Available online at www.sciencedirect.com

ScienceDirect

Information Science 00 (2015) 1–14

Information Science

Fast Graph Mining with HBase

Ho Lee[a], Bin Shao[b], U Kang[a]

[a]*KAIST, 291 Daehak-ro (373-1 Guseong-dong), Yuseong-gu, Daejeon 305-701, Republic of Korea*
[b]*Microsoft Research Asia, Tower 2, No. 5 Danling Street Haidian District Beijing, P.R. China 100080*

Abstract

Mining large graphs using distributed platforms has attracted a lot of research interests. Especially, large graph mining on Hadoop has been researched extensively, due to its simplicity and massive scalability. However, the design principle of Hadoop to maximize scalability often limits the efficiency of the graph algorithms. For this reason, the performance of graph mining algorithms running on top of Hadoop is not satisfactory.

In this paper, we propose Unicorn, a graph mining library on top of HBase, an open source version of Bigtable. Unicorn exploits the random write characteristic of HBase to improve the performance of generalized iterative matrix-vector multiplication (GIM-V), a core graph mining routine. Extensive experiments show that Unicorn outperforms its predecessors by an order of magnitude for a graph with 68 billion edges.

Keywords: Graph Mining, Hadoop, HBase

1. Introduction

How can we analyze big graphs which do not fit in the memory or disks of a single machine? Graphs are everywhere; examples include World Wide Web [1], social networks [4], computer networks [5], citation networks [7], and mobile call networks [20], among others. The sizes of these graphs are growing at an unprecedented rate; it becomes commonplace to observe big graphs with more than billions of nodes and edges spanning several hundreds of Terabytes. Mining big graphs help us solve many important problems including web search, fraud detection, social network analysis, recommendation, anomaly detection, etc.

To handle big graphs, a parallel or distributed platform is necessary. One of the most popular parallel or distributed platform for big graph mining is MapReduce [3], or its open source platform Hadoop. Due to its scalability, simplicity, and fault tolerance, several graph mining platforms have been successfully built on Hadoop [10, 11]. However, although providing good scalability, by default Hadoop is not optimized for graph algorithms as it only provides batch accesses to the data while many graph algorithms require random accesses to the graph data. This limitation is the major bottleneck of the graph mining algorithms on Hadoop.

To overcome the problem, in this paper we propose Unicorn, a graph mining library on top of HBase, an open source implementation of Bigtable [2]. HBase is a distributed storage system for big data, and it provides random access functionality which Hadoop lacks. Using the random access functionality of

Email addresses: `crtlife@kaist.ac.kr` (Ho Lee), `binshao@microsoft.com` (Bin Shao), `ukang@cs.kaist.ac.kr` (U Kang)

Table 1: Symbols and definitions.

Symbols	Definitions
n	number of nodes in a graph
m	number of edges in a graph
M_{ij}	the (i,j)-th element of the matrix M
$M_{:i}$	the i-th column of the matrix M
v_i	the i-th element of the vector v
R	ratio of transmitted data on a shuffle phase
B_{sr}	speed of sequential read (bytes/sec) in HBase
B_{rr}	speed of random read (bytes/sec) in HBase
B_{rw}	speed of random write (bytes/sec) in HBase
D_s	speed of a local disk (bytes/sec)
N_s	speed of network transmission (bytes/sec)
$start\_mapper(p)$	cost of starting p mappers
$start\_reducer(r)$	cost of starting r reducers

HBase, Unicorn provides a fast method for big graph mining. Specifically, Unicorn provides fast GIM-V which is a building block of many graph mining algorithms including PageRank, Random Walk with Restart, radius/diameter, and connected component, etc [10]. Extensive experiments show that Unicorn is fast and scalable, outperforming its counterparts by up to 10.5×. The main contributions are as follows.

1. **Algorithm**. We propose a fast HBase based algorithm for Generalized Iterative Matrix-Vector multiplication(GIM-V), a core operator used in various graph mining tasks such as PageRank, connected components, K-step neighbors, and single source shortest paths.
2. **Cost analysis**. We derive the cost functions of our proposed algorithm, and give theoretical analysis of the performance.
3. **Extensive evaluation**. Extensive experiments show that Unicorn outperforms its counterparts by 10.5× for a graph with 68 billion edges, as shown in Figure 3.

The outline of the paper is as follows. We give related work in Section 2, and we describe naive algorithms in Section 3. We propose our Unicorn library in Section 4. After presenting experimental results in Section 5, we conclude in Section 6. Table 1 gives the definitions of the symbols we use.

2. Related Work

Related works are divided into four groups: GIM-V, Hadoop, HBase, and large scale graph analysis systems.

2.1. GIM-V

GIM-V (Generalized Iterative Matrix-Vector multiplication) [10, 9] is a core operator in various graph mining tasks such as PageRank, diameter estimation, connected components, K-step neighbors, and single source shortest paths. GIM-V is based on the observation that many graph mining algorithms are essentially represented by repeated matrix-vector multiplications. GIM-V redefines the sub operations (multiply, sum, and assign) of matrix-vector multiplications, so that many graph operations are expressed with it. In this paper we improve the performance of GIM-V on top of HBase.

2.2. Hadoop

Hadoop (`http://hadoop.apache.org/`) is the open source implementation of MapReduce [3]. Hadoo p is a widely used platform for processing huge amount of unstructured data using thousands of machines. Hadoop uses the Hadoop Distributed File System (HDFS), an open source implementation of the Google File System (GFS) [6], as its storage. HDFS maintains 3 copies of a file by default, and thus it can serve the file despite the failures of some machines. The programming model of Hadoop comprises two functions: map and reduce. The map function takes an input (key,value) pair and produces a set of intermediate (key,value) pairs. The reduce function takes an intermediate key, and a set of values for the key. After doing some computations on the key and the related values, the reduce function outputs new (key, value) pairs. Between the map and reduce steps, there is a shuffle step where the output of the map function is transferred to the reduce function through network. The main advantages of Hadoop are its 1) simplicity, 2) massive scalability, 3) fault tolerance, and 4) low cost of building and maintaining the cluster.

