Hog Language Tutorial

April 16, 2012

Jason Halpern	Testing/Validation
Samuel Messing	Project Manager
Benjamin Rapaport	System Architect
Kurry Tran	System Integrator
Paul Tylkin	Language Guru

Introduction

Hog gives users with some programming experience a gentle introduction to MapReduce, a popular programming model for distributed computation. In a Hog program, a user specifies an `@Map` function, which operates on key-value pairs (read from a text file), and outputs intermediate key-value pairs. The user also specifies an `@Reduce` function, which groups the intermediate key-value pairs by key, and outputs a final set of key-value pairs. This model of computation has been widely adopted for distributing large computations that might be considered "embarassingly parallelizable."

Thinking in MapReduce

For a programmer new to the idea of distributed computation, MapReduce may appear to offer almost limitless possibilities. Part of the challenge of learning to use a MapReduce framework is to clearly understand its limitations. Intuitively, MapReduce can solve a problem only if there is an explicit parallelizable component. For example, finding the maximum value in a large unsorted list of numbers, computing a Fast Fourier Transform, and counting the occurrence of words in a large amount of text are problems with an easy-to-see parallelizable component. Here are the basic steps that can guide the programmer in identifying problems that are candidates for solution via MapReduce:

1. Construct a non-parallel solution to the problem.

2. Is there a component in the solution that can be broken up into smaller chunks that are not mutually dependent (the "map" phase)? If not, then MapReduce likely will not help.

3. Can the results of the computations of the independent smaller chunks be reassembled to produce a solution to a larger subproblem (the "reduce" phase)? If not, then MapReduce will not be able to build the solution to larger problems from smaller ones, and will not be useful.

4. Having identified the parallelizable component, check for hidden dependencies between the subproblems by analyzing whether reassembling the smaller components in different ways may produce different answers. If this happens, the reduce phase can not properly be completed.

Once the map (breaking up the problem into chunks) and the reduce (recombining the solutions of the subproblems) phases have been clearly understood, one is ready to write a Hog program.

Program Structure

Every Hog program has four sections, defined in the following order:

`@Functions`: An optional section which defines functions used throughout the program.

`@Map`: This section defines the map function that takes the input key-value pairs and outputs intermediate key-value pairs.

`@Reduce`: This section defines the reduce function that takes a single key from the set of intermediate key-value pairs output by the map function, and all of the values associated with that key, and reduces them to a final output.

`@Main`: The entry point for the program which initiates the MapReduce routine and can perform other local (non-distributed) computations.

Word Count

Let's assume we have thousands of large text files, and we would like to get a cross-file word count for each word that appears in any of the files. We also have a cluster of computers to help us complete this task. The following short Hog program will produce a single output file with each word and its associated count on a separate line.

Word Count Code

```
@Map (int lineNum, text line)  ->  (text, int) {
      foreach word in line.tokenize(" ") {
            emit(word, 1)
      }
}

@Reduce (text word, iter<int> values) -> (text, int) {
      int count = 0
      while (values.hasNext()) {
            count = count + values.next()
      }
      emit(count, word)
}

@Main {
      mapReduce()
}
```

Running the Word Count Code

To run the program `WordCount.hog`, the user types the following into the terminal:

```
hog --hdfs WordCount.hog --input inputFile.txt --output wordCounts.csv
```

Here, `--hdfs` indicates that we want to run the job on the user's Hadoop cluster (specified in a configuration file; see the Hog reference manual for details on how to properly orient Hog to your Hadoop cluster). If, alternatively, we wanted to run the job locally, we can type `--local`. The second parameter is the name of the program, here `WordCount.hog`. The last two parameters indicate the input file (or directory, in which case every file in the directory will be used as input in turn) and the desired name of the output file.

Word Count Explanation

The general idea of this program is that we want to read every line of text from every file, and then, grouping by word, output the total number of times we encountered each word. Since we want to group by word, we will use the words themselves as the intermediate key output by the `@Map` function. This will allow us to group each word's value and send them all together in one key-value pair to the `@Reduce` function.

`@Functions`

The first thing we notice is that this program does not contain an `@Functions` section. This section is optional, and only needs to be included if the user wants to write his or her own subroutines to be used elsewhere in the program.

`@Map`

This section's job is to read in a line of text from a file, and simply output each word as the key with a value that indicates we have just encountered it. We will use this value later to perform the summation.

The first line of this section is the `@Map` header, which defines the ***signature*** of the `@Map` function. In the current release, all Hog programs read input files one line at a time, where the file offset of the line is the key, and the text of the line is the value. *This means that for all Hog programs, the only allowable types for the input key-value pair is* `(int, text)`. The inputs are also ***named*** in the signature in order to reference them in the body of the function.

