What is MapReduce, Anyway?

MapReduce is inspired by three approaches from functional programming for applying a function to each item of a collection of data, namely, Map, Filter and Reduce. That is pretty abstract, so I will try to bring some of these ideas down to Earth. I'll use lists to represent the “data” in any examples, but the concepts in MapReduce can apply equally well to any data source: multiple streams from the Internet, a number of internal data stores from multiple sites, and even user keystrokes/mouse moves.

If a function (or operation) can be applied to each item in some kind of input data, you may be able to use map, filter and reduce.

Defining Terms

Map

When we use the expression Map(function: f, data: [1,2,3,4,5,6]), we are declaring that we want to apply the function "f" to each element in the data. In this case, we have a list of numbers, but the data could be names, employee records, or URLs for Internet documents from the Internet that we would like to parse to extract useful information.

Example: function f is square(x) = x * x, and the data is our list [1..6].

Map(square(x), [1,2,3,4,5,6]) = [square(1), square(2), ..., square(6)], or [1, 4, 9, 16, 25, 36]

Filter

Filtering data is essentially a variation of Map. You could think of it in two stages:

1. Apply a "test" function to Map each item to either True or False ("In" or "Out")
2. Use the results of that Map operation to drop any item that fails the test (False)

Having said this, a Filter does not have to be implemented in this way. By declaring that we want to use a Filter operation, we have specified WHAT we want to do. It really does not matter HOW it gets done.

Most functional programming tutorials would illustrate a Filter by selecting prime numbers from a list of integers, or to isolate numbers that are not multiples of 3. A more useful illustration of Filter is a search filter that reviews documents in a repository, or a set of search engine results, and returns only those that pass the "relevance test". The test itself could be defined using a "fuzzy" criterion for relevance (0-20% Not, 20-50% A Little, 50-75% Fairly, > 75% Very -- or what have you), but the end result is that you'll choose some documents to accept, and omit the rest.

In a filter operation on a large number of data items, you might want to drop the items as early as possible. There is no law that requires you to make these decisions in advance when you offer Map or Filter operations on a server.

In a MapReduce context, Map and Filter will often end up lumped together. This is fine, because you don't want to waste processing time to perform potentially expensive transformations on data or documents that you can rule out immediately with a less computationally expensive filter.

Reduce

A Reduce operation on a collection of data is any kind of aggregate operation that boils down all of the detail items into one or more summary metrics computed on the (filtered} data. The canonical examples of a Reduce operation would be a Sum or a Count, but there are other possibilities.

Reduce is usually defined as an operation:

Reduce(function(accumulator, data item) -> new accumulator value; initial value; data).

Sometimes, you may see the Reduce operation defined recursively:

Reduce(function: f, initial_value, data = {first_item, all_other_items}) is equal to
Reduce (function: f, new_value = f(initial_value, first_item), data: {all_other_items})

If you follow that script, you can just rinse-and-repeat until you've processed all of the items.

Why is this some kind of technological advance?

If you look at this characterization of Map and Reduce, you'll see that these operations are fairly abstract. The declarations typically state only what needs to be done, and the implementation steps that specify how it is to be done are left open.

For operations on data items that are fairly independent of each other, there are advantages in defining things in this way. If there are no dependencies between data items, in the sense of the two rules listed below, you can use distributed processing across several "servers" to get to the result for the entire data collection much faster.

Basic ground rules for the simplest case (Exceptions and additional constraints will apply in real projects):

1. Computations for each data item do not depend on those for other data items, so no communication, coordination or shared memory is needed between "worker" machines.

2. The order of the computations does not matter.

Under these conditions, Map and Reduce operations could be outsourced from a MapReduce server installation to a fleet of "worker" computers that can take on pieces of the overall computation, and send their results back to the Aggregation Server (or "Boss" machine). That could give you a tremendous speed-up over the alternative of running on a single computing cluster. So there can be speed advantages that come from MapReduce.

With the right infrastructure, you can relax these constraints and still get many of the same benefits on data that needs to be ordered or preprocessed into some kind of table structure.

Another advantage of the Boss/Workers paradigm for MapReduce operations, which may be less obvious, is fault tolerance. Computers sometimes fail to complete their assigned tasks. Network connections can be lost. In a Boss/Workers setup, a Worker could send a status report back to the Boss machine (or a Supervisor, since even the Boss role can be shared) that either contains a SUCCESS status flag and the results of its assignment, or a FAILED flag.

If a Boss receives a FAILED message, that piece of the overall computation could be re-assigned to other Worker(s). In the case of a network outage, the Boss could respond to a Timeout event for the Worker, flush that assignment to that Worker, and re-assign the unfinished task to other resources with a new unique ID. Any homework that is turned in after the Timeout event can then be ignored.

Note: This is just one way to build in parallelism and fault tolerance.

An additional advantage to this sort of vague definition of MapReduce tasks is the ability to work with distributed data in a way that allows greater use of local processing. A central server (Hub) processing model forces remote sites to transmit all the original data to the Hub, wait for the Hub to do the processing, and then possibly transfer the processed results back from the Hub to the remote data repository. That's a lot of network traffic, any part of which could be lost, corrupted or even intercepted by third parties.

In a Reduce operation, where everything is boiled down to some [set of] summary measures, the local site could do much of the processing work, and transmit only the needed intermediate results to the Boss back at the Hub for inclusion in the final totals over all Worker machines.

Summary: Leaving the implementation details out of the MapReduce specification allows for flexibility and some degree of optimization in getting these operations done in the most beneficial way.

- You can optimize to save time, even if that means spending more on hardware and communications.
- You can design to save money (local processing, servers that are easier to replace, etc.).

Whatever your objectives, you can adjust your implementation to get the best result for your application.

Enter Hadoop.

Hadoop is an open source project from the Apache Foundation that lets you set up massively parallel distributed processing schemes for computations that can be fit into the MapReduce paradigm. The best part is that you can make Hadoop work on varying types of hardware, so you don't need to run the pieces of computational work solely on high-end, expensive supercomputers or complex computing cluster installations.

Hadoop makes it possible to farm out the bits of computational "homework" to "commodity hardware" – whatever that may mean for your installation. Commodity hardware is also an abstract term. In practice, you can match the level of computing power for Workers to meet the requirements of the assigned work. The worker machines could be set up on computers that are easy to provision and replace, so you won't have to buy special-purpose servers that require extended periods for setup and configuration.

MapReduce does NOT refer to the process of splitting up a large data processing job into assignments. The concepts behind MapReduce help us to think about and plan classes of processing tasks that are frequently applied to large datasets, or to a lot of data streams that are coming in from many sources and locations.

So far, it sounds like MapReduce and Hadoop are a kind of silver bullet that can eliminate the time and expense required to solve “Big Data” problems. As helpful as these ideas and their supporting technologies may be, not every potential MapReduce job can be optimized as much as we might like. Hadoop will not offer a cure-all for every problem.

We still have to understand the problem, determine what is needed, and work hard to do the right thing.

But when there is a good fit between the problem and this approach toward providing a solution, Hadoop and MapReduce can be very helpful.