The VLDB Journal
DOI 10.1007/s00778-012-0283-9

SPECIAL ISSUE PAPER

GBASE: an efficient analysis platform for large graphs

U Kang · Hanghang Tong · Jimeng Sun · Ching-Yung Lin · Christos Faloutsos

Received: 15 August 2011 / Revised: 24 April 2012 / Accepted: 29 May 2012

Abstract Graphs appear in numerous applications including cyber security, the Internet, social networks, protein networks, recommendation systems, citation networks, 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 graph? How to answer the graph queries quickly? We propose GBASE, an efficient analysis platform for large graphs. The key novelties lie in (1) our storage and compression scheme for a parallel, distributed settings and (2) the carefully chosen graph operations and their efficient implementations. We designed and implemented an instance of GBASE using MAPREDUCE/HADOOP. GBASE provides a parallel indexing mechanism for graph operations that both saves storage space, as well as accelerates query responses. We run numerous experiments on real and synthetic 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.

Keywords Graph · Indexing · Compression · Distributed computing

U Kang (✉) · C. Faloutsos
Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: ukang@cs.cmu.edu

C. Faloutsos
e-mail: christos@cs.cmu.edu

H. Tong · J. Sun · C.-Y. Lin
IBM T. J. Watson, Yorktown Heights, NY, USA
e-mail: htong@us.ibm.com

J. Sun
e-mail: jimeng@us.ibm.com

C.-Y. Lin
e-mail: chingyung@us.ibm.com

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

Published online: 16 June 2012

graphs exist: the Twitter graph spans several Terabytes; clickstreams are reported to reach Petabyte scale [1]. 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 Terabytes 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 'compressed block encoding' to efficiently store homogeneous regions of graphs based on adjacency matrix representation. 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 43× less storage and 9.2× faster running time compared with the naive algorithm.
2. *Algorithms*. We identify a core graph operation, and use it to formulate *eleven* different types of graph queries including neighborhood, induced subgraph, egonet, K-core, cross-edges, and single source shortest distances. The novelty is in formulating edge-based queries (induced subgraph) as well as node-based queries (neighborhoods) using a unified framework.
3. *Query optimization*. We propose a grid selection strategy to minimize disk accesses and answer queries quickly. We also propose a MAPREDUCE [2] algorithm to support incidence matrix-based queries using the original adjacency matrix, without explicitly building the incidence matrix.

The rest of this paper is organized as follows. We first present the overall framework in Sect. 2. We describe the storage and indexing method in Sect. 3, and then the query execution in Sect. 4. We provide experimental evaluations and comparisons in Sect. 5. After reviewing the related work in Sect. 6, we conclude in Sect. 7.

2 Overall framework

The overall framework of our GBASE is summarized in Fig. 1. The design objective is to balance storage efficiency and query performance on large graphs. It comprises 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 Sects. 3 and 4, respectively.

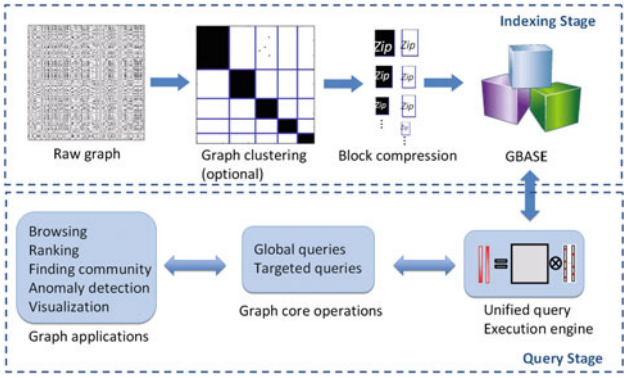


Fig. 1 Overall framework of GBASE. *1* Indexing Stage: raw graph is clustered and divided into compressed blocks. *2* Query Stage: global and targeted queries from various graph applications are handled by a unified query execution engine

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, GBASE reshuffles the nodes so that the nodes belonging to the same partition are put nearby. Third, GBASE compresses all non-empty blocks through standard compression algorithms such as Gzip and Elias-γ. Finally, the compressed blocks, together with some meta information (e.g., the block row id and column id), are stored 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 Fig. 1) or very sparse (e.g., the off-diagonal blocks in Fig. 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 Sect. 5) show that in some cases, we only need about 2 % storage space of the original after the indexing stage. Note that in this work, we are focusing on a static graph, and thus, we leave the update of the graphs as a future work.

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, for example, ranking, community detection, and anomaly detection. The key of the online 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 Sect. 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 [4]) scores among the neighborhood of a given edge/node is a strong indicator of abnormality of that node/edge [7]. The ratio

Table 1 Applications of GBASE

Query	Applications				
	Browsing	Ranking	Finding community	Anomaly detection	Visualization
Connected comp.			✓	✓	
Radius				✓	✓
PageRank, RWR	✓	✓		✓	
LineRank	✓	✓		✓	
Induced subgraph	✓		✓		✓
(K)-Neighborhood	✓		✓		✓
(K)-Egonet	✓		✓	✓	✓
K-core			✓		✓
Cross-edges				✓	✓
Single source shortest distances				✓	

Notice that GBASE answers wide range of both global (top 4 rows) and targeted queries (bottom 6 rows with bold fonts) with applications in browsing [3–5], ranking [3,4], finding communities [5,6], anomaly detection [6–9], and visualization [5,10]

