Hog Language Tutorial

March 21, 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."

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.

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(word, count)
  }

  @Main {
        mapReduce()
  }
```

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 valued `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 `int`s 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 word and its associated word count, `@Reduce` will output `text` and `int` 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 `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 word and its count as our 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.

Merge Sort

In this example, we will sort numbers in text files using a version of merge sort. 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.

Merge Sort Code

```
@Functions: {

# merge: Takes two sorted lists and merges them to return a new larger 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() && ind2 < sortedList2.size() ) {

            # insert the smaller of the 2 values and update index pointers
            if(sortedList1.get(ind1) < sortedList2.get(ind2)) {
                  mergedList.add(sortedList1.get(ind1))
                  ind1 = ind1 + 1
            }
            else {
                  mergedList.add(sortedList2.get(ind2))
                  ind2 = ind2 + 2
            }
      }

      # insert any remaining elements from sortedList1
      while (ind1 < sortedList1.size()) {
            mergedList.add(sortedList1.get(ind1)
    ind1 = ind1 + 1
}
```

```
# insert any remaining elements from sortedList2
        while (ind2 < sortedList2.size()) {
              mergedList.add(sortedList2.get(ind2))
      ind2 = ind2 + 1
}

    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.getNext()

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

        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.

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 much always be explicit. In order to cast, the programmer must put the type he or she wants to cast to in parenthesis before the value or variable name. In this case, we are casting text to an int, which is a very common operation in Hog, since all input is read in as text.

To sort the list of ints that have been read in from the input, we call a built-in function on lists that contain primitives called .sort(). This function sorts the list in ascending order.

Finally, we emit the sorted list as a value, with identical keys for each list, so that they are all sent to a single reducer.

@Reduce

The reduce function receives an iterator to 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 performed locally 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 to standard output and let the user know that the mapReduce job has completed.