# HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop

U Kang, Charalampos Tsourakakis, Ana Paula Appel,
Christos Faloutsos, Jure Leskovec

December 2008
CMU-ML-08-117


Carnegie Mellon

# HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop

**U Kang**[*]**,Charalampos Tsourakakis**[*]**, Ana Paula Appel**[††]
**Christos Faloutsos**[*]**, Jure Leskovec** [†]

Dec 2008
CMU-ML-08-117

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[†] Computer Science Department, Cornell/Stanford University
[††] ICMC, Univ. of São Paulo, Brazil

**Keywords:** diameter, graph, hadoop

**Abstract**

How can we quickly find the diameter of a petabyte-sized graph? Large graphs are ubiquitous: social networks (Facebook, LinkedIn, etc.), the World Wide Web, biological networks, computer networks and many more. The size of graphs of interest has been increasing rapidly in recent years and with it also the need for algorithms that can handle tera- and peta-byte graphs. A promising direction for coping with such sizes is the emerging map/reduce architecture and its open-source implementation, 'HADOOP'. Estimating the diameter of a graph, as well as the radius of each node, is a valuable operation that can help us spot outliers and anomalies. We propose HADI (HAdoop based DIameter estimator), a carefully designed algorithm to compute the diameters of petabyte-scale graphs. We run the algorithm to analyze the largest public web graph ever analyzed, with *billions* of nodes and edges. Additional contributions include the following: (a) We propose several performance optimizations (b) we achieve excellent scale-up, and (c) we report interesting observations including outliers and related patterns, on this real graph (116Gb), as well as several other real, smaller graphs. One of the observations is that the Albert et al. conjecture about the diameter of the web is over-pessimistic.

# 1 Introduction

Networked systems are ubiquitous. The analysis of networks such as the World Wide Web, social, computer and biological networks has attracted much attention recently. Some of the typical measures to compute are the (in- and out-) degree distribution(s), the clustering coefficient, the PageRank, the spectrum of the graph, and several more [15].

In this paper we focus on estimating the diameter of massive graphs. Diameter is one of the most interesting metrics of a network. Formally, a graph has diameter $d$ if every pair of nodes can be connected by a path of length at most $d$. Typically, $d$ is surprisingly small in real-world networks, also known as the "small-world" phenomenon ([20],[21],[13]). The maximum diameter is susceptible to outliers; this led to the definition of the *effective diameter* ([17]), which is defined as the minimum number of hops in which 90% of all connected pairs of nodes can reach each other. The diameter is important in both designing algorithms for graphs and understanding the nature and evolution of graphs. For example, real-world graphs including the web graph and social network graphs have small diameters (see [4] [5] [6]); moreover, in most real graphs, their diameter *shrinks* as the graph grows [14]. Unfortunately, existing algorithms do not scale up -at least directly- to tera- and peta-byte sized graphs.

In this paper, we show how to compute the diameter in a scalable, highly optimized way using HADOOP, the open source implementation of MAPREDUCE. We chose HADOOP because it is freely available, and because of the the power and convenience that it provides to the programmer. We run our algorithm, HADI, on the largest public web graph[1] ever analyzed, with several billion edges where we achieve excellent scale up. Finally, we show how HADI can be used for graph mining purposes: for example, thanks to HADI, we are able to dis-prove the conjecture of Albert et. al. [5] about the diameter of the web.

The rest of the paper is organized as follows. Sections 2 and 3 describe the HADI algorithm and the optimizations that allow excellent scale-up. Section 4 illustrates how HADI works through an example. In Section 5 and 6 we present the experimental result and analysis and in Section 7 we briefly present the related work. We conclude in Section 8.

# 2 HADI algorithm

In the next two sections we describe HADI. In order to reach the optimized version of HADI, we describe first two simpler versions namely HADI-naive and HADI-plain. Table 1 lists the symbols in this paper.

## 2.1 HADI Overview

The "skeleton" of HADI is shown in Algorithm 1. Given an edge file with `(source, destination)` pairs, HADI iteratively calculates neighborhood function $N(h)$ ($h$ = 1, 2, ...) until convergence, decided by a threshold parameter $\epsilon$. Recall that $N(h)$ is the count of pairs (a,b) such that 'a' reaches 'b' within $h$ hops. Using $N(h)$, we can compute the effective and average diameter (see Section 7). Line (1) is used to make the bitmask generation commands file (BC), which are needed to calculate $N(h)$ (Section 2.3).

The most important operation in HADI is line (2), in which we compute $N(h)$. In order to do so, HADI maintains $k$ Flajolet-Martin(FM)[9] bitmasks for each node. In each iteration, the bitmasks of a node $v$ are updated so that they contain the number of nodes reachable from $v$ within distance $h$. HADI works with two files: the input, edge file $E$ and a bitmask file B which is generated. Notice that HADI is eventually executing a database join, repeatedly. Let's think of the edge file $E$ as a table in a relational database whose

[1] Provided by Yahoo!.

| Symbol | Definition |
|---|---|
| $G$ | a graph |
| $n$ | number of nodes in a graph |
| $m$ | number of edges in a graph |
| $d$ | diameter of a graph |
| $B$ | input bitmask to HADI |
| $R$ | edge relation of the input graph, pairs of nodes $(u, v) \in G$ |
| $R'$ | reflexive closure of $R^{-1}$ |
| $h$ | number of hops |
| $N(h)$ | number of node-pairs reachable in $\leq h$ hops (neighborhood function) |
| $N(h, i)$ | number of neighbors of node $i$ reachable in $\leq h$ hops |
| $\mathcal{N}(h, i)$ | set of nodes reachable in $\leq h$ hops from node $i$ |
| $b(h, i)$ | Flajolet-Martin bitmask for node $i$ at $h$ hops |
| $\hat{b}(h, i)$ | Partial Flajolet-Martin bitmask for node $i$ at $h$ hops |

Table 1: Table of symbols

**input** : Edge file $E = \{(k_i, k_j)\}$
**output**: Diameter $d$,
Neighborhood $N(h)$ where $1 \leq h \leq d$
**begin**
**for** *h starting with 1 until MaxIteration* **do**
**if** *h == 1* **then**
Make Bitmask Generation Commands; // (1)
Calculate N(h); // (2)
**if** *h ¿ 1 AND* $N(h) < N(h-1)(1+\epsilon)$ **then**
$d \leftarrow h - 1$;
break for loop;
**end**
Print("Diameter=", d);

**Algorithm 1**: HADI algorithm: the big picture

attributes are source node id (sid) and destination node id (did). The list of bitmasks $B$ also corresponds to a table whose attributes are the node id (id) and the current bitmask (b). Let $b(h, i)$ be the bitmask of node $i$ after $h$ hops. Then the next bitmask $b(h + 1, i)$ of $i$ at the hop $h + 1$ is given as: $b(h + 1, i)$ = $b(h, i)$ BITWISE-OR $\{b(h, k) \mid E.sid = i$ AND $E.did = k\}$.