Table 2 Definitions of symbols

Symbol	Definition
$\mathcal{G}$	Graph
A	Adjacency matrix of the graph $\mathcal{G}$
B	Incidence matrix of the graph $\mathcal{G}$
n	Number of nodes
m	Number of edges
k	Number of partitions
p, q	Partition indices, $1 \leq p, q \leq k$
$I^{(p)}$	Set of nodes belonging to the pth partition
$l^{(p)}$	Partition size, $l^{(p)} \equiv \lvert I^{(p)} \rvert$, $1 \leq p \leq k$
$\mathcal{G}^{(p,q)}$	Subgraphs induced by pth and qth partitions
$m^{(p,q)}$	Number of edges in $\mathcal{G}^{(p,q)}$
$H(.)$	Shannon entropy function

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 [9]. The K-cores and cross-edges can be used for visualization and finding communities in large graphs.

3 Graph storage and indexing

In this section, we describe in detail the indexing and storage stage of GBASE. We use the symbols in Table 2.

3.1 Baseline storage scheme

A typical way to store the raw graph is to use the adjacency list format: for each node, it saves all the out-neighbors adjacent from the node. The adjacency list format is simple and might be good for answering out-neighbor queries. However, it is not an efficient format for answering general queries including in-neighbor queries and ego-net queries; for example, answering the in-neighbor of a query requires reading all the edges, which is not efficient. For the reason, we instead use the sparse adjacency matrix format, where we save each edge by a (source,destination) pair. Note that we only store nonzero elements of a matrix, and does not store empty elements. 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 section.

The storage system should be designed to be efficient in both storage cost and online 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, that is, 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, for example, METIS [11], Disco [12], Shingle [13], SlashBurn [14], etc., can be naturally plugged into GBASE.

Graph partitioning can be formally defined as follows. The input is the original raw graph denoted by $\mathcal{G}$. Given a graph $\mathcal{G}$, we partition the nodes into k groups. The set of nodes that are assigned into the pth partition for $1 \leq p \leq k$ is denoted by $I^{(p)}$. The subgraph or block induced by p-th source partition and qth destination partition is denoted as $\mathcal{G}^{(p,q)}$. The sets $I^{(p)}$ partition the nodes, in the sense that $I^{(p)} \cap I^{(p')} = \emptyset$ for $p \neq p'$, 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

$\mathcal{G}^{(p,q)}$ 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 example, a graph containing two cliques connected by an edge can be partitioned into two groups where each group contains all the nodes in a clique.

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 [15] 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 for finding 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 and disease type) 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 $\mathcal{G}^{(p,q)}$ 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 $\mathcal{G}^{(p,q)}$.

A naive way of encoding a block is *raw block encoding* which only stores the coordinates of the nonzero 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 ($log(\max(l^{(p)}, l^{(q)}))$ than the original, the savings are not great as we will see in Fig. 4 at Sect. 5.

To achieve better storage savings, we propose *compressed block encoding* which converts the adjacency matrix of the subgraph into a binary string and stores the compressed string as the payload. Any optimal compression algorithm can be used for the compressed block encoding. Compared to the raw block encoding, the compressed block encoding requires more cpu time to compress and uncompress blocks. However, the storage savings and the reduced data transfer size help to improve performance of GBASE as we will see in Sect. 5. We give two examples of the compressed block encoding using different compression algorithms: zip compression and gap Elias-γ encoding.

Zip Compression. In the zip compression method, we apply the standard Gzip algorithm to compress the binary string representation of the adjacency matrix blocks. For example, for the following adjacency matrix of a graph:

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

raw block encoding will just store the nonzero coordinates $(0, 0)$, $(1, 0)$, $(2, 1)$, and $(2, 2)$ as the payload. Compressed block encoding using zip algorithm converts the matrix into a binary string 110, 001, 001 (in the column major order) and then run the Gzip algorithm to generate the payload.

Gap Elias-γ Encoding. In the gap Elias-γ encoding, we first compute the gaps between nonzero elements inside a block, and compress the gaps using Elias-γ encoding. Elias-γ encoding stores a number x using $1+2\lfloor log x \rfloor$ bits which is close to the information-theoretic minimum [13]. For example, the offsets of the nonzero elements of the matrix in Eq. (1) are 0, 1, 4, 3 in the column major order. These offsets are then encoded with Elias-γ to create the payload.

Storage estimation. The storage needed for raw block encoding is $2 * m^{(p,q)} * log(max(l^{(p)}, l^{(q)}))$ for each block. Using compression algorithms achieving the information-theoretic minimum cost asymptotically (e.g., zip compression and the gap Elias-γ encoding), the storage needed for compressed block encoding is $l^{(p)} l^{(q)} H(d^{(p,q)})$ for each block (see Eq. (1) of [16]), where $d^{(p,q)} = \frac{m^{(p,q)}}{l^{(p)} l^{(q)}}$ is the density of $\mathcal{G}^{(p,q)}$, and $H(\cdot)$ is the Shannon entropy function $H(X) = -\sum_x p(x) \log p(x)$ where $p(x)$ is the probability that $X = x$. The total storage needed for all the blocks in the compressed block encoding is given by

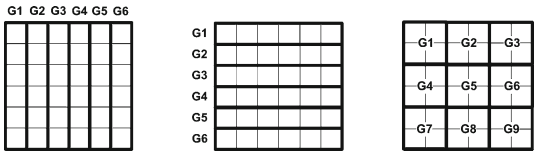
$$\sum_{1 \leq p,q \leq k} l^{(p)} l^{(q)} H(d^{(p,q)}). \tag{2}$$

Note that the number of bits to encode an edge in the compressed block encoding decreases as d increases, while it is constant in raw block encoding.

