5.2 Efficient Computation of PageRank

To compute the PageRank for a large graph representing the Web, we have to perform a matrix–vector multiplication on the order of 50 times, until the vector is close to unchanged at one iteration. To a first approximation, the map-reduce method given in Section 2.3.1 is suitable. However, we must deal with two issues:

1. The transition matrix of the Web M is very sparse. Thus, representing it by all its elements is highly inefficient. Rather, we want to represent the matrix by its nonzero elements.

2. We may not be using map-reduce, or for efficiency reasons we may wish to use a combiner (see Section 2.2.4) with the Map tasks to reduce the amount of data that must be passed from Map tasks to Reduce tasks. In this case, the striping approach discussed in Section 2.3.1 is not sufficient to avoid heavy use of disk (thrashing).

We discuss the solution to these two problems in this section.

5.2.1 Representing Transition Matrices

The transition matrix is very sparse, since the average Web page has about 10 out-links. If, say, we are analyzing a graph of ten billion pages, then only one in a billion entries is not 0. The proper way to represent any sparse matrix is to list the locations of the nonzero entries and their values. If we use 4-byte integers for coordinates of an element and an 8-byte double-precision number for the value, then we need 16 bytes per nonzero entry. That is, the space needed is linear in the number of nonzero entries, rather than quadratic in the side of the matrix.

However, for a transition matrix of the Web, there is one further compression that we can do. If we list the nonzero entries by column, then we know what each nonzero entry is; it is 1 divided by the out-degree of the page. We can thus represent a column by one integer for the out-degree, and one integer per nonzero entry in that column, giving the row number where that entry is located. Thus, we need slightly more than 4 bytes per nonzero entry to represent a transition matrix.

Example 5.7: Let us reprise the example Web graph from Fig. 5.1, whose transition matrix is

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

Recall that the rows and columns represent nodes A, B, C, and D, in that order. In Fig. 5.11 is a compact representation of this matrix.[5]

Source	Degree	Destinations
A	3	B, C, D
B	2	A, D
C	1	A
D	2	B, C

Figure 5.11: Represent a transition matrix by the out-degree of each node and the list of its successors

For instance, the entry for A has degree 3 and a list of three successors. From that row of Fig. 5.11 we can deduce that the column for A in matrix M has 0 in the row for A (since it is not on the list of destinations) and 1/3 in the rows for B, C, and D. We know that the value is 1/3 because the degree column in Fig. 5.11 tells us there are three links out of A. □

[5]Because M is not sparse, this representation is not very useful for M. However, the example illustrates the process of representing matrices in general, and the sparser the matrix is, the more this representation will save.

5.2.2 PageRank Iteration Using Map-Reduce

One iteration of the PageRank algorithm involves taking an estimated PageRank vector $\mathbf{v}$ and computing the next estimate $\mathbf{v}'$ by

$$\mathbf{v}' = \beta M\mathbf{v} + (1-\beta)\mathbf{e}/n$$

Recall β is a constant slightly less than 1, $\mathbf{e}$ is a vector of all 1's, and n is the number of nodes in the graph that transition matrix M represents.

If n is small enough that each Map task can store the full vector $\mathbf{v}$ in main memory and also have room in main memory for the result vector $\mathbf{v}'$, then there is little more here than a matrix–vector multiplication. The additional steps are to multiply each component of $M\mathbf{v}$ by constant β and to add $(1-\beta)/n$ to each component.

However, it is likely, given the size of the Web today, that $\mathbf{v}$ is much too large to fit in main memory. As we discussed in Section 2.3.1, the method of striping, where we break M into vertical stripes (see Fig. 2.4) and break $\mathbf{v}$ into corresponding horizontal stripes, will allow us to execute the map-reduce process efficiently, with no more of $\mathbf{v}$ at any one Map task than can conveniently fit in main memory.

5.2.3 Use of Combiners to Consolidate the Result Vector

There are two reasons the method of Section 5.2.2 might not be adequate.

1. We might wish to add terms for $\mathbf{v}'_i$, the ith component of the result vector $\mathbf{v}$, at the Map tasks. This improvement is the same as using a combiner, since the Reduce function simply adds terms with a common key. Recall that for a map-reduce implementation of matrix–vector multiplication, the key is the value of i for which a term $m_{ij}\mathbf{v}_j$ is intended.