If SQL had operator for `BITWISE-OR`, and if we augment the edge file $E$ with self referencing edges for all nodes, then the bitmask update operation would be done by:

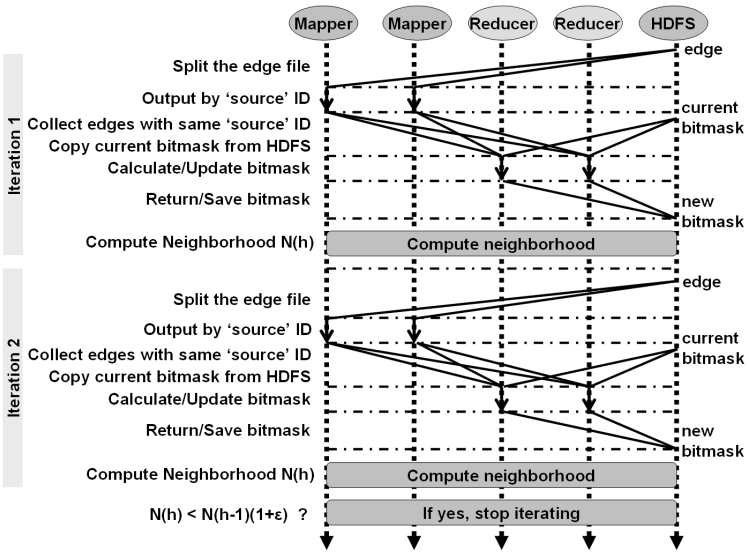



Figure 1: HADI-naive: Naive implementation of HADI.

SELECT B.id, BITWISE-OR(B.b)
FROM E, B
WHERE E.did=B.id
GROUP BY E.sid

In short, we need a join (one for each value of $h$), and aggregation. Thus, to compute the diameter, the question becomes how to efficiently execute these operations in the MapReduce framework. This is what HADI-naive and HADI-plain answer primarily, whereas HADI-optimized shows how to accelerate execution, using compression and related mechanisms.

## 2.2 HADI-naive

The main idea of HADI-naive is illustrated in the swim-lane diagram of Figure 1. Initially, the master node assigns pieces of the input edge file from HDFS, to the mappers. Each mapper gets a part of the graph, and it outputs ('source' node id, 'destination' node id) as (key, value) pairs. Then each reducer copies all the current FM (Flajolet-Martin) bitmasks (bitstrings) to the local file system. Upon obtaining the edges that share the same source node, each reducer updates the FM bitmask for that node using line 4-7 of Algorithm 4 and saves it to HDFS.

Then, by another map-reduce job, the current neighborhood $N(h)$ is calculated from the current FM bitmask using formula (7.1). The process repeats, until the ratio of $N(h)$ and $N(h-1)$ is below a threshold.

HADI-naive leaves room for optimization: each reducer copies all the $n$ bitmasks to its local file system, although it does not necessarily need them all. ($n$ is the number of nodes in the graph.)

## 2.3 HADI-plain

HADI-plain improves on HADI-naive, by copying only the necessary bitmasks to each reducer, through a carefully designed MAPREDUCE algorithm. For example, let's say we calculate the diameter of the graph

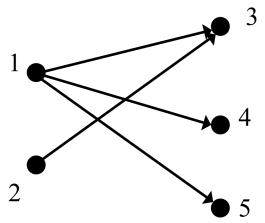



Figure 2: Example graph

in Figure 2 using two reducers A and B. Also assume that reducer A updates the bitmask of node 1, and reducer B updates the bitmask of node 2.

Reducer A does not need the bitmask of node 2, and reducer B does not need the bitmasks of nodes 1, 4 and 5. Thus, our goal is to only ship the necessary bitmasks. Next, we formalize this problem.

**Problem 1** (Join-Aggregating in MapReduce)**.** *Let the input data D =* $\{ (k_i, v_i) \mid k_i \in K, v_i \in V,$ *K is a set of key and V is a set of value.*$\}$*. Let the elements in D have dependency relation R =* $\{ (k_i, k_j) \mid (k_i, v_i)$ *depends on* $(k_j, v_j) \}$*. The problem of join-aggregating in MapReduce is to generate* $D^+ = \{(k_i, AGGR(v_j)) \mid k_i \in K, v_j \in V,$ *i=j or* $(k_i, k_j) \in R \}$ *given an aggregating function AGGR().*

In terms of relational database, Problem 1 is to find a result of
`SELECT R.`$k_i$`, AGGR(D.v) FROM R, D WHERE R.`$k_j$`=D.k GROUP BY R.`$k_i$
For subtle reasons, we need $R'$: the inverse relation $R^{-1}$, augmented with self-edges to all nodes. We define $R'$ as the *reflexive closure* of $R^{-1}$, as follows:

**Definition 1** (Reflexive closure of $R^{-1}$)**.** *For the relation R=*$\{(k_i, k_j)\}$*, the reflexive closure R' of* $R^{-1}$*(inverse of R) is given by R' =* $\{ (k_j, k_i) \mid (k_i, k_j) \in$ *R or i=j.* $\}$

Now we propose our method of solving problem 1.

**Lemma 1** (Solution of problem 1)**.** *Use a mapreduce stage to group relations* $(k_j, k_i) \in R'$ *and data* $(k_j, v_j) \in D$ *using key* $k_j$*. For each group, output edges(*$k_i$*,* $v_j$*) for all* $k_i$*. Use another mapreduce stage to group (*$k_i$*,* $v_j$*) using key* $k_i$*, and output (*$k_i, AGGR(\cup v_j)$*) in the reducer. Then the final output is* $D^+$*.*

*Proof.* Let the set of (key, value) pairs at the second reducer of Lemma 1 be $D^*$, and the set of (key, value) pairs which is used right before applying AGGR() in $D^+$ be $D'$. We prove by showing a bijection between the same elements in $D^*$ and $D'$. First, we show an injection from $D^*$ to $D'$. Elements $(k_i, v_j) \in D^*$ can belong to one of the two disjoint subset : $D^1*$, which satisfies i=j, and $D^2* = D^* \setminus D^1*$. For every data $d1 \in D^1*$, there exist only one corresponding data $d1' \in D'$ because i=j. In addition, for every data $d2 = (k_i, v_j) \in D^2*$, there exist only one corresponding data $d2' \in D'$ because there is a corresponding relation $(k_i, k_j) \in R$. Same argument applies when we start from $D'$ to $D^*$. Therefore, there is a one-to-one correspondence between $D^*$ and $D'$. □

