GBASE: A Scalable and General Graph Management System

U Kang
Carnegie Mellon University
ukang@cs.cmu.edu

Hanghang Tong
IBM T.J. Watson
htong@us.ibm.com

Jimeng Sun
IBM T.J. Watson
jimeng@us.ibm.com

Ching-Yung Lin
IBM T.J. Watson
chingyung@us.ibm.com

Christos Faloutsos
Carnegie Mellon University
christos@cs.cmu.edu

ABSTRACT

Graphs appear in numerous applications including cyber-security, the Internet, social networks, protein networks, recommendation systems, and many more. Graphs with millions or even billions of nodes and edges are common-place. How to store such large graphs efficiently? What are the core operations/queries on those graphs? How to answer the graph queries quickly? We propose GBASE, a scalable and general graph management and mining system. The key novelties lie in 1) our storage and compression scheme for a parallel setting and 2) the carefully chosen graph operations and their efficient implementation. We designed and implemented an instance of GBASE using MapReduce/Hadoop. GBASE provides a parallel indexing mechanism for graph mining operations that both saves storage space, as well as accelerates queries. We ran numerous experiments on real graphs, spanning billions of nodes and edges, and we show that our proposed GBASE is indeed fast, scalable and nimble, with significant savings in space and time.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—Data Mining

General Terms

Design, Experimentation, Algorithms

Keywords

Graph, Indexing, Compression, Distributed Computing

1. INTRODUCTION

Graphs have been receiving increasing research attention, being applicable in a wide variety of high impact applications, like social networks, cyber-security, recommendation systems, fraud/anomaly detection, protein-protein interaction networks, to name a few. In fact, any many-to-many database relationship can be easily treated as a graph, with myriads of additional applications (patients and symptoms; customers and locations they have been to; documents and terms in IR, etc). To add to the challenge of graph mining, even the volume of such graphs is unprecedented, reaching and exceeding billions of nodes and edges.

Problem Definitions.

Our goal is to build a general graph management system in parallel, distributed settings to support billion-scale graphs for various applications. For the goal, we address the following problems:

1. Storage. How can we efficiently store and manage such huge graphs in parallel, distributed settings to answer graph queries efficiently? How should we split the edges into smaller units? How should we group the units into files?
2. Algorithms. How can we define common, core algorithms to satisfy various graph applications?
3. Query Optimization. How can we exploit the efficient storage and general algorithms to execute queries efficiently?

For all the problems, scalability is a major challenge. The size of graphs has been experiencing an unprecedented growth. For example, one of the graphs we use here, the Yahoo Web graph from 2002, has more than 1 billion nodes and almost 7 billion edges. Similar size, or even larger graphs, exist: the Twitter graph spans several Terabytes; click-streams are reported to reach Petabyte scale. Such large graphs violate the assumption that the graph can be fit in main memory or at least the disk of a single workstation, on which most of existing graph algorithms have been built. Thus, we need to re-think those algorithms, and to develop scalable, parallel ones, to manage graphs that span Tera-bytes and beyond.

Our Contributions.

We propose GBASE, a scalable and general graph management system, to address the above challenges. The main contributions are the following:

1. Storage. We propose a novel graph storage method called ‘block compression’ to efficiently store homogeneous regions of graphs. We also propose a grid based method to efficiently place blocks into files. We run our algorithm on billion-scale graphs and show that the block compression method leads up to 50× less storage and faster running time. Our block compression method is agnostic to the underlying storage mechanism, which can be applied to distributed file systems as well as relational databases.
2. Algorithms. We identify a core graph operation, and use it to formulate seven different types of graph queries in-
The overall framework of our GBASE is summarized in Figure 1. The design objective is to balance storage efficiency and query performance on large graphs. It comprises of two components: the indexing stage and the query stage. In this section, we give a high level overview of each stage; and we will give more details in Section 3 and 4, respectively.