2. We might not be using map-reduce at all, but rather executing the iteration step at a single machine or a collection of machines.

We shall assume that we are trying to implement a combiner in conjunction with a Map task; the second case uses essentially the same idea.

Suppose that we are using the stripe method to partition a matrix and vector that do not fit in main memory. Then a vertical stripe from the matrix M and a horizontal stripe from the vector $\mathbf{v}$ will contribute to all components of the result vector $\mathbf{v}'$. Since that vector is the same length as $\mathbf{v}$, it will not fit in main memory either. Moreover, as M is stored column-by-column for efficiency reasons, a column can affect any of the components of $\mathbf{v}'$. As a result, it is unlikely that when we need to add a term to some component $\mathbf{v}'_i$, that component will already be in main memory. Thus, most terms will require that a page be brought into main memory to add it to the proper component. That situation, called *thrashing*, takes orders of magnitude too much time to be feasible.

An alternative strategy is based on partitioning the matrix into k^2 blocks, while the vectors are still partitioned into k stripes. A picture, showing the division for $k = 4$, is in Fig. 5.12. Note that we have not shown the multiplication of the matrix by β or the addition of $(1 - \beta)\mathbf{e}/n$, because these steps are straightforward, regardless of the strategy we use.

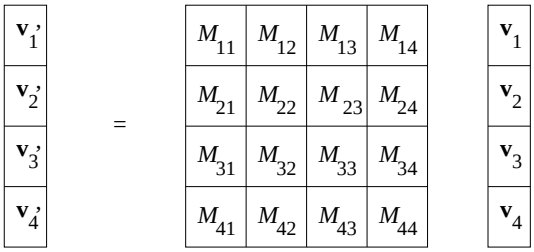

Figure 5.12: Partitioning a matrix into square blocks

In this method, we use k^2 Map tasks. Each task gets one square of the matrix M, say M_{ij}, and one stripe of the vector $\mathbf{v}$, which must be $\mathbf{v}_j$. Notice that each stripe of the vector is sent to k different Map tasks; $\mathbf{v}_j$ is sent to the task handling M_{ij} for each of the k possible values of i. Thus, $\mathbf{v}$ is transmitted over the network k times. However, each piece of the matrix is sent only once. Since the size of the matrix, properly encoded as described in Section 5.2.1, can be expected to be several times the size of the vector, the transmission cost is not too much greater than the minimum possible. And because we are doing considerable combining at the Map tasks, we save as data is passed from the Map tasks to the Reduce tasks.

The advantage of this approach is that we can keep both the jth stripe of $\mathbf{v}$ and the ith stripe of $\mathbf{v}'$ in main memory as we process M_{ij}. Note that all terms generated from M_{ij} and $\mathbf{v}_j$ contribute to $\mathbf{v}'_i$ and no other stripe of $\mathbf{v}'$.

5.2.4 Representing Blocks of the Transition Matrix

Since we are representing transition matrices in the special way described in Section 5.2.1, we need to consider how the blocks of Fig. 5.12 are represented. Unfortunately, the space required for a column of blocks (a "stripe" as we called it earlier) is greater than the space needed for the stripe as a whole, but not too much greater.

For each block, we need data about all those columns that have at least one nonzero entry within the block. If k, the number of stripes in each dimension, is large, then most columns will have nothing in most blocks of its stripe. For a given block, we not only have to list those rows that have a nonzero entry for that column, but we must repeat the out-degree for the node represented by the column. Consequently, it is possible that the out-degree will be repeated as many times as the out-degree itself. That observation bounds from above the

space needed to store the blocks of a stripe at twice the space needed to store the stripe as a whole.

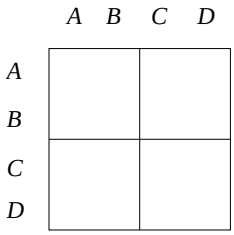

Figure 5.13: A four-node graph is divided into four 2-by-2 blocks

Example 5.8: Let us suppose the matrix from Example 5.7 is partitioned into blocks, with $k = 2$. That is, the upper-left quadrant represents links from A or B to A or B, the upper-right quadrant represents links from C or D to A or B, and so on. It turns out that in this small example, the only entry that we can avoid is the entry for C in M_{22}, because C has no arcs to either C or D. The tables representing each of the four blocks are shown in Fig. 5.14.