3.4 Block placement

After compressing 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 Fig. 2a. The other alternative is to use horizontal placement to place the horizontal blocks in a file as shown in Fig. 2b. However, both of the placement techniques are good only for one type of query: for example, horizontal and vertical placement are good for out-neighbor and in-neighbor queries, respectively.

To solve the problem, GBASE uses the grid placement, shown in Fig. 2c, which we demonstrate to be efficient for queries that access in-neighbors, out-neighbors, or both.



(a) Vertical Placement **(b)** Horizontal Placement **(c)** Grid Placement

Fig. 2 Adjacency matrices showing possible placement of blocks into files in HADOOP. The smallest rectangle represents a block in the adjacency matrix. The placement strategy determines which of the blocks are grouped into files G1 to G6 or G9. Vertical placement in **a** is good for in-neighbor queries, but inefficient for out-neighbor or egonet queries. Horizontal placement in **b** is good for out-neighbor queries, but inefficient for in-neighbor or egonet queries. GBASE uses the grid placement, shown in **c**, which is efficient for all types of queries

The advantage of the grid placement is that it minimizes the number of input files to answer queries. Suppose we store all the compressed blocks in K files. With the vertical and the horizontal placement, we need $O(K)$ file accesses to find the out- and in-neighbors of a given query node, respectively. In contrast, we need only $O(\sqrt{K})$ files accesses with grid placement. We will see this run-time query optimization in more detail at Sect. 4.3. We note that the parameter optimization for the grid placement (e.g., number of files, number of blocks per file) is left for a possible future work.

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, for example, diameter estimation. In contrast, "targeted" queries need to access only parts of the graph. GBASE supports eleven different queries including neighborhoods, induced subgraphs, egonets, K-core, cross-edges, and single source shortest distances.

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, discovery of connected components [6], and LineRank (i.e., PageRank on the line graph [17]). Our main contribution here is that our proposed storage and compression schemes reduce the graph storage significantly, and enable faster running time as shown in Sect. 5. The global queries also serve as primitives for targeted queries (see 'T6: K-core' in Sect. 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 the 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 nonzeros whose column ids are the node ids of the edge.

The matrix-vector multiplication observation has the extra benefit that it corresponds to a 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 most 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:

$$in^1(v) = A \times e_v, \tag{3}$$

where the matrix A is the adjacency matrix of the graph and the vector is the "indicator vector" e_v which is the n-vector whose vth 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 in 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-neighbors 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:

$$nh^k(v) = A \times nh^{k-1}(v), \tag{4}$$

where the 0-step in-neighbors $nh^0(v)$ is simply the indicator vector e_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 $\mathcal{G}$, 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 and only if they are adjacent in $\mathcal{G}$.

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 e_{vq} 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:

$$S(V_q) = B \times e_{vq}, \tag{5}$$

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_q 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 [10]. 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. Compute 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 $E_{12} - E_1 - E_2$.

T8: Single-source shortest distances. Given a query "source" node q, and the maximum path length k, the single-source shortest distances query finds the shortest path distances from q to the nodes reachable within k steps, where the maximum path length is limited by k.

Matrix-Vector version

Let d^k be an n-vector containing the answer to the query. We initialize d^0 by setting 0 for the element q, and ∞ for other elements. d^k is updated recursively from d^{k-1} as follows:

$$d^k = A \times d^{k-1}, \tag{6}$$

where the sub operations in the matrix-vector multiplication are redefined. In the standard matrix-vector multiplication, the ith element d_i^k of the vector d^k is determined by $d_i^k = \sum_j A_{ij} d_j^{k-1}$. Here we redefine the operations by $d_i^k = \min_j(A_{ij} + d_j^{k-1})$.

SQL version

Assume we have a table E with attributes src, dst, and val, representing source node id, destination node id, and weight of an edge, respectively. We also have a distance vector table V with attributes id and val, representing row id and the value of the element at the row, respectively. The next-step distance vector is computed by the following SQL statement:

```
SELECT E.src, MIN(sum2(E.val, V.val))
FROM E, V
WHERE E.dst=V.id
   GROUP BY E.sid
```

where $sum2$ is a UDF (user-defined function) which returns the sum of the two arguments.

4.3 Query execution engine

We describe the query execution engine of GBASE built on the top of HADOOP [18], an open source implementation of MAPREDUCE [2] 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.

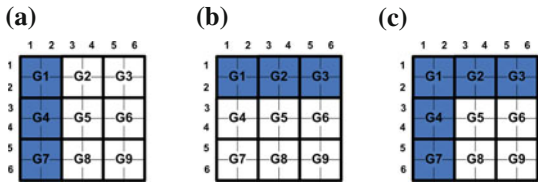


Fig. 3 Grid selection in 6 by 6 blocks where the query node belongs to the second block. The smallest *rectangle* corresponds to a block, and a *bigger rectangle* containing 4 blocks is a grid which is stored in a file. Notice that GBASE selects different grids based on the type of the query and the query node id. For example, GBASE selects G1, G4, and G7, instead of all the grids for in-neighbors query. This reduced input size results in the decreased running time. **a** 1-step in-neighbors, **b** 1-step out-neighbors, **c** 1-step in- and out-neighbors

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 Fig. 3a. For out-neighbor queries, we select grids whose row range contains the query node as shown in Fig. 3b. For in/out neighbors and egonet queries, we select grids whose row or column range contains the query. As we will see in Sect. 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 elements $([src, dst], src)$ 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 reflects 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.

