L23: MapReduce

Revolution in large-scale programming with a simple (appearing) system.

Big Data. MapReduce is a system for very large data with certain properties. Think logs of all users, or the web.

- The data is enormous. Several terabytes. It does not fit on one machine, it may require 10 or 100 or more machines.
- The data is static, in fact it would take a long time just to read the data (let along write it). When it is updated, it is typically only appended. Even rarer is data is deleted.

This type of data is synonymous with the *data science* revolution. Instead of the scientific method where one

(1) proposes a hypothesis, (2) gathers data, (3) checks whether the data supports the hypothesis,

in data science one

(1) searches an existing corpus of large data for structure.

This structure is the hypothesis and its validation. In stead of *trial and error*, it is data mining to the rescue.

Here we address the most common system infrastructure, MapReduce, to support such process on truly large datasets.

23.1 Distributed File System

Before understanding MapReduce, it is critical to understand the infrastructure that *it* is built on. Specifically a distributed key-value store file system. In MapReduce, and inside Google, this is called the Google File System (or GFS). In the open sources version Hadoop it is called the Hadoop File System (HDFS). Although these ideas were not completely new, they are likely to outlast MapReduce itself.

Item level. The data is typically stored in *key-value pairs*. This is a very limiting, but still general data format. Each data element is given a key, and the an associated set of values. Examples:

- key = log id value = actual log
- key = web address value = html or out-going links
- key = document id in set value = list of words
- key = word in corpus value = how often it appears

They key is a unique (or to some resolution) identifier of the object, and then the value can store anything. Sometimes there will be an assumed format to the value.

Block level. One could then image just filling up tons of hard drives with key-value pairs. However, this is unwieldy. Rather, the items are stored in *blocks* of medium size. A typically HDFS block size is 64 MB (but it can be set differently).

Each block is has a *replication factor*, a typically value is 3. This means it is stored 3 times.

So each file (terabytes) is broken down into blocks (of size 64 MB) and then replicated some number (say 3) times. Then each such block is stored on separate compute node. The replicas are always stored on separate nodes, and especially neighboring blocks are tried to be stored on separate nodes.

This might seem strange: the principal of "locality" says that similar data should be stored nearby! So why do this?

- Resiliency: If one node dies, then the other versions can be used in place (and re-replicate the 3rd version.
- Redundancy: If one node is slow, then another with same data can be used in place.
- Heterogeneity: The format is still very flexible. Usually 64MB is enough for some degree of locality.

23.2 How does MapReduce work

Now that data is stored in key-value pairs in some distributed file system, we are going to kid of ignore this. Just think of what you want to do to each item, that is each key-value pair.

1: Mapping: Typically, the initial format of key-value pairs is very rough. The data is just stored, but not in any particular order. It may contain many irrelevant parts of information. So the *mapping* phase is in charge of getting it to where it into the right format.

The mapping phase takes a file, and converts it into a set of key-value pairs. The values generally contain the data, and the keys are used to obtain locality in the next part.

2: Shuffling: Output of Map is typically as large as original data, so cannot all fit on one machine. The shuffle step puts all key-value pairs with same key on one machine. Like a shuffle of cards.

It is sometimes called *sort* since this can be done by sorting all keys, and putting the first set of ones that fit on the same machine. Sometimes it is useful to have nearby keys on the same machine (in addition to ones with same value).

3: Reducing: The reduce step takes the data that has been aggregates it by keys and does something useful. This data is now all in the same location.

1.5: Combine (optional): They key-value pair way of organizing data can be verbose. If from same block many values have same key, they can be aggregated in before the Shuffle step.

23.2.1 Implementation Notes

This system is designed to be extremely scalable (much of this is done behind the scenes), and to be extremely easy to use. The user only needs to write the *Mapping* and the *Reduce* step. The system takes care of the rest. And these code snippets are often extremely short.

Often the map and reduce phase are *streaming* in that they only make one pass over the data. The Map phase just scans the files once, and the reduce phase takes the key-value pairs in a stream (not necessarily in sorted key order) and produces an output. However, this assumes it has enough storage to maintain something for each key.

One of the powers of MapReduce it inherits from the Distributed File System is that it is resilient. There is one master node that keeps track of everything (by occasionally pinging nodes). If one node goes down, then the master node finds all other nodes with same blocks and asks them to do the work the down node was supposed to do. Again, this is all behind the scenes to the user.

23.2.2 Rounds

Sometimes it will be useful to use the output of one round of MapReduce as the input to another round. This is often considered acceptable if it uses a constant number of rounds, $\log n$ may be too many. (If $n = 1{,}000{,}000{,}000$, then $\log_2 n \approx 30$ may be too much).

In Hadoop, there may be a several minute delay between rounds (due to Java VM issues). This may be dramatically less in other related systems, such as Google's internet MapReduce.

In general, the algorithms takes as long as the last reducer. Sometimes many things map to the same key, and then this takes much longer. This effect is known as the *curse of the last reducer*.

23.3 Example

We'll go through several classic MapReduce examples:

23.3.1 Word Count

Consider as input an enormous text corpus (for instance all of English Wikipedia) stored in out DFS. The goal is to count how many times each word is used.

Map: For each word: w make key-value pair $\langle w, 1\rangle$

Reduce: For all words w have

$$\langle w, v_1\rangle, \langle w, v_2\rangle, \ldots$$

output

$$\langle w, \sum v_i\rangle.$$

The combiner can optionally perform the same thing as the reducer, but just on the words from one block. The reduce can still add the results.

23.3.2 Index

Again consider a text corpus (all pages of English Wikipedia). Build an index, so each word as a list of pages it is in.

Map: For each word: w make key-value pair $\langle w, p\rangle$ where p is the page that is currently being processed.

Reduce: For all words w have

$$\langle w, v_1\rangle, \langle w, v_2\rangle, \ldots$$

output is a set for each word.

$$\langle w, \bigcup v_i\rangle.$$

23.3.3 Phrases

Again consider a text corpus (all pages of English Wikipedia). Build an index, but now on shingles of length 4, and only those that occur on exactly one page.

Map: For each 4-shingle s (consecutive set of 4 words), make key-value pair $\langle w, p\rangle$ where p is the page that is currently being processed.

Reduce: For all shingles s have

$$\langle s, v_1\rangle, \langle s, v_2\rangle, \ldots \langle s, v_k\rangle$$

output could be empty, or is a shingle-page pair. Specifically, if $k = 1$, then output

$$\langle s, v_1\rangle,$$

otherwise output nothing.