Distributed data processing with Hadoop, Part 3: Application development

Developing a Ruby MapReduce application for Hadoop

Skill Level: Intermediate

M. Tim Jones (mtj@mtjones.com)
Independent Author

14 Jul 2010

With configuration, installation, and the use of Hadoop in single- and multi-node architectures under your belt, you can now turn to the task of developing applications within the Hadoop infrastructure. This final article in the series explores the Hadoop APIs and data flow and demonstrates their use with a simple mapper and reducer application.

The first two articles of this series focused on the installation and configuration of Hadoop for single- and multinode clusters. This final article explores programming in Hadoop—in particular, the development of a map and a reduce application within the Ruby language. I chose Ruby, because first, it’s an awesome object-oriented scripting language that you should know, and second, you'll find numerous references in the Resources section for tutorials addressing both the Java™ and Python languages. Through this exploration of MapReduce programming, I also introduce you to the streaming application programming interface (API). This API provides the means to develop applications in languages other than the Java language.

Let's begin with a short introduction to map and reduce (from the functional perspective), and then take a deeper dive into the Hadoop programming model and
its architecture and elements that carve, distribute, and manage the work.

The origin of map and reduce

So, what are the functional elements that inspired the MapReduce programming paradigm? In 1958, John McCarthy invented a language called Lisp, which implemented both numerical and symbolic computation but in a recursive form that is foreign to most languages in use today. (There’s actually a fascinating history of Lisp on Wikipedia that includes a useful tutorial—well worth the time to read.) Lisp was first realized on the IBM® 704, the first mass-produced computer, which also supported another old favorite: FORTRAN.

The map function, originating in functional languages like Lisp but now common in many other languages, is an application of a function over a list of elements. What does this mean? Listing 1 provides an interpreted session with Scheme Shell (SCSH), which is a Lisp derivative. The first line defines a function called square that takes an argument and emits its square root. The next line illustrates the use of the map function. As shown, with map, you provide your function and a list of elements to which the function is applied. The result is a new list containing the squared elements.

Listing 1. Demonstration of the map function in SCSH

```scheme
> (define square (lambda (x) (* x x)))
> (map square '(1 3 5 7))
'(1 9 25 49)
```

Reduction also applies over a list but typically reduces the list to a scalar value. The example provided in Listing 2 illustrates another SCSH function for reducing a list to a scalar—in this case, summing the list of values in the form (1 + (2 + (3 + (4 + (5))))). Note that this is classical functional programming, relying on recursion over iteration.

Listing 2. Demonstration of reduction in SCSH

```scheme
> (define (list-sum lis) (if (null? lis) 0 (+ (car lis) (list-sum (cdr lis)))))
> (list-sum '(1 2 3 4 5))
15
```

It's interesting to note that recursion is as efficient as iteration in imperative languages because the recursion is translated into iteration under the covers.
Hadoop's programming model

Google introduced the idea of MapReduce as a programming model for processing or generating large sets of data. In the canonical model, a \textit{map} function processes key-value pairs, resulting in an intermediate set of key-value pairs. A \textit{reduce} function then processes those intermediate key-value pairs, merging the values for the associated keys (see Figure 1). Input data is partitioned in such a way that it can be distributed among a cluster of machines for processing in parallel. In the same way, the generated intermediate data is processed in parallel, making the approach ideal for processing very large amounts of data.

\textbf{Figure 1. Simplified view of MapReduce processing}

For a quick refresher, look at the architecture from Figure 1 from the perspective of map and reduce for word count (because you'll develop a map and reduce application in this article). When input data is provided (into the Hadoop file system [HDFS]), it is first partitioned, and then distributed to map workers (via the job tracker). Although the example in Figure 2 shows a short sentence being partitioned, typically the amount of work to partition is in the 128MB-size range for one reason: It takes a small amount of time to set work up, so having more work to do minimizes this overhead. The map workers (in the canonical example) split the work into
individual vectors that contain the tokenized word and an initial value (1, in this case). When the map tasks are complete (as defined in Hadoop by the task tracker), the work is provided to the reduce worker. The reduce worker reduces the keys to a unique set, with the value representing the number of keys found.

**Figure 2. Simple MapReduce example**

Note that this process can occur on the same machine or different machines or be
done sequentially or in parallel using different partitions of data, and still the result is the same.

Although the canonical view (for search index generation using word count) is one way to view Hadoop, it turns out that this model of computing can be generically applied to a number of computational problems, as you'll see.

