MapReduce

Philip L. Bewig — pbewig@gmail.com

October 9, 2006

MapReduce is a programming idiom that provides a convenient expression for programs that combine like items into equivalence classes. The idiom was developed by Google as a way to exploit large clusters of computers operating in parallel on large bodies of data, but is also useful as a way of structuring certain types of programs. This note presents the MapReduce idiom, provides some suggestions for its tasteful use, and describes an implementation in Scheme.

1. Google's MapReduce

Jeffrey Dean and Sanjay Ghemawat, in their paper *MapReduce: Simplified Data Processing on Large Clusters* presented at the Sixth Symposium on Operating System Design and Implementation in San Francisco in December 2004 (available at `labs.google.-com/papers/mapreduce.html`), describe Google's MapReduce idiom as follows:

> MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.
>
> Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.
>
> Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

In their paper, Dean and Ghemawat focus on the automatic parallelization of their implementation, and rightly so, since that is the true novelty of their system. But MapReduce is also useful as an organizing idiom to structure programs when processing smaller data sets on uniprocessor systems.

2. MapReduce in Scheme

Our implementation of MapReduce consists of a single function

```
(map-reduce mapper reducer lt? items)
```

which has type

```
(α → (values β γ)) ×              mapper
((β × γ × γ) → γ) ×               reducer
((β × β) → boolean) ×             lt?
list α →                          items
    list (cons β γ)               result
```

`(mapper item)` is a function that extracts the key and value from an input item; it takes an item and returns `(values key value)`. `(reducer key value1 value2)` is a function that combines two intermediate key/value pairs with identical keys into a single value; it takes a key, an existing value and a new value and returns a new value. `(lt? key1 key2)` is a function that takes two keys and returns `#t` if the first key is less than the second and `#f` otherwise. `items` is a list of input items. The return value of `map-reduce` is a list of pairs whose `car` is a key and whose `cdr` is a value, ordered by the `lt?` predicate.

Here is a simple example to count the frequency of characters in a string:

```
(map-reduce
  (lambda (x) (values x 1))
  (lambda (k v1 v2) (+ v1 v2))
  char<?
  (string->list "banana"))
```

This expression evaluates to

```
((#\a . 3) (#\b . 1) (#\n . 2))
```

This example clarifies the types of the various `map-reduce` elements. Here, the mapping function is `(lambda (x) (values x 1))`, which returns the next character from the input string as the key and 1 as the value, which is both the recursive base and the increment for summing the frequency counts. The reducing function is `(lambda (k v1 v2) (+ v1 v2))`, which accumulates the count associated with a particular character. The ordering function is `char<?`, which compares the two characters alphabetically. The input is a string converted to a list of characters. And the output is a list of pairs, with characters in the `car`s in alphabetical order and summarized counts in the `cdr`s.

3. Exploiting MapReduce

We'll next show several examples that exploit the `map-reduce` function. All of these examples are based on reading words from a file and use the `(get-words . port)` function that reads a port (either a port passed as an argument, or the current input port if no argument is present) and returns a list of pairs containing the words and the line numbers on which they appear, in the reverse order they exist on the port. The implementation of `get-words` appears in Appendix 1. We'll use the source text of the map-reduce function, which appears in Appendix 2, for our sample input text.

The `get-words` function relies on a predicate `char-in-word?` that determines if the current character is part of a word. For english text we could use `char-alphabetic?`, but since our sample is scheme source code, we'll define `char-in-word?` according to the definition of identifiers in R5RS section 2.1 as follows:

```
(define (char-in-word? c)
  (let ((word-chars
         (string->list
           "!$%&*+-./:<=>?@^_~")))
    (or (char-alphabetic? c)
        (char-numeric? c)
        (member c word-chars))))
```

Given these functions, the expression

```
(with-input-from-file "mapreduce.ss"
  (lambda () (get-words)))
```

evaluates to

```
(("t" . 53)
 ("enlist" . 53)
 … 381 pairs elided …
 ("map-reduce" . 1)
 ("define" . 1))
```

3.1. Word frequencies

Our first example counts the number of occurrences of each word in a file. The logic is similar to the logic in the character-frequency example, and ignores the line-number returned by `get-words`:

```
(define (word-freq file)
  (with-input-from-file file
    (lambda ()
      (map-reduce
        (lambda (x) (values (car x) 1))
        (lambda (k v1 v2) (+ v1 v2))
        string<?
        (get-words)))))
```