In the indexing stage, given the original raw graph which is stored as a big edge file, GBASE first partitions it into several homogeneous blocks. Second, according to the partition results, we shuffle the nodes so that the nodes belonging to the same partition are put nearby. Third, we compress all non-empty block through standard compression such as GZip. Finally, we store the compressed blocks, together with some meta information (e.g., the block row id and column id, and all the encoded node ids), into the graph databases. For many real graphs, such homogeneous blocks, community-like structure, do exist. Therefore, after partition and reshuffling, the resulting blocks are either relatively dense (e.g., the diagonal blocks in Figure 1) or very sparse (e.g., the off-diagonal blocks in Figure 1). Both cases are space efficient for compression (i.e., the compression ratio is high). In the extreme case that a given block is empty, we do not store it at all. Our experiments (See Section 5) show that in some cases, we only need less than 2% storage space of the original after the indexing stage.

In the query stage, our goal is to provide a set of core operations that will be sufficient to support a diverse set of graph applications, e.g., ranking, community detection, anomaly detection, and etc. The key of the on-line query stage is the query execution engine, which unifies the different types of inputs as query vectors. It also unifies the (seemingly) different types of operations on the graph by a unified matrix-vector multiplication which we will introduce in Section 4. By doing so, GBASE is able to support multiple different types of queries simultaneously. Table 1 summarizes the queries (the first column) that are supported by GBASE. These queries construct the main building blocks for a variety of important graph applications (Table 1). For example, the diversity of RWR(Random Walk with Restart [35]) scores among the neighborhood of a given edge/node is a strong indicator of abnormality of that node/edge [34]. The ratio between the number of edges (or the summation of edge weights) and number of nodes within the egonet can help find abnormal nodes on weighted graphs [6]. The K-cores and cross-edges can be used for visualization and finding communities in large graphs.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>Graph.</td>
</tr>
<tr>
<td>$A$</td>
<td>Adjacency matrix of the graph $G$.</td>
</tr>
<tr>
<td>$B$</td>
<td>Incidence matrix of the graph $G$.</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of nodes.</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of edges.</td>
</tr>
<tr>
<td>$k$</td>
<td>Number of partitions.</td>
</tr>
<tr>
<td>$p, q$</td>
<td>Partition indices, $1 \leq p, q \leq k$.</td>
</tr>
<tr>
<td>$I^{[p]}$</td>
<td>Set of nodes belonging to the $p$-th partition.</td>
</tr>
<tr>
<td>$B^{[k]}$</td>
<td>Partition size, $I^{[k]} \equiv</td>
</tr>
<tr>
<td>$G^{[p,q]}$</td>
<td>Subgraphs induced by $p$-th and $q$-th partitions.</td>
</tr>
<tr>
<td>$m^{[p,q]}$</td>
<td>Number of edges in $G^{[p,q]}$.</td>
</tr>
<tr>
<td>$H(\cdot)$</td>
<td>Shannon entropy function.</td>
</tr>
</tbody>
</table>

Table 2: Definitions of symbols

In the query stage, global and targeted queries from various graph applications are handled by a unified query execution engine.
the node. The adjacency list format is simple and might be good for answering out-neighbor queries. However, it is not efficient format for answering general queries including in-neighbor queries and ego-net queries as we will see in Section 4. For the reason, we instead use the sparse adjacency matrix format, where we save each edge by a (source, destination) pair. The advantage of the sparse adjacency matrix format is its generality and flexibility to enable efficient storage and indexing techniques as we will see later in this and the next sections.

The storage system should be designed to be efficient in both storage cost and on-line query response. To this end, we propose to index and store the graph on the homogeneous block, community-like structure, levels. Next, we will describe how to form, compress and store/place such blocks.

3.2 Block Formulation

The first step is to partition the graph, i.e., re-order the rows and columns, and make homogeneous regions into blocks. Partitioning algorithms form an active research area, and finding optimal partitions is orthogonal to our work. Any partition algorithms, e.g., METIS [22], Disco [30], etc, can be naturally plugged into GBASE.

Graph partitioning can be formally defined as follows. The input is the original raw graph denoted by $G$. Given a graph $G$, we partition the nodes into $k$ groups. The set of nodes that are assigned into the $p$-th partition for $1 \leq p \leq k$ is denoted by $I^{(p)}$. The subgraph or block induced by $p$-th source partition and $q$-th destination partition is denoted as $G^{(p,q)}$. The sets $I^{(p)}$ partition the nodes, in the sense that $I^{(p)} \cap I^{(q)} = \emptyset$ for $p \neq q$, while $\bigcup_p I^{(p)} = \{1, \ldots, n\}$. In terms of storage, the objective is to find the optimal $k$ partitions which lead to smallest total storage cost of all blocks/subgraphs $G^{(p,q)}$ where $1 \leq p, q \leq k$. Intuitively, we want the induced subgraphs to be homogeneous (meaning the subgraphs are either very dense or very sparse), which captures not only community structure but also leads to small storage cost.

