As you may know Microsoft has recently announced plans for a [Hadoop adoption for both Windows Server and Windows Azure](http://blogs.technet.com/b/microsoft_blog/archive/2011/10/12/microsoft-expands-data-platform-to-help-customers-manage-the-new-currency-of-the-cloud.aspx). You can find out more about Hadoop and Windows Azure at [Apache Hadoop-based Services for Windows Azure](https://www.hadooponazure.com/) and [Availability of Community Technology Preview (CTP) of Hadoop based Service on Windows Azure](http://blogs.technet.com/b/dataplatforminsider/archive/2011/12/14/availability-of-community-technology-preview-ctp-of-hadoop-based-service-on-windows-azure.aspx).

If you are not familiar with MapReduce then there are some useful resources for F# and MapReduce that would also be worth reading [Exploring MapReduce with F#](http://weblogs.asp.net/podwysocki/archive/2009/03/03/exploring-mapreduce-with-f.aspx) and [Parsing Log Files with F#, MapReduce and Windows Azure](http://msdn.microsoft.com/en-us/magazine/gg983490.aspx).

As Hadoop is written in Java the main integration point for MapReduce for .Net developers is through Hadoop Streaming. Hadoop streaming being a utility that allows you to create and run MapReduce jobs with any executable or script as the mapper and/or the reducer. You can find out everything you want to know about Hadoop Streaming at: <http://hadoop.apache.org/common/docs/current/streaming.html>

**MapReduce Code**

Before starting it is worth noting that the complete code for this blog post can be found at:

<http://code.msdn.microsoft.com/Hadoop-Streaming-and-F-f2e76850>

When writing a Hadoop Streaming MapReduce job it will quickly become apparent that there is a codebase that one can reuse. As such, I thought it would be useful to put together this codebase to enable F# developers to write MapReduce libraries through a simple API. The idea was to provide reusable code such that one only needed to be concerned with implementing the MapReduce code with the following function prototype’s:

Map : string –> (string \* obj) option

Reduce : string -> seq<string> –> obj option

The idea is that the Hadoop text input is processed and each input line is passed into the Map function which parses and filters the key/value pair for the data. The values are then sorted and merged, by Hadoop. The processed mapped data is then passed into the Reduce function, as a key and corresponding sequence of strings, which then defines the optional output value.

So why the use of the obj type? Hadoop Streaming is based on text data, albeit a binary interface is also available (more on this in a later post). Thus the inputs into the MapReduce are strings. However, when performing the MapReduce operations strings are not always suitable, but they do need to be able to be represented as strings. This is normally handled through serialization but in this case I have used sprintf with a “%O” pattern. Thus any type needs to have a meaningful ToString() implementation such that the data can once again be parsed back into a workable type.

As a sample consider the following data:

11075 19:07:56 en-US iPhone OS Apple iPhone 4.2.1 Georgia United States 0 0  
11081 01:46:19 en-US Android Samsung SCH-i500 Colorado United States 4.2620937 0 0  
11086 04:07:25 en-US Android Unknown Android 2.3 California United States 0 0  
11090 03:34:59 en-US iPhone OS Apple iPod Touch 4.3.x Hawaii United States 0 0  
11095 19:34:47 en-US Android Samsung SCH-i500 Illinois United States 0.4621525 1 0  
11095 02:31:19 en-US Android Samsung SCH-i500 Nebraska United States 1.2662282 0 0  
11095 02:31:21 en-US Android Samsung SCH-i500 Nebraska United States 0.2905647 0 1  
11095 19:34:49 en-US Android Samsung SCH-i500 Illinois United States 1.3336967 1 1  
11097 10:22:41 en-US iPhone OS Apple iPhone 4.0 Illinois United States 0 0  
11102 12:54:27 en-US Android Samsung SCH-i400 Florida United States 0 0  
11106 12:54:25 en-GB Windows Phone HTC 7 Trophy Greater London United Kingdom 9.94 2 0  
11106 08:50:46 en-GB Windows Phone HTC 7 Trophy Greater London United Kingdom 3.12 0 0  
11106 11:07:31 en-GB Windows Phone HTC 7 Trophy Greater London United Kingdom 15.538 1 0  
11106 11:13:27 en-GB Windows Phone HTC 7 Trophy Greater London United Kingdom 1.5066558 1 1  
11106 11:13:28 en-GB Windows Phone HTC 7 Trophy Greater London United Kingdom 1 2  
11112 00:42:52 en-US iPhone OS Apple iPhone 4.2.x Illinois United States 18.1075543 0 0  
11112 00:43:14 en-US iPhone OS Apple iPhone 4.2.x Illinois United States 2.6342826 0 1  
11130 10:48:20 en-GB iPhone OS Apple iPhone 4.2.1 Greater London United Kingdom 0 0  
11131 12:19:52 en-US Android Unknown Android 2.3 Massachusetts United States 0 0