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_q 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 ∈ 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 ∈ 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
```

5 Experiments

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

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

We first describe the experimental settings, and presents results which provide answers to the questions.

5.1 Experimental setting

Datasets. We use large graph datasets summarized in Table 3. The YahooWeb dataset is a web graph from Yahoo! with 1.4 billion nodes, 6.6 billion edges, and 120 GB in space. Twitter is a social network containing the "who follows whom" relationships. YahooWeb and Twitter are few of the largest real graphs which help us test the scalability of our GBASE system on real workload. LinkedIn is a social network containing friends relationships. Wikipedia is a document graph showing the links between articles. In order to show the performance across different data scales, we use two synthetic data generators: Kronecker [19] and Erdős-Rényi [20] to generate multiple 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.
- GBASE NCB (No clustering, Compression, Blocking): compressed block encoding without clustering.
- GBASE CCB (Clustering, Compression, Blocking): compressed block encoding with clustering.
- GBASE CCB+GS (CCB with Grid Selection): CCB with grid selection as described in Section 4.3.

For compression, we use gap Elias-γ encoding since it achieves higher compression rate than Gzip algorithm. To evaluate the effect of clustering, we use the following settings:

- YahooWeb: the original YahooWeb graph is already well clustered, since its nodes are lexicographically numbered, and thus, many intra edges within domains exist. For the reason, we use the original graph as the clustered graph. We randomly permuted the node ids of the original graph to generate the non-clustered graph.
- Twitter, LinkedIn, Wikipedia: we use the Shingle ordering [13] to cluster the graphs.
- Kronecker and Erdős-Rényi: since Kronecker graphs are highly clustered from its construction, we use the

Table 3 Order and size of networks

Graph	Nodes	Edges	File size	Description
YahooWeb	1,413 M	6,636 M	0.12 TB	Web graph snapshot at 2002
Twitter	104 M	3,730 M	83 GB	Twitter who follows whom at June 2010
LinkedIn	7.5 M	58 M	1 GB	Who connected to whom at 2006
Wikipedia	3.6 M	42 M	0.6 GB	document network at 2007
Kronecker	177 K	1,977 M	25 GB	Synthetic Kronecker graph
	120 K	1,146 M	13.9 GB	
	59 K	282 M	3.3 GB	
Erdős-Rényi	177 K	1,977 M	25 GB	Synthetic Erdős-Rényi graph
	120 K	1,146 M	13.9 GB	
	59 K	282 M	3.3 GB	

M: million, K: thousand
YahooWeb: http://webscope.sandbox.yahoo.com
Twitter: http://www.twitter.com
LinkedIn:underNDA
Wikipedia: http://en.wikipedia.org/wiki/Wikipedia:Database_download
Kronecker, Erdős-Rényi: http://www.cs.cmu.edu/~ukang/dataset

Kronecker graph as the clustered representation of the Erdős-Rényi graph. That is, we use Erdős-Rényi graph for RAW, NNB, and NCB, and Kronecker graph for CCB experiment: this graph is called "Random" in the experiments.

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.

5.2 Space efficiency comparison

We show the data sizes of real and synthetic graphs across different storage schemes in Fig. 4 and Table 4. We have the following observations:

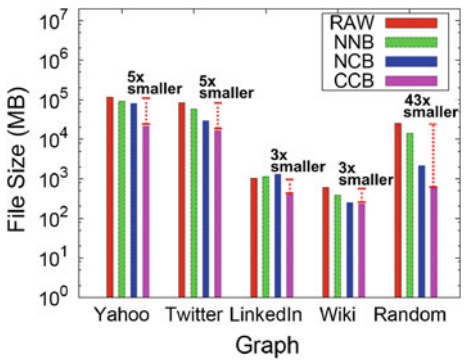


Fig. 4 Effect of different encoding methods for GBASE. The *Y*-axis is in log scale. "Yahoo" and "Wiki" denote the YahooWeb and the Wikipedia graphs, respectively. For "Random" graph, Erdős-Rényi graph is used for RAW, NNB, and NCB, and Kronecker graph is used for CCB experiment. Notice our proposed compressed block encoding on clustered graph (CCB) achieves the best compression, reducing up to 43× smaller than the original (RAW). The "Random" graph (Kronecker and the Erdős-Rényi) has better performance gain than real-world graphs since the density is much higher. The Kronecker graph has better compression than the Erdős-Rényi graph since it has a block-like structure from the construction

Size reduction. The raw block encoding (NNB), and the compressed block encoding without clustering (NCB) method reduce the data sizes at most 1.8× and 11.7×, respectively. However, for LinkedIn graph, they in fact increases the data sizes than the original. The reason of this increase in data size is that blocks from the original LinkedIn graph are very sparse, and thus, the storage overhead of the meta information (block row id, column id, etc.) outweighs the savings from the encodings. In contrast, our proposed clustered block encoding (CCB) method reduces the data size for all cases, achieving 43× storage savings at maximum.

Density and compression. The compressed block encoding compression ratio is better for the denser graphs ("Random") than the sparse real-world graphs (YahooWeb, Twitter, LinkedIn, and Wikipedia). The reason is that denser graphs lead to denser blocks which allow reduced bits per edge by compression algorithms.

