Caveat emptor

Last time, I described my adventures with Eclipse 3.3 with Eclipse Hadoop Plugin Yahoo Hadoop VM 0.18.0.

Problems:
Eclipse Hadoop Plugin doesn't work perfectly except with 0.13.0, and generates deprecated java stubs!
Hadoop 0.18.0 is too old and syntax has changed for 0.20.1.

I concluded that this is the wrong way to learn Hadoop. Last night, I got much farther by using:
Text editing instead of Eclipse
Videos, notes, example projects, and exercises from cloudera.com
The apache ant Jar builder.
Cloudera Hadoop VM 0.20.1
Cloudera Hadoop Distribution 0.20.1
Learning Hadoop my way:
Get cloudera examples from /comp/150CPA/Hadoop-projects (these are configured for hadoop 0.20.1 and compatible with the Cloudera VM.)
Use existing simple projects as templates.
Edit the text files.
Compile and build using "ant" command.
This works on any of our linux machines.
Copy built "jar" file to running vm instance using scp.
Populate HDFS using "hadoop fs" command.
Run your code using "hadoop jar" command.
Some resources

The hadoop VM:
/comp/150CPA/hadoop/cloudera-training-0.3.3.tar.bz2

The Cloudera hadoop distribution:
/comp/150CPA/hadoop/cloudera-hadoop-0.20.tar.gz

Cloudera examples:
/comp/150CPA/Hadoop-projects

See Homework 03!
Abstraction
   It should be obvious that a MapReduce program is mostly boilerplate code.
Key to MapReduce is being able to understand programs at an appropriately high level of abstraction.
A MapReduce program implements a dataflow pipeline.
Each step ("map" or "reduce") does a data transformation whose result is used in the next step.
We will document data flow and ignore class structure.
Given data flow, class structure can be derived.
A simple notation

Data in MapReduce is a sequence of **tuples**.

- `<k,v>`: a key/value pair for Map.
- `<k,List<v>>`: a key/value pair for Reduce

(Actually, the iterator is over a **tree** instead of a list, but we don't need to know that.)
A succinct version of wordcount

Wednesday, March 03, 2010
9:27 AM

Let

- List<X> represent a list of things X
- sum(List<X>) be the sum of a list of numbers X
- split(X) split a line into a list of words List<w>.
- foreach X (Y) set x to each element of a list Y in turn.

Then we can express last time's wordcount succinctly as

**Map:**

Given <offset,line>

- foreach word (split(line))
  - output <word,1>

**Reduce:**

Given <word,List<count>>

- output <word,sum(List<count>)>

\[ sum \]
A picture of wordcount

```latex
\text{Here is a test.}
\text{This is another test.}
\text{reduce}
\text{"this", 1} \rightarrow \text{"another", 1} \rightarrow \text{"this", 2} \rightarrow \text{"another", 1}
```
While Java is a serial language, wordcount is a functional (dataflow) program. Can express functional programs as composition of functions. Sometimes, we have to define functions with obvious meanings, e.g., split and sum.
Some more notation:

Let document(k) be the name of the document in which key k resides. This is a function supplied in MapReduce.

Notate composite keys consisting of two things as <k,l>, e.g., the input to a map might be <<k,l>,<m,n>>, which means that the pair k and l are together a key for the pair <m,n>.

Notate lists of two or more things as List<m,n>, which represents a list of pairs <m,n>.

Thus, the input to Reduce might be <<k,l>,List<m,n>> which means that the key is a pair <k,l> and the values are a list of pairs <m,n>.
This is not java parameterized class notation!
List<k,l> means a list of pairs k,l, which in Java would
be notated as Iterator<Pair<k,l>>
There is nothing in Java that could represent
List<k,l,m> without constructing, e.g., a new class
Triple, e.g., Iterator<Triple<k,l,m>>.
Problem 2 from last time: count the minimum and maximum frequency of words in documents, i.e., for each document d, output <d,<min,max>>.