Evaluating `(word-freq "mapreduce.ss")` yields:

```
(("0" . 1)
 ("1" . 1)
 ("2" . 1)
 ("3" . 1)
 ("4" . 1)
 ("and" . 5)
 ("balance" . 3)
 ("base" . 4)
 ("black" . 11)
 ("black?" . 5)
 ("c" . 12)
 ("call-with-values" . 1)
 ("car" . 1)
 ("cdr" . 1)
 ("color" . 10)
 ("cond" . 3)
 ("cons" . 4)
 ("define" . 15)
 ("else" . 3)
 ("empty" . 5)
 ("empty?" . 5)
 ("enlist" . 4)
 ("eqv?" . 3)
 ("if" . 1)
 ("ins" . 4)
 ("insert" . 2)
 ("items" . 6)
 ("k" . 15)
 ("key" . 13)
 ("l" . 24)
 ("lambda" . 2)
 ("let" . 2)
 ("let*" . 1)
 ("lkid" . 21)
 ("loop" . 2)
```

```
("lt?" . 3)
("map-reduce" . 1)
("mapper" . 2)
("nil" . 4)
("pair?" . 1)
("r" . 24)
("red" . 6)
("red?" . 9)
("reducer" . 2)
("rkid" . 21)
("t" . 34)
("tc" . 4)
("tk" . 6)
("tl" . 4)
("tr" . 4)
("tree" . 18)
("tv" . 4)
("v" . 13)
("value" . 13)
("vector" . 1)
("vector-ref" . 5)
("z" . 5)
("zk" . 2)
("zl" . 2)
("zr" . 2)
("zv" . 2))
```

3.2. Inverting a frequency table

The frequency listing shown above is useful, but it might be preferable in descending order of frequency. That can be done by a function that inverts a table:

```
(define (invert table)
  (map-reduce
    (lambda (x) (values (cdr x) (car x)))
    (lambda (k v1 v2)
      (string-append v1 " " v2))
    (lambda (k1 k2) (> k1 k2))
    table))
```

The mapping function extracts the frequency count as the key and the word as the value, the reducing function appends two words in input order, and the ordering function sorts in descending order by reversing the sense of the comparison. We can see the descending frequency counts in the sample text by evaluating `(invert (word-freq "mapreduce.ss"))`:

```
((34 . "t")
 (24 . "l r")
 (21 . "lkid rkid")
 (18 . "tree")
 (15 . "define k")
 (13 . "key v value")
 (12 . "c")
 (11 . "black")
 (10 . "color")
 (9 . "red?")
 (6 . "items red tk")
 (5 . "and black? empty empty?
        vector-ref z")
 (4 . "base cons enlist ins nil
       tc tl tr tv")
 (3 . "balance cond else eqv? lt?")
 (2 . "insert lambda let loop mapper
       reducer zk zl zr zv")
 (1 . "0 1 2 3 4 call-with-values
       car cdr if let* map-reduce
       pair? vector"))
```

3.3. Cross-referencing a file

`Map-reduce` and `get-words` can produce a cross-reference listing of a file, associating each word with the line numbers where it appears:

```
(define (xref file)
  (with-input-from-file file
    (lambda ()
      (map-reduce
        (lambda (x)
          (values (car x)
                  (list (cdr x))))
        (lambda (k v1 v2)
          (if (eq? (car v1) (car v2))
              v1
              (cons (car v2) v1)))
        string<?
        (get-words)))))
```

The mapping function forms a singleton list of each line number, and the reducing function conses additional line numbers onto the list. Since `get-words` returns the input words in reverse order, the line numbers for each input word are consed onto the output list in ascending order. Applied to the sample text by `(xref "mapreduce.ss")`, the result is:

```
(("0" 5)
 ("1" 6)
 ("2" 7)
 ("3" 8)
 ("4" 9)
 ("and" 13 18 22 26 43)
 ("balance" 12 36 37)
 ("base" 41 42 44 47)
 ("black" 3 11 15 17 20 21
          24 25 28 29 40)
 ("black?" 11 13 18 22 26)
 ("c" 2 10 11 12 13 18 22 26 31)
 ("call-with-values" 50)
 ("car" 51)
 ("cdr" 52)
 ("color" 5 13 18 22 26 34)
 ("cond" 13 35 42)
 ("cons" 44 46)
 ("define" 1 2 3 4 5 6 7 8 9
           10 11 12 32 33 41)
 ("else" 31 38 45)
 ("empty" 3 4 35 48)
 ("empty?" 4 35 42 43)
 ("enlist" 41 45 47 53)
 ("eqv?" 4 10 11)
```

```
("if" 49)
("ins" 33 36 37 39)
("insert" 32 52)
("items" 1 48 49 51 52)
("k" 2 12 17 21 24 28 31 32
       35 36 37 38 52)
("key" 6 14 15 19 20 23 25 27
         29 34 39 44 46)
("l" 2 12 13 14 15 16 17 18
       19 20 21 24 28 31)
("lambda" 51 52)
("let" 34 48)
("let*" 39)
("lkid" 8 13 15 16 20 22 23 24
          25 28 30 34 39 43 45)
("loop" 48 52)
("lt?" 1 36 37)
("map-reduce" 1)
("mapper" 1 51)
("nil" 3)
("pair?" 49)
("r" 2 12 17 21 22 23 24 25
       26 27 28 29 30 31)
("red" 10 14 19 23 27 35)
("red?" 10 13 18 22 26)
("reducer" 1 38)
("rkid" 9 16 17 18 19 20 21 25
          26 29 30 34 39 43 47)
("t" 4 5 6 7 8 9 32 33 34 35 39 41 42
       43 44 45 46 47 48 52 53)
("tc" 34 36 37 38)
("tk" 34 36 37 38)
("tl" 34 36 37 38)
("tr" 34 36 37 38)
("tree" 2 3 14 15 17 19 20 21 23
          24 25 27 28 29 31 35 38 40)
("tv" 34 36 37 38)
("v" 2 12 17 21 24 28 31 32 35 38 52)
("value" 7 14 15 19 20 23 25 27
           29 34 39 44 46)
("vector" 2)
("vector-ref" 5 6 7 8 9)
("z" 39)
("zk" 39 40)
("zl" 39 40)
("zr" 39 40)
("zv" 39 40))
```