In HADI-plain, the input data is the bitmask B = $\{ (k_i, v_i) \mid k_i$ is the node number, $v_i$ is the bitmask of the node $\}$, and edge file R = $\{ (k_i, k_j) \mid$ there is an edge from node $k_i$ to node $k_j \}$. HADI-plain uses Lemma 1 to implement line (2) in Algorithm 1 by three-stage MAPREDUCE functions which are shown in Algorithm 2 and Algorithm 3.

```
//Stage 1: Invert edge, match bitmasks to node id
input : Edge data R = {(k_i, k_j)},
        Current bitmask B = {(k_i, bm_i)}
//For the first iteration, BC is used in place of B

Stage1-Map ;
begin
    Get key k, value v;
    if (k, v) is of type BC or B then
        Output(k, v);                                              // (3)
    else if (k, v) is of type R then
        k_1 ← v;
        Output(k_1, k);                                            // (4)
end

Stage1-Reduce ;
begin
    Get key k, array of values v[];
    K ← [];
    foreach v ∈ v[] do
        if (k, v) is of type BC then
            b̂(h − 1, k) ← newly generated FM bitmask;
        else if (k, v) is of type B then
            b̂(h − 1, k) ← v;
        else if (k, v) is of type R then
            k_1 ← v;
            Add k_1 to K;
    for k' ∈ K do
        Output(k', b̂(h − 1, k));
    Output(k, b̂(h − 1, k)) if not already outputted;               // (5)
end
```

**Algorithm 2**: HADI `Stage1`.

The first two stages join and aggregate the two files as described in Lemma 1; in the last stage, we calculate $N(h)$. In `Stage1`, we derive R' from R, and joins R.$k_j$ and B.k. Specifically, in `Stage1-Map` the input data has two different types("bitmask" data and "edge" data). If the data is of type "bitmask", the algorithm passes it to the reducer directly ( line (3) ). If the data is of type "edge", the inverse of the relation is saved to make R' from R ( line (4) ). The self referencing edge is not added to R' at `Stage1-Map`, but it is implicitly considered ( line(5) ).

In `Stage1-Reduce`, we join R'.$k_j$ and B.k to create partial FM bitmasks $\hat{b}(h-1, k_i)$ for each node $k_i$.

In `Stage2`, we group data based on key B.k that were spread over all the reducers in `Stage1`, and apply aggregating function BITWISE-OR. As a result, after `Stage2` we get the current bitmasks $b(h, k_i)$ of every node $k_i$. These bitmasks are used as the input B of `Stage1` at the next iteration. `Stage3` uses these current bitmasks to calculate the individual neighborhood for every node, and it sums them up to calculate total number of neighborhood $N(h)$. This $N(h)$ is used to determine when to stop the iteration

```
//Stage 2 : Merge bitmasks for each node
input : Bitmask B = {(k_i, bm_i)}

Stage2-Map ;
begin
    Get key k, value v;
    Output(k, v);
end

Stage2-Reduce ;
begin
    Get key k, array of values v[];
    b(h, k) ← 0;
    foreach v ∈ v[] do
        b(h, k) ← b(h, k) BITWISE-OR v;
    Output(k, b(h, k));
end

//Stage 3: Calculate neighborhood function N(h)
input : Bitmask B = {(k_i, bm_i)}

Stage3-Map ;
begin
    Get key k, value v;
    b(h, k) ← v;
    b ← the leftmost zero bit position in b(h, k);
    N(h, k) ← 2^b/0.77351;
    Output('N', N(h, k));
end

Stage3-Reduce ;
begin
    Get key k, array of values v[];
    s ← 0;
    foreach v ∈ v[] do
        s ← s + v;
    Print("N at h = ", s);
end
```

**Algorithm 3**: HADI `Stage2` and `Stage3`.

process. In this process, one natural question arises : what is used as the input B in the first iteration? In the first iteration, initial FM bitmask should be assigned to each node. Since MapReduce operation is centered around input data, we make a bitmask generation commands file BC (Bitmask Creation) that instructs hadoop to generate FM bitmask (line (1) in Algorithm 1). This BC is used in place of the input data B only in the first iteration.

### 2.4 Time and Space Analysis

We analyze the algorithm complexity of HADI for a graph G with $n$ nodes and $m$ edges. Note that this analysis applies to both HADI-plain and HADI-optimized. Let $M$ be the number of machines in the MapReduce or Hadoop cluster. We are interested in the time complexity, as wells space complexity.

**Lemma 2** (Time Complexity of HADI)**.** *HADI takes $O(d(n+m)/M)$ time.*

*Proof.* In all stages, HADI has O($|input\ size|$) running time. In `Stage1`-Map, the running time is $O((n+m)/M)$ since each machine gets $(n+m)/M$ lines of input. In `Stage1`-Reduce, the running time is $O((n+m)/M)$ since each machine again gets $(n+m)/M$ lines of input assuming edges are uniformly distributed to nodes. In `Stage2`, the running time for map and reduce is $O((n+m)/M)$ since each machine gets $\approx (n+m)/M$ lines of input. In `Stage3`-Map, each machine gets $\approx n/M$ lines of input, and the running time is $O(n/M)$. In `Stage3`-Reduce, the running time is $O(n)$ since the reducer gets $n$ lines of input.

Since one iteration from `Stage1` to `Stage3` requires $O((n+m)/M)$ time, $d$ iterations of HADI require $O(d(n+m)/M)$ time. □

Similarly, for space we have:

**Lemma 3** (Space Complexity of HADI)**.** *HADI requires $O((n+m)\log n)$ space.*

*Proof.* The input of HADI requires $O(n\log n + m)$ space because each node has bitmasks whose size is proportional to $\log n$. In `Stage1`, the intermediate step between map and reduce requires $O(n\log n + m)$ space. The output of `Stage1`-Reduce requires $O((n+m)\log n)$ space. In `Stage2`, the intermediate step between the map and the reduce phase requires $O((n+m)\log n)$ space. The output of `Stage2`-Reduce requires $O(n\log n)$ space since it contains the current bitmasks of all nodes. Recall that the number of bitmasks $k$, is constant. In `Stage3`, the intermediate step between map and reduce requires $O(n)$ space. The output of `Stage3`-Reduce requires $O(1)$ space.

