Notes on Map-Reduce and Hadoop – CSE 40822
Prof. Douglas Thain, University of Notre Dame, February 2016

Caution: These are high level notes that I use to organize my lectures. You may find them useful for reviewing main points, but they aren't a substitute for participating in class.

References:

- Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
- Jimmy Lin and Chris Dyer, Data-Intensive Text Processing with MapReduce, Morgan & Claypool Publishers, 2010.
- K. Shvachko, H. Kuang, S. Radia, R. Chansler, "The Hadoop Distributed Filesystem", IEEE Mass Storage Systems and Technologies, 2010.

Background and Context

Early days of the Google web search engine. (2004)
Complex programs mixed up logic with fault tolerance.
Simplified computing model: Map-Reduce
Result: Much greater productivity at scale.

The Map-Reduce Programming Model

User provides two functions: Map and Reduce, and asks for them to be invoked in a given data set. They must have the following form:

```
Map( key, value ) -> list( key, value )
Reduce( key, list(values) ) -> output
```

The **framework** is responsible for locating the data, applying the functions, and then storing the outputs. The user is not concerned with locality, fault tolerance, optimization, and so forth.

The Map functions are applied to each of the files comprising the data sets, and emit a series of (key,value) pairs. Then, for each key, a bucket is created for all of the values with that key. The Reduce function is then applied to all values in that bucket.

(Blackboard diagram of how this works.)

Example Map-Reduce Programs

WordCount is the "hello world" of Map-Reduce. This program reads in a large number of files and computes the frequency of each unique word in the input.

```
Map( key, value ) {
        // key is the file name
        // value is the file contents
        For each word in value {
                Emit( word, 1 )
        }
}

Reduce( key, list(values) ) {
        count = 0;
        For each v in list(values) {
                count++;
        }
        Emit( key, count );
}
```

Sometimes you need to run multiple rounds of Map-Reduce in order to get the desired effect. For example, suppose you now want to generate the top ten most frequently used words in this set of documents. Run Map-Reduce on the output of the previous, but with this program:

```
Map( key, value ) {
        word = key
        count = value
        Emit( 1, "count word");
}

Reduce( key, list(values) ) {
        For first ten items in list(values) {
        Emit( value )
        }
}
```

Example Problems to Work in Class

Suppose you have the following weather data. A set of (unsorted) tuples, each consisting of a year, month, day, and the maximum observed temp that day:

(2007,12,10,35)
(2008,3,22,75)
(2015,2,15,12)
. . .

1. Write a Map-Reduce program to compute the maximum temperature observed each month for which data is present.

2. Write a Map-Reduce program to compute the average temperature for the day of the year (over all years).

Suppose that you have data representing a graph of friends:

A -> B,C,D
B -> A, C, D
C -> A,B
D -> A,B

3 . Write a Map-Reduce program that will identify common friends:

(A,B) -> C,D
(A,C) -> B
(A,D) -> . . .

The Hadoop Distributed System

Hadoop began a an open-source implementation very similar in spirit to the Google File System (GFS) and the Map-Reduce programming model. It has grown into a complex ecosystem of interacting pieces of software.

HDFS - Hadoop Distributed Filesystem

Architecture:
- One Name Node + Many Data Nodes
- Files divided into large 64MB chunks.
- Files once written, are immutable.
- Chunks are replicated three times in two different racks.

Interface:
- Java library.
- Hadoop command-line tool.
- Status web page.

Considerations:
- Fault tolerance.
- High access latency.
- Uploading can be slow, due to replication.
- Very high throughput on parallel reads.
- Secondary name node performs log compression.
- Multiple disks per data node.

MR – Hadoop Map-Reduce

Architecture:

One JobTracker per cluster coordinates the entire M-R computation.
TaskTrackers on each node dispatch and monitor each M-R task.
HDFS -> Maps -> Temporary Space -> Shuffle -> Reducers -> HDFS

Interface:

Native M-R code in Java.
Other languages use the streaming interface.
Hadoop command-line tool.

Considerations:

Fault tolerance.
Stragglers.
Data balance.
Number of "reducers".

Question:

Which part of a Map-Reduce program is naturally scalable,
and which part is likely to be a bottleneck?

Does that affect how you would design a M-R program?

Overview of Hadoop Map-Reduce Assignment