

The Web Laboratory

PageRank Calculation using Map Reduce

Spring 2008 Report

Asif-ul Haque (asif@cs.cornell.edu)
Vijayanand Chokkapu (vc223@cornell.edu)

Cornell University
Advisor: Professor William Arms

Table of Contents

Abstract.....	3
Introduction.....	4
Description.....	5
Experiments & Results	12
Key Insights	15
Conclusions & Future Work.....	16
Acknowledgements.....	17
References.....	18

Abstract

The main goal of this project was to implement the well-known Pagerank algorithm using Map-Reduce programming. We have come up with several formulations of Pagerank using Map-Reduce programming. While the formulations appear different, at the heart of the Pagerank computation is computing the product of a sparse matrix and a vector. The sparse matrix is stored as a list of tuples. Map-Reduce programming is very suitable for doing aggregation operations like summing the product bits and hence the applicability of Map-Reduce programming to compute Pagerank of large graphs.

We implemented our different formulations on the Hadoop distributed system with six nodes. However we report the results of only one of the implementations. Hadoop is an open source system and is being increasingly used by different companies as their production system. We used a small synthetic dataset generated according to "the referential attachment" model of random graphs to match the degree distribution of many real world networks. We tested different parameter settings of the Hadoop system on the small dataset to get a better understanding of how parameters like number of Map tasks, number of Reduce tasks, choice of an intermediate Combiner, etc affects the performance of our implementation of Pagerank.

Introduction

The purpose of the Web Laboratory is to provide data and computing tools for research about the Web and the information on the Web. “The Cornell Web Lab is funded in part by National Science Foundation grants CNS-0403340 SES-0537606, IIS-0634677, and IIS-0705774”. One of the team’s that is working towards achieving this goal is the Pagerank team. The PageRank team works on the web graph represented in the form of adjacency lists generated by the indexing team to compute PageRank[2] of the webpages. This document describes the several methods that were formulated to compute PageRank.

Prior work on this project was done by Kim, Chen[1] using parallel computing. However this semester, it was decided to do this computation based on a different approach called Map Reduce[4] running on a Hadoop[3] cluster. Hadoop is a framework for running applications on large clusters built of commodity hardware. The Hadoop framework transparently provides applications both reliability and data motion. It implements a computational paradigm named Map/Reduce, where the application is divided into many small fragments of work, each of which may be executed or reexecuted on any node in the cluster. In addition, it provides a distributed file system (HDFS) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster. Both Map/Reduce and the distributed file system are designed so that node failures are automatically handled by the framework.

Description

We have come up with three different approaches for computing the Pagerank. The first approach basically iterates using the normalized link matrix represented in the form of adjacency lists.

Approach 1

Notation

U is a set of URLs defining the nodes of a graph.

u, v, w are elements of U

An edge of the graph is defined by a pair (u, v) , where u is a from-URL and v is a to-URL.

Transitions

A transition probability is defined by a triple, (u, v, p) , where p is the conditional probability, $p(v|u)$, that, with no damping, a random surfer will jump from node u to node v given that the surfer is at node u .

A state probability is defined by a pair, (w, q^i) , where q^i is the probability, $q^i(w)$, that at iteration i the random surfer is at node w . In the limit over many iterations, the values of q^i converge to the PageRank.

Iterations

$q^{i+1}(w) = \sum q^i(u) * p(w|u)$ where the summation is over all u for which $p(w|u)$ is not zero.

Note: Rather than work with actual probabilities, which can be very small, it is convenient at iteration zero to give every q the value 1, i.e., q is n times the probability, where n is the total number of nodes. The sum of the PageRanks calculated by this method equals n . An advantage of this approach is that it is not necessary to know the value of n .

Each iteration requires two MapReduce steps.

Step 1

The purpose of Step 1 is to associate with each node, represented by a (w, q^i) pair, the non-zero transitions (u, v, p) from that node and calculate the transition probabilities.

Map

Input:

(u, v, p) for each link. [This input is the same for each iteration.]
 (w, q^i) for each node.

Output (keys are underlined):

(\underline{u}, v, p)
 (\underline{w}, q^i)

Reduce

Input:

$(\underline{w}, q^i, \{v, p\})$ for each node, i.e., the probability of being at the node and the non-zero transitions from that node.

Output:

$(\underline{w}, \{v, q^i * p\})$ for each node, w , calculate the probability of a transition to v .

Step 2

The purpose of Step 2 is to calculate the probability of being at node v after the transition by summing all the possible transitions that end at that node.