The flexibility of Hadoop

From the simple example shown in Figure 2, notice that the two primary elements are the map and reduce processes. Although there is a traditional view for how these processes work, it's not a requirement of the architecture for map and reduce to behave this way. This is the real power of Hadoop—its flexibility to implement map and reduce processes that behave in a way that solves a particular application. The word count example is useful and applicable to a large number of problems, but other models still fit within this general framework. All that's required is the development of a map and reduce application making the processes visible to Hadoop.

Among other applications, Hadoop has even been used for machine learning applications implementing algorithms as diverse as neural networks, support vector machines, and k-means clustering (see the Resources section for more information).

Data streaming

Although Hadoop is a Java-based framework, it's possible to write map and reduce applications in languages other than the Java language. Streaming makes this possible. The streaming utility within Hadoop implements a type of data flow glue. With the streaming utility, you can define your own map and reduce executables (with each taking input from standard input [stdin] and providing output through standard output [stdout]), and the streaming utility reads and writes data appropriately, invoking your applications as needed (see Listing 3).

**Listing 3. Using the Hadoop streaming utility**

```
hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
   -input inputData \
   -output outputData \
   -mapper map_exec \
   -reducer reduce_exec
```

Listing 3 illustrates how to use the streaming utility within Hadoop, while Figure 3 shows graphically how the flow is defined. Note that this is a simple example of streaming use. Numerous options are available to tailor how data is parsed, to tailor how images are invoked, to specify replacement images for the partitioner or
combiner, and other configuration tweaks (see the Resources section for more information).

**Figure 3. Graphical streaming example**

```ruby
#!/usr/bin/env ruby

# Our input comes from STDIN
STDIN.each_line do |line|
```

Ruby example

With a basic understanding of the streaming utility under your belt, you're ready to write a simple Ruby map and reduce application and see how to use the processes within the Hadoop framework. The example here goes with the canonical MapReduce application, but you'll see other applications later (along with how you would implement them in map and reduce form).

Begin with the mapper. This script takes textual input from stdin, tokenizes it, and then emits a set of key-value pairs to stdout. Like most object-oriented scripting languages, this task is almost too simple. The mapper script is shown in Listing 4 (with some comments and white space to give it a bit more size). This program uses an iterator to read a line from stdin and another iterator to split the line into individual tokens. Each token (word) is then emitted to stdout with an associated value of 1 (separated by a tab).

**Listing 4. Ruby map script (map.rb)**

```ruby
#!/usr/bin/env ruby

# Our input comes from STDIN
STDIN.each_line do |line|
```
# Iterate over the line, splitting the words from the line and emitting
# as the word with a count of 1.
ligne.split.each do |word|
  puts "#{word}\t1"
end

Next, look at the reduce application. This one is slightly more complicated but uses a
Ruby hash (associative array) to simplify the reduction operation (see Listing 5). This
script again works through the input data from stdin (passed by the streaming
utility) and splits the line into a word and value. The hash is then checked for the
word; if found, the count is added to the element. Otherwise, you create a new entry
in the hash for the word, and then load the count (which should be 1 from the
mapper process). When all input has been processed, you simply iterate through the
hash and emit the key-value pairs to stdout.

Listing 5. Ruby reduce script (reduce.rb)

#!/usr/bin/env ruby
# Create an empty word hash
wordhash = {}
# Our input comes from STDIN, operating on each line
STDIN.each_line do |line|
  # Each line will represent a word and count
  word, count = line.strip.split
  # If we have the word in the hash, add the count to it, otherwise
  # create a new one.
  if wordhash.has_key?(word)
    wordhash[word] += count.to_i
  else
    wordhash[word] = count.to_i
  end
end
# Iterate through and emit the word counters
wordhash.each { |record, count| puts "#{record}\t#{count}"}

With the map and reduce scripts done, test them from the command line. Remember
to change these files to executable using chmod +x. Start by generating an input
file, as shown in Listing 6.

Listing 6. Generating a file of input

# echo "Hadoop is an implementation of the map reduce framework for " \
"distributed processing of large data sets." > input
#

With this input, you can now test your mapper script, as shown in Listing 7. Recall
that this script simply tokenizes the input to key-value pairs, where each value will be
1 (non-unique input).

**Listing 7. Testing the mapper script**

```ruby
# cat input | ruby map.rb
Hadoop 1
is 1
an 1
implementation 1
of 1
the 1
map 1
reduce 1
framework 1
for 1
distributed 1
processing 1
of 1
large 1
data 1
sets. 1
#
```

So far, so good. Now, pull the entire application together in the original streaming
form (Linux® pipes). In Listing 8, you pass your input through your map script, sort
that output (optional step), and then pass the resulting intermediate data through the
reducer script.

