24 MapReduce and Distributed File Systems

MapReduce is the system synonymous with a revolution in large-scale programming with simple (appearing) system. Other systems include Hadoop, Spark, and many others more specialized ones.

Big Data. MapReduce is a system for very large data with certain properties. Think logs of all uses, or the web. There are two key properties:

- The data is enormous. At least several terabytes ... could be much much more. It does not fit on one single machine, it may require 100 or 1000 or more machines.
- The data is (fairly) static, in fact it would take a long time just to read the data (let alone write it). When it is updated it is typically only appended. Even rarer is data deleted.

This type of data is also synonymous with the *data science* revolution (although that often occurs at smaller scales). Instead of the scientific method where one

$$(1) \text{ proposes a hypothesis } \rightarrow (2) \text{ gathers data } \rightarrow (3) \text{ checks whether the data supports the hypothesis,}$$

in data science one

$$(1) \text{ searches an existing corpus of large data for structure.}$$

This "structure" is the hypothesis and its validation. Instead 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.

24.1 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, it is called the Google File System (or GFS). In the open source version Hadoop, it is called the Hadoop File System (HDFS). Although these ideas were not completely new, they are outlasting MapReduce itself.

Item level. The data is typically stored in *key-value pairs*. This is very limiting (from a structure/DB point of view), but is yet a very general data format. Each data element is given a key, and 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 or list of pages on which it appears.

The 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. Sometimes there are many key-value pairs that have the same key; this allows the information associated with they key to be stored in multiple places, and updated without touching the original data.

Note, in high contrast to a traditional database system, we have eschewed locality. As we will see, we will "bring the computation to the data," to make up for this.

Block level. One could then imagine 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 can be set differently).

Each block has a replication factor, a typical value is 3 (although 2 is not unheard of). 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 a 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! (e.g. sorting, search trees). So why do this?

- Resiliency: If one node dies, then the other versions can be used in place (and re-replicated the 3rd version).
- Redundancy: If one node is slow, then another with same data can be used in place.
- Heterogeneity: The format is flexible. Usually 64 MB is enough for some degree of locality. Also, by forcing nodes to carry assorted data, the computation on each node will tend to look more uniform (but still not always).

24.2 How does MapReduce Work

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

1: Mapping: Typically, the initial format of the key-value pair is very rough. The data is just stored, but not processed much yet, and in any particular order. It may contain many irrelevant parts of information (for the current task). So the *mapping* phase is in charge of getting it into the right format. Keep in mind, this data set is very large, so it is stored once, but may be used for many many different purposes.

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

2: Shuffling: Output of Map is typically (roughly) as large as the original data, so it also cannot all fit on one machine. The shuffle step puts all key-value pairs with the same key on one machine (if they can all fit). The data is redistributed and mixed up, kind of what it would look like in the mapping of card positions when shuffling a deck of cards...

It is sometimes called *sort* since this can be done by sorting all keys, and putting them in sorted order on another set of machines (e.g. smallest in order on first machine, next smallest on next machine, and so forth – the actually implementation is more robust than this in several ways, but conceptually this is useful). Sometimes it is useful to have nearby keys on the same machine (in addition to ones with the same values).

3: Reducing: The reduce step takes the data that has been aggregated by keys and does something useful (application specific). This data is now all in the same location – we have locality.

1.5: Combine (optional): The key-value pair way of organizing data can be verbose. If from the same block many values have the same key, then they will all be sent to the same Reduce node. So the Combine step aggregates them into a single key-value pair before the Shuffle step.

This can make streaming on the Mapper more tricky.

24.2.1 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 that 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 the 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.

24.2.2 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, and since it writes all data from disk, and then re-reads it in the next round – for resiliency issues). New systems (most noteably *Spark*) have been introduced recently to specifically address this issue. They get around this rewrite-to-disk-each-round by being able to recover the content of a crashed machine by reconstructing it from the memory of all active machines (this uses some complicated bookkeeping schemes, that were quite non-trivial to get correct.)

