Hadoop and Map-reduce computing

1 Introduction

This activity contains a great deal of background information and detailed instructions so that you can refer to it later for further activities and homework.

Map-reduce is a strategy for solving problems using two stages for processing data that includes a sort action between those two stages. This problem-solving strategy has been used for decades in *functional programming* languages such as LISP or Scheme. More recently, Google has adapted the map-reduce programming model (Dean and Ghemawat, 2004) to function efficiently on large clusters in order to process vast amounts of data — for example, Google's selection of the entire web.

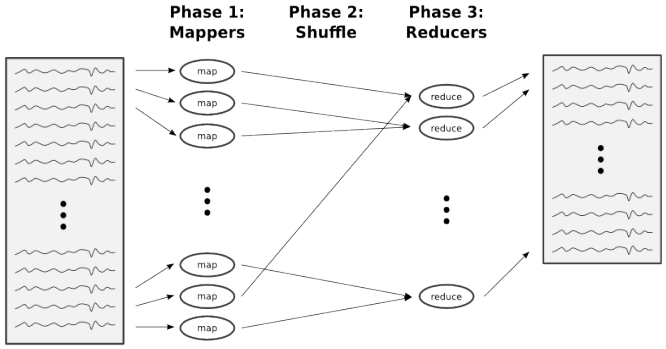
The Apache Foundation provides an open-source implementation of map-reduce for clusters called Hadoop, which has primarily been implemented by Yahoo!. St. Olaf student researchers have created a user interface called *WebMapReduce (WMR)* that uses Hadoop to make map-reduce programming convenient enough for CS1 students to use.

2 Map-reduce computing

In map-reduce programming, a programmer provides two functions, called the *mapper* and the *reducer*, for carrying out a sequence of two computational stages on potentially vast quantities of data.

- The *mapper* function is applied to each line of data, and breaks up that data into labeled units of interest, called *key-value pairs*.
- The *reducer* function is then applied to all the key-value pairs *that share the same key*, allowing some kind of consolidation of those pairs.

Between the mapper and reducer stages, a map-reduce system automatically reorganizes the intermediate key-value pairs, so that each call of the reducer function can receive a complete set of key-value pairs *for a particular key*, and so that the reducer function is called for every key in sorted order. We will refer to this reorganization of key-value pairs between the mapper and reducer stages as *shuffling*. A map-reduce implementation such as `Hadoop` takes care of details such as splitting the data and shuffling, so a map-reduce programmer needs only provide mapper and reducer algorithms. The following diagram illustrates the three steps of mapping, shuffling, and reducing.



3 Wordcount, an example

The goal is to start with an input text (of arbitrary size), and print a table of frequencies of each word appearing in that text. For example, if the input is the phrase "`the cat in the hat`", then the following table should be produced:

```
cat     1
hat     1
in      1
the     2
```

The mapper function will receive one line of input at a time. It will split up that line into words, then produce the key-value pairs for each *word* found in the line.

```
word    1
```

A call to the reducer function will receive all the key-value pairs for some word, e.g.,

```
the     1
the     1
the     1
the     1
```

That call to the reducer function will add the values, then produce one key-value pair representing the count for that word.

```
the     4
```

We can represent this algorithm schematically as follows:

- Mapper:

```
                    "the cat in the hat"
                             |
                             V
                    [mapper for this line]
                             |
        /----------+---------+-----------+-----------\
        |          |         |           |           |
        V          V         V           V           V
("the" "1") ("cat" "1") ("in" "1") ("the" "1") ("hat" "1")
```

- Reducer:

```
 ("the" "1") ("the" "1")
      |           |
      \-----+-----/
            |
            V
[reducer for the key "the"]
            |
            V
       ("the" "2")
```

Python implementation for the mapper function, for our *WebMapReduce* (WMR) system:

```
def mapper(key, value):
  words = key.split()
  for word in words:
    Wmr.emit(word, "1")
```

Notes:

- The WMR system calls a function named `mapper()` once for each line of input. When the input data consists of plain text, not key-value pairs, then each line of input text becomes a key, and the value is treated as empty.
- The WMR system for Python defines a class `Wmr` that includes a class method `emit()` for producing key-value pairs to be forwarded (via shuffling) to a reducer. `Wmr.emit()` requires two string arguments, so we used the string `'1'` instead of the number `1` when we called `Wmr.emit()` in the `mapper()`.

Python implementation for the reducer function, for WMR:

```
def reducer(key, iter):
  sum = 0
  for s in iter:
    sum = sum + int(s)
  Wmr.emit(key, str(sum))
```

Notes:

- The function `reducer()` is called once for each distinct key that appears among the key-value pairs emitted by the mapper, and that call processes all of the key-value pairs that use that key. The two arguments of `reducer()` are that common `key` and an *iterator* `iter`, which provides access to all the values for that key. Iterators in Python3 are designed for `for` loops; for example, the Python3 function `range()` returns an iterator (recall that one must perform a type conversion `list(range(...))` in order to produce a *list* of values from `range()`).

  Rationale: WMR `reducer()`'s use iterators instead of lists because the number of values may be very large in the case of large data. For example, there would be billions of occurences of the word "the" if our data consisted of all pages on the web. When there are a lot of key-value pairs, it is more efficient to dispense pairs one at a time through an iterator than to create a gigantic complete list and hold that list in main memory; also, an enormous list may overfill main memory.
- The `reducer()` function adds up all the 1's that appear in key-value pairs for the `key` that appears as `reducer()`'s first argument. Each 1 provided by the iterator `iter` is a

string, so we must first convert it to an integer before adding it to the running total `sum`.