**Listing 8. Simple MapReduce using Linux pipes**

```bash
# cat input | ruby map.rb | sort | ruby reduce.rb
large 1
of 2
framework 1
distributed 1
data 1
an 1
the 1
reduce 1
map 1
sets. 1
Hadoop 1
implementation 1
for 1
processing 1
is 1
#
```

**Ruby with Hadoop**

With your map and reduce scripts working as expected in the shell environment, put
them to the test with Hadoop. I'm going to skip the Hadoop setup tasks (refer to Part
1 or Part 2 of this series to get Hadoop up and running).
The first step is to create an input directory within HDFS for your input data, and then provide a sample file on which you'll test your scripts. Listing 9 illustrates this step (see Part 1 or Part 2 for more information on these steps).

**Listing 9. Creating input data for the MapReduce process**

```bash
# hadoop fs -mkdir input
# hadoop dfs -put /usr/src/linux-source-2.6.27/Documentation/memory-barriers.txt input
# hadoop fs -ls input
Found 1 items
-rw-r--r--  1 root  supergroup  78031 2010-06-04 17:36 /user/root/input/memory-barriers.txt
```

Next, using the `streaming` utility, invoke Hadoop with the custom scripts, specifying the input data and location for output (see Listing 10). Note in this example that the `-file` options simply tell Hadoop to package your Ruby scripts as part of the job submission.

**Listing 10. Using Hadoop streaming with custom Ruby MapReduce scripts**

```bash
# hadoop jar /usr/lib/hadoop-0.20/contrib/streaming/hadoop-0.20.2+228-streaming.jar
#   -file /home/mtj/ruby/map.rb -mapper /home/mtj/ruby/map.rb
#   -file /home/mtj/ruby/reduce.rb -reducer /home/mtj/ruby/reduce.rb
#   -input input/* -output output
packageJobJar: [/home/mtj/ruby/map.rb, /home/mtj/ruby/reduce.rb, /var/lib/hadoop-0.20/...]
10/06/04 17:42:38 INFO mapred.FileInputFormat: Total input paths to process : 1
10/06/04 17:42:39 INFO streaming.StreamJob: getLocalDirs(): [/var/lib/hadoop-0.20/...]
10/06/04 17:42:39 INFO streaming.StreamJob: To kill this job, run:
10/06/04 17:42:39 INFO streaming.StreamJob: /usr/lib/hadoop-0.20/bin/hadoop job ...
10/06/04 17:43:17 INFO streaming.StreamJob: map 100% reduce 0%
10/06/04 17:43:26 INFO streaming.StreamJob: map 100% reduce 100%
10/06/04 17:43:29 INFO streaming.StreamJob: Job complete: job_201006041053_0001
```

Finally, explore the output using the `cat` file system operation through the `hadoop` utility (see Listing 11).

**Listing 11. Exploring the Hadoop output**

```bash
# hadoop fs -ls /user/root/output
Found 2 items
drwxr-xr-x  - root  supergroup  0 2010-06-04 17:42 /user/root/output/_logs
-rw-r--r--  1 root  supergroup  23014 2010-06-04 17:43 /user/root/output/part-00000
# hadoop fs -cat /user/root/output/part-00000 | head -12
+---> | 4
   immediate 2
   Alpha) 1
   enable 1
   _mandatory_ 1
   Systems 1
   DMA. 2
```
So, in less than 30 lines of script, you’ve implemented the map and reduce elements and demonstrated their execution within the Hadoop framework. A simple example, but one that illustrates the real power behind Hadoop and why it’s becoming such a popular framework for processing large data sets with custom or proprietary algorithms.

Other applications for Hadoop

Hadoop can be used in many applications beyond simply computing word counts of large data sets. All that’s needed is a representation of the data in a vector form that the Hadoop infrastructure can use. Although canonical examples use the vector representation as a key and value, there’s no restriction on how you can define the value (such as an aggregate of a number of values). This flexibility can open new opportunities for Hadoop in a richer set of applications.

One interesting application that fits squarely in the MapReduce word count model is tabulating the frequency of Web server access (discussed in the seminal Google paper). For this application, the URLs serve as the keys (as ingested from the Web server access logs). The result of the reduce process is the total number of accesses per URL for a given Web site based on the Web server logs.