Map

Input:

$(\underline{w}, \{v, q^i * p\})$

Output:

$(\underline{v}, q^{i * p})$ // each input creates several outputs

Reduce

Input:

$(\underline{v}, \{q^{i * p}\})$ // the sorting brings together all the transitions to v .

Output:

(\underline{v}, q^{i+1}) where $q^{i+1} = \sum q^{i * p}$

Pagerank Formulation for Map-Reduce Programming

Mathematical reformulation of Pagerank

Suppose M is the stochastic transition matrix of the directed graph $G = (V, E)$. So M is an $n \times n$ matrix with $n = |V|$ and $\sum_j M_{ij} = 1$ for all i . Suppose $A = M^T$. Let $\mathbf{1}_n$ be the vector of all one's. If $0 < \epsilon < 1$ and \mathbf{p} is the vector of pagerank values then the Pagerank formulae is written as

$$\mathbf{p} = \epsilon A \mathbf{p} + \frac{1 - \epsilon}{n} \mathbf{1}_n$$

If we define $\mathbf{q} = \frac{n}{1 - \epsilon} \mathbf{p}$ then the formulae becomes

$$\mathbf{q} = \epsilon A \mathbf{q} + \mathbf{1}_n$$

Since \mathbf{q} is proportional to the pagerank values \mathbf{p} , they can be used wherever pagerank is used except that $\|\mathbf{q}\|_1 = \frac{n}{1 - \epsilon}$ whereas $\|\mathbf{p}\|_1 = 1$. This formulae can be rewritten as

$$\mathbf{q} = (I_n - \epsilon A)^{-1} \mathbf{1}_n$$

For any vector $\|\mathbf{x}\|_2 = 1$

$$\underbrace{\mathbf{x}^T A \mathbf{x}}_{\text{scalar}} = (\mathbf{x}^T A \mathbf{x})^T = \mathbf{x}^T A^T \mathbf{x} = \mathbf{x}^T M \mathbf{x} \leq 1$$

So $\mathbf{x}^T \epsilon A \mathbf{x} < 1$ implying $I_n - \epsilon A$ is positive definite and thus $(I_n - \epsilon A)^{-1}$ exists. Expanding the inverse into an infinite series we get

$$\mathbf{q} = \sum_{k=0}^{\infty} \epsilon^k A^k \mathbf{1}_n$$

If we want δ precision in the values of \mathbf{q} then we need to sum up the first N terms where $\epsilon^N \leq \delta$. In other words $N \geq \frac{\log(1/\delta)}{\log(1/\epsilon)}$. So $N = \lceil \frac{\log(1/\delta)}{\log(1/\epsilon)} \rceil$ suffices. So the formulae becomes

$$\mathbf{q} = \sum_{k=0}^N \epsilon^k A^k$$

Thus the pagerank algorithm can be succinctly written as

$$\mathbf{q} = \mathbf{1}_n, \mathbf{r} = \mathbf{1}_n$$

For $k = 1$ to N do

$$\mathbf{r} = \epsilon A \mathbf{r}$$

$$\mathbf{q} = \mathbf{q} + \mathbf{r}$$

Map-Reduce implementation

The sparse matrix M can be represented by tuples $\langle u, v, p_{uv} \rangle$ where $p_{uv} = \frac{1}{\text{outdegree}(u)}$. And if we maintain tuples $\langle u, r_u, q_u \rangle$ for every vertex u of the graph with r_u and q_u initially being 1, every iteration of the algorithm readily translates to two Map-Reduce programs.

The core of the iteration is a sparse matrix-vector product and that is essentially summations of some products. The first Map-Reduce program computes the products while the second Map-Reduce program sums them up. Both the Maps are identities i.e. emitting the same values as it is taking as input. But in a real implementation the maps do a “join” between the two kinds of tuples. The Map-Reduce programs are listed below. The underlined variables are the keys.

