By: Steve Krenzel

Finding Friends

MapReduce is a framework originally developed at Google that allows for easy large scale distributed computing across a number of domains. Apache Hadoop is an open source implementation.

I'll gloss over the details, but it comes down to defining two functions: a map function and a reduce function. The map function takes a value and outputs key:value pairs. For instance, if we define a map function that takes a string and outputs the length of the word as the key and the word itself as the value then map(steve) would return 5:steve and map(savannah) would return 8:savannah. You may have noticed that the map function is stateless and only requires the input value to compute it's output value. This allows us to run the map function against values in parallel and provides a huge advantage. Before we get to the reduce function, the mapreduce framework groups all of the values together by key, so if the map functions output the following key:value pairs:

3 : the
3 : and
3 : you
4 : then
4 : what
4 : when
5 : steve
5 : where
8 : savannah
8 : research

They get grouped as:

3 : [the, and, you]
4 : [then, what, when]
5 : [steve, where]
8 : [savannah, research]

Each of these lines would then be passed as an argument to the reduce function, which accepts a key and a list of values. In this instance, we might be trying to figure out how many words of certain lengths exist, so our reduce function will just count the number of items in the list and output the key with the size of the list, like:

3 : 3
4 : 3
5 : 2
8 : 2

The reductions can also be done in parallel, again providing a huge advantage. We can then look at these final results and see that there were only two words of length 5 in our corpus, etc...

The most common example of mapreduce is for counting the number of times words occur in a corpus. Suppose you had a copy of the internet (I've been fortunate enough to have worked in such a situation), and you wanted a list of every word on the internet as well as how many times it occurred.

The way you would approach this would be to tokenize the documents you have (break it into words), and pass each word to a mapper. The mapper would then spit the word back out along with a value of 1. The grouping phase will take all the keys (in this case words), and make a list of 1's. The reduce phase then takes a key (the word) and a list (a list of 1's for every time the key appeared on the internet), and sums the list. The reducer then outputs the word, along with it's count. When all is said and done you'll have a list of every word on the internet, along with how many times it appeared.

Easy, right? If you've ever read about mapreduce, the above scenario isn't anything new... it's the "Hello, World" of mapreduce. So here is a real world use case (Facebook may or may not actually do the following, it's just an example):

Facebook has a list of friends (note that friends are a bi-directional thing on Facebook. If I'm your friend, you're mine). They also have lots of disk space and they serve hundreds of millions of requests everyday. They've decided to pre-compute calculations when they can to reduce the processing time of requests. One common processing request is the "You and Joe have 230 friends in common" feature. When you visit someone's profile, you see a list of friends that you have in common. This list doesn't change frequently so it'd be wasteful to recalculate it every time you visited the profile (sure you could use a decent caching strategy, but then I wouldn't be able to continue writing about mapreduce for this problem). We're going to use mapreduce so that we can calculate everyone's common friends once a day and store those results. Later on it's just a quick lookup. We've got lots of disk, it's cheap.

Assume the friends are stored as Person->[List of Friends], our friends list is then:

A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D

Each line will be an argument to a mapper. For every friend in the list of friends, the mapper will output a key-value pair. The key will be a friend along with the person. The value will be the list of friends. The key will be sorted so that the friends are in order, causing all pairs of friends to go to the same reducer. This is hard to explain with text, so let's just do it and see if you can see the pattern. After all the mappers are done running, you'll have a list like this:

For map(A -> B C D) :

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

For map(B -> A C D E) : (Note that A comes before B in the key)

(A B) -> A C D E
(B C) -> A C D E
(B D) -> A C D E
(B E) -> A C D E

For map(C -> A B D E) :

(A C) -> A B D E
(B C) -> A B D E
(C D) -> A B D E
(C E) -> A B D E

For map(D -> A B C E) :

(A D) -> A B C E
(B D) -> A B C E
(C D) -> A B C E
(D E) -> A B C E

And finally for map(E -> B C D):

(B E) -> B C D
(C E) -> B C D
(D E) -> B C D

Before we send these key-value pairs to the reducers, we group them by their keys and get:

(A B) -> (A C D E) (B C D)
(A C) -> (A B D E) (B C D)
(A D) -> (A B C E) (B C D)
(B C) -> (A B D E) (A C D E)
(B D) -> (A B C E) (A C D E)
(B E) -> (A C D E) (B C D)
(C D) -> (A B C E) (A B D E)
(C E) -> (A B D E) (B C D)
(D E) -> (A B C E) (B C D)

Each line will be passed as an argument to a reducer. The reduce function will simply intersect the lists of values and output the same key with the result of the intersection. For example, reduce((A B) -> (A C D E) (B C D)) will output (A B) : (C D) and means that friends A and B have C and D as common friends.

The result after reduction is:

(A B) -> (C D)
(A C) -> (B D)
(A D) -> (B C)
(B C) -> (A D E)
(B D) -> (A C E)
(B E) -> (C D)
(C D) -> (A B E)
(C E) -> (B D)
(D E) -> (B C)

Now when D visits B's profile, we can quickly look up (B D) and see that they have three friends in common, (A C E).

About the author
I'm Steve Krenzel, a software engineer and co-founder of Thinkfuse. Contact me at steve@thinkfuse.com.