For many real graphs, the community/clustering structure can be naturally identified. For instance, in Web graphs, the lexicographic ordering of the URL can be used as an indicator of community [10] since there are usually more intra-domain links compared with the inter-domain links. For authorship network, the research interest is often a good indicator to find communities since authors with the same or similar research interest tend to have more collaborations. For patient-doctor graph, the patient information (e.g., geography, disease type, etc) can be used to find the communities (patients with similar disease and living in the same neighborhood have higher chance to visit the same doctor).

3.3 Block Compression

The homogeneous block representation provides a more compact representation of the original graph. It enables us to encode the graph in a more efficient way. The encoding of a block $G^{(p,q)}$ consists of the following information:

- source and destination partition ID $p$ and $q$;
- the set of sources $I^{(p)}$ and the set of destinations $I^{(q)}$;
- the payload, the bit string of subgraph $G^{(p,q)}$.

A naive way of encoding a block is raw block encoding which only stores the coordinates of the non-zero entries in the block. Although this method saves the storage space since the nonzero elements within the block can be encoded with a smaller number of bits ($log_{2}(max(I^{(p)}, I^{(q)}))$) than the original, the savings are not great.

To achieve better compression, we propose zip block encoding which converts the adjacency matrix of the subgraph into a binary string and stores the compressed string as the payload. Compared to the raw block encoding, the zip block encoding requires more cpu time to zip and unzip blocks. However, the storage savings and the reduced data transfer size help to improve performance of GBASE as we will see in Section 5.

For example, we have the following adjacency matrix of a graph:

$$ G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} $$

raw block encoding will just store the non-zero coordinates (0,0), (1,0), (2,1), and (2,2) as the payload. Zip block encoding will converts the matrix into a binary string 110,001,001 (in the column major order) and then use the compression of this string as the payload.

Storage Estimation. The storage needed for raw block encoding is $2 \times n^{(p,q)} \times \log_{2}(max(I^{(p)}, I^{(q)}))$. The storage needed for zip block encoding is $l^{(p,q)} H(d)$, where $d = \frac{n^{(p,q)}}{\log_{2}(max(I^{(p)}, I^{(q)}))}$ is the density of $G^{(p,q)}$. $H(\cdot)$ is the Shannon entropy function: $H(X) = -\sum x p(x) \log p(x)$ where $p(x)$ is the probability that $X = x$. Note that the number of bits to encode an edge in zip block encoding decreases as $d$ increases, while it is constant in raw block encoding.

3.4 Block Placement

After we have the blocks, we need to store/place them in the file system (e.g., HDFS of HADOOP, relational DB). Here, the main idea is to place several blocks together into a file, and select only relevant files as inputs in the query stage. The question is, how do we place blocks into files? A typical approach is to use vertical placement to place the vertical blocks in a file as shown in Figure 2(a). The other alternative is to use horizontal placement to place the horizontal blocks in a file as shown in Figure 2(b). However, both of the placement techniques are good only for one type of query: for example, horizontal and vertical placement is good for out-neighbor and in-neighbor queries, respectively.

To solve the problem, GBASE uses the grid placement as shown in Figure 2(c). The advantage of the grid placement is that it can answer various types of queries efficiently as we will see in Section 4. Suppose we store all the compressed blocks in $K$ files. With vertical/horizontal placement, we need $O(K)$ file accesses to find the in- and out-neighbors of a given query node. In contrast, we only need $O(\sqrt{K})$ file accesses with grid placement.
4. HANDLING GRAPH QUERIES

In this section, we describe query execution in GBASE. GBASE supports both “global” queries, as well as “targeted” queries for one or a few specific nodes. The answer to global queries requires traversal of the whole graph, like, e.g., diameter estimation. In contrast, “targeted” queries need to access only parts of the graph. GBASE supports seven different queries including neighborhoods, induced subgraphs, egonets, K-core, and cross-edges.