1. • Map $\langle \underline{u}, (v, p_{uv}) \rangle, \langle \underline{u}, (r_u, q_u) \rangle \Rightarrow \langle \underline{u}, (v, p_{uv}) \rangle, \langle \underline{u}, (r_u, q_u) \rangle$
- Red $\langle \underline{u}, \{(r_u, q_u), \{(v, p_{uv})\}\} \rangle \Rightarrow \langle \underline{u}, \epsilon r_u p_{uv} \rangle$
2. • Map $\langle \underline{v}, \epsilon r_u p_{uv} \rangle, \langle \underline{v}, (r_v, q_v) \rangle \Rightarrow \langle \underline{v}, \epsilon r_u p_{uv} \rangle, \langle \underline{v}, (r_v, q_v) \rangle$
- Red $\langle \underline{v}, \{(r_v, q_v), \{\epsilon r_u p_{uv}\}\} \rangle \Rightarrow \langle \underline{v}, (\underbrace{\sum_u \epsilon r_u p_{uv}}_{r_v}, q_v + \underbrace{\sum_u \epsilon r_u p_{uv}}_{r_v}) \rangle$

Improved implementation

In the above implementation most of the “real” work is being done by the two Reduce programs whereas the two Maps are repeatedly joining the two kinds of tuples. So the obvious question is whether we can join the two tuples only once initially and use both Map and Reduce for computation so that there is just one Map-Reduce task instead of two. The motivation to reduce the number of Map-Reduce tasks is that on a real system Map-Reduce programs take time to setup.

Instead of keeping the two kinds of tuples we mentioned earlier, we keep tuples of the form $\langle u, v, r_u, \epsilon r_u p_{uv} \rangle$. We will still need tuples of the form $\langle u, q_u \rangle$. But joining these tuples is very easy as is shown in the following Map-Reduce program.

- Map $\langle \underline{u}, (v, r_u, \epsilon r_u p_{uv}) \rangle, \langle \underline{v}, q_v \rangle \Rightarrow \langle \underline{v}, q_v \rangle, \langle \underline{v}, \epsilon r_u p_{uv} \rangle, \langle \underline{v}, (w, r_v, \epsilon r_v p_{vw}) \rangle$
- Red $\langle \underline{v}, \{q_v, \epsilon r_u p_{uv}, (w, r_v, \epsilon r_v p_{vw})\} \rangle \Rightarrow \langle \underline{v}, q_v + \sum_u \epsilon r_u p_{uv} \rangle, \langle \underline{v}, (w, \sum_u \epsilon r_u p_{uv}, \frac{\epsilon r_v p_{vw}}{r_v} \sum_u \epsilon r_u p_{uv}) \rangle$

The first tuple on the left hand side in the Map program is output exactly as the last tuple on the right hand side except that the variables u and v are renamed as v and w . The middle tuple on the right hand side is also obtained from the first tuple on the left hand side. The Reduce program sums up the values and adds the sum to the current sum q as well as updates the tuples $\langle u, v, r_u, \epsilon r_u p_{uv} \rangle$ by appropriately dividing by the old value of r_u (r_v in the Reduce program) that is being carried around and multiplying by the new value of r_u (r_v in the Reduce program) just computed. The old value is set to the new value as well.

The important idea here is that the Map program cleverly splits tuples so that the Reduce program can sum things up as well multiply things to make it ready for summing up in the next iteration. Thus instead of multiplying and then summing up in two steps, we sum up and multiply

at the same step. The tuples are also maintained so that the output from the Map-Reduce program can be fed back to itself and the pagerank iterations, as we formulated it, can be run.

Now the initial values of the tuples of the form $\langle u, v, r_u, \epsilon r_u p_{uv} \rangle$ should be $\langle u, v, 1, \epsilon p_{uv} \rangle$ and for the tuples of the form $\langle v, q_v \rangle$ should be $\langle v, 1 \rangle$. One interesting outcome of this improved Map-Reduced implementation is that the constant ϵ does not appear anywhere in the iteration! It appears only during initialization of the tuples.

Approach 3

The third approach is basically implements the page rank formula from a functional perspective. Page Rank can be viewed as distribution of a Page's value of PageRank equally to all of it's out link nodes in each iteration.

Notation

Let U be a set of URLs, defining the nodes of a graph.
 u, v, w are elements of U .
 L is the set of links, $\{(u, v)\}$, with duplicates removed.

For a given u ,

F_u is the set of v , s.t. $(u, v) \in L$.
 $|u|$ is the number of elements of F_u .

For a given v ,

T_v is the set of u , s.t. $(u, v) \in L$.

Iteration formula

Associated with each $w \in U$, there is a number $p(w)$, known as the PageRank of w .
The Page/Brin formula:

$$PR(A) = (1-d) + d (PR(T1)/C(T1) + \dots + PR(Tn)/C(Tn))$$

can be written as the iteration:

$$\begin{aligned} p^0(w) &= 1 \\ p^{i+1}(w) &= (1-d) + d \sum p^i(u)/|u| \end{aligned} \quad (1)$$

where the summation is over all $u \in T_w$.

