MapReduce

As discussed in the previous session, currently there are a couple of computation models which might be used by different companies for different purposes. MapReduce is a programming model and an associated implementation for processing and generating large data sets. The terms Map and Reduce come from Lisp and functional programming.

What do we know about programming? Most of us have learned “programming” techniques in Java or C++ classes. Basically this is the way we have developed our programming:

First, we have learned the simple commands: if-statements, for loops, etc.

But we have seen that by using only the simple commands, some parts of the code (which we had spent lot of time writing them) are painfully similar.

So we wrote our first “function”! For decades (I think!), functional programming ruled the world! However, some of the functions together with specific data were closely related, and thus another abstraction level, “Classes” was invented. Using classes and keeping encapsulation in mind, we invented “Object Oriented Programming”.

However, the very idea of encapsulation (keeping related stuff hidden and near each other) may cause problems in parallel computing. Indeed object oriented programming is not the only way of constructing an abstract level over the functions.

Consider a recycling industry. Every day the company gets tons of used materials. Assume that we only have three types of materials: plastic, glass, and metal materials. Of course each type of material needs a different recycling process. If we want to sequentially recycle the materials, each recycling unit should inspect each input material, determine its type, and recycle the material according to its type. The problem is that each unit should be able to inspect and recycle any type of material. Thus creating and maintaining a new recycling unit is costly.

An alternative parallel solution is to separate the inspection phase and the recycling phase. Each inspection unit simply *maps* each material to its type, e.g., maps each material to “plastic recycling”, “glass recycling”, or “metal recycling”. Then in the *shuffling* phase, we move the materials to the specific recycling units according to their types and finally each recycling unit recycles its input. In this parallel solution we can easily use the advantage of large number of units.

In *MapReduce* framework we use a similar method. The programmer should specify a map function, which maps each *value* in the input to a *key*, just like mapping a material to its type. Next, the programmer should specify a *reduce* function which given all the pairs with the same key, it produces the appropriate output, e.g., how to recycle the plastic materials. The execution framework applies the map function on every input. After this mapping phase, we have a set of “(key, value)” pairs. In the shuffle phase, the execution framework aggregates values by keys. Thus after this phase all the pairs with the same key are bundled together as “(key, array of values)”. In the reduce phase, the execution framework applies the reduce function on each “(key, array of values)” and produces the output.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time 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.

Therefore, all that the programmer should do is to define two functions:

- map(val) -> emits a set of (key, val) : is run on each val in the input.
- reduce(key, val[]) -> emits the final output : is run for each unique key emitted by map functions.

Now let us see a few algorithms for basic problems.

Search

Given a pattern and a set of files, each file containing a string in each line, output the files which contain the pattern in at least one line. Note that the files might have large number of lines.

```
method map(filename-linenumber fl, string st, pattern p)
    if st matches p
        Emit(filename-linenumber fl, _)

method reduce(filename-linenumber fl, -)
    Output(fl)
```

Word Count

Given a set of documents, count the number of appearances of different terms.

```
method map(url u, content cnt)
    for all term t in cnt do
        Emit(term t, count 1)

method reduce(term t, counts [c1,c2,c3,…])
    sum := 0
    for all count c in counts [c1,c2,c3,…] do
        sum := sum + c
    Output(term t, count sum)
```

For example: url1=look at him; url2=look at her.
Mapping phase: (look,1) (at,1) (him,1) (look,1) (at,1) (her,1)
Reducing phase: (look,2) (at,2) (him,1) (her,1)

In some implementations there are also two other functions: *combine* and *partition*. A combine function runs on the pairs that are emitted from map function which may reduce the number of pairs by combining the pairs with the same key. A partition function divides up key space for parallel reduce operations.

How can we use combiner to reduce network I/O in the last two problems?

Sort

Give a set of files, each file containing some values, sort all the numbers.

If we use a simple map here, we will need n reducers. Thus map functions should use a hash function such that $range(h) = \#reducers$ and if $x \leq y$ then $h(x) \leq h(y)$.

In general we may need more than one round of map/reduce in our algorithm. Thus in their general forms, the map function processes a key/value pair to generate a set of intermediate key/value pairs, and the reduce function merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this multi-round model (see [1]).

For example: BFS!

Intuition: $DistanceTo(startNode) = 0$; For all nodes n reachable from some set of nodes S, $DistanceTo(n) = 1 + min(DistanceTo(m), m \in S)$.

A map task receives

- Key: node x
- Value: D (distance from start), points-to (list of nodes reachable from x))

```
method map(x, D, points-to)
    for all nodes x in points-to do
        Emit(x, D+1)
    Emit(x, points-to)

method reduce(x, [Dis], points-to)
    Emit(x, min([Dis]), points-to)
```

Different Implementations I found:

- **Hadoop doesn’t have much to do with SQL or traditional database management.**
- **Aster Data’s MapReduce implementation.**
- **Greenplum’s MapReduce implementation.**

Sources:

[1] **MapReduce: Simplified Data Processing on Large Clusters** by Jeffrey Dean and Sanjay Ghemawat.

[2] **http://www.joelonsoftware.com/items/2006/08/01.html** by Joel Spolsky.

[3] **http://www.cs.umd.edu/class/fall2010/cmsc433/lectures/mapreduce.pdf**

[4] **http://www.cs.rutgers.edu/~muthu/mapreduce_spr11.html**

[5] **http://www.umiacs.umd.edu/~jimmylin/cloud-2010-Spring/session3-slides.pdf**

[6] **http://www.cloudera.com/wp-content/uploads/2010/01/5-MapReduceAlgorithms.pdf**

[7] **http://www.dbms2.com/2008/09/05/three-different-implementations-of-mapreduce/**

[8] **www.umiacs.umd.edu/~jimmylin/cloud-2008-Fall/Session5.ppt**