Therefore, the maximum space required is $O((n+m)\log n)$. □

Note that the fastest previously known algorithm for diameter calculation, ANF, has $O(d(n+m))$ time and $O(n\lg n)$ space complexity. The time complexity of HADI is lower than that by a factor of $M$. The space complexity of HADI is higher than that by $m\log n$; however this is not an issue, since HADI is run on distributed machines with large aggregate disk capacity.

## 3 HADI optimizations

In this section, we discuss HADI-optimized which is the algorithm we propose to compute the diameter of massive graphs. HADI-optimized deals with both algorithmic and system issues.

### 3.1 Bit Shuffle Encoding

In HADI, we use $k$ (e.g., 32, 64) hashing functions for each node, to increase estimation accuracy. Since the output of the `Stage1` is proportional to $(n+m) * k * |a\ bitmask|$, we need many disk I/Os when $k$ is large. For example, in the Yahoo web graph of Section 6, we need (1.4G+6.6G)*(32)*(8) = 2 Tera bytes of disk when we use $k$=32 hashing functions, with bitmasks of size 8 bytes each. We propose to decrease the bitmask size by the following bit shuffle encoding method.

The main idea is to carefully reorder the bits of the bitmaps of each node, and then use run length encoding. Specifically, we try to make the reordered bit strings to contains long sequences of 1's and 0's: We get all the first bits from all $k$ (say, 32) bitmasks, get the second bits, and so on. As a result we get a single bit sequence of length $k * |a\ bitmask|$, but most of the first bits are '1's, and most of the last bits are '0's. Then we encode only the length of each bit sequence, achieving good space savings (and, eventually, time savings, through fewer I/Os).

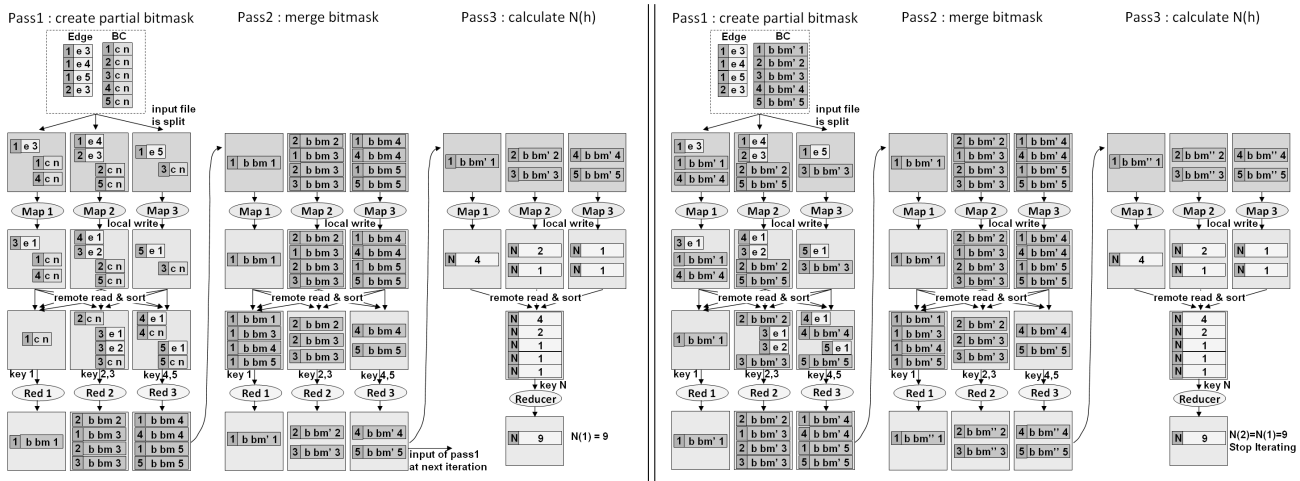
### 3.2 Pruning Stable Nodes

HADI performs 3-Stage map-reduces until the number of neighborhoods converges. In its basic form, the input and output of each map-reduce function have same size for every iteration. However we observed that the bitmask of a node doesn't change once the hop number reached its breadth first search limit - we refer to such a node as a *stable* node. Therefore any nodes that point to stable nodes do not need to consider them in their bitmask update operation. Specifically, we don't output partial bitmasks for stable nodes in `Stage1` reducer. The effect of this optimization is that the running time of each iteration decreases as the current hop number increases.

### 3.3 Checkpointing

Sometimes HADOOP hangs, due to failures in the clusters or in the network, and this would cause us to lose all our work so far. Thus, in HADI-optimized, we carefully implemented a checkpointing function. This checkpointing is possible thanks to the fact that the calculation of the current neighborhood $N(h)$ depends only on the previous neighborhood $N(h-1)$.

## 4 HADI example




(a) **First Iteration of HADI-plain** (b) **Second Iteration of HADI-plain**

Figure 3: Execution of HADI-plain on graph of Figure 2

In this section, we illustrate the HADI-plain algorithm via an example on the toy-graph of Figure 2. The first iteration is shown in Figure 3(a), and the second in Figure 3(b). In this example we assume that three mappers and reducers are used. Also, reducer 1 collects data whose key is 1, reducer 2 collects data whose key is 2 or 3, and reducer 3 collects data whose key is 4 or 5. The notation bmX means the FM bitmask of node X, which is equal to $b(h, X)$. Finally, $(x, y)$ represents an input line where $x$ is the key and y is the value.

**First Iteration** In the first iteration, the input files are the Edge file and BC(Bitmask Creation) file. The Edge file contains all the edges in the input graph and a flag indicating that each line is an edge. Specifically, each line in the edge file contains the key which is the source node id and the value which is composed of the flag('e') and the destination node id. Two other flags used in HADI-plain are the following: 'b' indicates that the input file is a Bitmask file, and 'c' indicates that bitmasks should be created (BC file). Therefore, flag 'c' is seen only in the first iteration of the execution when the BC file is given as input. Each line of the BC file contains the key(node id) and the value('c' flag and $n$ which is the number of total nodes in the input graph).