Last reducer. In general, the algorithms take as long as the last reducer takes to finish; the next round does not start until all reducers finish. 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*.

In some cases, MapReduce implementations can overcome this by seeing if a reducer is slow (just since the machine semi-crashed) and splitting its load to already finished machines. But if one key has a huge fraction of the data, then that part cannot be split (e.g. see "the" in word count). Often this requires *intelligent algorithmic design*.

24.3 Examples

Lets go through several classic MapReduce examples:

24.3.1 Word Count

Consider as input an enormous text corpus (for instance all of English Wikipedia) stored in our 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

$$\langle w, v_1\rangle, \langle w, v_2\rangle, \langle w, v_3\rangle$$

output

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

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. This helps enormously for instance when on many corpuses of data the word "the" occurs 7% of the time (which is a huge reducer if there are 100 or more machines). And text already follows Zipf's law and is heavy-tailed. Light-tailed distributions could be even worse.

24.3.2 Inverted Index

Again consider a text corpus (all pages of English Wikipedia). Build an index, so each word has a list of pages it is in. In a search engine, most queries are preprocessed this way – only for very rare/obscure queries would they compute something on the fly.

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

$$\langle w, p_1\rangle, \langle w, p_2\rangle, \langle w, p_3\rangle$$

output

$$\langle w, \bigcup_i p_i\rangle.$$

We could modify the reduce operation to sort all of these pages $\bigcup_i p_i$ (say based on a score related to pagerank) so that the most important ones are first.

24.3.3 Phrases

Again consider a text corpus (all pages of English Wikipedia). Build an index, but now on k-grams of length 3, and only those that occur on exactly one page.

Map: For each 3-gram g (consecutive set of 3 words), make key-value pair $\langle g, p\rangle$ where p is the page that is currently being processed.

Combine: For all 3-grams g have

$$\langle g, p\rangle, \langle g, p\rangle, \ldots$$

output (so only one copy)

$$\langle g, p\rangle.$$

Reduce: For all shingle g have

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

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

$$\langle w, \bigcup_i v_i\rangle = \langle w, w_1\rangle,$$

otherwise output nothing since it does not occur (by default) or occurs on more than one page.

24.4 PageRank on MapReduce

The Internet is stored as a big matrix M (size $n \times n$). Specifically the column-normalized adjacency matrix where each column represents a webpage and where it links to are the non-zero entries.

M is sparse. Even as n grows, each webpage probably on average only links to about 20 other webpages. Over 99.99% of entries in M are $M[a, b] = 0$.

We define another matrix

$$P = \beta M + (1 - \beta)B$$

where $B[a, b] = 1/n$ and typically $\beta = 0.85$ or so.

Recall that we want to calculate the PageRank vector using Markov Chain theory. The PageRank of a page a is $q_*[a]$ where $q_* = P^t q$ as $t \to \infty$. Here we can usually use $t = 50$ or 75. This describes the *importance* of a webpage from a random surfer's perspective.

Problems: The matrix M is sparse and can be stored (on many machines at Google). Its size is roughly $20n$, where n is on the order of 1 billion. But the matrix P is dense (since B is dense). So of course P^t is also dense. Since n is about 1 billion, it is too big to store.

But q_i is only size n, and we need to store this. So we can just iterate as $q_{i+1} = Pq_i$ and then we only need to store P implicitly as M and B.

$$q_{i+1} = \beta M q_i + (1 - \beta)1/n$$

And repeat this step t times.

Still, n is very big, so we cannot store M or q_i on any one machine. This could be terabytes of data. And when working with many machines, some will crash!
MapReduce to the rescue.