The input signature is followed by an arrow, followed by the type signature of the outputted intermediate key-value pairs. In this case, we will output each word as `text` and its count as an `int`. These values are ***unnamed***, as they cannot be referenced in the `@Map` section.

The `int` type represents an ***integer number*** such as 0, 1, -2, 3, 5, etc. In addition, Hog has the type `real` which represents ***real numbers*** such as 0.1, 2.141, etc. The `text` type is Hog's string type, and represents a sequence of characters. To create a `text` object, simply include a string of characters between two double quotes (e.g. `"hello world 123"`).

In the body of the function, we split the line of text passed in as the value into words delineated by whitespace by using the built-in function tokenize(). We then iterate through the list of words (of type list<text>) that tokenize() returns using a foreach loop. Notice that you call tokenize() "on" a text object. text objects are the only type of objects that support this function. Attempting to call the function on an object of a different type (e.g. count.tokenize() for the variable count in this example) would lead to an error, called an ***exception***. Exception handling is outside of the scope of this tutorial. Please see the language reference manual for guidance on how to anticipate and handle exceptions.

In the body of the foreach loop, we use the built-in function emit() to output a key-value pair, which the framework then groups by key when passing to the @Reduce section. In this case, since we want to group by the word itself, we emit the word and the value 1, which we will later use to calculate the totals in the @Reduce section.

@Reduce

In this section, for each word (the key) emitted by the @Map section, we will simply add up all the counts (the values) emitted for each particular word to get the final count. It should now be clear why we emitted the value 1 for each word in the @Map section, as we do so once for every instance of seeing a particular word.

Since the inputs to this section are grouped by key, @Reduce will receive a word and an ***iterator*** (referred to as an iter in Hog) over all of that word's values (the 1's we emitted in the @Map section). For *every* word, this function will receive an iterator over all of the values emitted by the @Map function for *that* word. This is why the header for this section has the word as the key and an iterator over a list of ints as the value. The key type of the input to the reduce function *must match* the key type of the output of the map function. Similarly, the values type of the reduce function is *always* an iterator over the type of the value output by the map function.

Since we want to output a count and its associated word, @Reduce will output int and text for each word.

In the body, we initialize the ***variable*** count to 0, and then iterate through the list of values using a familiar while loop, adding each value of 1 to a running total (recall that count has type int, which means it can represent an integer value). To do this, we use the built-in functions on iterators hasNext()—which returns true if the iterator contains more values and false otherwise—and next()—which returns the next value in the list and moves the iterator position forward. The statements inside the while loop continue to execute until we have seen every variable in the iter object (when values.hasNext() evalutes to false).

After we have a full count for the input word, we emit the total count and the word as our final output. The framework then takes care of writing these emitted key-value pairs to an output file, the name of which is specified by the user when the program is run (see the section above, **Running the Word Count Code**). In this example, we are emitting the word count as the key so that the final result is sorted by the count. If the word was used as the key, the final output would be in alphabetical order. It is also worth noting that even if two words have the same count (and thus the same key), this does not raise any problems, since duplicate keys are only combined in between the `@Map` and `@Reduce` phases. Any inidividual key-value pair emitted in the `@Reduce` phase will be in the final output.

@Main

In this section, we simply call the built-in function `mapReduce()`, which initiates the MapReduce program as specified by the previous sections and the command line arguments.

Sample Output

Here is the first fifty lines of output generated by `WordCount.hog` when run on a single file containing the English text of *War and Peace*:

```
31784, the
21049, and
16389, to
14895, of
10056, a
8314, in
7847, he
7645, his
7425, that
7255, was
5540, with
5316, had
4492, not
4209, at
4162, her
4009, I
3757, it
3744, as
3495, on
3488, him
3308, for
3134, is
2888, but
2762, The
2718, you
2636, said
2620, she
```

```
2526, from
2390, all
2387, were
2354, be
2333, by
2031, who
2006, which
1910, have
1812, He
1777, one
1727, they
1693, this
1645, what
1566, or
1561, an
1554, Prince
1550, so
1541, Pierre
1466, been
1439, did
1424, up
1409, their
1342, would
```

MergeSort

In this example, we will sort numbers in text files using a distributed version of the divide-and-conquer algorithm MergeSort. We will assume that our text files contain lines of integers, delimited by commas. The idea is for each call to map to sort a small list of numbers on a single line of text, and for reduce to merge all of the sorted lists it receives.