2.3. HBase

Overview. HBase (`http://hbase.apache.org/`) is an open source implementation of Bigtable [2]. HBase is a distributed storage system for structured data. The data are stored in the Hadoop Distributed File System (HDFS). HBase consists of a master server, region servers, and Zookeeper, a distributed lock manager. In HBase, data are split into regions. A region has a row range, and it stores rows corresponding to the range. A master server assigns regions to region servers, and balances the load of region servers. Region servers manage a set of regions, and handle read and write requests. Zookeeper stores the location of HBase data, HBase schema information, and access control lists.

Data Model. HBase is a multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; and the corresponding value is an array of bytes. The data are sorted in the lexicographic order by row key. Thus, clients can select the row keys so that they get good localities (e.g. reversing the host part of URL to store web pages). Column keys are grouped into sets called column families, and the column families form the basic unit of access control. Timestamps are used to store multiple versions of the same data.

Data Access. Unlike the standard Hadoop framework which provides only batch accesses to data, HBase provides random accesses to data. In HBase, read and write requests to regions are served by region servers. Regions are stored in the form of HFile, a persistent, ordered immutable map from keys to values, in HDFS. When a write request reaches a region server, the server checks the validity of the request, and writes the request in the commit log in HDFS. A group of requests in the commit log is committed together, and the committed writes are inserted into a memstore. When a read request arrives at a tablet server, the server checks the validity, and answers the request from a merged view of the sequence of HFiles and the memstore. Due to the group commits, the write operation is faster than the read operation in HBase. Also, sequential read is much faster than random read because 1) the sequential read exploits a special mechanism to reduce the number of RPC calls, and 2) it further utilizes Block Cache which will be described below.

Caching for read performance. HBase supports Block Cache to improve the read performance. The Block Cache is useful for reading data that are close to the recently read data because the Block Cache keeps the HFiles blocks that were read from HDFS.

2.4. Large Scale Graph Analysis Systems

Large scale graph analysis systems are divided into two categories: single-machine based systems and distributed systems.

The single-machine based systems focus on medium sized graphs that fit in the memory or disks of a single machine. A representative system is GraphChi [12] which showed how graph computation can be performed on massive graphs with billions of nodes and edges using a commodity computer, with the speed matching distributed frameworks. TurboGraph [8] improves on GraphChi using the multi-core and SSD parallelisms for increasing performance. X-Stream [18] is an edge-centric graph system using streaming partitions. By using streaming, it removes the necessity of pre-processing and building an index which causes random accesses into set of edges.

The distributed systems for graph analysis are divided into memory based systems (Pregel [17], GraphLab [15, 16] and Trinity [19]) and disk based systems (Pegasus [10] and GBase [9]). Pregel, and its open-source version Giraph (`http://giraph.apache.org/`), uses Bulk-Synchronous Parallel (BSP) model, which updates vertex states by using message passing at each sequence of iterations called super-step. GraphLab is a parallel machine learning library for graphs. It uses multiple cores to achieve high computation speed. Trinity is a distributed graph management system designed for efficient in-memory graph processing. For huge graphs that cannot fit into the memory of multiple machines, distributed disk-based approaches are popular. Pegasus and GBase are disk-based graph systems on top of Hadoop. These systems express graph computation by matrix-vector multiplication, and provide a good scalability. However, due to the sequential access patterns of Hadoop, the speed of Pegasus and GBase are not satisfactory. The proposed Unicorn exactly aims to improve the performance of these Hadoop based graph mining by exploiting the random access capability of HBase.

3. Naive Algorithms

In this section, we introduce naive graph mining algorithms for GIM-V. The naive algorithms comprise two groups: one based on Hadoop, the others based on HBase. In the following, a graph is represented by an adjacency matrix whose (i, j)th element represents the existence of an edge from node i to node j (see Figure 1).

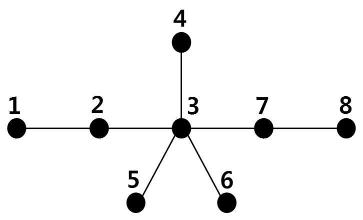
3.1. On Hadoop

The Hadoop based graph mining system [10], PEGASUS, receives two input files: the adjacency matrix file and the vector file. Each line of the matrix file contains an edge in the form of (source, destination). Each line of the vector file contains an element of the vector in the form of (id, value). PEGASUS's implementation of matrix-vector multiplication consists of two MapReduce steps. The first step groups the relevant elements of the matrix and the vector, and the second step sums up intermediate values to get the final result. The Hadoop based implementation has a good scalability, but it has a performance issue that comes from the design of the Hadoop framework. The Hadoop framework accesses data in a form of stream to maximize scalability. The framework reads data stored in a distributed file system, and feeds the data as input to mappers and reducers. Mappers and reducers can access the data only sequentially, and random accesses to the data are not allowed. This fixed access pattern leads to inflexible implementations of matrix-vector multiplication in Hadoop. In the matrix-vector multiplication, an element of the matrix should be multiplied with the corresponding vector element. However, because of the fixed access pattern, the implementation requires a reduce step to group the relevant elements of the matrix and the vector. Since the reduce step requires additional disk I/Os and network traffic, the implementation has a performance issue.