4.1 Global Queries

Global queries are performed by repeated joins of edge blocks and vector blocks. GBASE supports the following graph queries: degree distribution, PageRank, RWR (“Random Walk with Restart”), radius estimations, and discovery of connected components. Our contribution here is that our proposed storage and compression scheme reduce the graph storage significantly, and enable faster running time as shown in Figure 5. The global queries also serve as primitives for targeted queries (see ‘T6: K-core’ in Section 4.2), enabling a variety of applications as shown in Table 1.

4.2 Targeted Queries

Many graph mining operations can be unified as matrix-vector multiplication. Here that matrix is either the adjacency matrix $A$ of size $n \times n$ or the incidence matrix $B$ of size $m \times n$ where $n$ and $m$ are the number of nodes and edges in the graph, respectively. Each row of the incidence matrix corresponds to an edge, and it has two non-zeros whose column ids are the node ids of the edge.

The matrix-vector multiplication observation has the extra benefit that it corresponds to an SQL join. Thus, graph mining could use all the highly optimized join algorithms in the literature (hash join, indexed join etc), while still leverages the proposed block compression storage scheme.

In fact, for each of the upcoming primitives, we shall first give the matrix-vector details, and then the SQL code.

T1: 1-step neighbors.

The first query is to find 1-step in-neighbors and out-neighbors of a query node $v$.

**Matrix-Vector version**

Given a query node $v$, its 1-step in-neighbors can be found by the following matrix-vector multiplication:

$$in^1(v) = A \times ev_v$$

where the matrix is the adjacency matrix of the graph $A$ and the vector is the ‘indicator vector’ $ev_v$ which is the $n$-vector whose $v$-th element is 1, and all other elements are 0s. The 1-step in-neighbors of the query node $v$ are those nodes whose corresponding values in $in^1(v)$ are 1s.

The 1-step out-neighbors can be obtained in the similar way by replacing $A$ with its transpose $A^T$.

**SQL version**

We can also find 1-step in-neighbors and out-neighbors by the standard SQL. Assume we have a table $E(src, dst)$ storing the edges, with attributes ‘source’ ($src$) and ‘destination’ ($dst$). The 1-step out-neighbors of a query node ‘$q$’ are given by

```
SELECT dst
FROM E
WHERE src='q'
```

without even requiring a join. 1-step in-neighbor can be answered in a similar way.

**T2: K-step neighbors.**

The next query is to find ‘within k-step’ neighbors. Let us only consider the $k$-step in-neighbors. $k$-step out-neighbors can be found in similar way - we only need to replace the matrix $A$ by its transpose $A^T$ in the matrix-vector multiplication version; and switch $src$ and $dst$ in the SQL version.

**Matrix-Vector version**

The $k$-step in-neighbors $nh^k(v)$ of the query node $v$ is defined recursively by $(k-1)$-step neighbors $nh^{k-1}(v)$ in terms of matrix-vector multiplication as follows:

$$nh^k(v) = A \times nh^{k-1}(v),$$

where the 0-step in-neighbors $nh^0(v)$ is just the indicator vector $ev_v$. After the $k$ multiplications, the $k$-step in-neighbors are those nodes whose corresponding values in $nh^k(v)$ or $nh^{k-1}(v)$ are 1s.

**SQL version**

As before, assume we have a table $E$ with attributes $src$ and $dst$. The $k$-step in-neighbors can also be found by SQL join. In general, the $k$-step in-neighbors is a $(k-1)$-way join. For example, the 2-step in-neighbors of a query node ‘$q$’ is given by the following SQL join:

```
SELECT E2.src
FROM E as E1, E as E2
WHERE E1.dst='q'
AND E1.src = E2.dst
```

**T3: Induced subgraph.**

Given a set of nodes $V_q$ in a graph $G$, the induced subgraph is defined to be a graph whose nodes are $V_q$ and an edge between two nodes $v_1$ and $v_2$ exist if they are adjacent in $G$.

**Matrix-Vector version**

Let $B$ be the $m \times n$ incidence matrix where $m$ and $n$ are the number of edges and nodes of the graph, respectively. Let $ev_{eq}$ be the $n$-vector, whose corresponding elements for $V_q$ are 1s, and 0s otherwise.