Cross-reference listings like this are a standard service for programmers; `map-reduce` makes them trivially easy to produce.

3.4. Anagrams

Our last example shows how to identify anagram classes in an input dictionary defined as a list of words (strings). The mapping function "signs" each word by sorting its characters into alphabetical order, and the reducing function brings together words with common signatures. As an example, `(anagrams '("time" "stop" "pots" "cars" "emit"))` evaluates to `("cars" "time emit" "stop pots")`. The code is:

```
(define (anagrams dict)
  (map cdr
    (map-reduce
      (lambda (x)
        (values
          (list->string
            (sort char<?
              (string->list x)))
          x))
      (lambda (k v1 v2)
        (string-append v1 " " v2))
      string<?
      dict)))
```

4. Implementing the `map-reduce` function

The `map-reduce` function `cdr`s through the input list, submitting each item to the mapping function, then submitting the key/value pair returned by the mapping function to a dictionary that either inserts the key/value pair, if the key isn't in the dictionary, or uses the reducing function to combine the new value with the existing value if the key already exists in the dictionary. Once the input list is exhausted, the key/value pairs are retrieved from the dictionary and inserted into the output list in order. The complete implementation appears in the appendix; the main processing loop is shown below:

```
(let loop ((items items) (t empty))
  (if (pair? items)
      (call-with-values
        (lambda () (mapper (car items)))
        (lambda (k v)
          (loop (cdr items)
            (insert t k v))))
      (enlist t '())))
```

4.1. The dictionary

There are many ways to implement the dictionary; we choose a red/black tree, which is easy to implement and, being nearly balanced, is reasonably fast. We prefer a tree to a hash table because the final output must be sorted, and the tree does the sort for free.

Our implementation of red/black trees is stolen from section 3.3 of Chris Okasaki's book *Purely Functional Data Structures*; those interested in the details of the trees' operation should refer to the book. We provide two functions: `(insert tree key value)` and `(enlist tree base)`.

The `insert` function recursively winds down the red/black tree until it finds the proper place for the key. If the key doesn't exist, a new node is added to the tree using the current key/value combination, and balancing rotations and re-colorings are performed as the stacked function calls unwind. However, if the key already exists, the reducing function is called, passing the key, the current value, and the new value as arguments, and the result replaces the value currently in the tree.

The `enlist` function performs in-order traversal of the tree, building the list right-to-left so that it is properly ordered when the function completes. The initial recursive base is the null list.

The dictionary functions are local within `map-reduce` to avoid name-space pollution and because that makes the `reducer` and `lt?` functions available without being passed as arguments to the dictionary.

By the way, the frequency table shown above provides a useful sanity check on the red/black tree implementation. The `lkid` and `rkid` variables each appear the same number of times, as do the `l` and `r` variables; since each access of the tree must touch both its left and right sub-trees, it is reassuring that the references appear the same number of times. The `black`/`red` and `black?`/`red?` counts differ because `black` and `red` nodes perform different functions in the tree.

4.2. The input list

In the current implementation of the `map-reduce` function, input is presented to the function in a list. Sometimes, it may be useful for the input to take other forms. For instance, a database cursor may return thousands of records, so that it is inconvenient or impossible to store them all in memory.