Block structure and compression. For "Random" graphs, CCB achieves 43× savings while NCB achieves 11.7× savings. Note that CCB and NCB applied compressed block encoding on Kronecker and Erdős-Rényi graphs, respectively. The reason of Kronecker graph's better compression rate than Erdős-Rényi graph is that the Kronecker graph is block structured from the construction [19], and thus, it benefits the compression algorithm better than the Erdős-Rényi graph.

To summarize, compressed block encoding on clustered graph (CCB) has shown great space savings (up to 43×) across all datasets outperforming all competitors (RAW, NNB, NCB), which confirms the design objective of GBASE.

5.3 Indexing time comparison

So far, we have compared the resulting space efficiency of different methods. Next, we evaluate the indexing time required by each method. In Fig. 5, we show the running time of GBASE indexing process vs. the number of edges for graphs generated by both Kronecker (KR) and Erdős-Rényi (ER) generators. We use 200 machines.

Table 4 Effect of different encoding methods for GBASE

Graph	RAW	NNB	NCB	CCB	$\frac{\text{RAW}}{\text{CCB}}$
YahooWeb	116,518,939,878	90,927,149,331	80,271,934,486	21,327,750,493	5×
Twitter	83,156,286,766	58,092,370,573	29,718,767,360	16,522,432,151	5×
LinkedIn	1,036,553,688	1,138,819,798	1,320,553,373	397,694,425	3×
Wikipedia	604,698,633	378,842,305	250,924,451	224,976,600	3×
Random	25,199,902,253	14,121,708,962	2,148,362,975	584,395,280	43×

The numbers show the graph sizes in bytes. Note our proposed compressed block encoding on clustered graph (CCB) achieves the best compression, leading up to 43× smaller storage than the original (RAW)

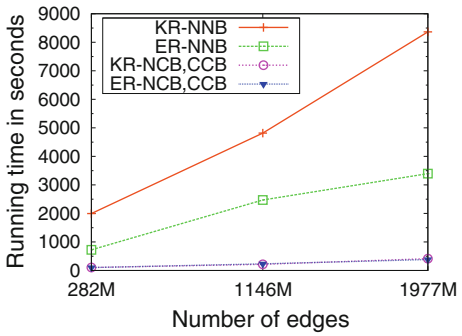


Fig. 5 Scalability of indexing in GBASE. KR-NNB: Kronecker graph with raw block encoding. ER-NNB: Erdős-Rényi graph with raw block encoding. KR-NCB,CCB: Kronecker graph with compressed block encoding. ER-NCB,CCB: Erdős-Rényi graph with compressed block encoding. Notice that the indexing time is linear on the number of edges. Also notice that the compressed block encoding is up to 22× faster than the raw block encoding, since the output size is smaller

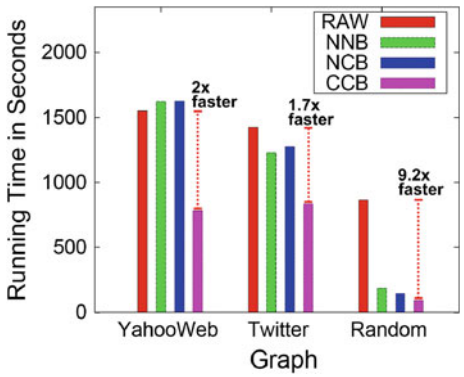


Fig. 6 Running time comparison of global (PageRank) queries over different storage methods, using 100 machines. We use two largest real-world graphs (YahooWeb and Twitter), and two synthetic graphs (Kronecker and Erdős-Rényi graphs which are called "Random"). The CCB method, which combines the clustering and the compressed block encoding, performs the best, outperforming RAW method up to 9.2×. Note that the time savings rates are smaller than the storage savings rates shown in Fig. 4 and Table 4, due to the additional cpu time for decoding compressed blocks

Running time. Compressed block encoding (NCB and CCB) requires much less time than raw block encoding (NNB), despite the additional compression step: NCB and CCB perform 22.4× and 20.3× faster than NNB for 1145M and 1977M edges, respectively. The reason is that the resulting compressed blocks are much smaller than those from the raw block encoding without compression. Thus, the running time for writing the compressed blocks to disks is much smaller than in the case of the uncompressed block. Also note that the KR-NNB takes longer time than ER-NNB. The reason is that some blocks in Kronecker graphs are very dense, while in Erdős-Rényi graphs all the blocks are sparse. The dense blocks in Kronecker graphs result in long encoding time, and it increases the total running time since the running time of a MAPREDUCE job is bounded by the longest mapper or reducer time.

Linear scalability. The indexing times for both compressed (NCB and CCB) and raw block encoding (NNB) increase linearly as the number of edges for both Kronecker and Erdős-Rényi graphs. This confirms the scalability of our encoding schemes.

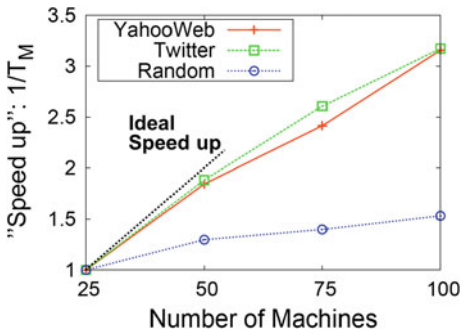


Fig. 7 Machine scalability of our proposed CCB method. The Y-axis shows the ratio of the running time T_M with M machines, and T_{25}, for PageRank queries. Note the speed-up grows near-linear to the number of machines

5.4 Global query time

So far, we confirmed the scalability and efficiency of the indexing phase. Next we evaluate the performance of different schemes on the query phase. Here, we show the running time and the scalability of GBASE global queries in Figs. 6, 7, and 8. We choose to run the PageRank query, since PageRank

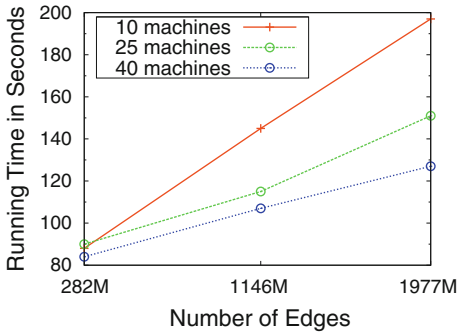


Fig. 8 Edge scalability of our proposed CCB method. The Y-axis shows the running time in seconds, for PageRank queries on Kronecker graphs. Note the running times scale up near-linearly with the number of edges for all the settings (10, 25, and 40 machines)

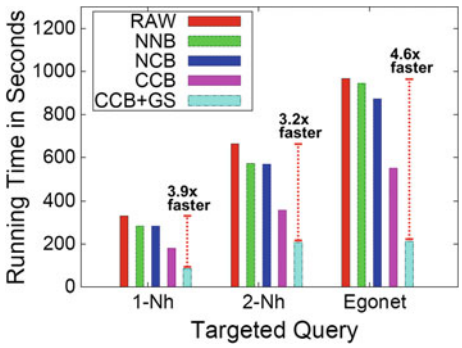