Then, the induced subgraph $S(V_q)$ from $V_q$ is expressed by the following matrix-vector multiplication:

$$S(V_q) = B \times ev_q,$$

where the resulting vector $S(V_q)$ is $m$-vector and the elements in $S(V_q)$ have values of 0, 1, or 2. The induced subgraph is given by those edges whose corresponding values in $S(V_q)$ are 2s since it means that the incident nodes (both the source and the target) of the edges are in $V_q$.

**SQL version**

Assume we have an incidence matrix as table $B$, with attributes $eid$, $srcid$, and $dstid$, representing the edge id, the source node id, and the destination id of a row in the incident matrix, respectively. Also assume we have a query vector table $Q$ with an attribute $nodeid$. Then the induced subgraph is given by the following join:

```
SELECT B.eid, B.srcid, B.dstid
FROM B, Q as Q1, Q as Q2
WHERE B.srcid=Q1.nodeid
AND B2.dstid=Q2.nodeid
```

**T4: 1-step egonet.**

Informally, the 1-step-away egonet (or just ‘egonet’) of a node $v$ is its 1-step-away vicinity. Formally, it is defined as the induced subgraph that includes $v$ and its 1-step neighbors. Extracting the egonet of a query node $v$ is a special case of extracting induced subgraph. That is, the set of nodes $V_q$ is defined to be the $v$ and its 1-step in-neighbors and out-neighbors.

The details are omitted, since we can combine earlier expressions (for both the matrix-vector case, as well as for the SQL case).
**T5: K-step egonet.**

K-step egonet of a node \( v \) is defined to be the induced subgraph from \( v \) and its within-\( k \) step neighbors. Extracting the k-step egonet of a query node \( v \) is also a special case of extracting induced subgraph. That is, the set of nodes \( V_k \) is defined to be the \( v \) and its within-\( k \) step neighbors. Thus, the same expression for the \( k \)-step neighbors and the induced subgraph can be used for extracting \( k \)-step egonet.

**T6: K-core.**

K-core of a graph is a maximal connected subgraph in which all vertices have degree at least \( K \). K-core is useful for finding communities and visualizing graphs. Although it seems complicated at first, all K-cores of a large graph can be enumerated by GBASE using primitives defined before:

1. Compute degrees of all nodes. Let \( C \) be the set of nodes with degree \( \geq K \).
2. Compute induced subgraph \( G' \) using \( C \).
3. Find connected components of \( G' \). The resulting components are the K-core.

**T7: Cross-edges.**

Given two disjoint sets \( V_1 \) and \( V_2 \) of nodes, how can we find the cross edges connecting the two sets? Cross-edges are useful for visualizing the interaction of two distinct sets of nodes, as well as anomaly detection (e.g., a set of nodes having few edges to the rest of the world are suspicious). Cross-edges can be computed by GBASE using induced subgraph queries:

1. Computed induced subgraphs \( S(V_1), S(V_2), S(V_1 \cup V_2) \) using nodes in \( V_1, V_2 \), and \( (V_1 \cup V_2) \), respectively.
2. Let \( E_1, E_2, \) and \( E_{12} \) be the set of edges in \( S(V_1), S(V_2), \) and \( S(V_1 \cup V_2) \), respectively. The cross edges are exactly the edges in \( E_{12} - (E_1 - E_2) \).

**4.3 Query Execution Engine**

We describe the query execution engine of GBASE built on the top of HADOOP, an open source implementation of MAPREDUCE which is a distributed large scale data processing platform.

**Overview.**

As described in previous sections, the main operation of GBASE is the matrix-vector multiplication. GBASE handles queries by executing appropriate block matrix-vector multiplication modules. The global queries are typically handled by multiple matrix-vector multiplications since the answer to the queries is often a fixed point of the multiplication (e.g., the first eigenvector in case of PageRank). The local queries require one or few multiplications.

Most of the operations require the adjacency matrix of the graph. Thus, GBASE uses the adjacency matrix directly as its input. However, some operations including the induced subgraph require the incidence matrix which is different from the adjacency matrix. We will see how to handle the queries requiring incidence matrix efficiently at the end of this subsection.