MergeSort Code

```
@Functions: {

  #{ merge: Takes two sorted lists and merges them into
          one combined and properly sorted list. }#
  list<int> merge(list<int> sortedList1, list<int> sortedList2) {

        list<int> mergedList()

        # pointers to next value of each sorted list
        int ind1 = 0
```

```
        int ind2 = 0

        # merge all values while neither list is empty
        while(ind1 < sortedList1.size() and ind2 < sortedList2.size()) {

              # insert the smaller of the 2 values and update indices
              if (sortedList1.get(ind1) < sortedList2.get(ind2)) {
                    mergedList.add(sortedList1.get(ind1++))
              }
              else {
                    mergedList.add(sortedList2.get(ind2++))
              }
        }

        # insert any remaining elements from sortedList1
        while (ind1 < sortedList1.size()) {
              mergedList.add(sortedList1.get(ind1++))
        }

        # insert any remaining elements from sortedList2
        while (ind2 < sortedList2.size()) {
              mergedList.add(sortedList2.get(ind2++))
        }

        return mergedList
  }
}

@Map (int lineNum, text line) -> (text, list<int>) {

      text reduceKey = "reduceKey"
      list<int> sortedInts()

      # put every number from line into list
      foreach number in line.tokenize(",") {
            sortedInts.add((int) number)
      }

      # sort list
      sortedInts.sort()

      # for every line of numbers, emit the sorted ints with an identical key
      emit(reduceKey, sortedInts)

}
```

```
# reduce will get a list of sorted lists, and merge them 2 at a time
@Reduce (text key, iter<list<int>> allSortedLists) -> (text, list<int>) {

      # only one output key
      text reduceKey = ""

      # begin with the first list as the fully sorted list
      list<int> allSortedNums = allSortedLists.next()

      #{ merge the allSortedNums with the next
         sorted list until all lists have been merged }#
      while(allSortedLists.hasNext()) {
            allSortedNums = merge(allSortedNums, allSortedLists.next())
      }

      emit(reduceKey, allSortedNums)

}

@Main {
      print("Beginning sort.\n")
      mapReduce()
      print("Sort complete.\n")
}
```

`@Functions`

In this section, we define a function called `merge`, which takes two sorted `lists` of `ints`, and returns a merged `list` of the two in sorted order. The way to define a function should be familiar to programmers comfortable with C or Java. In the first line of the body of the function, we are creating a new, empty list. Following that, we demonstrate a few flow of control statements such as `while` loops, and `if else` statements, the `and` boolean operator, and comparators all of which should also be familiar.

The function works as follows: it starts with cursors at the beginning of both input lists. It continues to add elements from the first list to the combined list until there is a smaller element in the second list, at which point it starts adding elements from that list instead. It continues switching back and forth between the two lists in this manner to ensure that it adds all the elements from both lists in the proper order. It finally checks that no elements in either list have been left out, and then returns the new, combined list.

Also included in this section are some built-in list functions, such as `.size()`, to get the number of elements in a list, `.add()`, to add an element onto the end of the list, and `.get()` to get an element at a specific index in the list.

`@Map`

The map function reads in a line of comma-separated integers as `text`, and outputs a list of the integers in sorted order. To do this, we introduce ***casting***, which enables the user to transform a value of one type into another type. In order to cast, the programmer must put the type he or she wants to cast to in parentheses before the value or variable name. In this case, we are casting a `text` to an `int`, which is a very common operation in Hog, since all input is read in as `text`. If the string contained in the `text` object is not a valid number (e.g. `"1ab2"`), the cast will result in an exception. A more robust version of the MergeSort program would include some exception handling to handle these cases. It's worth noting that the current code is somewhat brittle, in that if such a string of characters is encountered, the program will fail.

To sort the list of `int`s that have been read in from the input, we call a built-in function on lists called `.sort()`. This function sorts the elements of a `list` in ascending order according to their lexicographic ordering. The `sort` method is well-defined in part because `list` objects can only contain elements of the same type, and only supports primitive types. See the language reference manual for more details about `lists` and their built-in functions.

Finally, we emit the sorted list as a value, with identical keys for each list, so that they are all sent to the same reducer. This is a common trick employed in Hog when we only care about one value instead of two (i.e. a key-value pair).

`@Reduce`

The reduce function receives an iterator over *all* of the sorted lists from the map function, and merges them together one by one using the `merge()` function we defined earlier.

`@Main`

In this section, we demonstrate that arbitrary code can be executed locally (i.e. not on the cluster) in the `@Main` block. While the `@Main` must always call the `mapReduce()` function to begin the map reduce program, it can also perform locally any code that could be written in a function. In this example, we use the built-in function `print()` to print messages to standard out and let the user know that the MapReduce job has completed.

Sample Output

Because we used the same, empty `text` (`text reduceKey = ""`) for the keys for all key-value pairs sent to the `@Reduce` section, we end up with a sorted list of numbers all on one line. An example is given below,

```
,-123,-1,0,0,0,1,1,2,3,4,9,15,1234,6234,53854,123495,82737473745
```

Notice that the string begins with a space and then a comma, this is because of the empty `text` key. If we had instead put some non-empty `text` for the key, it would've shown up here.