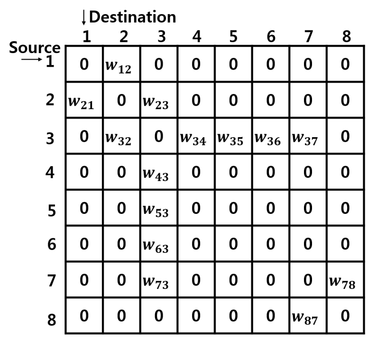
3.2. On HBase

One way to overcome the limit of Hadoop is to use HBase which supports the random access functionality. There are two naive approaches for matrix-vector multiplication using HBase. The first approach, called HBMV-DL, uses the random read capability of HBase, thereby removing the reduce step of the method in Section 3.1. The second approach, called HBMV-E, improves HBMV-DL by further exploiting the cache of HBase.

HBMV-DL, shown in Algorithm 1, uses two tables in HBase: the matrix table D and the vector table T (Figure 2(a)). Each row of the matrix table D contains the adjacency list of a node: each row has a source id as the *key*, and (the destinations ids, and the edge weights for the source id) as the *value*. Each row of the vector table T has a node id as the *key*, and the vector element for the node id as the *value*. To multiply the matrix and the vector, the mapper is given the matrix table D in HBase as its main input. On receiving a row of D, the mapper performs random reads on T to get the corresponding vector elements



(a) Example graph.



$$\begin{array}{c|cccccccc}
\text{Source} \backslash \text{Destination} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & 0 & w_{12} & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & w_{21} & 0 & w_{23} & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & w_{32} & 0 & w_{34} & w_{35} & w_{36} & w_{37} & 0 \\
4 & 0 & 0 & w_{43} & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & w_{53} & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & w_{63} & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & w_{73} & 0 & 0 & 0 & 0 & w_{78} \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & w_{87} & 0
\end{array}$$

(b) Adjacency matrix of (a).

Figure 1: An example graph and the adjacency matrix of it.

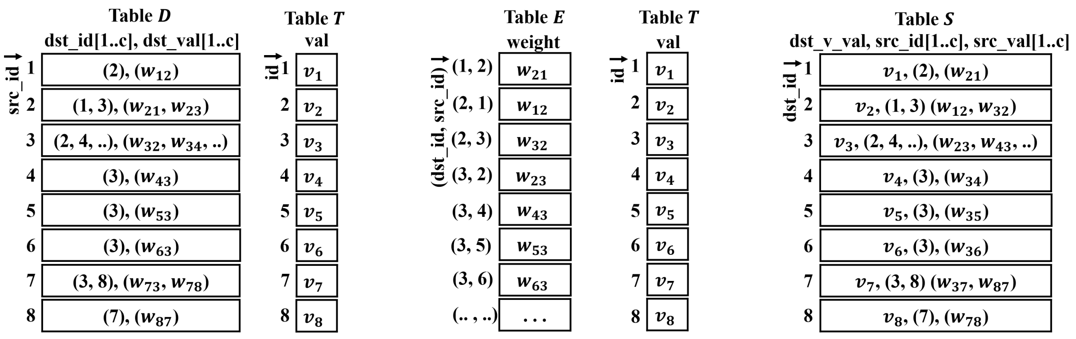


Table *D*

src_id	dst_id[1..c], dst_val[1..c]
1	(2), (w_{12})
2	(1, 3), (w_{21}, w_{23})
3	(2, 4, ..), (w_{32}, w_{34}, ..)
4	(3), (w_{43})
5	(3), (w_{53})
6	(3), (w_{63})
7	(3, 8), (w_{73}, w_{78})
8	(7), (w_{87})

Table *T*

id	val
1	v_1
2	v_2
3	v_3
4	v_4
5	v_5
6	v_6
7	v_7
8	v_8

(a) HBMV-DL

Table *E*

(dst_id, src_id)	weight
(1, 2)	w_{21}
(2, 1)	w_{12}
(2, 3)	w_{32}
(3, 2)	w_{23}
(3, 4)	w_{43}
(3, 5)	w_{53}
(3, 6)	w_{63}
(.., ..)	...

Table *T*

id	val
1	v_1
2	v_2
3	v_3
4	v_4
5	v_5
6	v_6
7	v_7
8	v_8

(b) HBMV-E

Table *S*

dst_id	dst_v_val, src_id[1..c], src_val[1..c]
1	v_1, (2), (w_{21})
2	v_2, (1, 3) (w_{12}, w_{32})
3	v_3, (2, 4, ..), (w_{23}, w_{43}, ..)
4	v_4, (3), (w_{34})
5	v_5, (3), (w_{35})
6	v_6, (3), (w_{36})
7	v_7, (3, 8) (w_{37}, w_{87})
8	v_8, (7), (w_{78})

(c) UNICORN

Figure 2: Comparison of table design in the HBase-based algorithms, on the graph in Figure 1. (a) HBMV-DL: each row of the matrix table D contains the adjacency list of a node; each row of the vector table T has a node id as the *key*, and the vector element for the node id as the *value*. (b) HBMV-E: each row of the matrix table E uses (destination id, source id) as the *key*, and the weight of the edge (source id, destination id) as the *value*; the vector table T is the same as that of HBMV-DL. (c) Unicorn: each row of the table S has a destination id as the *key*, and (the vector element for the destination, source ids for the destination, and the edge weights for the destination) as the *value*.