**Stage1** Before starting `Stage1`, HADOOP distributes the input file to three mappers[2]. If the input line contains edge(flag 'e'), the mapper of `Stage1` outputs the inverted edge. For example, mapper 1 outputs (3, e 1) because it gets (1, e 3) as the input line. If the input line contains the bitmask creation flag 'c', the mapper outputs the same data. Now the reducer starts operating. Reducer 1 gets (1, c n) as input, and outputs the FM bitmask for node 1. Reducer 2 gets (2, c n) as input in one function call, and (3, e 1), (3, e 2), (3, c n) in another function call. When reducer 2 gets (3, e 1), (3, e 2), (3, c n) as input, it first collects all the destination nodes (1 and 2) in the edge line. Also, it creates the FM bitmask for node 3 after seeing the (3, c n) line. After reading all the inputs, reducer 2 associates all the destination nodes it collected with the bitmask of node 3. Also, it associates node 3 with bitmask 3 because node 3 is not in the collected destination nodes list. The effect of this operation is that the bitmask 3 is associated with node 3 and all the nodes that point to node 3. This is what we wanted: each node should be associated with the bitmask of nodes which are linked from it. After the reduce step of `Stage1`, we get all the (nodeid, bitmask) pairs for updating the bitmask of every node. For example, for updating the bitmask of node 1 we require previous bitmask of node 1, 3, 4, and 5. Those bitmasks(1, b bm1), (1, b bm3), (1, b bm4), (1, b bm5) exist in the `Stage1`'s output lines whose key is 1. However, those lines are spread over all the three reducers. We need to group these bitmasks based on key(node 1). That is the job of `Stage2`.

**Stage2** The input of `Stage2` is the output of `Stage1` reducers. In `Stage2`, the output of mappers is same as the input. The bitmasks for each node are collected in the reducer. For example, all the bitmasks that are used to update the bitmask of node 1 are gathered in reducer 1, all the bitmasks that are used to update the bitmask of node 2 and 3 are gathered in reducer 2, and all the bitmasks that are used to update the bitmask of node 4 and 5 are gathered in reducer 3. The reducers of `Stage2` get these bitmasks, and do BITWISE-OR operation to merge them. For example, reducer 1 performs bm1 $\leftarrow$ (bm1 | bm2 | bm3 | bm4) operation to create the new bitmask of node 1. Therefore, reducer of `Stage2` outputs updated bitmasks for each node. These updated bitmasks(presented as bm') are used as the input of `Stage3` mapper in the current iteration and `Stage1` mapper in the next iteration.

**Stage3** In `Stage3`, the input data is the node id and the bitmask of each node. These bitmasks contain all the information to estimate $N(1, i)$, the number of nodes within distance 1 of each node $i$. Each mapper gets the bitmask lines, and outputs the number of nodes within distance 1 from the node in the line. For example, mapper 2 outputs ('N', 2) after it reads (2, b bm'2), because there are two nodes (2,3) from node 2 within 1 hop. Also, mapper 2 outputs ('N', 1) after it reads (3, b bm'3), because there are only one node

[2]This is for illustration purposes. In fact, mappers are assigned to the input data.

from node 3 within 1 hop, which is node 3 itself. Because all the output of mappers in `Stage3` have same key('N'), it is processed in only one reducer. The reducer of `Stage3` sums all the neighborhood values to calculate the total neighborhood $N(1)$=9. Because this is the first iteration, we proceed to the second iteration.

**Second Iteration** The second iteration proceeds in the same way as the first iteration except two differences. First, the Bitmask file is used as input for `Stage1` instead of the Bitmask Creation (BC) file. This Bitmask file comes from the output of `Stage2` reducers in the first iteration. Note that the BC file is used only in the `Stage1` of the first iteration, and all the other iterations use the Bitmask file instead as input for `Stage1` mappers. Second, the `Stage1` reducers don't have to create FM bitmasks, because the bitmasks are generated from the previous iteration. Except these two differences, each stage performs the same operations: `Stage1` creates all the (nodeid, related bitmask) pairs, `Stage2` gathers and merges these pairs based on node id, and `Stage3` computes the neighborhood function $N(h)$. As the diameter of this example graph is 1, the `Stage3` reducer outputs 9 as $N(2)$. Since this $N(2)$ is same as $N(1)$, HADI-plain stops iterating and outputs 1 as the diameter of the graph.

# 5 Performance of HADI

Next, we perform experiments to answer the following questions:

- Q1: How fast is HADI?
- Q2: How does it scale up with the number of nodes?
- Q3: What observations does HADI lead to on real graphs?

We focus on Q1 and Q2 in this section, leaving the observations for the next.

## 5.1 Experimental Setup

We use both real and synthetic graphs for our experiments, with the following details.

- YahooWeb : Real, directed graph showing links between real web pages. The data was indexed by Yahoo! Altavista search engine in 2002. The file portion of the url is masked; we can only see the host and the depth of the url.
- IMDB : Real, undirected bipartite graph from `IMDB.com`, recording which actor played in what movie [2].
- AS-Caida : Real, undirected graph showing the autonomous system (AS) relationships [1].
- Kronecker : Synthetic, undirected Kronecker graphs [12] using a chain of length two as the seed graph.

We chose the Kronecker graph generator because it yields graphs that mirror several real-graph characteristics, including small and constant diameters, power-law degree distributions, e.t.c.

Table 2 shows detailed information on our data set. HADI was run on M45, one of the fifty most powerful supercomputers in the world. M45 has 480 hosts (each with 2 quad-core Intel Xeon 1.86 GHz, running RHEL5), with 3Tb aggregate RAM, and over 1.5 PetaByte aggregate disk capacity. The cluster is running Hadoop on Demand (HOD).

| Graph | Nodes | Edges | File Size |
|---|---|---|---|
| YahooWeb | 1,413,511,390 | 6,636,600,779 | 116G |
| IMDB | 757,395 | 4,539,634 | 60M |
| AS-Caida | 65,535 | 211,444 | 2.4M |
| Kronecker | 177,147 | 1,977,149,596 | 25G |
| | 59,049 | 282,416,200 | 3.3G |
| | 19,683 | 40,333,924 | 439M |
| | 6,561 | 5,758,240 | 56M |

Table 2: Datasets

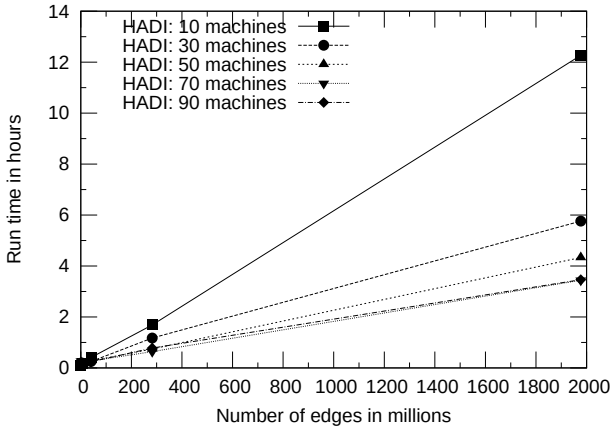



Figure 4: Wall-clock time versus number of edges, for different number of machines. Notice the excellent scalability: linear on the graph size.