**Grid Selection.**

Before running the matrix-vector multiplication, GBASE selects the grids containing the blocks relevant to the queries. Only the files corresponding to the grids are fed into HADOOP jobs that GBASE executes. For global queries, we need to select all the grids since all the blocks are relevant. For targeted queries, however, we can select only relevant grids. For in-neighbor queries, we select grids whose column range contains the query node as shown in Figure 3(a). For out-neighbor queries, we select grids whose row range contains the query node as shown in Figure 3(b). For egonet queries, we select grids whose row or column range contains the query. As we will see in Section 5, this grid selection has advantages of decreasing the running time.

**Handling Incidence Matrix Queries.**

While the majority of operations use the adjacency matrix, the induced subgraph queries use the incidence matrix. Thus, GBASE need to access the incidence matrix to support the queries. A naive approach is to build the incidence matrix \( B_{m \times n} \) by numbering edges sequentially. However, it requires the storage to save \( B \) which is twice the size of the original adjacency matrix. The question is, can we answer incidence matrix queries efficiently without the additional storage?

Our proposed main idea is to derive the incidence matrix from the original adjacency matrix as required. That is, an adjacency matrix element \((src, dst)\) can be interpreted as \(([src, dst], src)\) and \(([src, dst], dst)\) of the incidence matrix where \([src, dst]\) is the edge id. Thus, the query execution algorithm for handling incidence matrix can work on the original adjacency matrix by treating each adjacency matrix element as two incidence matrix elements.

The HADOOP algorithm for the induced subgraph, which reflect the main idea, is shown in Algorithm 1. The algorithm is composed of two stages. In the first stage, the elements in the incidence matrix and the query vector are grouped together to generate partial results. Notice that two incidence matrix elements are generated (line 6,7 of Algorithm 1) for an adjacency matrix element. In the second stage, the partial results are summed to get the final result. Note that only edges having the sum 2 are included in the egonet since it means that the two incidence nodes of the edges are contained in the query node set.

5. EXPERIMENTS

To evaluate our GBASE system, we perform experiments to answer the following questions:

- **Q1** How much does our zip block encoding reduce the data size?
- **Q2** How do our algorithms scale up with the graph sizes and the number of machines?
- **Q3** How do our indexing and query execution methods save query response time?

**Datasets.** We use large graph datasets summarized in Table 3. The YahooWeb dataset is a web graph from Yahoo! with 1.4 billion edges.
nodes, 6.6 billion edges, and 120GB in space. YahooWeb is one of the largest real graph which helps us test the scalability of the GBASE system on real workload. In order to show the performance across different data scales, we use two synthetic data generators: Kronecker [23] and Erdős-Rényi [14] to generate multiple realistic graphs with different sizes.

**Storage Schemes.** We use the following notations to distinguish different storage and indexing methods:

- **GBASE RAW** (original RAW encoding): raw encoding which is the original adjacency matrix format.
- **GBASE NNB** (No clustering, No compression, Blocking): raw block encoding without compression and clustering.

**Algorithm 1: HADOOP algorithm for Induced Subgraph**

```plaintext
Algorithm 1: HADOOP algorithm for Induced Subgraph

Input: Edge \( E = \{(src, dst)\} \) of a graph \( G = (V, E) \), Query Node Set \( V_q = \{nodeid\} \)
Output: Edges belonging to the subgraph induced from \( V_q \)

1 InducedSubgraph-Map1(Key k, Value v):
2 begin
3 if \((k, v)\) is of type \( E \) then
4 \((src, dst) <- (k, v)\);
5 // Emit incidence matrix elements
6 Output(src, [src, dst]);
7 Output(dst, [src, dst]);
8 else if \((k, v)\) is of type \( V \) then
9 \((nodeid) <- (k, v)\);
10 Output((nodeid, '1'));
11 end
12 end

13 InducedSubgraph-Reduce1(Key k, Value v[1..r]):
14 begin
15 if \(v[]\) contains '1' then
16 Remove '1' from \(v[]\);
17 foreach \(p \in v[1..r - 1]\) do
18 \([src, dst] <- p\);
19 // Emit partial multiplication result
20 Output([src, dst], 1);
21 end
22 end
23 end
24 InducedSubgraph-Map2(Key k, Value v);
25 begin
26 Output(k, v); // Identity Mapper
27 end
28 InducedSubgraph-Reduce2(Key k, Value v[1..r]):
29 begin
30 sum = 0;
31 foreach num \(\in v[1..r]\) do
32 \(sum = sum + num\);
33 end
34 // Select edges whose incident nodes belong to the query node set
35 if \(sum=2\) then
36 \([src, dst] <- k\);
37 Output(src, dst);
38 end
39 end
```