Algorithm 1: HBMV-DL: each row of the matrix table has a source id as the *key*, and the list of destination ids and that of the edge weights for the source id as the *value*.

```
Input:
1) Matrix table $D = \{(src\_id, (dst\_id[1..c], dst\_val[1..c]))\}$: input of Map(), and
2) Vector table $T = \{(id, val)\}$: accessed via random read inside Map()
Output: Vector table $T = \{(id, val)\}$
1 Map(key $src\_id$, value $(dst\_id[1..c], dst\_val[1..c])$);
2 begin
3     $sum \leftarrow 0$;
4     foreach $i \leftarrow 1$ to $c$ do
5         $v\_val \leftarrow T$.read($dst\_id[i]$); //read the value of the vector element from HBase;
6         $sum \leftarrow sum + dst\_val[i] * v\_val$;
7     emit($src\_id$, $sum$) ;
8 end
```

of the nodes in the destination list (line 5). After multiplying the matrix and the vector elements (line 6), the result is written back to the vector table T in HBase (line 7). Note that HBMV-DL does not require an additional reduce step since the map step can access all the information to compute an element of the output vector for a given row of the matrix table.

Algorithm 2: HBMV-E: each row of the matrix table has a (*dst_id*, *src_id*) pair as the *key*, and the weight of the edge as the *value*.

```
Input:
1) Matrix table $E = \{((dst\_id, src\_id), weight)\}$: input of Map(), and
2) Vector table $T = \{(id, val)\}$: accessed via random read inside Map()
Output: Vector table $T = \{(id, val)\}$
1  Map(key $(dst\_id, src\_id)$, value $weight$);
2  begin
3      $v\_val \leftarrow T$.read($dst\_id$); //read the value of the vector element from HBase;
4      emit($src\_id$, $weight * v\_val$);
5  end
6  Reduce(key $src\_id$, value $wv[1..r]$);
7  begin
8      $sum \leftarrow 0$;
9      foreach $temp \in wv[1..r]$ do
10         $sum \leftarrow sum + temp$;
11     emit($k$, $sum$) ;
12 end
```

HBMV-E, shown in Algorithm 2, is similar to HBMV-DL, except that the matrix table E stores each edge separately in its own row (Figure 2(b)). This design is chosen to more efficiently use the cache functionality of HBase. The cache of HBase is designed so that reading consecutive rows of a table leads to better cache hit. To exploit this feature, each row of the matrix table E in HBMV-E uses (destination id, source id) as the *key*, and the weight of the edge (source id, destination id) as the *value*. Note that the order (destination id, source id) of the node ids in the key is reversed from the order (source id, destination id) in the edge. The reason is that a mapper reading an edge (a row of the matrix table E) requires a vector element corresponding to the *destination* id of the edge. Thus, using the (destination id, source id) as the key guarantees that consecutive rows of the matrix table E will read the same or similar range of vector elements, thereby achieving better cache (Block Cache) hit in HBase. Note that using the (source id, destination id) as the key does not benefit from the cache since consecutive rows of the matrix table E would request non-redundant vector elements corresponding to diverse destination ids. Note that since HBMV-E reads each edge separately, it requires a reduce step to sum up the intermediate results (lines 6-12).

4. The UNICORN Library

In this section, we propose Unicorn, an efficient graph mining library for large graphs using HBase. We first give the main idea, and then describe Unicorn in detail. Next, we show applications of Unicorn including PageRank, connected components, K-step neighbors, and single source shortest paths. Finally, we discuss the costs of Unicorn.

4.1. Main Idea

The main ideas to improve the naive algorithms in Section 3 are: 1) designing algorithms to exploit the random write capabilities of HBase, and 2) designing the table structure accordingly. The random write

Table 2: Characteristics of matrix-vector multiplication methods based on HBase. Unicorn stores a column of a matrix and the corresponding vector element together in a table, and thus it needs only the random write operation. However, the other two methods, HBMV-DL and HBMV-E (both in Section 3.2), store the vector elements in a separate vector table, and thus they require random read operations. Between HBMV-DL and HBMV-E, the main difference lies in whether the method is map-only: unlike HBMV-E which treats each edge separately, HBMV-DL groups edges with a same source, and thus it works only with mappers, without requiring reducers.

	HBMV-DL	**HBMV-E**	**Unicorn**
Random Read on Vector	Yes	Yes	No
Random Write on Vector	No	No	Yes
Map Only	Yes	No	No
Separate Vector Table	Yes	Yes	No

capability of HBase allows a column of the adjacency matrix to be stored together with the vector element. Unicorn combines the adjacency matrix and the vector together in a table in HBase (Figure 2(c)), and exploits the random write capability of HBase to use the combined table. As we will see in Section 5, Unicorn provides the best performance. Table 2 compares the characteristics of HBase-based methods. Below, we describe the algorithm in detail. Note that in the algorithm, *emit*(k, v) is a function to write a key, value pair (k, v) as an output of a mapper or a reducer; since all the algorithms use HBase as the output storage, the function invokes a write to HBase.

4.2. UNICORN Graph Mining Algorithm

Algorithm 3: Matrix-Vector Multiplication in Unicorn: each row of the input table has *dst_id* as the *key*, and $(dst\_v\_val, src\_id[1..c], src\_val[1..c])$ as the *value*.

```
Input:
Table (for both matrix and vector elements) $S = \{(dst\_id, (dst\_v\_val, src\_id[1..c], src\_val[1..c]))\}$:
input of Map()
Output: None- the vector element is updated via random write inside Reduce()
1  Map(key dst_id, value $(src\_id[1..c], src\_val[1..c], dst\_v\_val)$);
2  begin
3      foreach $i \leftarrow 1$ to $c$ do
4          emit($src\_id[i], src\_val[i] * dst\_v\_val$);
5  end
6  Reduce(key src_id, value $sd[1..r]$);
7  begin
8      $sum \leftarrow 0$;
9      foreach $temp \in sd[1..r]$ do
10         $sum \leftarrow sum + temp$;
11     $S$.write(src_id, dst_v_val, sum); // write the value of the vector element to HBase;
12 end
```

