24 PageRank on MapReduce

PageRank and MapReduce go well together, and this synergy was important for the growth of Google, and proliferation of the MapReduce paradigm (and related ideas).

MapReduce. Recall that MapReduce works roughly as follows.

We start with some big data set D that has been partitioned into blocks $D_1, D_2, \ldots, D_m$. These are distributed among different machines, and we can think of them as being each on one machine; that is, let D_i be on machine i. These blocks are also replicated, but (part of the beauty of MapReduce is that) we can ignore this.

The core constraint to keep in mind is that each machine has limited memory compared to D (but can easily store any one D_i).

We now proceed in rounds, each with 3 steps.

1. **Mapper:** Convert all $d \in D$ to $(\mathsf{key}(d), \mathsf{value}(d))$
2. **Shuffle:** Moves all (k, v) and $(\mathsf{k}', \mathsf{v}')$ with $\mathsf{k} = \mathsf{k}'$ to same machine.
3. **Reducer:** Transforms $\{(\mathsf{k}, \mathsf{v}_1), (\mathsf{k}, \mathsf{v}_2), \ldots\}$ to an output $D'_k = f(\mathsf{v}_1, \mathsf{v}_2, \ldots)$.

... and don't forget ...

1.5. **Combiner:** If one machine has multiple $(\mathsf{k}, \mathsf{v}_1), (\mathsf{k}, \mathsf{v}_2)$ with same k then it can perform part of Reduce before Shuffle.

We can think of the output of the reducer as D_i on machine i, and then pump this into another round of MapReduce. For many simple analytics task (the non-PageRank biggest use), this is unnecessary, but for certain more involved problems, we may want to iterate a constant number of times.

A programmer only needs to specify the Mapper and Reducer, the rest is taken care of behind the scenes; this includes the hard parts: Shuffle, any node going down, replication, ...

Example: Histogram
Mapper: $d \in D$ converts to $(k = \text{bin}(d), 1)$
(combiner)
Reducer: $\{(k = i, v)\}$ converts to $\text{bin}_i = \sum v$.

24.1 PageRank

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.

24.1.1 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: $j \rightarrow$ (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.

24.1.2 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.

24.1.3 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.