Parallel Processing of Multiple Graph Queries Using MapReduce

Song-Hyon Kim, Kyong-Ha Lee[†], Hyebong Choi, and Yoon-Joon Lee
Department of Computer Science
KAIST, Daejeon, Republic of Korea
Email: {songhyon.kim, egnever, yoonjoon.lee}@kaist.ac.kr, bart7449@gmail.com[†]

***Abstract*—Recently the volume of the graph data set is often too large to be processed with a single machine in a timely manner. A multi-user environment deteriorates this situation with many graph queries given by multiple users. In this paper, we address the problem of processing multiple graph queries over a large set of graphs. We devise several methods that support efficient processing of multiple graph queries based on MapReduce. Particularly, we focus on processing multiple queries for graph data in parallel with a single input scan. We show that our methods improve the performance of multiple graph query processing with various experiments.**

***Keywords*-parallel processing; MapReduce; graph query; big data;**

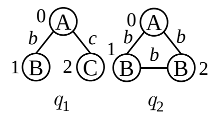
I. Introduction

Graphs are widely used to model complex structures such as chemical compounds, protein interactions, and Web data in many applications [1]. However, many graph data sets are often hard to handle within a single machine because of their size and complexity, e.g., the PubChem project now serves more than 30 million chemical compounds, the storage size of which hits tens of terabytes [2].

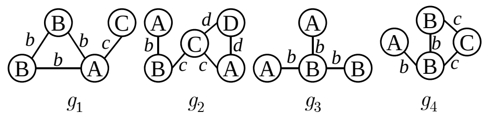
It is required for users to find graphs that contain the patterns in which they are interested from a graph data set. This is formally called a *graph query* or *subgraph isomorphism* problem, which belongs to NP-complete [3]. In a multi-user environment, many users may describe their interesting patterns with their own graph-structured queries. With the massive volume of graph data set and many query graphs, it is more difficult to process graph queries within a single machine in a timely manner.

Meanwhile, MapReduce has gained a lot of attention from both of industry and academia [4]. MapReduce presents a distributed method to processing data-intensive jobs with no hassle of managing the jobs across nodes. In addition, MapReduce has advantages in its scalability, fault-tolerance, and simplicity [5].

In this paper, we address the problem of processing multiple graph queries over a large set of graphs using Hadoop [6], an open source implementation of MapReduce. We first start by discussing a naïve approach which performs all pair-wise subgraph isomorphism tests between a graph query set and a graph data set. Definitely, the naïve approach is very time-consuming. To reduce the number of expensive subgraph isomorphism tests, we introduce a filter-and-verify



(a) Graph query set Q



(b) Graph data set D

Figure 1. A running example

scheme and propose two implementations of the scheme, i.e., *map-side verification* and *reduce-side verification*. In addition, we devise several alternatives considering feature types and feature set comparison, which affect the overall query processing time. We show our experimental results with both synthetic and real data sets. To the best of our knowledge, this is the first work to consider parallel processing of multiple graph queries over a large graph data set with MapReduce.

The rest of the paper is organized as follows. We discuss related work in Section II. Section III presents preliminaries. We propose our methods in Section IV and optimizations for them in Section V. We provide our experimental results in Section VI. Finally, we conclude this paper in Section VII.

II. Related Work

Graph databases are categorized into two types: a *graph-transaction setting* and a *single-graph setting* [7]. In the graph-transaction setting, a graph database consists of a set of relatively small graphs, whereas in the single-graph setting, the data of interest is a single large graph. In this paper, we focus on the graph-transaction setting consisting of tens of millions of graphs.

MapReduce programming model relies on both data parallelism and data shipping paradigms [5]. The MapReduce framework works in two stages: *map* and *reduce*. Input data are partitioned into equal-sized blocks and each of blocks is assigned to a mapper at map stage. Outputs of map stages are stored in local disks, then shuffled and pulled to reducers at reduce stage. This implies I/O inefficiency during processing. Readers are referred to a recent survey for the MapReduce framework and its up-to-date improvements [5].

Blanas et al. [8] compared many join techniques available on MapReduce for analysis of click stream logs at Facebook. Vernica et al. [9] proposed a method to parallelize set-similarity joins with MapReduce by utilizing prefix-filtering scheme. Our work is motivated by some techniques for supporting join processing in the MapReduce programming model.

Some scientists also studied graph processing with MapReduce. Luo et al. [10] solved a single graph query processing problem under a very restricted assumption, i.e., each edge in a query graph must be uniquely identified by labels of its endpoints and itself. Lin et al. [11] proposed several design patterns applicable to iterative graph algorithms such as PageRank [12]. There are also other studies about large scale graph processing [13], [14]. However, they are different from our work in that they focused on processing a single large graph such as Web data or social network, rather than a large set of small graphs. They also do not address the subgraph isomophism problem at all.