Fast matrix-vector multiplication of Unicorn is shown in Algorithm 3. Unicorn has only one table S which stores both the matrix and the vector elements. Each row of the table S has a destination id as the *key*, and (the vector element for the destination, source ids for the destination, and the edge weights for the destination) as the *value*. The mapper is given the table S in HBase as its input. On receiving a row of S, the mapper multiplies the vector element with the edge weights for the destination id; for each edge a partial multiplication result is emitted (line 4). The reduce step aggregates the intermediate values (line 10), and writes the result to HBase using the random write functionality (line 11). Note that the random write is performed n times where n is the the number of nodes in the input graph, while the naive HBase-based algorithms perform random read m times where m is the number of edges in the input graph. Given that $n \ll m$ in real world graphs, and that random write is faster than random read in HBase, Unicorn is expected to perform faster than the naive HBase-based algorithms; we verify this claim experimentally in Section 5.

For using the combined table which combines the adjacency matrix and the vector, Unicorn needs preprocessing. We implement preprocessing on Hadoop. The running time of preprocessing is similar to the running time of one matrix-vector multiplication step. Graph mining algorithms consists of multiple matrix-vector multiplication steps, and preprocessing is needed at most two times for the adjacency matrix and the transpose of the adjacency matrix in each graph. Hence, preprocessing does not have a significant impact on the time for analyzing the graph.

Applications. As discussed in Section 2.1, Unicorn supports many graph algorithms that are expressed by GIM-V, including PageRank, connected component, diameter estimation, random walk with restart, K-step neighbors, and single source shortest paths [9, 10].

4.3. Cost Analysis

In this section, we analyze the cost of Unicorn by deriving cost equations. This analysis of costs helps better understand the characteristics and the performances of Unicorn and other competitors. We first derive costs of random accesses on HBase.

Cost Analysis 1. *Random Read Cost (from* HBase*) - the cost $C_{RR}(U, s)$ of random read of s bytes from* HBase *using U machines is given by*

$$C_{RR}(U, s) = \frac{s}{U} * \frac{1}{B_{rr}} \tag{1}$$

Explanation. U machines read the s bytes at the speed of B_{rr} per second. Thus, it takes $\frac{s}{U} * \frac{1}{B_{rr}}$ time for the read. □

Cost Analysis 2. *Random Write Cost (to* HBase*) - the cost $C_{RW}(U, s)$ of random write of s bytes to* HBase *using U machines is given by*

$$C_{RW}(U, s) = \frac{s}{U} * \frac{1}{B_{rw}} \tag{2}$$

Explanation. U machines write the s bytes at the speed of B_{rw} per second. Thus, it takes $\frac{s}{U} * \frac{1}{B_{rw}}$ time for the write. □

Next, we derive costs of steps in running a Hadoop job using HBase as a distributed storage. Since the task of matrix-vector multiplication is I/O intensive, we focus only on the I/O costs, without considering the CPU costs.

Cost Analysis 3. *Map Cost (from* HBase*) - the map cost $C_M(p, U, s)$ of reading data of size s using p mappers and U machines is given by*

$$C_M(p, U, s) = start\_mapper(p) + \frac{s}{U} * \frac{1}{B_{sr}} \tag{3}$$

Explanation. In a map function, the cost of starting p mappers is $start\_mapper(p)$. The U machines, which run the mappers, read the matrix from HBase in parallel at the rate of B_{sr} bytes per second. Hence, the cost of reading the data is $\frac{s}{U} * \frac{1}{B_{sr}}$. □

Cost Analysis 4. *Shuffle Cost - the cost $C_F(U,s)$ of shuffling data of size s using U machines is given by:*

$$C_F(U,s) = \frac{s*R}{U} * \frac{1}{N_s} + 4 * \frac{s}{U} * \frac{1}{D_s} \tag{4}$$

Explanation. In the shuffle step, the data of size s are transferred from the machines running mappers to those running reducers. When the source and the destination of the transfer are the same, however, the data are not sent through the network. Assuming that R is the ratio of data transmitted through the network, then the time for the network transfer is given by $\frac{s*R}{U} * \frac{1}{N_s}$. The term $4 * \frac{s}{U} * \frac{1}{D_s}$ is the cost for 4 I/Os required for the mapper's output written to and read from disks, and the reducer's input written to and read from disks. □

Cost Analysis 5. *Reduce Cost (to* HBase*) - the reduce cost $C_R(r,U,s)$ of writing data of size s using r reducers and U machines is given by*

$$C_R(r,U,s) = start\_reducer(r) + C_{RW}(U,s) \tag{5}$$

Explanation. In a reduce function, the cost of starting r reducers is $start\_reducer(r)$. The U machines, which run the reducers, write the data of size s to HBase in parallel with the cost $C_{RW}(U,s)$. □

Now we are ready to derive the cost of Unicorn. We also derive the cost of HBMV-DL and HBMV-E for comparison.