## 5.2 Running Time and Scale-up

Figures 4 and 5 give running time results. Figure 4 gives the wall-clock time versus the number of edges in the graph. Each curve corresponds to a different number of machines used (from 10 to 90). Notice that the running time is linear on the number of edges, as expected. Thus, HADI has excellent scalability.

Figure 5 gives the throughput $1/T_M$. We also tried HADI with one machine; however it didn't complete, since the machine would take so long that it would often fail in the meanwhile. For this reason, we do not report the typical scale-up score $s = T_1/T_M$ (ratio of time with 1 machine, over time with $M$ machine), and instead we report just the inverse of $T_M$ (see Figure 5). We give the result from the largest synthetic graph (25Gb, 1.9 billion edges). We see the typical behavior of an effective algorithm: the throughput increases near-linearly with $M$, until we hit some bottleneck (network bandwidth, JVM load-time), in which case the throughput flattens. For the graph we show, the throughput/scale-up was near-linear, up to $M$=70 machines. The curves were similar for the other large graphs, and, of course, for small graphs, they were flat from the very beginning, as expected.

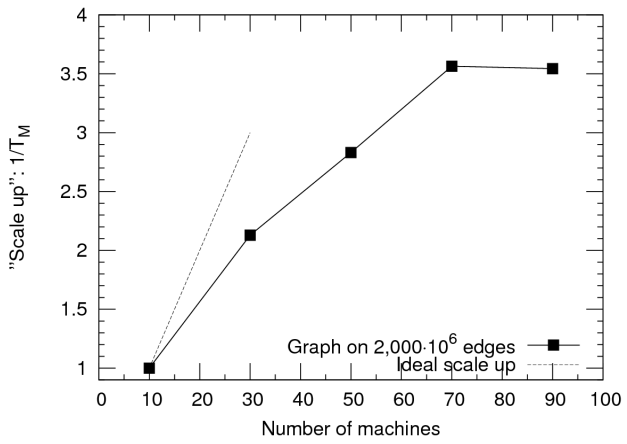



Figure 5: "Scale-up" (actually, throughput $1/T_M$), versus number of machines $M$, for our 25Gb Kronecker graph. Notice the near-linear growth in the beginning, close to the ideal, dotted line.

### 5.3 Effect of Optimizations

Among the optimizations that we mentioned earlier, which one helps the most, and by how much? Figure 6 plots the running time versus graph-size, for HADI-plain and HADI-optimized, with all combinations of optimizations. The speed improved by **1.73x** as the result of our optimizations. Most of the gains were thanks to the bitmask encoding.

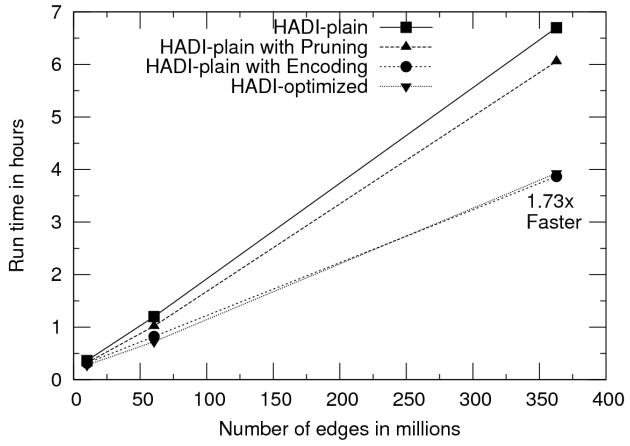



Figure 6: Comparison of running time of HADI with/without optimizations.

## 6 HADI at Work

Here we report findings on real graphs, that we were able to analyze thanks to HADI.

## 6.1 Diameter

The Yahoo web graph has *1.4 billion* nodes and *6.6 billion* edges, spanning hundreds of GigaBytes (see Table 2). How large is its diameter? Is it true that the diameter of the web grows as $\log N$, as is conjectured by Albert et al. [5]? HADI settled this questions, on what is probably the largest publicly available graph that has ever been studied:

The 90% effective diameter is 19, and the average diameter is 15.64. Moreover, as shown in Figure 7, the Albert et al. conjecture is pessimistic, predicting 19.2 as opposed to 15.64. However, our result is consistent with the more recent observation that the diameter of real graphs is constant or even shrinks over time[14]. Also notice that the smaller sample of Broder et al. [6] had a larger diameter (16.18) than ours.

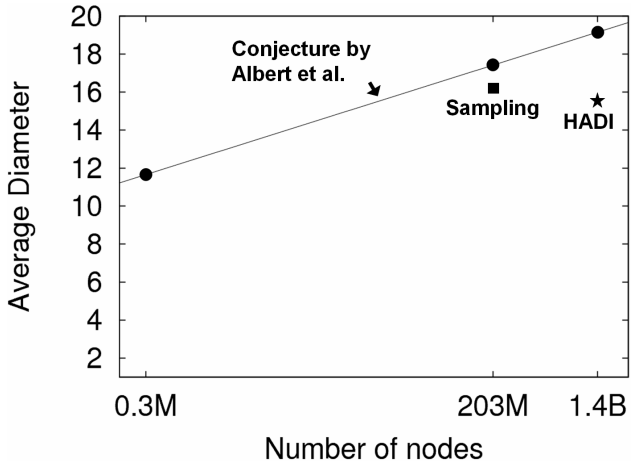



Figure 7: Comparison of average diameter of web graphs - average diameter, vs number of nodes $N$ in the crawl (in log scale). Circle at left: the result of Albert et al.; the line and the rest circles are their conjectured formula, $O(\log N)$. The square point ('Sampling') is by Broder et al. The star point is our result, with HADI. Notice the *deviation* of the formula ('circle') from reality ('star')

## 6.2 Radius-plot of Large Web Graph

The radius $r_{exact,i}$ of node $i$ is the number of hops that we need to reach the farthest-away node from node $i$. For the usual robustness reasons, we use the *effective radius* $r_i$, which is the 90th-percentile of all the distances from node $i$. The *radius plot* of a graph is the PDF (probability density function) of the radius: The horizontal axis is the radius of a node, and the vertical is the count of nodes with such radii. HADI can easily produce the *radius plot* of a graph, thanks to the FM-bitstrings that it maintains. Figure 8 shows the radius plot of the Yahoo web graph.

The radius plot gives a lot of information:

*Zero radius pages*: For example, in Figure 8, we first observe that there are many web pages with a radius of 0. These can be either pages with no out links, or pages which are excluded from further explorations due to policy (e.g., distance limit, maximum depth and so on) of crawlers. To verify this, we sampled several top-level pages[3] with degree 0, and tracked their contents using the Internet Archive

[3]These are the only pages whose exact urls are fully exposed.