Kronecker, Erdős-Rényi: http://www.cs.cmu.edu/~ukang/dataset

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>YahooWeb</td>
<td>1.413M</td>
<td>6.636M</td>
<td>0.12 TB</td>
</tr>
<tr>
<td>Kronecker</td>
<td>177 K</td>
<td>1.977 M</td>
<td>25 GB</td>
</tr>
<tr>
<td></td>
<td>120 K</td>
<td>1.145 M</td>
<td>13.9 GB</td>
</tr>
<tr>
<td></td>
<td>59 K</td>
<td>282 M</td>
<td>3.3 GB</td>
</tr>
<tr>
<td></td>
<td>20 K</td>
<td>6 M</td>
<td>439 MB</td>
</tr>
<tr>
<td>Erdős-Rényi</td>
<td>177 K</td>
<td>1.977 M</td>
<td>25 GB</td>
</tr>
<tr>
<td></td>
<td>120 K</td>
<td>1.145 M</td>
<td>13.9 GB</td>
</tr>
<tr>
<td></td>
<td>59 K</td>
<td>282 M</td>
<td>3.3 GB</td>
</tr>
<tr>
<td></td>
<td>20 K</td>
<td>6 M</td>
<td>439 MB</td>
</tr>
</tbody>
</table>

**Figure 4: Effect of different encoding methods for GBASE.**

KR-2B: Kronecker graph with 2 billion edges. ER-2B: Erdős-Rényi random graph with 2 billion edges. Notice our proposed zip block encoding(NZB) decreases the input sizes significantly, reducing to 50\times smaller than the original(RAW). The Kronecker and the Erdős-Rényi graphs have better performance gain than the YahooWeb graph since the first two are denser than the last and thus take advantage of the compression. The Kronecker graph has better compression than the Erdős-Rényi graph since it has a block-like structure from the construction.

- **GBASE NZB** (No clustering, Zip compression, Blocking): zip block encoding without clustering.
- **GBASE CZB** (Clustering, Zip compression, Blocking): zip block encoding with clustering.
- **GBASE CZB+GS** (CZB with Grid Selection): grid selection as described in Section 4.3.

We deploy our GBASE HADOOP implementation onto the M45 HADOOP cluster by Yahoo!. The cluster has total 480 machines with 1.5 Petabyte total storage and 3.5 Terabyte memory.

5.1 Space Efficiency Comparison

We show the data size over three different graphs across different storage schemes in Figure 4: 1) KR-2B is a graph of 177K nodes and about 2 billion edges generated using Kronecker generator; 2) ER-2B is of the same size as KR-2B but generated by Erdős-Rényi generator; 3) YahooWeb is the real Yahoo web graph of 1.4 billion nodes and 6.6 billion edges. We have the following observations:

**Size Reduction.** The zip block encoding(NZB) reduces the raw data size significantly (50\times, 9\times, and 3.6\times smaller than the original(RAW) size for Kronecker, Erdős-Rényi, and YahooWeb graphs, respectively). In contrast, the raw block encoding(NNB) decrease the size at most 2.3\times smaller than the original(RAW).

**Density and Compression.** The zip block encoding compression ratio is better for the dense graphs (Kronecker and Erdős-Rényi) than the sparse YahooWeb graph. The reason is that the
of nodes and edges since the Erdős-Rényi graph has nonzeros randomly distributed in the adjacency matrix.

**Running Time.** We see that CZB, which combines the clustering and the zip block encoding, performs the best. It outperforms RAW, NNB, NZB by $14 \times$, $4.6 \times$, and $2.6 \times$, respectively, for 10 machines. The main reason of the better performance is the decreased I/O time due to reduced storage.

