`MapReduce` Reading Guide

Parth Upadhyay (mostly), with a few additions by Michael Walfish

April 14, 2013

For this paper, skip sections 4 and 7.

Here are some hints for reading a research paper. First read the abstract and intro very carefully, line by line. Then read all typographically distinct items in the rest of the paper: section headings, figure captions, etc. (Skim lists and equations.) Then read the conclusion. By this point, you'll have a pretty good idea of the structure of the paper. (This is important because authors are usually deliberate about the structure of the paper, and you can understand a paper much more quickly if you have that structure in your head as you read.) Then go back and read the individual sections. When something doesn't make sense, write a note in the margin. Then at the end, go back and look at your margin questions and see which of them remain. Bring the remaining ones to class, or ask a friend.

Now, here are some questions to answer for this particular paper. Keep these in mind as you read.

1. `map` and `reduce` are constructs borrowed from the functional language Lisp. If you've ever programmed in a functional language, you know how powerful these constructs can be in describing computation. Look carefully at the example given in section 2.1, and consider the types defined in 2.2. The example shown computes the number of occurrences of each word. Let's do a slightly different problem; compute the number of occurences of words of a certain length! The input to the MapReduce framework will be a list of key-value pairs: `(document, value)`. The output should be a list of key-value pairs `(length of word, number of occurrences of words of that length)`.

```
map(String key, String value) {
    // key: document name
    // value: document contents
    // TODO: fill in code here
    // hint: it'll look similar to the example in the paper




}

reduce(String Key, Iterator values) {
    // key: number, length of word
    // values: a list of counts
    // TODO: fill in this code
```

```
}
```

2. Fault Tolerance is a huge concern when running on this many machines. What does the master do to check if a machine has failed? How is the failure of a machine handled? How is the failure of the master handled?

3. What was the motivation behind backup tasks? What performance optimization do they provide?

4. In practice, what kind of machines did the author's run their jobs on? Specifically, are they inexpensive machines? Expensive machines?