Fig. 9 Running times of targeted queries over different storage and indexing methods, on Twitter graph. 1-Nh and 2-Nh denote the 1-step and the 2-step neighborhood queries, respectively. Note that the CCB+GS (grid selection method combined with the clustered zip block encoding) outperforms the others by 4.6× at maximum

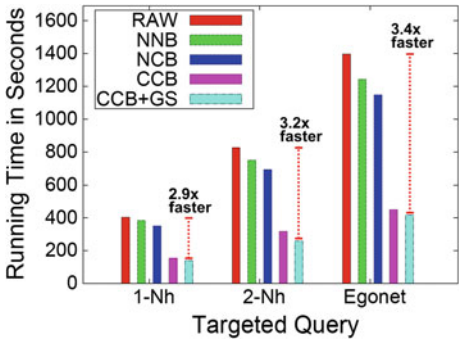


Fig. 10 Running time of targeted queries over different storage and indexing methods, on YahooWeb graph. 1-Nh and 2-Nh denote the 1-step and the 2-step neighborhood queries, respectively. Note that the CCB+GS (grid selection method combined with the clustered zip block encoding) outperforms the others by 3.4× at maximum

is one of the most representative matrix-vector multiplication based algorithm. The PageRank query is evaluated on YahooWeb, Twitter, Kronecker and Erdős-Rényi graphs. We have the following observations.

Running time. In Fig. 6, we see that our proposed CCB method, which combines the clustering and the compressed block encoding, performs the best for all graphs on PageRank queries. Specifically, it outperforms RAW, NNB, NCB methods up to 9.2×, 2×, and 1.6×, respectively, for the Random graph. Note that the time savings rates are smaller than the storage savings rates shown in Fig. 4 and Table 4. The reason is that the compressed block encoding requires additional cpu time for decoding blocks, thereby increase the running time. However, the effect of the decreased I/O time overshadowed the additional decoding time, resulting in smaller total running time.

Machine scalability. Figure 7 shows the scalability of CCB method with regard to the number of machines. The Y-axis shows the "speed up", that is, the ratio of the running time T_M with M machines, and T_{25}. We see that for all the graphs, the running times scale up near-linearly with the number of machines. Putting more machines will eventually decrease the slope, but we leave the limit of this linear scale up open for future work. We also note that figuring out the minimum number of machines required to handle a certain data size is another future work.

An interesting observation is that the slope of the speed-up is small for the "Random" graph. The reason is that after the block compression, the Random data become much smaller (43×) than the original, and thus, even the 25 machines can handle the compressed data in a fairly small amount of time. Putting 100 machines in this case does not help for the speed-up much, since there is a limit of speed-up due to the fixed costs to run MapReduce jobs.

Edge scalability. Figure 8 shows the scalability of CCB method with regard to the number of edges. We used the Kronecker graphs for the experiments. Note that for all the settings (10, 25, and 40 machines), the running time scales up near-linearly with the number of edges.

5.5 Targeted query time

We show the performance of targeted queries on Twitter and YahooWeb graphs in Figs. 9 and 10, respectively. 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 10 randomly selected query nodes.

Effect of compression and clustering. For all the queries in both of the graphs, the compressed block encoding *without* clustering (NCB) performs better than the naive block encoding (NNB), but the performance gain is small. For example, NCB for the 1-Nh query on YahooWeb graph performs 1.09× better than NNB as shown in Fig. 10. In contrast,

the compressed block encoding *with* clustering (CCB) performs much better than the naive block encoding (NNB). For example, CCB for the egonet query on YahooWeb graph performs 2.8× better than NNB as shown in Fig. 10. We can see that our proposed compression, combined with clustering, helps for faster running time. We note that the maximum speed-up gains (2.8×) is smaller than the maximum storage savings (5×) by compression, due to the additional cpu time for decoding data.

Effect of grid selection. The CCB-GS method, which combines the CCB with grid selection, achieves the fastest running time for all the queries in both of the graphs. In Twitter graph, the CCB-GS method outperforms RAW and CCB methods up to 4.6× and 2.6×, respectively, for egonet queries. The reason of this performance gain is that CCB-GS selects only relevant grids ($\sqrt{K}$ for total K grids) for input blocks, while CCB reads all the blocks for query execution. We see that our proposed run-time optimization for query execution pays off, leading to faster running times.

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, (4) matrix computation, and (5) 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, Trißl et al. [21] proposed to index the graph using pre- and postorder number to answer the reachability queries. Chierichetti et al. [13] explored link reciprocity for adjacency queries. Aggarwal et al. [22] proposed using edge sampling to handle graph connectivity queries. Sarkar et al. [23] explored the clustering properties to proximity queries on graphs. Maserrat et al. [24] 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 computational mode, which limits the size of the graph such indexing techniques can support. These limitations are carefully addressed in the GBASE, which supports multiple different types of queries simultaneously and is naturally applicable to the distributed computing environment.