Cost Analysis 6. HBMV-E *Cost - the cost for* HBMV-E *is given by*

$$\begin{aligned} costE = & C_M(p,U,m) + C_{RR}(U,m) + C_F(U,m) \\ & + C_R(r,U,n) \end{aligned} \tag{6}$$

Explanation. In HBMV-E, mappers are given the matrix elements which are sequentially read from HBase. Given a matrix element, a mapper performs a random read of the vector element corresponding to the destination of the matrix element. Thus up to now, the cost $C_M(p,U,m) + C_{RR}(U,m)$ is needed. The multiplied matrix elements are transmitted to reducers with the cost $C_F(U,m)$. Finally, the reducers write the result vector back to HBase with the cost $C_R(r,U,n)$. □

Cost Analysis 7. HBMV-DL *Cost - the cost for* HBMV-DL *is given by*

$$costDL = C_M(p,U,m) + C_{RR}(U,m) + C_{RW}(U,n) \tag{7}$$

Explanation. HBMV-DL is similar to HBMV-E, but HBMV-DL is a map-only method which does not need the shuffle and reduce steps. Thus, it requires $C_M(p,U,m) + C_{RR}(U,m)$ for mappers to sequentially read the matrix elements and randomly read the corresponding vector elements. The result vector is written directly back to HBase in the mappers with the cost $C_{RW}(U,n)$. □

Cost Analysis 8. Unicorn *Cost - the cost for* Unicorn *is given by*

$$costUNI = C_M(p,U,m+n) + C_F(U,m) + C_R(r,U,n) \tag{8}$$

Explanation. In Unicorn, mappers are given the matrix elements and the vector elements which are sequentially read from HBase with the cost $C_M(p,U,m+n)$. The multiplied matrix elements are transmitted to reducers in the shuffle step with the cost $C_F(U,m)$. Finally, the reducers output data of size s with the cost $C_R(r,U,n)$. □

We verify these cost functions experimentally in Section 5.3.

Table 3: Summary of graphs used.

Graph	**Nodes**	**Edges**	**Description**
Kronecker	387 M	68,719 M	Synthetic graphs [13]
YahooWeb	1,413 M	6,636 M	WWW pages in 2002
Twitter	62 M	1,838 M	Who-follows-whom in 2009/11
	40 M	755 M	Subgraph of Twitter in 2009/11
	30 M	424 M	Subgraph of Twitter in 2009/11
LinkedIn	6.8 M	58 M	Friendship network in 2006/10

5. Experiments

We present experimental results to answer the following questions.

Q1 How does Unicorn improve the performance of matrix-vector multiplication?
Q2 How well does Unicorn scale out?
Q3 How accurate are the cost functions?

Data. The graphs we use in our experiments are described in Table 3. We use the synthetic Kronecker[13] graphs for performance experiments since we can generate graphs of any size with the characteristics of realistic graphs including heavy-tailed degree distribution, shrinking diameter, densification power law, etc. [14]. The subgraph of Twitter indicates the principal submatrix of the adjacency matrix of Twitter graph.

Cluster. All experiments are performed in an HBase cluster with 20 machines. Each machine has 12 Terabytes of disks, 32 Gigabytes of memory, and a 3.30 GHz Intel(R) Xeon(R) CPU E3-1230 v3. We use Hadoop version 1.2.1 and HBase version 0.94.19.

5.1. Q1 - performance of matrix-vector multiplication

We compare the performance of Unicorn with that of Hadoop-based method (PEGASUS, shown in Section 3.1) and naive HBase-based methods in Figure 3 and Table 4 which shows the average running time of PageRank per iteration. Note that Unicorn has the best performance, outperforming PEGASUS up to 10.5 times. Other HBase-based methods are slower than PEGASUS. We have the following observation.

- Unicorn is the fastest since it carefully uses the random write operation which is fast and scalable in HBase. On the other hand, PEGASUS uses only the sequential read and write operations which limit the speed. HBMV-E and HBMV-DL are slower than Unicorn because they use random read operations which are relatively slow in HBase.

5.2. Q2 - Scalability

In this section, we report the edge scalability and the machine scalability of Unicorn.

For the edge scalability, we show how the performance of Unicorn changes as the input data grows. We use subgraphs of Twitter, and measure the running time of PageRank algorithms in Unicorn, PEGASUS, HBMV-DL, and HBMV-E using 20 machines. Figure 4 shows the result: note that although all methods scale linearly with the number of edges, Unicorn has the smallest slope.

For the machine scalability, we show the relative throughput $\frac{T_4}{T_M}$ with M machines for the Twitter graph, where T_M is the running time with M machines. T_4 is the running time with 4 machines which we assume to be contained in a smallest cluster. Figure 5 shows the result: note that although all methods scale near-linearly with the number of edges, Unicorn has the smallest slope. HBMV-DL is not shown since it died while running due to the huge amount of HBase read request which do not benefit from the cache of HBase (more details in Sections 3.2 and 5.3).

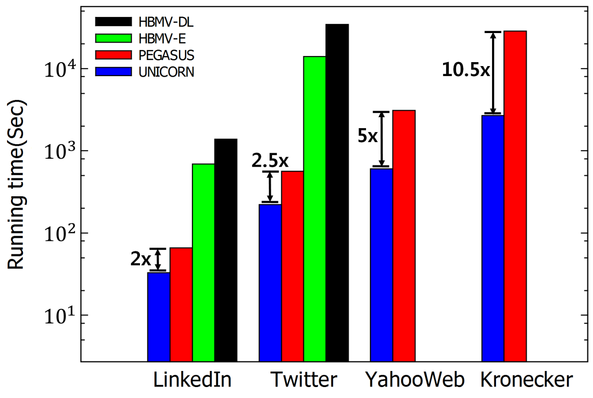


Figure 3: Performance of Unicorn compared with the Hadoop-based method (PEGASUS) and naive HBase-based methods (HBMV-*). For all the graphs, Unicorn is the fastest, outperforming the second best method (Pegasus) by up to 10.5 times. Note that HBMV-E and HBMV-DL are not shown for YahooWeb and Kronecker graphs since the experiments took too long (more than a day). HBMV-E and HBMV-DL are slower than PEGASUS due to the slow speed of random reads in HBase.