There are several alternatives. `Map-reduce` can be changed to take input from a port, a generator (see section 13.3 of Dorai Sitaram's book *Teach Yourself Scheme in Fixnum Days*, available at `www.ccs.neu.edu/-home/dorai/t-y-scheme/t-y-scheme.-html`), a stream (see my paper *A Library of Streams* at `srfi.schemers.org/srfi-40`) or from some other data structure that discards items as they are used. Or the `map-reduce` function can be customized for a particular application, fetching individual items as needed. All these alternatives are accommodated by simple changes to the main loop of the `map-reduce` function.

5. Conclusion

`Map-reduce` conflates processing, searching, combining and sorting into a useful programming idiom. It makes some simple programs very easy to express, as we have seen, and is also useful as a way to structure larger programs. It permits a large data set to be treated as a unit instead of in its constituent pieces, and promotes code reuse. `Map-reduce` belongs in the toolbox of every Scheme programmer.

Appendix 1: The `get-words` function

```
(define (get-words . port)
  (define (get-word p)
    (let loop ((c (peek-char p)) (rev-word '()))
      (cond ((eof-object? c) (if (pair? rev-word) (list->string (reverse rev-word)) c))
            ((char-in-word? c)
              (let ((x (read-char p))) (loop (peek-char p) (cons x rev-word))))
            ((pair? rev-word) (list->string (reverse rev-word)))
            ((char=? #\newline c) (read-char p) "")
            (else (read-char p) (loop (peek-char p) rev-word)))))
  (let ((p (if (null? port) (current-input-port) (car port))))
    (let loop ((w (get-word p)) (line 1) (word-list '()))
      (cond ((eof-object? w) word-list)
            ((string=? "" w) (loop (get-word p) (add1 line) word-list))
            (else (loop (get-word p) line (cons (cons w line) word-list)))))))
```

Appendix 2: `mapreduce.ss`

```
(define (map-reduce mapper reducer lt? items)
  (define (tree c k v l r) (vector c k v l r)
  (define empty (tree 'black 'nil 'nil 'nil 'nil)
  (define (empty? t) (eqv? t empty))
  (define (color t) (vector-ref t 0)
  (define (key t) (vector-ref t 1))
  (define (value t) (vector-ref t 2))
  (define (lkid t) (vector-ref t 3))
  (define (rkid t) (vector-ref t 4))
  (define (red? c) (eqv? c 'red))
  (define (black? c) (eqv? c 'black))
  (define (balance c k v l r)
    (cond ((and (black? c) (red? (color l)) (red? (color (lkid l))))
           (tree 'red (key l) (value l)
             (tree 'black (key (lkid l)) (value (lkid l))
               (lkid (lkid l)) (rkid (lkid l)))
             (tree 'black k v (rkid l) r)))
          ((and (black? c) (red? (color l)) (red? (color (rkid l))))
           (tree 'red (key (rkid l)) (value (rkid l))
             (tree 'black (key l) (value l) (lkid l) (lkid (rkid l)))
             (tree 'black k v (rkid (rkid l)) r)))
          ((and (black? c) (red? (color r)) (red? (color (lkid r))))
           (tree 'red (key (lkid r)) (value (lkid r))
             (tree 'black k v l (lkid (lkid r)))
             (tree 'black (key r) (value r) (rkid (lkid r)) (rkid r))))
          ((and (black? c) (red? (color r)) (red? (color (rkid r))))
           (tree 'red (key r) (value r)
             (tree 'black k v l (lkid r))
             (tree 'black (key (rkid r)) (value (rkid r))
               (lkid (rkid r)) (rkid (rkid r)))))
          (else (tree c k v l r))))
  (define (insert t k v)
    (define (ins t)
      (let ((tc (color t)) (tk (key t)) (tv (value t)) (tl (lkid t)) (tr (rkid t)))
        (cond ((empty? t) (tree 'red k v empty empty))
              ((lt? k tk) (balance tc tk tv (ins tl) tr))
              ((lt? tk k) (balance tc tk tv tl (ins tr)))
              (else (tree tc tk (reducer k tv v) tl tr)))))
    (let* ((z (ins t)) (zk (key z)) (zv (value z)) (zl (lkid z)) (zr (rkid z)))
      (tree 'black zk zv zl zr)))
  (define (enlist t base)
    (cond ((empty? t) base)
          ((and (empty? (lkid t)) (empty? (rkid t)))
           (cons (cons (key t) (value t)) base))
          (else (enlist (lkid t)
                        (cons (cons (key t) (value t))
                              (enlist (rkid t) base))))))
  (let loop ((items items) (t empty))
    (if (pair? items)
        (call-with-values
          (lambda () (mapper (car items)))
          (lambda (k v) (loop (cdr items) (insert t k v))))
        (enlist t '()))))
```