In the exercise 8 answers, I "cheated" and changed the input driver to supply documents as a whole. Here's a way **without** changing the input driver:

**Map:**
- Given <offset, line>
  - let doc = document(offset)
  - foreach word (split(line))
    - output <<doc,word>,1>

**Reduce:**
- Given <<doc,word>,List<count>>
  - output <<doc,word>,sum(List<count)>

# after this, <<doc, word>, count> means count is the word count for word in doc.

**Map:**
- Given <<doc, word>, count>
  - output <doc, <count,count>>

**Reduce:**
- Given <doc, List<low, high>>
  - output <doc, <min(List<low>), max(List<high>)>>

(where min(L) and max(L) are the minimum and maximum values of a WritableComparable, and List<low> and List<high> are the obvious projections of List<low,high> into component lists)
The acid test of whether a reduce step is a combine step is associativity:
Know that $\min(\min(1,5),10) = \min(1,\min(5,10))$
Thus one can do reduce "early", in the sense that any partial reduce is allowed.
Understanding min and max line counts

Wednesday, March 03, 2010
11:37 AM
In my notation, java classes are **implicit**.

E.g., the map input \( \langle k, l \rangle, c \rangle \) means the key has a type like:

```java
class KLstuff implements WritableComparable {
    private Ktype k;
    private Ltype l;

    ...
}
```

while the value might be declared as

```java
class CStuff implements Writable {
    private CType c;

    ...
}
```

Likewise the reduce input \( \langle k, \text{List}<r,s> \rangle \) has a key class that contains k and implements WritableComparable while the list contains instances of a class that contains r and s and implements Writable.
Problem 4 from last time: compute minimum distance between cities given <city1, <distance, city2>>

The key to this is to realize two things:

Dijkstra’s algorithm for shortest path relies upon the construction "if dist(A,B)+dist(B,C)<dist(A,C) then dist(A,C)=dist(A,B)+dist(B,C)"

Each call of the reduce phase for <city1, List<distance, city2>> gets a simplex of the graph:

So, one can act on the simplex by computing new candidate distances between city2 and city3!
Shortest distance in a graph

Input: <c1, <d, c2>> representing distance d between cities c1 and c2.
Output: <<c1,c2>,d> where d is the shortest distance between c1 and c2 using intermediate cities.

Map:
Given <c1, <d, c2>>
output <<c1,c2>,d>
output <<c2,c1>,d>

Reduce:
Given <<c1,c2>, List<d>>
output <<c1,c2>, min(List<d>)>

Repeat while distances change:
Map:
Given <<c1,c2>,d>
output <c1,<d,c2>>
Reduce:
Given <c1, List<d,c2>>
foreach <dist1,city1> (List<d,c2>)
output <<c1, city1>, dist1>
foreach <dist2, city2> (List<d,c2>)
if (city1 != city2)
output <<city1,city2>,dist1+dist2>
Map: # do nothing
Reduce:
  Given <<c1,c2>,List<d>>
    output <<c1,c2>,min(List<d>)>
end repeat
How to tell whether a dataset has changed.
Can merge output of two pipelines into one input.
Can thus compare them.

Algorithm: has data changed?
Input: two pipelines <oldk, oldv> and <newk, newv>
Output: differences, or nothing if they haven't changed.

Map:
Given <oldk, oldv>
    output <oldk, oldv,"old">>
Reduce: # does nothing

Map:
Given <newk, newv>
    output <newk, newv, "new">
Reduce: # does nothing

# now, using the driver, merge <oldk, oldv, "old">>
# and <newk, newv, "new">> into <allk, allv, state>>

Map: # does nothing
Reduce:
    Given <allk, List<allv, state>>>
      Let <firstv, firststate> be first element of list
      Let <secondv, secondstate> be second element
if secondv is not defined or firstv != secondv)
output <allk, "oops">

# if output is empty, pipelines agree completely.
A picture of comparison

\[ \langle k_1, v_{1a} \rangle \]
\[ \langle k_2, v_{2a} \rangle \]

\[ \langle k_1, v_{1b} \rangle \]
\[ \langle k_2, v_{2b} \rangle \]

\[ \langle k_1, v_{1b1}, \text{"new"} \rangle \]
\[ \langle k_2, v_{2b}, \text{"old"} \rangle \]

\[ \langle k_1, \text{false} \rangle \]
Some other dirty tricks

Passing a scalar variable as a local static variable in Java.

Reading the HDFS directly instead of using MapReduce, to turn a tuplespace into a scalar.