Table 4: Performance of Unicorn compared with the Hadoop-based method (PEGASUS) and naive HBase-based methods (HBMV-*). This table shows the average running time of PageRank per iteration in Figure 3.

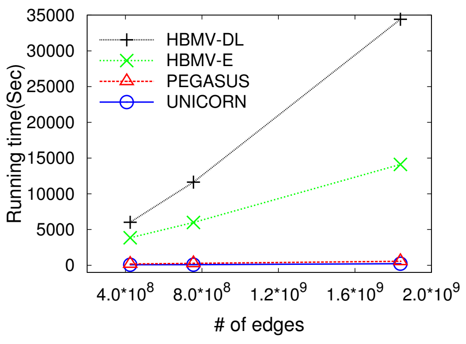
Graph	**UNICORN (Sec)**	**PEGASUS (Sec)**	**HBMV-E(Sec)**	**HBMV-DL(Sec)**
Kronecker	2697	28559	–	–
YahooWeb	603	3094	–	–
Twitter	222	566	14111	34418
LinkedIn	33	66	690	1379

5.3. Q3 - Accuracy of the cost functions

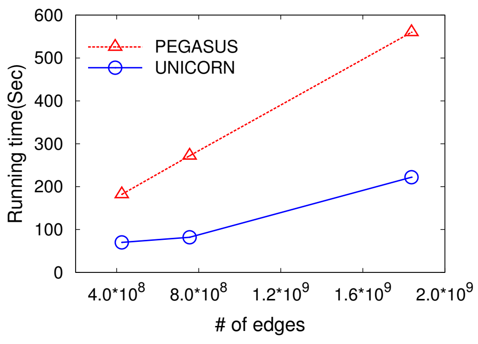
In this section, we verify our cost function's accuracy. We first compare HBMV-E and HBMV-DL, and then HBMV-E and Unicorn, using the Twitter graph. We compare the cost results with the experimental results presented in Section 5.1.

HBMV-E *and* HBMV-DL. We compare the costs of HBMV-E and HBMV-DL. From Lemmas 6 and 7, we remove the terms C_M and C_{RW}; the remaining terms for HBMV-E is $C_{RR} + C_F + start\_reducer(r)$, and those for HBMV-DL is C_{RR}. We measure the costs of the remaining terms using the Twitter graph. Since directly measuring the costs require modifying the HBase source code which we treat as a black-box, we take an indirect measurement approach: we first run an experiment containing all the terms, and then another experiment containing only the non-remaining terms, then compute the difference with regard to the running time. Table 5 shows the result. At first, the result that HBMV-E is faster than HBMV-DL seems counter intuitive, since HBMV-E's remaining terms contain the HBMV-DL's remaining terms. It turned out that the random read cost (time) C_{RR} of HBMV-E is much smaller than that of HBMV-DL due to the cache effect of HBase. Table 6 shows the comparison of random read costs of HBMV-E and HBMV-DL. We can see that HBMV-E has smaller random read cost than HBMV-DL, due to the cache effect. The HBMV-E (Reverse) is a result from an experiment which uses (source, destination), instead of (destination, source) as the key for the matrix table in HBMV-E. It is the slowest, since HBMV-E (Reverse) cannot exploit the Block Cache of HBase, as described in Section 3.2.

HBMV-E *and* Unicorn. We compare the cost of HBMV-E and Unicorn. Similarly to the experiments in the previous paragraph, we compare the running times of remaining terms ($C_M(p, U, m) + C_{RR}(U, m)$ for HBMV-E and $C_M(p, U, m + n)$ for Unicorn) after removing the common terms. Table 7 shows the



(a) UNICORN, PEGASUS, and HBMV



(b) UNICORN and PEGASUS

Figure 4: Running time vs number of edges in UNICORN, PEGASUS, HBMV-DL, and HBMV-E for the subsets of Twitter graph. Note that although all methods scale linearly, UNICORN has the smallest slope. We magnify (a) for comparing PEGASUS and UNICORN in (b).

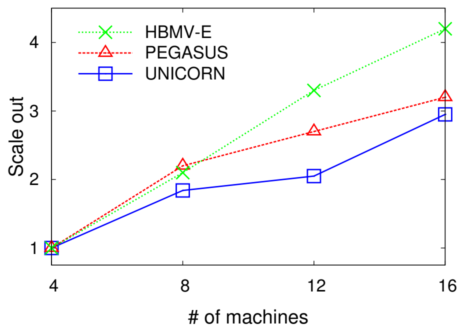


Figure 5: The relative throughput $\frac{T_4}{T_M}$ with M machines for the Twitter graph, where T_M is the running time with M machines. Note that the relative throughputs of HBMV-E scales near-linearly. HBMV-DL is not shown since it died while running due to the huge amount of HBASE read requests which do not benefit from the cache of HBASE.

Table 5: Cost comparison of HBMV-E and HBMV-DL. Despite the additional shuffle step, HBMV-E is faster than HBMV-DL thanks to better utilizing cache, and the combiner.

Method	**Cost (Sec)**
HBMV-E ($C_{RR} + C_F + C_R$)	14261
HBMV-DL ($C_{RR} + \frac{n}{U} * \frac{1}{B_{rw}}$)	31047

Table 6: Cost comparison of HBMV-E, HBMV-DL, and a variant of HBMV-E by reversing the source and destination of the key. Note that HBMV-E is the fastest due to the effective use of cache in HBase.