If we examine Fig. 5.14(a), we see the representation of the upper-left quadrant. Notice that the degrees for A and B are the same as in Fig. 5.11, because we need to know the entire number of successors, not the number of successors within the relevant block. However, each successor of A or B is represented in Fig. 5.14(a) or Fig. 5.14(c), but not both. Notice also that in Fig. 5.14(d), there is no entry for C, because there are no successors of C within the lower half of the matrix (rows C and D). □

5.2.5 Other Efficient Approaches to PageRank Iteration

The algorithm discussed in Section 5.2.3 is not the only option. We shall discuss several other approaches that use fewer processors. These algorithms share with the algorithm of Section 5.2.3 the good property that the matrix M is read only once, although the vector $\mathbf{v}$ is read k times, where the parameter k is chosen so that $1/k$th of the vectors $\mathbf{v}$ and $\mathbf{v}'$ can be held in main memory. Recall that the algorithm of Section 5.2.3 uses k^2 processors, assuming all Map tasks are executed in parallel at different processors.

We can assign all the blocks in one row of blocks to a single Map task, and thus reduce the number of Map tasks to k. For instance, in Fig. 5.12, M_{11}, M_{12}, M_{13}, and M_{14} would be assigned to a single Map task. If we represent the blocks as in Fig. 5.14, we can read the blocks in a row of blocks one-at-a-time, so the matrix does not consume a significant amount of main-memory. At the same time that we read M_{ij}, we must read the vector stripe $\mathbf{v}_j$. As a result, each of the k Map tasks reads the entire vector $\mathbf{v}$, along with $1/k$th of the matrix.

Source	Degree	Destinations
A	3	B
B	2	A

(a) Representation of M_{11} connecting A and B to A and B

Source	Degree	Destinations
C	1	A
D	2	B

(b) Representation of M_{12} connecting C and D to A and B

Source	Degree	Destinations
A	3	C, D
B	2	D

(c) Representation of M_{21} connecting A and B to C and D

Source	Degree	Destinations
D	2	C

(d) Representation of M_{22} connecting C and D to C and D

Figure 5.14: Sparse representation of the blocks of a matrix

The work reading M and $\mathbf{v}$ is thus the same as for the algorithm of Section 5.2.3, but the advantage of this approach is that each Map task can combine all the terms for the portion $\mathbf{v}'_i$ for which it is exclusively responsible. In other words, the Reduce tasks have nothing to do but to concatenate the pieces of $\mathbf{v}'$ received from the k Map tasks.

We can extend this idea to an environment in which map-reduce is not used. Suppose we have a single processor, with M and $\mathbf{v}$ stored on its disk, using the same sparse representation for M that we have discussed. We can first simulate the first Map task, the one that uses blocks M_{11} through M_{1k} and all of $\mathbf{v}$ to compute $\mathbf{v}'_1$. Then we simulate the second Map task, reading M_{21} through M_{2k} and all of $\mathbf{v}$ to compute $\mathbf{v}'_2$, and so on. As for the previous algorithms, we thus read M once and $\mathbf{v}$ k times. We can make k as small as possible, subject to the constraint that there is enough main memory to store $1/k$th of $\mathbf{v}$ and $1/k$th of $\mathbf{v}'$, along with as small a portion of M as we can read from disk (typically, one disk block).

5.2.6 Exercises for Section 5.2

Exercise 5.2.1: Suppose we wish to store an $n \times n$ boolean matrix (0 and 1 elements only). We could represent it by the bits themselves, or we could represent the matrix by listing the positions of the 1's as pairs of integers, each integer requiring $\lceil \log_2 n \rceil$ bits. The former is suitable for dense matrices; the latter is suitable for sparse matrices. How sparse must the matrix be (i.e., what fraction of the elements should be 1's) for the sparse representation to save space?

Exercise 5.2.2: Using the method of Section 5.2.1, represent the transition matrices of the following graphs:

(a) Figure 5.4.

(b) Figure 5.7.

Exercise 5.2.3: Using the method of Section 5.2.4, represent the transition matrices of the graph of Fig. 5.3, assuming blocks have side 2.

Exercise 5.2.4: Consider a Web graph that is a chain, like Fig. 5.9, with n nodes. As a function of k, which you may assume divides n, describe the representation of the transition matrix for this graph, using the method of Section 5.2.4