This represents some mobile phone data with the 2nd column representing the query time and the 4th column representing the platform device. A simple MapReduce could be:

Pull the Device Platform and Query Time from the data:

**Mapper**

module MobilePhoneQueryMapper =  
  
// Performs the split into key/value  
let private splitInput (value:string) =  
try  
let splits = value.Split('\t')  
let devicePlatform = splits.[3]  
let queryTime = TimeSpan.Parse(splits.[1])  
Some(devicePlatform, box queryTime)  
with  
| :? System.ArgumentException -> None  
  
  
// Map the data from input name/value to output name/value  
let Map (value:string) =  
splitInput value

Calculate the Min, Average, and Max Query Times.

**Reducer**

module MobilePhoneQueryRangeReducer =   
  
let Reduce (key:string) (values:seq<string>) =  
let initState = (TimeSpan.MaxValue, TimeSpan.MinValue, 0L, 0L)  
let (minValue, maxValue, totalValue, totalCount) =  
values |>   
Seq.fold (fun (minValue, maxValue, totalValue, totalCount) value ->  
(min minValue (TimeSpan.Parse(value)), max maxValue (TimeSpan.Parse(value)), totalValue + (int64)(TimeSpan.Parse(value).TotalSeconds), totalCount + 1L) ) initState  
  
Some(box (minValue, TimeSpan.FromSeconds((float)(totalValue/totalCount)), maxValue))

The box function is used to ensure the types returned from the MapReduce calls is of the obj type.

So onto the executable code.

As this is Hadoop Streaming applications, both the mapper and the reducer are executables that read the input from StdIn (line by line) and emit the output to StdOut. Thus one just needs a console application that does a Console.ReadLine() to get the data, and perform a Console.WriteLine() to emit the output.

As with all F# Console applications the MainEntry point is defined as:

module Program =   
  
[<EntryPoint>]  
let Main(args) =   
  
Controller.Run args  
  
// main entry point return  
0

Each executable then contains a Controller module that is run to process the data.

**Mapper Executable**

The purpose of the Mapper code is to perform Input Format Parsing, Projection, and Filtering. The code to perform this is as follows:

module Controller =   
  
let Run (args:string array) =   
  
// Define what arguments are expected  
let defs = [  
{ArgInfo.Command="input"; Description="Input File"; Required=false };  
{ArgInfo.Command="output"; Description="Output File"; Required=false } ]  
  
// Parse Arguments into a Dictionary  
let parsedArgs = Arguments.ParseArgs args defs   
  
// Ensure Standard Input/Output and allow for debug configuration  
use reader =   
if parsedArgs.ContainsKey("input") then  
new StreamReader(Path.GetFullPath(parsedArgs.["input"]))  
else  
new StreamReader(Console.OpenStandardInput())  
  
use writer =  
if parsedArgs.ContainsKey("output") then  
new StreamWriter(Path.GetFullPath(parsedArgs.["output"]))  
else  
new StreamWriter(Console.OpenStandardOutput(), AutoFlush = true)  
  
// Combine the name/value output into a string  
let outputCollector (outputKey, outputValue) =  
let output = sprintf "%s\t%O" outputKey outputValue  
writer.WriteLine(output)  
  
// Read the next line of the input stream  
let readLine() =   
reader.ReadLine()  
  
// Define the input sequence  
let rec inputs() = seq {  
let input = readLine()  
if not(String.IsNullOrWhiteSpace(input)) then  
// Yield the input and the remainder of the sequence  
yield input  
yield! inputs()  
}  
  
// Process the lines from the stream and pass into the mapper  
inputs()  
|> Seq.map MobilePhoneQueryMapper.Map  
|> Seq.filter Option.isSome  
|> Seq.iter (fun value -> outputCollector value.Value)

A majority of this code provides a means for specifying input and output files to better aid testing (more on this in a bit).

The code boils down to the last three lines which performs the main functions:

inputs()  
|> Seq.map MobilePhoneQueryMapper.Map  
|> Seq.filter Option.isSome  
|> Seq.iter (fun value -> outputCollector value.Value)

This parses the input, maps the data into a key value pairs, filters out unwanted rows, and outputs the data using the outputCollector function.

The outputCollector function takes the processed key/value pair and outputs them to the correct stream. In the Java API the types used for output are based on a Writable interface; rather than Java Serialization. In this implementation the outputCollector takes the obj.ToString(); through the use of sprintf(). Thus if the provided types does not have a useful ToString() you will have to create the string representation before calling the outputCollector. The outputCollector performs the formatting of the key/value pair into a Tab delimited string; as required for Hadoop Streaming.