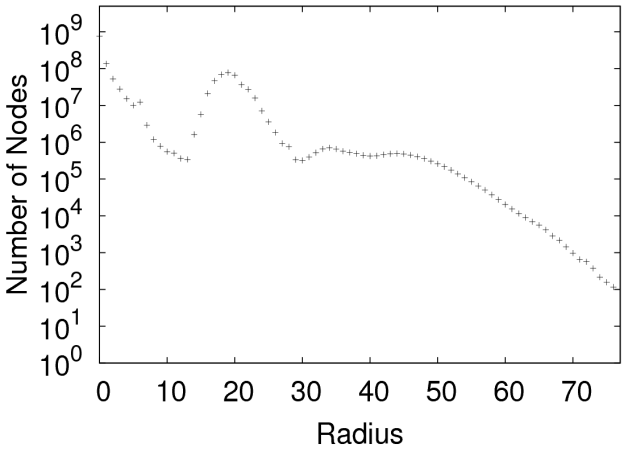



Figure 8: Radius plot of Yahoo web graph.

(`www.archive.org`). We found that some pages with out-degree 0, including `images.google.com` and `health.netscape.com`, have in fact several out links in every snapshot taken in 2002. This is strong evidence that the crawlers deliberately stopped at some of the pages, despite the fact that they had out-links.

*Max radius pages:* We also further examined those pages that had the top 3 largest radii. These 488 pages belong to only 11 web sites. None of the sites are active now; they are unpopular sites which need a long chain of links to reach the bulk of the remaining pages.

*Shape of radius plot*: It is interesting that there is a local maximum at radius 20, with some additional local maxima at 33 and 45. The shape agrees with the *bow-tie* observation, since the nodes with radius 0-10 are probably on the 'out' component, while the nodes with radius 10-30 are probably in the 'core'.

### 6.3 Radius of IMDB and Caida

The radius plot is a powerful tool that can help us spot outliers and extract information about the structure of a graph. We show some more examples next:

In the IMDB graph of Figure 9 (a), we can easily spot outlier nodes with large radius. Famous actors and famous movies have smaller radii, for example, actors such as Brad Pitt, Julia Roberts, Tom Cruise, and movies such as *Ocean's Eleven* and *Star Wars: Episode III* have radius between 6 and 8. Conversely, actor nodes with high radius would signal a long chain of lesser-known actors, playing in obscure movies. Such actor nodes are depicted in Figure 10: The Russian actress "Valeriya Malaya" (left), and the actress "Lyudmila Pogorelova" (top middle) all have radius 20. These actors played in only one movie, "Botinki iz America" (Shoes from America), whose radius is 19. The rest of the actors in that movie have radius 18. Not surprisingly, neither the movie nor the involved actors are known, at least to the authors of this paper.

Similar outliers we can spot through the AS-Caida radius plot of Figure 9 (b): there, we see that there is one node with radius 15, another with 14 and so on. Further investigation shows that (a) all these nodes belong to a chain; (b) using `whois`, the name of all the autonomous systems in this long chain is "US Sprint". This suggests that they are managed by the same company, which is a large telecom in the US.

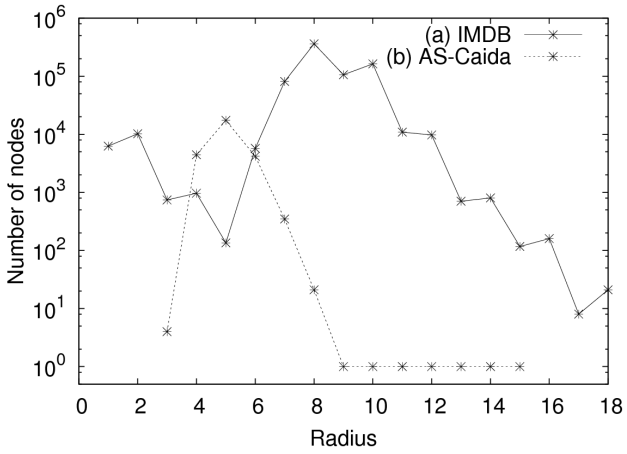



Figure 9: Radius Plot of (a)IMDB and (b)AS-Caida

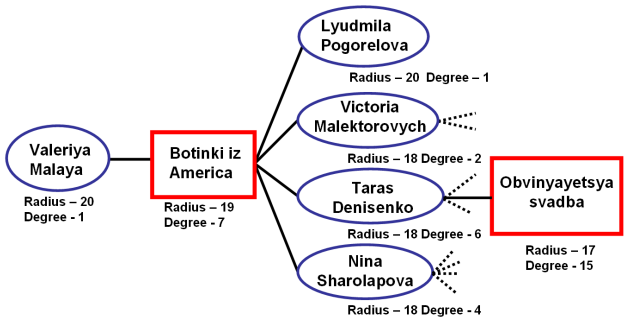



Figure 10: Russian actress "Valeriya Malaya" and the tail of the corresponding long chain in IMDB.com dataset. She has radius 20, and most of the other actors/actresses in her only movie has similar radius.

# 7 Related Work

In this section, we provide an overview of the related work: algorithms for diameter computation, the MAPREDUCE programming framework and related work from the database community.

## 7.1 Computing the Diameter

Running a Breadth First Search or any other exact algorithm on a massive graph in order to compute the diameter, is not practical due to the high time complexity ($O(n^2 + nm)$). An interesting observation is that the problem of finding the diameter of the graph can be reduced to the problem of counting the number of distinct elements in a set $A$. Roughly speaking, after performing more hops than the diameter of the graph the set of reachable nodes does not increase in size. Therefore, any streaming algorithm for the counting of distinct elements can be used. This is exactly the idea behind HADI.

Formally, define the neighborhood function $N(h)$ for $h = 0, 1, 2, \ldots, \infty$ as the number of pairs of nodes that can reach each other in $h$ hops or less. $N(0)$ is thus the number of nodes in the graph. We can find the effective diameter and average diameter of a graph using $N(h)$ as well. The effective diameter of a graph is defined to be the $h$ where $N(h)$ start to be bigger than 90% of $N(h_{max})$. The average diameter of a

graph is the expected value of $h$ over the distribution of $N(h) - N(h-1)$, *i.e.*, $\sum_{h=1}^{h_{max}} h * (N(h) - N(h-1))/(N(h_{max}) - N(0))$.

The pseudocode that illustrates the aforementioned concept is shown in Algorithm 4.