Method	**Cost (Sec)**
HBMV-E	14147
HBMV-DL	31047
HBMV-E (Reverse)	46993

result. Clearly, HBMV-E has a larger cost due to the additional term C_{RR} for random read. Unicorn has an additional sequential read cost of n compared to HBMV-E, but note that the sequential read is much faster than random read as explained in Section 2. Also, the amount m of random read in HBMV-E is much larger than the additional amount n of sequential read in Unicorn.

From the above experiments, we see that our cost functions accurately estimate that Unicorn is the fastest, and HBMV-E is faster than HBMV-DL, which is exactly reflected in Figure 3.

6. Conclusions

In this paper, we propose Unicorn, a scalable graph mining library on top of HBase, an open source version of Bigtable. Unicorn provides a fast GIM-V algorithm with which many graph mining algorithms are expressed. Unlike previous graph mining algorithms based on HDFS which provide only the sequential access, and naive HBase based algorithms which use random read functionality of HBase, Unicorn exploits the random write functionality of HBase to expedite the graph processing. We analyze the costs of our proposed algorithm. We also perform extensive experiments on large real and synthetic graphs. The experimental results show that Unicorn is efficient, outperforming the HDFS based system by up to 10.5 times for a graph with 68 billion edges. Future works include extending Unicorn for incremental graph computations (e.g. time evolving graphs).

Acknowledgment

This research was supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ICT/SW Creative Research program (NIPA-2013-H0503-13-1008) supervised by the NIPA(National IT Industry Promotion Agency), and by the IT R&D program of MSIP/IITP. [10044970, Development of Core Technology for Human-like Self-taught Learning based on Symbolic Approach].

References

[1] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J., Jun. 2000. Graph structure in the web. Comput. Netw. 33 (1-6), 309–320.
URL `http://dx.doi.org/10.1016/S1389-1286(00)00083-9`

Table 7: Cost comparison of HBMV-E and Unicorn. HBMV-E has a larger cost since random read is slower than random write, and also the amount of read is much larger than that of Unicorn.

Method	**Cost (Sec)**
HBMV-E ($C_M + C_{RR}$)	14414
Unicorn (C_M)	50

[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., Gruber, R. E., 2006. Bigtable: a distributed storage system for structured data, 15–15.
URL `http://dl.acm.org/citation.cfm?id=1267308.1267323`

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

[4] Ellison, N. B., Steinfield, C., Lampe, C., 2007. The benefits of facebook friends: social capital and college students use of online social network sites. Journal of Computer-Mediated Communication 12 (4), 1143–1168.
URL `http://dx.doi.org/10.1111/j.1083-6101.2007.00367.x`

[5] Faloutsos, M., Faloutsos, P., Faloutsos, C., Aug. 1999. On power-law relationships of the internet topology. SIGCOMM Comput. Commun. Rev. 29 (4), 251–262.
URL `http://doi.acm.org/10.1145/316194.316229`

[6] Ghemawat, S., Gobioff, H., Leung, S.-T., 2003. The google file system. In: SOSP. pp. 29–43.

[7] Giles, C. L., Bollacker, K. D., Lawrence, S., 1998. Citeseer: an automatic citation indexing system. In: INTERNATIONAL CONFERENCE ON DIGITAL LIBRARIES. ACM Press, pp. 89–98.

[8] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., Yu, H., 2013. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '13. ACM, New York, NY, USA, pp. 77–85.
URL `http://doi.acm.org/10.1145/2487575.2487581`

[9] Kang, U., Tong, H., Sun, J., Lin, C.-Y., Faloutsos, C., 2012. Gbase: an efficient analysis platform for large graphs. VLDB J. 21 (5), 637–650.

[10] Kang, U., Tsourakakis, C., Faloutsos, C., 2009. Pegasus: A peta-scale graph mining system - implementation and observations. ICDM.

[11] Kang, U., Tsourakakis, C. E., Appel, A. P., Faloutsos, C., Leskovec, J., February 2011. Hadi: Mining radii of large graphs. ACM Trans. Knowl. Discov. Data 5, 8:1–8:24.

[12] Kyrola, A., Blelloch, G., Guestrin, C., 2012. Graphchi: Large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. OSDI'12. USENIX Association, Berkeley, CA, USA, pp. 31–46.
URL `http://dl.acm.org/citation.cfm?id=2387880.2387884`

[13] Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., 2005. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In: In PKDD. Springer, pp. 133–145.

[14] Leskovec, J., Kleinberg, J., Faloutsos, C., 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining. KDD '05. ACM, New York, NY, USA, pp. 177–187.
URL `http://doi.acm.org/10.1145/1081870.1081893`

[15] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J. M., 2012. Distributed graphlab: A framework for machine learning and data mining in the cloud. Proc. of the VLDB Endowment 5 (8), 716–727.

[16] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J. M., 2010. Graphlab: A new framework for parallel machine learning. arXiv preprint.

[17] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., Czajkowski, G., 2010. Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. SIGMOD '10. ACM, New York, NY, USA, pp. 135–146.
URL `http://doi.acm.org/10.1145/1807167.1807184`

[18] Roy, A., Mihailovic, I., Zwaenepoel, W., 2013. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In: Proc. of the 24th ACM Symposium on Operating Systems Principles. ACM.

[19] Shao, B., Wang, H., Li, Y., 2013. Trinity: A distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. SIGMOD '13. ACM, New York, NY, USA, pp. 505–516.
URL `http://doi.acm.org/10.1145/2463676.2467799`

[20] Wang, D., Pedreschi, D., Song, C., Giannotti, F., Barabasi, A.-L., 2011. Human mobility, social ties, and link prediction. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '11. ACM, New York, NY, USA, pp. 1100–1108.
URL `http://doi.acm.org/10.1145/2020408.2020581`