III. PRELIMINARIES

We formally define a graph query problem that we deal with in this paper. Then, we describe the way of representing graph data in MapReduce. In addition, we introduce the concept of features in graph query processing.

A. Problem Definition

A graph is denoted by a tuple $g = (V,\, E,\, L,\, l)$, where V is the set of vertices and E is the set of undirected edges such that $E \subseteq V \times V$. L is the set of labels of vertices or edges, and the labeling function l defines the mapping: $V \cup E \to L$. We also denote the vertex set and the edge set of graph g by $V(g)$ and $E(g)$, respectively. Moreover, we denote the label of $u \in V(g)$ and $(u, v) \in E(g)$ by $l(u)$ and $l(u, v)$, respectively.

Definition 1: Given two graphs $g = (V, E, L, l)$ and $g' = (V',\, E',\, L',\, l')$, g is **subgraph isomorphic** to g', denoted $g \subseteq^s g'$ if and only if there exists an injective function $f : V \to V'$ such that

1) $\forall u \in V,\ f(u) \in V'$ and $l(u) = l'(f(u))$,
2) $\forall (u, v) \in E,\ (f(u), f(v)) \in E'$ and $l(u, v) = l'(f(u), f(v))$.

Problem Statement: Let $D = \{g_1, g_2, \cdots, g_n\}$ be a graph data set. Furthermore, let $Q = \{q_1, q_2, \cdots, q_m\}$ be a graph query set such that $|Q| << |D|$. For each graph query $q \in Q$, we find all the graphs to which q is subgraph isomorphic from D.

Figure 1 shows a running example which will be used throughout this paper. In the example, the answers of two graph queries are $A(q_1) = \{g_1\}$ and $A(q_2) = \{g_1\}$, where $A(q_i)$ represents a set of answers for q_i such that $A(q_i) = \{g_j | q_i \subseteq^s g_j \wedge g_j \in D\}$.

B. Graph Representation

Data in MapReduce are modeled by a list of `<key, value>` pairs, which are generally typed strings. Thus, all graph data must be serialized to be the `<key, value>` pairs for processing. We use the the following terms to refer to our serialized graph data.

- `gid`: a unique identifier for a single graph g.
- `gcode`: a serialized format of g, which enumerates vertices and edges in g, i.e., $\{|V(g)|,\ |E(g)|,\ l(V(g)),\ E(g))\}$, where $e \in E(g)$ is represented as 'from-`gid`, to-`gid`, $l(e)$'.

We call a pair of `gid` and `gcode` a *graph record*. For example, query graph q_2 in Figure 1(a) is modeled by the pair of `gid` and `gcode` ('q_2', '$3, 3, A, B, B, 0, 1, b, 0, 2, b, 1, 2, b$').

C. Features in graph query processing

A feature is a substructure of a graph, which represents partial structural information of the graph. There are various kinds of features such as path, subtree, and subgraph [15]–[17].

IV. THE PROPOSED METHOD

In this section, we first discuss a naïve approach for processing multiple graph queries using MapReduce. Two feature set comparison methods are introduced in the following subsection. Finally, we propose two different implementations based on MapReduce, both of which follow a *filter-and-verify* scheme.

A. Naïve approach

A simple solution for parallel processing of multiple graph queries over a massive graph data set is to perform subgraph isomorphism test in parallel for each pair of query graph q and data graph g such that $q \in Q$ and $g \in D$. Since $|Q| << |D|$, we partition D into equal-sized blocks and simultaneously perform the test for each block with query set Q. This is akin to partitioned nested-loop join techniques in parallel DBMS [18]. However, the naïve approach requires to perform subgraph isomorphism test $|Q| \times |D|$ times in total. This consumes a lot of time for query processing.

Therefore, we rather use a filter-and-verify scheme to reduce the number of subgraph isomorphism tests. In the scheme, we first get candidate data graphs then test subgraph isomorphism with only the candidate graphs, rather than directly testing subgraph isomorphism for all data graphs. For this, we filter irrelevant graphs out by comparing a set of features from a query graph with a set of features from a data graph. Although the filtering phase may produce false positives, it reduces the overall execution time significantly since it excludes many graphs in advance so that the number of graphs to be verified is quite reduced.