Finally, there are works on indexing *many small* graphs using frequent subgraph [25,26] or significant graph patterns [27], which is quite different from our setting where we have *one large* graph.

Graph queries. There are numerous different queries on graphs. To name a few, graph-level queries answer some global statistics of the whole graph, for example, estimating diameters [8], counting connected components [6], etc. Node-level queries, on the other hand, focus on the relationship among individual nodes. Representative queries include neighborhood [24], proximity [4], PageRank [3], centrality [28], etc. Between the graph-level and individual node-level, there are also queries on the sub-graph level, for example, community detection [29,30], finding induced subgraph [31], etc. GBASE covers a wide range of queries, including the global and the node-level ones, by a unified matrix-vector multiplication framework.

Column store. Column-oriented DBMS has gained its popularity in the recent years, due to (among other merits) its excellent I/O efficiency for read-extensive analytical workloads. From research community, some representative works include [32–36]. A notable work of column store database from industrial side is HBase (http://hbase.apache.org/). HBase is designed for large sparse data, built on the top of HADOOP core. Different from HBase, our GBASE partitions the data in two dimensions (both columns and rows) and it is tailored for large real graphs. By leveraging the block and community-like property which exists in many real graphs, GBASE enjoys the advantages of both row-oriented and column-oriented storages.

Matrix computation. A remotely related work includes large scale matrix computation software including NASA's General Purpose Solver [37] and SciDB [38]. Although GBASE works on the adjacency matrix of a graph, GBASE is based on compressed representation of the graphs which is not provided in the aforementioned works.

Parallel data management. Parallel data processing has attracted a lot of industrial attention recently due to the success of MAPREDUCE, a parallel programming framework [2], and its open source version HADOOP [18]. Due to its excellent scalability, ease of use, and cost advantage, MAPREDUCE and MAPREDUCE-like systems have been extensively explored for various data processing. Representative work include Pregel [39], PEGASUS [6], SCOPE [40], Dryad [41], PIG Latin [42], Sphere [43], and Sawzall [44], etc. Among them, both PEGASUS and Pregel focus on large graph querying/mining and are most related to our work. The proposed GBASE (preliminary version in [45]) provides an even lower-level support in terms of storage cost by indexing the graph on the homogeneous block levels, which are ignored in either PEGASUS or Pregel. In addition, both PEGASUS and Pregel essentially perform node/vertex-centralized computation. Our GBASE is more flexible in the sense that it also supports edge-centralized processing (e.g., induced subgraphs, egonet, etc.) in addition to node-centralized processing.

7 Conclusion

In this paper, we propose GBASE, an efficient analysis platform for large graphs. 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 "compressed block encoding". Experiments on billion-scale graphs show that the storage and the running time reduced up to 43× and 9.2× of the original, respectively.
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 *eleven* 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 a 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, efficient update of the graphs, and better support for time evolving graphs.

Acknowledgments Funding was provided by the U.S. ARO and DARPA under Contract Number W911NF-11-C-0088, by DTRA under contract No. HDTRA1-10-1-0120, and by ARL under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions are those of the authors and should not be interpreted as representing the official policies, of the U.S. Government, or other funding parties, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation here on.

References

1. Liu, C., Guo, F., Faloutsos, C.: BBM: Bayesian browsing model from petabyte-scale data. In: KDD (2009)
2. Dean, J., Ghemawatm, S.: MapReduce: simplified data processing on large clusters. In: OSDI (2004)
3. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web, Stanford Digital Library Technologies Project (1998)
4. Tong, H., Faloutsos, C., Pan, J.-Y.: Fast random walk with restart and its applications. In: ICDM (2006)
5. Lin, C.-Y., Cao N., Liu, S., Papadimitriou, S., Sun, J., Yan, X.: smallblue: social network analysis for expertise search and collective intelligence. In: ICDE (2009)
6. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: a peta-scale graph mining system—implementation and observations. In: ICDM (2009)
7. Sun, J., Qu, H., Chakrabarti, D., Faloutsos, C.: Neighborhood formation and anomaly detection in bipartite graphs. In: ICDM (2005)
8. Kang, U., Tsourak, Appelakis, C.E., Appel, A.P., Faloutsos, C., Leskovec J.: Radius Plots for Mining Tera-byte Scale Graphs: Algorithms, Patterns, and Observations. In: SDM (2010)
9. Akoglu, L., McGlohon, M., Faloutsos, C.: oddball: Spotting Anomalies in Weighted Graphs. In: PAKDD (2010)
10. Alvarez-Hamelin, I., Dall'Asta, L., Barrat, A., Vespignani, A.: k-core decompositions: a tool for the visualization of large scale networks, http://arxiv.org/abs/cs.NI/0504107
11. Karypis, G., Kumar, V.: (1999) Multilevel k-way Hypergraph Partitioning In: DAC
12. Papadimitriou, S. Sun, J.: DisCo: distributed co-clustering with map-reduce, ICDM (2008)
13. Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., Raghavan, P.: On compressing social networks. In: KDD (2009)
14. Kang, U., Faloutsos, C.: Beyond 'Caveman Communities': Hubs and spokes for graph compression and mining. In: ICDM (2011)
15. Boldi, P., Vigna, S.: The webgraph framework I: compression techniques. In: WWW (2004)
16. Chakrabarti, D., Papadimitriou, S., Modha, D.S., Faloutsos, C.: Fully automatic cross associations. In: KDD (2004)
17. Kang, U., Papadimitriou, S., Sun, J., Tong, H.: Centralities in large networks: algorithms and observations. In: SDM (2011)
18. Hadoop information, http://hadoop.apache.org/
19. Leskovec, J., Chakrabarti, D., Kleinberg, J.M., Faloutsos, C.: Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In: PKDD (2005)
20. Erdős, P., Rényi, A.: On random graphs, Publicationes Mathematicae (1959)
21. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD (2007)
22. Aggarwal, C.C., Xie, Y., Yu, P.S.: GConnect: a connectivity index for massive disk-resident graphs. In: PVLDB (2009)
23. Sarkar, P., Moore, A.W.: Fast nearest-neighbor search in disk-resident graphs. In: KDD (2010)
24. Maserrat, H., Pei, J.: Neighbor query friendly compression of social networks. In: KDD (2010)
25. Xin, D., Han, J., Yan, X., Cheng, H.: Mining compressed frequent-pattern sets. In: VLDB (2005)
26. Zhao, P., Yu, J.X. Yu, P.S.: Graph indexing: tree + delta >= graph. In: VLDB (2007)
27. Yan, X., Cheng, H., Han, J., Yu, P.S.: Mining significant graph patterns by leap search. In: SIGMOD (2008)
28. Bader, D.A., Kintali, S., Madduri, K., Mihail, M.: Approximating betweenness centrality. In: WAW (2007)
29. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning for irregular graphs. In: SIAM Review (1999)
30. Andritsos, P., Miller Renée, J., Tsaparas, P.: Information-theoretic tools for mining database structure from large data sets. In: SIGMOD (2004)
31. Addario-Berry, L., Kennedy, W.S., King, A.D., Li, Z. Reed, B.A.: Finding a maximum-weight induced k-partite subgraph of an i-triangulated graph. Discret. Appl. Math. 158(7):765–770 (2010)
32. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., Zdonik, S.B.: C-Store: a column-oriented DBMS. In: VLDB (2005)
33. Abadi, D.J., Madden, S., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: SIGMOD (2008)
34. Abadi, D.J., Boncz, P.A., Harizopoulos, S.: Column oriented database systems. In: PVLDB (2009)
35. Ivanova, M., Kersten, M.L., Nes, N.J., Goncalves, R.: An architecture for recycling intermediates in a column-store. In: SIGMOD (2009)

36. Héman, S., Zukowski, M., Nes, N.J., Sidirourgos, L., Boncz, P.A.: Positional update handling in column stores. In: SIGMOD (2010)
37. Watson, W.R., Storaasli, O.O.: Application of NASA General-Purpose Solver to Large-Scale Computations in Aeroacoustics, NASA Langley Technical Report Server (1999)
38. Cudré-Mauroux, P., Kimura, H., Lim, K.-T., Rogers, J., Simakov, R., Soroush, E., Velikhov, P., Wang, D.L., Balazinska, M., Becla, J., DeWitt, D.J., Heath, B., Maier, D., Madden, S., Patel, J.M., Stonebraker, M., Zdonik, S.B.: A demonstration of SciDB: a science-oriented DBMS, PVLDB 2:2 (2009)
39. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C. Horn, I. Leiser, N. Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD (2010)
40. Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: easy and efficient parallel processing of massive data sets. In: VLDB (2008)
41. Isard, M., Yu, Y.: Distributed data-parallel computing using a high-level programming language. In: SIGMOD (2009)
42. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD (2008)
43. Grossman, R.L., Gu, Y.: Data mining using high performance data clouds: experimental studies using sector and sphere. In: KDD (2008)
44. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: parallel analysis with Sawzall. Sci Program J. 13(4):277–298 (2005)
45. Kang, U., Tong, H., Sun, J., Lin, C.-Y. Faloutsos, C.: GBASE: a scalable and general graph management system. In: KDD (2011)