24.4.1 PageRank on MapReduce : v1

Here is a first attempt. Break M into k vertical stripes $M = [M_1\ M_2\ \ldots\ M_k]$ so each M_j fits on a machine. Break q into $q^T = [q_1\ q_2\ \ldots\ q_k]$ (a horizontal split), again so each q_j fits on a machine with M_j. (This can be assumed how the data is stored, or can be done in a earlier round of MapReduce if not.)

Now in each round:

- Mapper: $j \to$ (key= $j' \in [k]$; value = row r of $M_j * q_j$)
- Reducer: adds values for each key i to get $q_{i+1}[j] * \beta + (1 - \beta)/n$.

Note that the output of each mapper is an entire vector q_{i+1} (or at least part needs to be added together with other components to obtain q_{i+1}), of length n. This follows since each stripe M_j has n/k full columns. Is this feasible?
... Yes, since q_{i+1} only has as many non-zero entries as M_j.

However, we are not getting that much out of the combiner phase. We will see next how this can be improved.

24.4.2 PageRank on MapReduce : v2

Let $\ell = \sqrt{k}$ and *tile* M into $\ell \times \ell$ blocks

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} & \ldots & M_{1,\ell} \\ M_{2,1} & M_{2,2} & \ldots & M_{2,\ell} \\ \ldots & \ldots & \ldots & \ldots \\ M_{\ell,1} & M_{\ell,2} & \ldots & M_{\ell,\ell} \end{bmatrix}$$

- Mapper: Each of k machines gets one block $M_{i,j}$ and gets sent q_i for $i \in [\ell]$.
- Reducer: On each row i' adds $M_{i,j}q_i$ to $q[i']$. Then does $q_+[i'] = q[i']\beta + (1 - \beta)/n$.

slight problems still... Each q_i (for $i \in [\ell]$) is stored in $\ell = \sqrt{k}$ places.

Thrashing on $M_{i,j}$. It may fit on disk, but not in memory. Solution: blocking on $M_{i,j}$ so it fits in disk, but its sub-blocks $\{B_s\}_s$ fit in memory. Now only need to read each B_s once, but read/write q and q_+ for each block (this takes up less space). But this is becoming less of a problem as MapReduce type machines have more and more memory.

24.4.3 Example

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

Stripes:

$$M_1 = \begin{bmatrix} 0 \\ 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} \quad M_2 = \begin{bmatrix} 1/2 \\ 0 \\ 0 \\ 1/2 \end{bmatrix} \quad M_3 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad M_4 = \begin{bmatrix} 0 \\ 1/2 \\ 1/2 \\ 0 \end{bmatrix}$$

These are stored as $\big(1 : (1/3, 2), (1/3, 3), (1/3, 4)\big)$, $\big(2 : (1/2, 1)(1/2, 4)\big)$, $\big(3 : (1, 3)\big)$, and $\big(4 : (1/3, 1), (1/2, 2)\big)$.

Blocks:

$$M_{1,1} = \begin{bmatrix} 0 & 1/2 \\ 1/3 & 0 \end{bmatrix} \quad M_{1,2} = \begin{bmatrix} 0 & 0 \\ 1 & 1/2 \end{bmatrix} \quad M_{2,1} = \begin{bmatrix} 1/3 & 0 \\ 1/3 & 1/2 \end{bmatrix} \quad M_{2,2} = \begin{bmatrix} 0 & 1/2 \\ 0 & 0 \end{bmatrix}$$

These are stored as $\big(1 : (1/2, 2)\big)$, $\big(2 : (1/3, 1)\big)$, as $\big(2 : (1, 3), (1/2, 4)\big)$, as $\big(3 : (1/3, 1)\big)$, $\big(4 : (1/3, 1), (1/2, 2)\big)$, and as $\big(3 : (1/2, 4)\big)$.

Note that some blocks have no effect on the some vector elements they are responsible for. $M_{2,2}$ has no effect on $q_+[3]$ and $M_{1,2}$ has no use for $q[3]$. Both effects are quite common and can be used to speed things up.