MapReduce calculation of (1)

Map

Input

$(u, \{p^i(u), |u|, F_u\})$

Output

$(u, \{0, \{|u|, F_u\}\})$

and for each $v \in F_u$,

$(v, \{(p(u)/|u|), \{\}\})$

Reduce

Input

$(w, \{(0, \{|w|, F_w\})\}, \{(p(u)/|u|)\})$

where there is a separate value of $p(u)/|u|$ for each $u \in T_w$.

Output

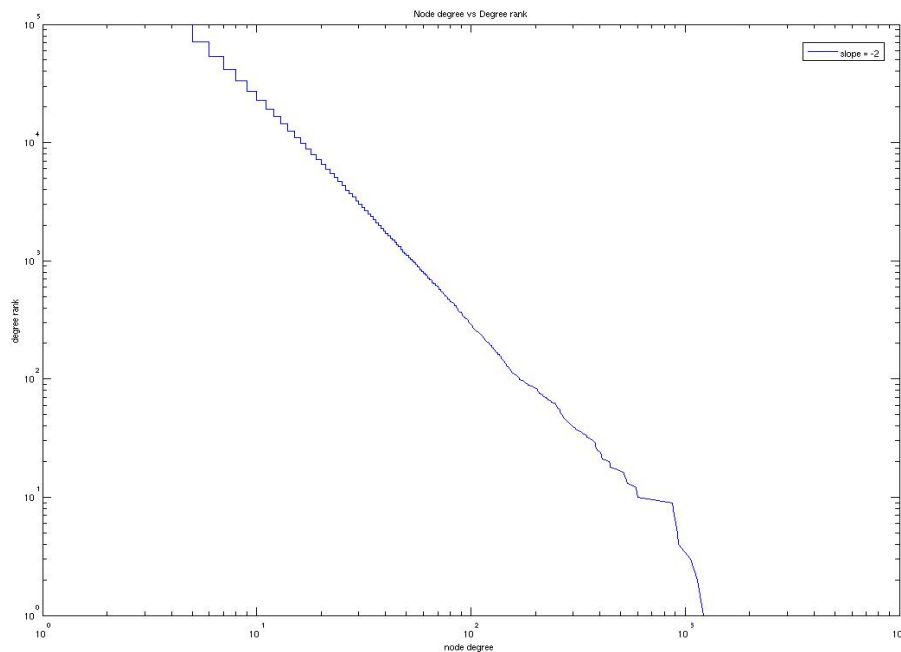
$(w, \{p^{i+1}(w), |w|, F_w\})$

Experiments & Results

The Map-Reduce implementation based on the third was tested on a synthetic dataset. The data was generated using the preferential attachment model or Barabasi-Albert (BA) model. Ideally we would have liked to test the implementation on real data such as the world wide web (or a subset). Such data would have tested the scalability of the Map-Reduce implementation. However it was not possible due to a number of reasons. So we generated some data that forms a scale-free network.

The preferential attachment model was one of the earliest models used to explain the citation networks and the World Wide Web. The degree distribution generated by the model has a power law form with an exponent of -3 i.e. $P(k) \sim k^{-3}$.

We use the model to generate an undirected graph and then interpret each undirected edge as two directed edges. For our data set we started with a small set of initial nodes (about 10), all of them connected to each other through undirected edges. After that we add 100,000 nodes sequentially. When we add a node to the graph it gets linked to a small (constant) number of nodes already present in the graph. In our data we link each new node to 5 existing nodes. Thus for 100,000 vertices we have a million directed edges. For every node added to the graph we choose random nodes to link it to. The nodes are chosen with probability proportional to their current degree. Thus nodes with higher degree are more likely to be connected to the new node (thus the name preferential attachment or rich-getting-richer model). Here is the Zipf plot (Rank-Frequency plot) of node degrees of the graph on a log-log scale. This is the cumulative degree distribution of the graph. Clearly the cumulative distribution follows a power law with exponent -2. So the degree distribution follows a power law with exponent -3.



Test Process

The experiments were conducted on synthetic data which consisted of 100,000 nodes having 1,000,000 outlinks. The aim of the experiments was to gain insights on the ratio of Map and Reduce tasks which would minimize the total running time. We also wanted to see how using a Combiner affects the overall running time of the computation.