**for** *each node* $i \in G$ **do**
  $\mathcal{N}(0, i) = \{i\}$
**for** *each number of hops* $h$ **do**
  **for** *each node* $i \in G$ **do**
    $\mathcal{N}(h, i) = \mathcal{N}(h-1, i)$;
  **for** *each edge* $(i, j) \in G$ **do**
    $\mathcal{N}(h, i) = \mathcal{N}(h, i) \cup \mathcal{N}(h-1, j)$;
  $N(h) = \sum_i |\mathcal{N}(h, i)|$;

**Algorithm 4**: Naive algorithm for computing the neighborhood function $N(h)$

In order to handle a large disk-resident edge-file, we apply the idea of Flajolet-Martin ([9]), which was used in the ANF algorithm ([18]) as well. The main idea is to represent the each neighborhood set *approximately*, by using the Flajolet-Martin (FM) method. Therefore neighborhood set $\mathcal{N}(h, i)$ is represented by its FM-bitstring $b(h, i)$ and instead of union-ing two neighborhood sets, we execute bitwise ORs on the FM-bitstrings. Then we can estimate $N(h)$ from the position of leftmost '0' in each $b(h, i)$ using the following formula:

$$N(h) = \frac{1}{0.77351} \sum_{i \in V} 2^{\frac{1}{k} \sum_{l=1}^{k} b_l(i)} \tag{7.1}$$

where $b_l(i)$ is the smallest leftmost bit index of a zero-bit of each of the $l^{th}$ FM-bitstring of node $i$ and $k$ is the number of hashing functions.

### 7.2 MAPREDUCE, HADOOP and RDBMS

Scalable, parallel algorithms attract increasing attention. For example, see [3], that uses vertical partitioning to speed up queries in RDF data. Dean *et al.* [8] introduced MAPREDUCE. The MAPREDUCE programming framework processes huge amounts of data in a massively parallel way, using hundreds or thousands commodity machines. This model has two major advantages: (a) it shields the programmer from the details about the data distribution, replication, fault-tolerance, load balancing etc. and (b) it uses familiar concepts from functional programming. In short, the programmer needs to provide only two functions, a *map* and a *reduce*. The typical framework is as follows [11]: (a) the *map* stage sequentially passes over the input file and outputs (key, value) pairs; (b) the *shuffling* stage: the MAPREDUCE framework automatically groups of all values by key, (c) the *reduce* stage processes the values with the same key and outputs the final result. The approach is also related to the `group by` construct of SQL, where group-by corresponds to the *shuffling* stage, and the SQL aggregate function corresponds to the *reduce* stage.

HADOOP is the open source implementation of MAPREDUCE. HADOOP provides the Distributed File System (HDFS) [22] which is similar to the Google File System (GFS) [10], HBase [23] which is a way of efficiently storing and handling semi-structured data as Google's BigTable storage system [7], and PIG, a high level language for data analysis [16]. Due to its power and simplicity, HADOOP is a very promising tool for massive parallel graph mining applications (e.g [19]).

Recently Yang et al. [24] suggested the Map-Reduce-Merge model that can accommodate heterogeneous data sets and support join-like relational operations between them. Our method in Section 2 is similar to their model in the sense that it essentially performs a hash-join and grouping/aggregating operations on two different data sets.

However our method is different from their model in several aspects since our method compactly implements join and aggregating operations using only the two map-reduce operations, although the model would have used three map-reduce and one merge operations to do the same work. Furthermore, HADI is operational, and it uses sophisticated algorithmic and system optimizations to deal with large graphs. These fine-tunings are the ones that allow us to analyze the Yahoo! Web Graph since HADI-naive and HADI-plain would fail without them.

# 8 Conclusions

The motivation for this work has been to develop a graph mining tool that can handle giga-, tera- and peta-byte scale social networks and graphs. One of the most difficult and expensive operations involved is the estimation of the diameter which, on the surface, would be at best quadratic on the number of nodes, and thus impossible for large graphs. The contributions of this work are the following:

- We propose HADI, a carefully designed and tuned map/reduce algorithm for diameter estimation, on arbitrarily large graphs.
- We present extensive experiments on real and synthetic graphs, showing excellent scalability and findings.
- We calculate the diameter of the largest public web graph ever analyzed.

Future work could focus on the parallelization of additional graph mining algorithms, like community detection, and scree plots.

# 9 Acknowledgments

This material is partially supported by the National Science Foundation under Grants No. IIS-0705359. We would like to thank YAHOO! for providing us with the web graph and access to the M45, Adriano A. Paterlini for feedback, and Lawrence Livermore National Laboratory for their support.

# References

[1] The caida as relationships dataset 11/12/2007. `http://www.caida.org/data/active/as-relationships/`.

[2] The internet movie database. `http://www.imdb.com/`.

[3] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[4] R. Albert and A.-L. Barabasi. Emergence of scaling in random networks. *Science*, pages 509–512, 1999.

[5] R. Albert, H. Jeong, and A.-L. Barabasi. Diameter of the world wide web. *Nature*, (401):130–131, 1999.

[6] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks 33*, 2000.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *USENIX'06*, Berkeley, CA, USA, 2006.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[9] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[10] S. Ghemawat, H. Gobioff, and S. Leung. The google file system, 2003.

[11] R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70:1–30, 2008.

[12] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *PKDD*, pages 133–145, 2005.

[13] J. Leskovec and E. Horvitz. Worldwide buzz: Planetary-scale views on an instant-messaging network. Technical report, Microsoft Research, June 2007.

[14] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proc. of ACM SIGKDD*, pages 177–187, Chicago, Illinois, USA, 2005. ACM Press.

[15] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110, New York, NY, USA, 2008.

[17] C. Palmer, G. Siganos, M. Faloutsos, C. Faloutsos, and P. Gibbons. The connectivity and fault-tolerance of the internet topology. In *NRDM 2001*, Santa Barbara, CA, May 25 2001.

[18] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. Anf: a fast and scalable tool for data mining in massive graphs. *KDD*, pages 81–90, 2002.

[19] S. Papadimitriou and J. Sun. Disco: Distributed co-clustering with map-reduce. *ICDM*, 2008.

[20] D. J. Watts. *Six degrees : the science of a connected age*. 2003.

[21] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, (393):440–442, 1998.

[22] web. http://hadoop.apache.org/. Hadoop general documentation.

[23] wiki. http://wiki.apache.org/hadoop/hbase. Hadoop's Bigtable-like structure.

[24] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified relational data processing on large clusters. *SIGMOD*, 2007.


Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Carnegie Mellon