**Reducer Executable**

After running the Mapper, the data being parsed into the Reducer will be a key/value pair delimited with a Tab. Using the aforementioned sample data and Map, a sample, but selective, input dataset for a Reducer would be:

Android 18:54:20   
Android 19:19:44   
Android 19:19:46   
RIM OS 17:19:36   
RIM OS 17:17:18   
RIM OS 00:44:41   
Windows Phone 12:54:25   
Windows Phone 08:50:46   
Windows Phone 11:13:28   
iPhone OS 19:07:56   
iPhone OS 03:34:59   
proprietary development 14:29:20   
proprietary development 14:30:17

The processing of the mapped data within the Reducer is a little more complex than the Mapper. The idea is that the data is grouped by the input key and the resulting sequences are passed into the Reduce function; a function call for each key along with the corresponding sequence.

Whereas the Seq.groupBy can perform this operation, the groupBy function makes no assumption on the ordering of the original sequence. As a consequence the resulting sequence is not lazily evaluated, and is thus not suitable for large sequences; an absolute must for Hadoop MapReduce jobs. The code thus has to create a lazily evaluated sequence for each input key. This can be achieved as one knows the input data is sorted.

To achieve this the processing has to be state dependant to handle the transition from one key value to the next. The input data is processed in such a fashion that any change in key value causes a transition to the next sequence. The state persistence is needed to track the key change, and to ensure the first yield of the new sequence is the value that caused this transition; transition values are not lost.

The code for the Reducer is as follows:

module Controller =   
  
let Run (args:string array) =   
  
// Define what arguments are expected  
let defs = [  
{ArgInfo.Command="input"; Description="Input File"; Required=false };  
{ArgInfo.Command="output"; Description="Output File"; Required=false } ]  
  
// Parse Arguments into a Dictionary  
let parsedArgs = Arguments.ParseArgs args defs  
  
// Ensure Standard Input/Output and allow for debug configuration  
use reader =   
if parsedArgs.ContainsKey("input") then  
new StreamReader(Path.GetFullPath(parsedArgs.["input"]))  
else  
new StreamReader(Console.OpenStandardInput())  
  
use writer =  
if parsedArgs.ContainsKey("output") then  
new StreamWriter(Path.GetFullPath(parsedArgs.["output"]))  
else  
new StreamWriter(Console.OpenStandardOutput(), AutoFlush = true)  
  
// Combine the name/value output into a string  
let outputCollector outputKey outputValue =   
let output = sprintf "%s\t%O" outputKey outputValue  
writer.WriteLine(output)  
  
// Read the next line of the input stream  
let readLine() =   
reader.ReadLine()  
  
// Parse the input into the required name/value pair  
let parseLine (input:string) =   
let keyValue = input.Split('\t')  
(keyValue.[0].Trim(), keyValue.[1].Trim())  
  
// Converts a input line into an option  
let getInput() =   
let input = readLine()  
if not(String.IsNullOrWhiteSpace(input)) then  
Some(parseLine input)  
else  
None  
  
// Creates a sequence of the input based on the provided key  
let lastInput = ref None  
let continueDo = ref false  
let inputsByKey key (firstValue:string option) = seq {  
// Yield any value from previous read  
if firstValue.IsSome then  
yield firstValue.Value  
  
continueDo := true  
while !continueDo do  
match getInput() with  
| Some(input) when (fst input) = key ->  
// Yield found value and remainder of sequence  
yield (snd input)   
| Some(input) ->  
// Have a value but different key  
lastInput := Some(fst input, snd input)  
continueDo := false  
| None ->  
// Have no more entries  
lastInput := None  
continueDo := false  
}  
  
// Controls the calling of the reducer  
let rec processInput (input:(string\*string) option) =  
if input.IsSome then  
let key = fst input.Value  
let value = MobilePhoneQueryReducer.Reduce key (inputsByKey key (Some(snd input.Value)))  
if value.IsSome then  
outputCollector key value.Value  
if lastInput.contents.IsSome then  
processInput lastInput.contents  
  
processInput (getInput())

The code also contains the same testing structure to allow one to pass in an input and output file; once again to aid in debugging.

The code uses a string.Split() to derive the tuple of key/value pair; as this is how the Mapper writes out the data. Once the data has been Reduced, the outputCollector once again performs the formatting of the key/value data into a Tab delimited string, sending the text to the StdOut stream. The value in this case is the reduced data corresponding to the key.

The output for the sample data and Reduce sample code, listed above, would be:

Android (00:00:06, 12:54:39, 23:59:54)   
RIM OS (00:01:07, 13:52:56, 23:59:58)   
Unknown (00:00:36, 10:29:27, 23:52:36)   
Windows Phone (00:00:32, 12:38:31, 23:55:17)   
iPhone OS (00:00:01, 11:51:53, 23:59:50)   
proprietary development (14:29:20, 14:29:44, 14:30:17)

The value corresponding to the key is the string representing of a tuple of type (TimeSpan\*TimeSpan\*TimeSpan).

