulator. The destination is the folder structure that will be reflected under the Azure Blob container.

2. Run the following commands to get a list of the .txt files in the source file folder:

```
# Get a list of the txt files
$filesAll = Get-ChildItem $localFolder

$filesTxt = $filesAll | where {$_.Extension -eq ".txt"}
```

3. Run the following snippet to copy the files:

```
# Copy the file from local workstation to the Blob container
foreach ($file in $filesTxt){

    $fileName = "$localFolder\$file"
    $blobName = "$destFolder/$file"

    write-host "Copying $fileName to $blobName"

    Set-AzureStorageBlobContent `
        -File $fileName `
        -Container $containerName `
        -Blob $blobName `
        -Context $destContext
}
```

4. Run the following command to list the uploaded files:

```
# List the uploaded files in the Blob storage container
Get-AzureStorageBlob `
    -Container $containerName `
    -Context $destContext `
    -Prefix $destFolder
```

Upload the word count applications

1. In the Azure PowerShell window, set the following variables

```
$mapperFile = "C:\Tutorials\WordCount\WordCountMapper\bin\Debug\WordCountMapper.exe"
$reducerFile = "C:\Tutorials\WordCount\WordCountReducer\bin\Debug\WordCountReducer.exe"
$blobFolder = "WordCount/Apps"
```

   Notice the destination folder is WordCount/Apps, which is the structure that will be reflected in the Azure Blob container.

2. Run the following commands to copy the applications:

```
Set-AzureStorageBlobContent `
    -File $mapperFile `
    -Container $containerName `
    -Blob "$blobFolder/WordCountMapper.exe" `
    -Context $destContext

Set-AzureStorageBlobContent `
    -File $reducerFile `
    -Container $containerName `
    -Blob "$blobFolder/WordCountReducer.exe" `
    -Context $destContext
```

3. Run the following command to list the uploaded files:

```
# List the uploaded files in the Blob storage container
Get-AzureStorageBlob `
    -Container $containerName `
    -Context $destContext `
    -Prefix $blobFolder
```

You shall see both application files listed there.

Run the MapReduce job on Azure HDInsight

The PowerShell commands to create and run the MapReduce job has been described previously in this module.

Retrieve the MapReduce job output

This section shows you how to download and display the output.

1. Open Azure PowerShell window
2. Set the values and then run the commands:

```
$subscriptionName = "<AzureSubscriptionName>"
$storageAccountName = "<TheDataStorageAccountName>"
$containerName = "<TheDataBlobStorageContainerName>"
$blobName = "WordCount/Output/part-00000"
```

3. Run the following commands to create an Azure storage context object:

```
Select-AzureSubscription $subscriptionName

$storageAccountKey = Get-AzureStorageKey $storageAccountName | %{ $_.Primary }

$storageContext = New-AzureStorageContext `
                        -StorageAccountName $storageAccountName `
                        -StorageAccountKey $storageAccountKey
```

4. Run the following commands to download and display the output:

```
Get-AzureStorageBlobContent `
    -Container $ContainerName `
    -Blob $blobName `
    -Context $storageContext `
    -Force

cat "./$blobName" | findstr "there"
```

Handling failures in MapReduce

Debugging distributed programs is always difficult, because very few debuggers will let you connect to a remote program that wasn't run with the proper command line arguments.[16]

Task failure

Task instances are the actual MapReduce jobs which are run on each slave node. The TaskTracker starts a separate JVM processes to do the actual work (called as Task Instance) this is to ensure that process failure does not take down the task tracker. Each Task Instance runs on its own JVM process. There can be multiple processes of task instance running on a slave node. This is based on the number of slots configured on task tracker. By default a new task instance JVM process is spawned for a task.[17]

- Occurs mostly due to a run-time exception in user code-either map or reduce function.
- Can also occur due to a bug in JVM itself , which is exposed because of the child process.
- In such case the child JVM running the process exits
- TaskRunner notices this and informs the same to the JobRunner.
- JobRunner then reschedules the task to be run on different task tracker.
- If a task fails for 4 (configurable) times, it marked as failed and is not tried again.
- Now a job can fail even if a single task fails. or we can configure the % of tasks to fail before a job is considered failed by job runner.
- In case of streaming task , it is considered failed if the streaming process exists with a non zero code. However this can also be configured to be considered as success
- In case of speculative execution, tasks are killed. This is not considered as task failure.
- Hanging tasks are killed automatically, if a task tracker has not received any update for a configurable amount of time. These tasks are then considered as failed.
- If the time ouu time is set to 0, then tasks will never time out[18]

TaskTracker Failure

A TaskTracker is a slave node daemon in the cluster that accepts tasks (Map, Reduce and Shuffle operations) from a JobTracker. There is only One Task Tracker process run on any hadoop slave node. Task Tracker runs on its own JVM process. Every TaskTracker is configured with a set of slots, these indicate the number of tasks that it can accept. The TaskTracker starts a separate JVM processes

[16] http://wiki.apache.org/hadoop/HowToDebugMapReducePrograms

[17] http://www.fromdev.com/2010/12/interview-questions-hadoop-mapreduce.html#What-is-TastTracker

[18] http://blog.abhinavmathur.net/2013/01/failure-handling-in-classic-map-reduce.html

to do the actual work (called as Task Instance) this is to ensure that process failure does not take down the task tracker. The TaskTracker monitors these task instances, capturing the output and exit codes. When the Task instances finish, successfully or not, the task tracker notifies the JobTracker. The TaskTrackers also send out heartbeat messages to the JobTracker, usually every few minutes, to reassure the JobTracker that it is still alive. These message also inform the JobTracker of the number of available slots, so the JobTracker can stay up to date with where in the cluster work can be delegated.[19]

- This happens when there is a failure of hardware running the task tracker.
- JobTracker notices that the task tracker is failed, as it stopped sending heartbeat
- JobTracker schedules all the tasks running on the failed task tracker on some other task tracker.
- Even the tasks which were run on the failed task tracker needs to be run again as there output might not be accessible now.
- If more than 4 (configurable) tasks of a particular jobs fails on a particular task tracker it is considered as fault.
- If more than configured number of faults occur for a particular task tracker, it is blacklisted by JobRunner. No further tasks are assigned to a black listed task tracker.
- Faults expire over time (by default 1 day) or if a task runner is restarted and it joins the Job runner again.

Job Tracker Failure

JobTracker is the daemon service for submitting and tracking MapReduce jobs in Hadoop. There is only One Job Tracker process run on any hadoop cluster. Job Tracker runs on its own JVM process. In a typical production cluster its run on a separate machine. Each slave node is configured with job tracker node location. The JobTracker is single point of failure for the Hadoop MapReduce service. If it goes down, all running jobs are halted.[20]

- This is a serious failure.
- All the jobs needs to be submitted and run again once new job tracker is brought up
- Although there is provision to continue from previous state by configuration, it doesnt work reliably.

[19] http://www.fromdev.com/2010/12/interview-questions-hadoop-mapreduce.html#What-is-TastTracker

[20] http://www.fromdev.com/2010/12/interview-questions-hadoop-mapreduce.html