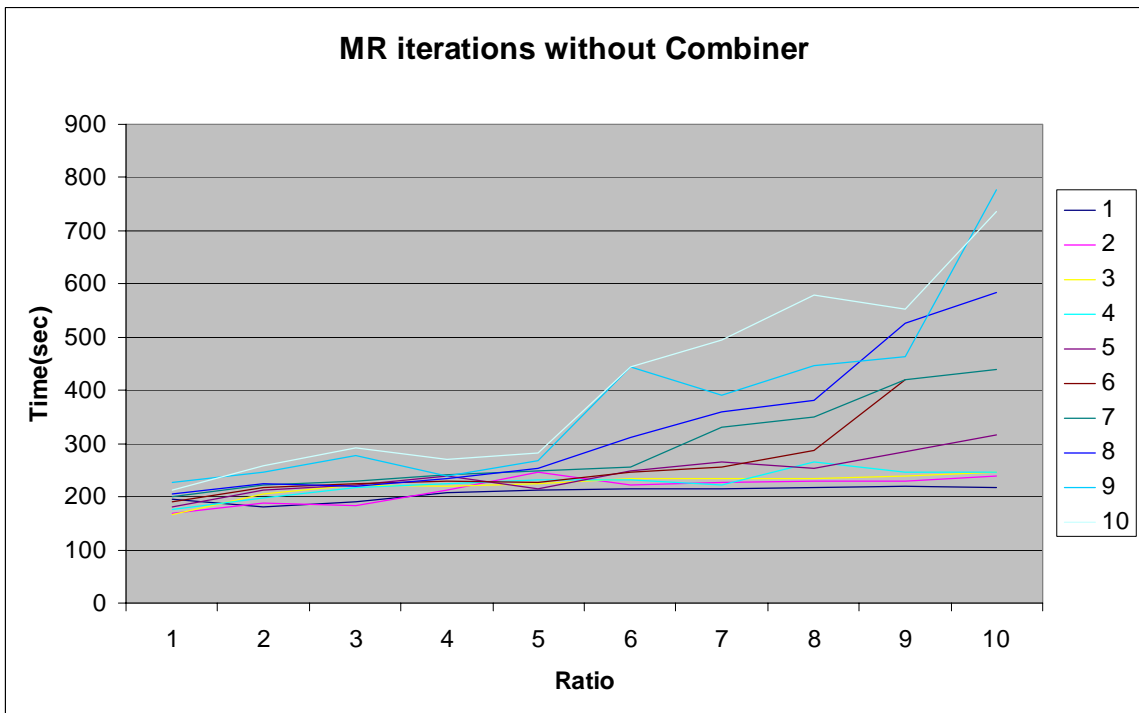
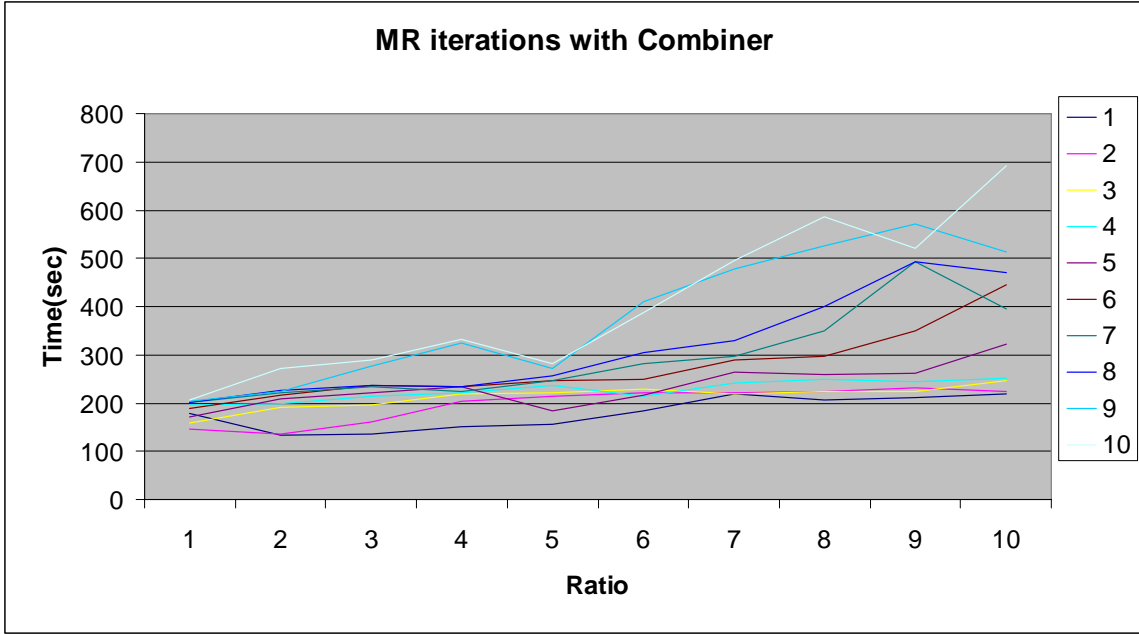
The first test consisted of runs with 10 iterations for each Map/Reduce tasks having variable number of Map and Reduce tasks with the Map tasks ranging from 1 to 20 and the Reduce tasks ranging from 1 to 20 with increments of 1 Map and Reduce per run.