- The method `Wmr.emit()` is used to produce key-value pairs as output from the mapper. This time, only one pair is emitted, consisting of the word being counted and `sum`, which holds the number of times that word appeared in *all* of the original data.

4 Trying it out

1. In a browser, visit the WMR site. Login using the WMR username and password provided for you, then select "Launch WMR".
2. Enter a jobname (perhaps involving your username, for uniqueness; I'd avoid spaces in the jobname).
3. Choose the Python language.
4. Enter the input data, e.g., the `cat in the hat` line above. You can use the "Direct Input" option and enter that data in the text box provided.
5. Enter the mapper. You can use the "Direct Input" option as before for the data; alternatively, you can use the "Upload" option and navigate to a function that contains the mapper (this is more convenient for repeated runs). Check that the appropriate radio button is clicked to indicate the source option you're actually using.
6. Also enter the reducer.
7. Click on the submit button.
8. A status page should appear indicating that the job started successfully and is running. Refresh the page (titled "Job in Progress") as desired until the job completes.
9. Once the job runs to completion, refreshing the job status page will lead to a Job Complete page. This page will include your output. If you used the cat input, your output should match the illustration above.
10. Try running the program again, this time with large data (ask your instructor where you can find larger data sets like whole books).

5 More about parallel computing

Map-reduce computing is designed for computer clusters. A *cluster* of computers consists of multiple computers that are networked together and equipped with software so they can perform cooperative computations on a single problem. The individual computers in a cluster are often called *nodes* in that cluster.

Map-reduce computations are performed on a cluster in order to carry out large-scale data operations in a reasonable amount of time. We say that map-reduce computations are *scalable*, meaning that the computations can remain feasible even when there is an increase in the size or complexity of those computations.

The strategy of performing multiple physical computations at the same time is called *parallel* computing. Running map-reduce in parallel on multiple computers in a cluster enables the map-reduce to scale to large data sets. Google and other companies run map-reduce on clusters with thousands of nodes, enabling them to compute with large sections of the entire world-wide web. The total number of bytes in such a computation may be on the order of tens of

petabytes (1 petabyte is about 1 quadrillion characters)!

Here are two basic categories of parallelism:

1. *Data parallelism* is when the same algorithms are carried out on different computers with different data at the same time. For example, if map-reduce is applied to a large data set (e.g., gigabytes or terabytes, where a terabyte is about a trillion bytes), that data is first divided up into "slices" that may be processed on separate computers in a cluster.
2. *Task parallelism* is when different algorithms are carried out at the same time. An assembly line is an example of task parallelism: multiple stations carry out different tasks at the same time. For another example, the mapper and reducer functions represent two different algorithms. If the key-value pairs produced by a mapper running on one computer were forwarded directly to a reducer running on a different computer as soon as those key-value pairs were produced, that would have an example of task parallelism. (In practice, the mapping, shuffling, and reducing stages of map-reduce are each completed in order, for scalability and other reasons discussed below.)

WMR uses the *Hadoop* implementation of map-reduce, which is an open-source software project (meaning that the program code is made publicly accessible) under the Apache Software Foundation. The primary authors of Hadoop are employees of Yahoo!, Inc., and this software is used professionally by Yahoo! and other companies as well as at educational institutions. (Google's implementation of map-reduce is proprietary, i.e., a company secret.)

Programming Hadoop directly requires more programming background than a beginning student can handle, in the Java programming language. WMR makes it possible for beginning programmers to program map-reduce computations in the language they are learning. Although the programming is simplified, genuine Hadoop map-reduce computations are carried out (except for the testing option), so significant large-scale computing involving parallelism can be performed.

Some examples of large data:

- Netflix data, originally distributed for a competition sponsored by Netflix, consists of lines containing a movie ID number, a viewer ID number, a rating, and a date. (Approx 2 GB = 2 gigabytes)
- "Snapshots" of Wikipedia are available. (approximately 150 MB)
- Facebook data is obviously large, but privacy protections make it difficult to deal with.
- For linguistic analysis via map-reduce, there are many potential sources of data on the Internet, such as online newspapers.
- Scientific data may be taken from experimental readings or generated from a computational model of a phenomenon.
- Project Gutenberg and Google Books projects provide public-domain complete literary works. For example, we have retrieved Dostoevsky novels and the complete works of Edgar Allen Poe through this source.

6 Multiple map-reduce cycles

WMR enables you to use the output from one map-reduce cycle as the input for a subsequent map-reduce cycle. In that case, the mapper function in the second cycle may use both the key and value for its computations.

Example:

- Ordering word-count output in frequency order
- Computing average ratings per movie on the first map-reduce cycle; ordering by rating on a second cycle.

One could write an *identity reducer*, which emits key-value pairs without any change (though the output will be sorted as it has gone through the shuffling step.) An *identity mapper* is similar in that it emits key-value pairs from lines of input data without alteration.

An identity reducer in Python:

```
def reducer(key, iter):
  for s in iter:
    Wmr.emit(key, s)
```

An identity mapper in Python:

```
def mapper(key, value):
  Wmr.emit(key, value)
```

7 Creating an index

- Suppose that input for the mapper stage consists of a location identifier (e.g., line number) as the key and a line of text as the value. How could one produce an index, where each word *in the text* is listed together with the line numbers it appears in?
- A *concordance* consists of the words in a text together with the line that contains them, for context. How could you create a concordance using WMR?