As you can see, there is a fair amount of code controlling the input and output streams for calling the Map and reduce functions; that can be reused for all Hadoop Streaming jobs.

**Testing the Code**

So now the code has been put together how can it be tested? Debugging a job once it has been deployed to a Hadoop cluster is not an easy task. As such one is much better off if testing can be performed without Hadoop in the picture. Whereas this will not cover all test cases, as some issues will only be found when deployed to a cluster, it does provide a means to cover most test scenarios.

Unit Testing the individual map and Reduce functions is relatively straight forward. However performing testing on sample input data is a little trickier.

To assist in testing with data files I have put together a Tester application. This application:

* Defines and executes a Process for the Mapper executable in which
  + The StdIn is modified to be the file specified in the “–input” command line argument
  + the StdOut is modified to be a file with a “mapper” extension
* When the Mapper has completed, Sorts the output file from the Mapper into a file with a “reducer” extension
* When the Sort is complete, defines and executes a Process for the Reducer executable in which
  + The StdIn is modified to be the sorted “reducer” file
  + The StdOut is modified to be the file specified in the “–output” command line argument

Running the Tester application allows one to check inputs and outputs, in a flow similar to running within Hadoop. The full listing for the Tester application can be found in the download code:

<http://code.msdn.microsoft.com/Hadoop-Streaming-and-F-f2e76850>

Whereas this allows for simple data testing there is an easier option when one needs to attach a debugger. Bother the Mapper and Reducer executables accept a “–input” and “–output” command line arguments. These options define the files to be used, rather than the Standard Input/Output streams, for data processing. Using these arguments one can easily debug the executable with a set of test data and view the output. To create a Reducer input, for testing, one can run the Tester application with the required input data, and use the output file with a “reducer” extension as the input.

One final thing that has been useful for testing is the concept of a Null controller. This is merely an executable that just takes the input and passes it to the output stream:

module ControllerNull =   
  
let Run (args:string array) =   
  
let rec inputs() =  
let input = Console.ReadLine()  
if not(String.IsNullOrWhiteSpace(input)) then  
Console.WriteLine input  
inputs()  
inputs()

This Null controller is useful when deploying your application to a Hadoop cluster to ensure data is being input and output correctly.

**Hadoop and F#**

I am not going to say much about running the Hadoop job other than to show the command to run the Streaming job:

C:\Apps\dist\bin>hadoop.cmd jar ../lib/hadoop-streaming.jar -D keep.failed.task.file=true  
-input "/mobile/debug/sampledata.txt"  
-output "/mobile/querytimesdebug"  
-mapper "..\..\jars\FSharp.Hadoop.Mapper.exe"  
-reducer "..\..\jars\FSharp.Hadoop.Reducer.exe"  
-file "C:\MyDirectory\FSharp.Hadoop.MapReduce\FSharp.Hadoop.Mapper\bin\Debug\FSharp.Hadoop.Mapper.exe"  
-file "C:\MyDirectory\FSharp.Hadoop.MapReduce\FSharp.Hadoop.Reducer\bin\Debug\FSharp.Hadoop.Reducer.exe"  
-file "C:\MyDirectory\FSharp.Hadoop.MapReduce\FSharp.Hadoop.MapReduce\bin\Debug\FSharp.Hadoop.MapReduce.dll"

The sample download code includes all the necessary command line code to run the job, and also includes a sample data file and some commands to copy this data to the cluster.

There is however a point to be made about using F#, namely the dependency on the F# Runtime. Importantly the F# Runtime contains many useful functions and types, including APIs for collections such as lists, arrays, maps, sets and sequences. One should install the F# Runtime on all the nodes in the cluster. The installer can be found at:

[Visual Studio 2010 F# 2.0 Runtime SP1](http://www.microsoft.com/download/en/details.aspx?id=15834)

If one is running the job on a server on which the FSharp Runtime has not been deployed there is another option. One can copy the appropriate runtime file to the temp execution directory. To achieve this one needs to add the following file option to the execution command:

-file C:\Program Files (x86)\Reference Assemblies\Microsoft\FSharp\2.0\Runtime\v4.0\FSharp.Core.dll

**Conclusion**

In terms of reusability of the code one could create a Project Template for the base code. Another options would be to Activator.CreateInstance() to create a Type specified in a configuration file. In this case the Mapper and Reducer would have to inherit from a common Interface.

To conclude, don’t forget to check out Channel9 for more information:

<http://channel9.msdn.com/Events/windowsazure/learn/Learn-about-Hadoop-on-Windows-Azure-with-Alex-Stojanovic>

I will endeavor to cover other topics, such as Binary Streaming, in future posts.