B. *Feature set comparison*

In our approach, filtering is a set-inclusion test between two feature sets, one of which comes from a query graph q and the other comes from a data graph g. To become a candidate, a feature set of data graph g must include all items in a feature set of query graph q. The problem here is how quickly to perform the set-inclusion test between two feature sets. A basic approach is akin to nested-loop join i.e., each feature of a query graph is iteratively compared with the features of a data graph one by one until all features of a query graph are tested with features of all data graphs.

To address this problem, we devise a filtering method based on Bloom filter [19], a space-efficient probabilistic structure that is useful for existence test. The detailed procedure is as follows. We first assign each item in a feature set for data graph g a unique number. Then, we convert the feature set to a bitarray by applying multiple hash functions to the unique number associated with each feature. During query processing, if bit positions in the bitarray are set to 1 for all items in the feature set of query graph q, the corresponding data graph becomes a candidate of q. Of course, Bloom filter may generate false positives. We test how such false positives affect the overall performance of our system in experiments. Note that we build Bloom filters only for data graphs, but not for query graphs as the average number of features extracted from query graphs is much less than that from data graphs.

C. *Two MapReduce implementations*

With the filter-and-verify scheme, we examine if we benefit from where to put the verification step into a MapReduce job: *map-side* and *reduce-side*. In the map-side verification method, both filtering and verification are performed at map stage. The results of map stage are directly emitted into HDFS [6], a distributed filesystem used in Hadoop, and reduce stage works nothing at all. Workload across mappers are expected to be balanced by runtime scheduling scheme in the MapReduce framework [4]. On the contrary, in the reduce-side verification method, mappers perform only the filtering step and reducers perform subgraph isomorphism tests. The map-side verification method is similar to a map-merge join technique in MapReduce except that it requires to perform verification step for each pair of graphs, instead of join operation with two relations [8].

1) Map-side verification: Our map-side verification method is described in Algorithm 1. Reduce stage is omitted, since it contains no action. In MapReduce, input data are first partitioned into equal-sized blocks and each of the blocks is assigned to a mapper at map stage. Each mapper reads a block of graph data set D. When launching a MapReduce job, the MapReduce framework delivers a set of graph queries to each mapper via `distributedCache` [6], a facility to cache read-only files in Hadoop. Thus, we load a list of query graphs from a local disk, although Algorithm 1 generally describes loading of query graphs from HDFS (line 3). Each mapper generates features from graph queries in Q (line 4). For each graph record in the block of D, the `map` method also generates features (line 6). Then, the `map` method compares the feature set of each query graph q with the feature set of data graph V (line 8). If contained, the `map` method verifies the candidate by testing subgraph isomorphism (line 9). Note that `SubIsoTest`(q, g) requires `gcode` of two graphs to check whether graph q is subgraph isomorphic to graph g. A feature comparison operator $\subseteq$ is set-inclusion relation in the algorithm, however its details depend on which feature type is used.

```
1  class Mapper
2    method initialize()
3      $Q \leftarrow$ load a list of [$gid$, $gcode$] from HDFS
4      $Q.gfeature[\,] \leftarrow$ generate features from $Q$
5    method map($K$ : null, $V$ : [$gid, gcode$])
6      $feature \leftarrow$ generate features from $V$
7      foreach query graph $q$ in $Q$ do
8        if $q.gfeature \subseteq feature$ then
9          if SubIsoTest($q.gcode$, $V.gcode$) then
10           emit($q.gid$, $V.gid$)
```

Algorithm 1: The map-side verification method

```
1  class Mapper
2    method initialize()
3      $Q \leftarrow$ load a list of [$gid$, $gcode$] from HDFS
4      $Q.gfeature[\,] \leftarrow$ generate features from $Q$
5      initialize a buffer $B$
6    method map($K$ : null, $V$ : [$gid, gcode$])
7      $feature \leftarrow$ generate features from $V$
8      foreach query graph $q$ in $Q$ do
9        if $q.gfeature \subseteq feature$ then
10         $B[V] \leftarrow B[V] \cup \{q.gid\}$
11   method close()
12     foreach data graph $V$ in $B$ do
13       emit($V$, $B[V]$)
14 class Reducer
15   method initialize()
16     $Q \leftarrow$ load a list of [$gid$, $gcode$] from HDFS
17   method reduce($K'$ : [$gid, gcode$],
       $V'$ : a list of $gid$s)
18     foreach query id $qid$ in $V'$ do
19       $code \leftarrow$ retrieve gcode with $qid$ from $Q$
20       if SubIsoTest($code$, $K'.gcode$) then
21         emit($qid$, $K'.gid$)
```

Algorithm 2: The reduce-side verification method

2) Reduce-side verification: Algorithm 2 describes the reduce-side verification method. This scheme is more faithful to the MapReduce programming model. Unlikely to the Mapper in Algorithm 1, the verification step moves to reduce stage. Moreover, we perform pre-aggregation of intermediate data in mapper, i.e., *in-mapper combining* [11], to reduce the size of intermediate data delivered to reducers.