One checking the logs we found that often from the 2nd iteration onwards, Hadoop used to reset the number of Map tasks to a ratio of the Reduce tasks and was not using the numbers that we had passed as parameters. For example if we start off with the run with the number of mappers as 20 and the number of reducers as 6, Hadoop automatically sets the number of mappers to 24. In case of 20 mappers and 7 reducers, it sets mappers to 21 from 2nd run onwards and similarly for 8 it sets it to 24. However in case of 20 Mappers and 10 Reducers there would be no change.

Basically the number of Map Reduce tasks parameters we send to Hadoop act only as a hint to the framework and it can change the number of Map and Reduce tasks. The test results indicate that it was setting it to the next multiple of the number of reducers greater than the number of map tasks.

Since we were not getting clean results in this experiment due to the numbers of mappers not necessarily being the ones we are specified, we conducted a second test with the number of mappers set in multiples of the number of reducers.

The second test consisted of runs having 10 iterations for each Map/Reduce tasks having ratio's from 1 to 10 with the number of Reduce tasks varying from 1 till 15. In addition each run was also conducted with and without Combiner. The followings graphs show the time (sec) took for each run with the Ratio of Map and Reduce tasks varying from 1 to 10 both with and without Combiner.



Key Insights

1) Map Reduce tasks ratios between 4 and 5 are giving good performance (anything less than 5 is OK). However the moment we are increasing the ratio greater than 5 we are getting a large drop in performance. This can be explained by the fact that each Map Reduce invocation involves an overhead and if there are too many Maps for each Reduce, the overhead of having an additional Map task becomes more than the time saved due to an additional Map task.

2) The combiner is making a difference in although it not very significant due to the small size of present data set. We expect greater reduction in overall run duration with bigger size data sets having over Terabyte of data.

3) As the number of reduce tasks are increasing (with Mappers increasing in proportion) , the effect of combiner is getting diminished and in some cases it worse than without combiner. This could be because with more number of mappers, the number of data splits would increase thereby reducing the chances of effective combining. Also the overhead of invoking a combiner would increase with more number of Mappers per Reducer thereby increasing total duration of the run.

Conclusions & Future Work

Although we had come up with many different approaches for computing the Pagerank, our experimentation was largely limited due to lack of actual data. In particular we believe the combiner can make a huge difference in the time required to compute the Pagerank. However this requires testing on a huge collection of real data which we hope will be done by the team that continues this project.

The Pagerank code has been written in a way to allow the user to experiment with different values of damping factor, number of iterations by passing them as parameters. We hope that this would be useful in calibrating the parameters in production environment according to the requirement.

The user also has the option of getting the output in the form of PageID's and the Pagerank values. These could be stored in Hadoop's database Hbase which is an open-source, distributed; column-oriented store modeled after Google's BigTable and can store billions of rows X millions of columns on top of clusters of commodity hardware.

Acknowledgements

We would like to thank our advisor Professor William Arms, for his guidance. We would also like to thank Lucia Walle from the Cornell Theory Center for her support in setting up the hadoop cluster and helping us conduct experiments.

This work is funded in part by National Science Foundation grants CNS-0403340, DUE-0127308, SES-0537606, and IIS 0634677.

References

- [1] C. Kim & T. Chen, PageRank Calculation using Sparse Matrix in Clustered Computer Environment, <http://www.infosci.cornell.edu/SIN/WebLab/papers/Kim2007c.pdf>
- [2] S. Brin, & L. Page, The Anatomy of a Large-Scale Hypertextual Web Search Engine, Stanford, USA, 1998
- [3] Hadoop, <http://hadoop.apache.org/>
- [4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in Usenix SDI, 2004.