In machine learning applications, Hadoop has been used as a way to scale genetic algorithms for processing large populations of GA individuals (potential solutions). The map process performs the traditional genetic algorithm, seeking the best individual solution from the local pool. The reduce application then becomes a tournament of individual solutions from the map phase. This permits individual nodes to identify their best solution, and then to allow these solutions to compete in the reduce phase in a distributed display of survival of the fittest.

Another interesting application was created to identify botnets for email spam. The first step in this process was to classify email messages for the purpose of reducing them (based on a set of fingerprints) as coming from a given organization. From this filtered data, a graph was built for email that was connected in some way (for example, referring to the same link in the email message body). These related emails were then reduced to hosts (static or dynamic IP address) to identify the botnet in question.

Outside of applications that view the world through map and reduce primitives, Hadoop is useful as a means of distributing work among a cluster of machines. Map and reduce don't necessarily force a particular type of application. Instead, Hadoop
can be viewed as a way to distribute both data and algorithms to hosts for faster parallel processing.

**Hadoop application ecosystem**

Although Hadoop provides a flexible framework, other applications are available that can transform its interface for other applications. One interesting example is called *Hive*, which is a data warehouse infrastructure with its own query language (called *Hive QL*). Hive makes Hadoop more familiar to those with a Structured Query Language (SQL) background, but it also supports the traditional MapReduce infrastructure for data processing.

*HBase* is another interesting application that resides on top of the HDFS. It's a high-performance database system similar to Google BigTable. Instead of traditional file processing, HBase makes database tables the input and output form for MapReduce processing.

Finally, *Pig* is a platform on Hadoop for analyzing large data sets. Pig provides a high-level language that compiles to map and reduce applications.

**Going further**

This final article in the Hadoop series explored the development of a map and reduce application in Ruby for the Hadoop framework. Hopefully, from this article, you can see the real power of Hadoop. Although Hadoop restricts you to a particular programming model, that model is flexible and can be applied to a large number of applications.
Resources

Learn

- **MapReduce: Simplified Data Processing on Large Clusters** is the seminal paper on MapReduce, written in 2004 by Jeff Dean and Sanjay Ghemawat. This paper remains an enjoyable read.

- This article explored Hadoop’s streaming utility, which permits the development of map and reduce scripts in languages other than the Java language. Apache provides a great set of resources for streaming, including the Hadoop Streaming documentation and the streaming wiki (which provides a good introduction to the various command-line options).

- Wikipedia provides great introductions to the Lisp and Scheme languages as well as a general introduction to functional programming concepts (and MapReduce).

- To demonstrate the functional programming elements of map and reduce, this article used the Scheme shell. If you've ever wanted to experiment with Scheme, SCSH is a great sandbox to experiment with this powerful language. You can also learn about Scheme and scripting with C in Tim's article Scripting with Guile (developerWorks, January 2009), or read a great Scheme introduction.

- In the paper Map-Reduce for Machine Learning on Multicore, the MapReduce model is used to implement a variety of machine learning algorithms for multicore processors. It's an interesting read to explore how the MapReduce model can apply to a variety of computational algorithms.

- **Hive** is a data warehouse infrastructure built on top of Hadoop. It provides a query language over Hadoop data while supporting the traditional Hadoop programming model. **HBase** is a database representation over Hadoop's HDFS, permitting MapReduce to operate on database tables over simple files. Finally, **Pig** is a platform for large data set analysis that includes a high-level language for Hadoop programming.

- **Ruby** is the latest of the object-oriented scripting languages. It is dynamic with a focus on programmer productivity.

- Check out this list of Mapreduce and Hadoop algorithms in academic papers. This site provides an interesting perspective on how Hadoop is used for a variety of applications (from science, machine learning, web services, and more).

- Yahoo! provides a great set of resources for Hadoop at the developer network. In particular, the Yahoo! Hadoop Tutorial introduces Hadoop and provides a detailed discussion of its use and configuration.
In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.

Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.

Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools, as well as IT industry trends.

Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.

Follow developerWorks on Twitter, or subscribe to a feed of Linux tweets on developerWorks.

Get products and technologies

Cloudera offers a number of free (and stable!) Hadoop distributions. You can download a distribution for a variety of Linux distributions, a virtual machine (VM), and even a VM for Amazon Elastic Compute Cloud (Amazon EC2).

Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming (now in its second edition), AI Application Programming (in its second edition), and BSD Sockets Programming from a Multilanguage Perspective. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Senior Architect for Emulex Corp. in Longmont, Colorado.