**Machine Scalability.** All the methods scale up near-linearly with the number of machines as we see in Figure 6(a).

**Edge Scalability.** The methods also scale up near-linearly with the number of edges as we see in Figure 6(b).

### 5.4 Targeted Query Time

We show the performance on targeted queries in Figure 6(c). Since the targeted queries are often against a small subset of the data, increasing the number of machines does not matter here for improving an individual query. Therefore, we only demonstrate the result with fixing the number of machines to 100. All the experiments report the average running time of 5 randomly selected query nodes. The node ids in the YahooWeb graph is encoded in a clustered manner since all the pages in a domain are numbered sequentially. Thus, we use the zip blocked YahooWeb graph for the CZB experiment.

**Grid Selection.** We see that GBASE CZB+GS, which combined the clustering, the zip block encoding, and the grid selection, works the best for all the targeted queries. Especially, it works the best for 1-neighborhood query outperforming all other competitors from $1.6 \times$ to $4 \times$. The reason is that the grid selection method works better if the portion of the relevant grids is small. For 1-neighborhood query, the portion is the smallest ($\sqrt{K}$ for total $K$ grids), while other queries can have many relevant grids depending on the number of neighbors of the query node.

**Effect of Zip Block Encoding.** The clustered zip block encoding(CZB) performs slightly better than the raw block encoding(NNB) for 1-neighborhood and egonet queries, while it worked slightly worse than NNB for the 2-neighborhood query. The reason is that the size gain of the zip block encoding in CZB is not big enough to overshadow the increased running time for the zip compression. However, the performance of zip block encoding will continuously increase as better clustering algorithm is developed, as shown in the well clustered graph results of Figure 6(a,b). Moreover, the zip block encoding enjoys additional benefits of less storage and indexing time.

### 6. RELATED WORK

In this section, we review the related work, which can be categorized into four parts: (1) graph indexing techniques, (2) graph queries, (3) column store and (4) parallel data management.

**Graph Indexing.** Graph indexing is very active in both databases community as well as data mining community in the recent years. To name a few, Štrbl et al [36] proposed to index the graph using pre- and postorder number to answer the reachability queries. Chierichetti et al [12] explored link reciprocity for adjacency queries. Aggarwal et al [5] proposed using edge sampling to handle graph connectivity queries. Sarkar et al [32] explored the clustering properties to proximity queries on graphs. Maserra et al [27] proposed a Eulerian data structure for neighborhood queries. Despite of their success, there are two major limitations of these work. First, all the indexing techniques are designed for one particular type of queries. Therefore, their performance might be highly optimized for that particular type of query, but they are far sub-optimal for the remaining, vast majority types of queries. Second, they are implicitly designed for the centralized computa-
Figure 6: (a,b): Running time and scalability in one iteration of global queries in GBase on Kronecker and Erdős–Rényi graphs. The CZB method which combines the clustering and the zip block encoding outperforms the RAW method by 14×. Notice also that all the methods scale up near-linearly on the number of machines and edges. (c): Running time of targeted queries over different storage and indexing methods, on YahooWeb graph. K-Nh denotes K step neighborhood query. Note that the CZB+GS(grid selection method combined with the clustered zip block encoding) outperforms the others by 4× at maximum.

7. CONCLUSION

In this paper, we propose GBase, a scalable and general graph management system. The main contributions are the followings.

1. Storage. We carefully design GBase to efficiently store homogeneous regions of graphs in distributed settings using a novel ‘block compression’. Experiments on billion-scale graphs show that the storage and running time reduced up to 50× of the original.

2. Algorithms. We unify node-based and edge-based queries using matrix-vector multiplications on the adjacency and the incidence matrices. As a result, we get seven different types of versatile graph queries supporting various applications.

3. Query Optimization. We propose a fast graph query execution algorithm using a grid selection. Also, we provide an efficient MapReduce algorithm to support incidence matrix based queries using the original adjacency matrix, without explicitly building the incidence matrix.

Researches on large graph mining can benefit significantly from GBase’s efficient storage, widely applicable primitive operations, and fast query execution engine. Future research directions include query optimization for multiple, heterogeneous queries, and better support for time evolving graphs.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grants No. IIS-0705359 and IIS0808661, by the Defense Threat Reduction Agency under contract No. HDTRA1-10-1-0120, and by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. This work is also partially supported by an IBM Faculty Award, and the Gordon and Betty Moore Foundation, in the eScience project. The views and conclusions contained in this document are those of the authors and should not be interpreted as...
8. REFERENCES