The `initialize` method is similar to that of Algorithm 1 except for buffer B, which holds a pair of a data graph V and the corresponding `gids` of its candidate graph queries (line 5). In the `map` method, candidates are identified by comparing features and those are pushed into buffer B for delivery (lines 9–10). When closing the mapper, all pairs of data graph V and the corresponding `gids` of graph queries are emitted (lines 12–13). The `reduce` method reads data graph K' and a list of `gids` of candidate graph queries V' as input (line 17). Here, the input means that the data graph V is a candidate of every graph query in V'. For each graph query, the `reduce` method tests subgraph isomorphism to data graph K' (line 20). Final results are emitted with a pair of `gids` of query graph and data graph (line 21). Figure 2 shows an illustration of the reduce-side verification method. In the figure, *gfeature* represents a set of features. Details about the features will be explained in the following section.

V. OPTIMIZATION

We discuss two optimization techniques for the proposed methods in this section.

A. Reducing I/Os

As noted in many studies, MapReduce is not optimized for I/O efficiency. Thus I/O cost is a dominant factor for performance in MapReduce [5]. For that reason, we delicately model our data format for I/O efficiency. In Algorithm 2, we make `map` stage emit its result as a pair of a data graph and a list of `gids` of candidate query graphs, instead of a pair of a query graph and its candidate data graph. Two reasons lie in the decision. First, it saves many I/Os which are required to deliver the overall structure of query graphs for each candidate data graph separately. The `reduce` stage eliminates redundant loading of query graphs across different data graphs, then reads once the `gcode` of each query graph from HDFS. Furthermore, since $|Q| << |D|$, the lookup cost of loading graph structure with a given `gid` is reduced when choosing Q rather than D. Second, this strategy generates more groups of intermediate results. It is more suitable for load balancing. If groups of intermediate results are small, sometimes data skewness may involve performance degradation [20].

B. Feature type

The choice of feature types affects the number of candidates. A conventional feature type of a graph g is *edge label* (EL), i.e., $(l(u), l(v), l(u, v))$ for edge $(u, v) \in E(g)$. Edge label must be unique in a feature set, meaning that identical edges are not shown in a feature set. Luo et al. [10] adapts this feature type. Based on edge label, we propose two optional feature types. First, we add the number of occurrences of each edge label in a graph, which is denoted by *edge label with count* (ELC). This is simple but largely drops false positives. The other is to use cycle information in a graph, since cycles are rare in a sparse graph [17]. We call this *edge label with count and cycle* (ELC+CL).

Table I
SUMMARY OF PROPOSED METHODS

Verification position	Feature type	Feature set comparison
- map-side - reduce-side	- edge label - edge label with count info. - edge label with count and cycle info.	- nested-loop - Bloom filter

Example 1: Query graph q_2 in Figure 1(a) has features of *edge label* $\{(A, B, b), (B, B, b)\}$, *edge label with count* $\{(A, B, b, 1), (A, B, b, 2), (B, B, b, 1)\}$ where the last number of each feature is the occurrences of that edge label, and *edge label with count and cycle* $\{(A, B, b, 1), (A, B, b, 2), (B, B, b, 1), cycle(0, 1, 2)\}$ where $cycle(...)$ denotes a cycle consisting of those vertices.

Table I summarizes our methods discussed so far.

VI. EXPERIMENTS

A. Experimental Setup

Experiments were performed on a 9-node cluster, one of which was designated as a name node. Each data node is equipped with an Intel i5-2500 3.3GHz processor, 8GB memory and a 7200RPM HDD, running on CentOS 6.2. All nodes are connected via a Gigabit switching hub. We select the datasets given by authors of `iGraph` [15], a graph processing benchmark tool. Algorithms were implemented in C++ and executed via Hadoop Pipes [6], a C++ library that provides communications with Hadoop. We use VF2 [21] to test subgraph isomorphism between two graphs. For real graph data sets, we generated various sized chemical data also used in [15], each of which has 23.98 vertices and 25.76 edges in average. For synthetic data sets, we chose the dataset named *Synthetic.10K.E30.D3.L50* from [15], each of which has 14.23 vertices and 30.53 edges with 50 distinct vertex/edge labels in average. The number of graphs in a graph data set, $|D|$, is 10 million. We also randomly generated several sets of graph queries, which contain a various number of queries, i.e., $|Q| = 1, 10, 100,$ and 1000.

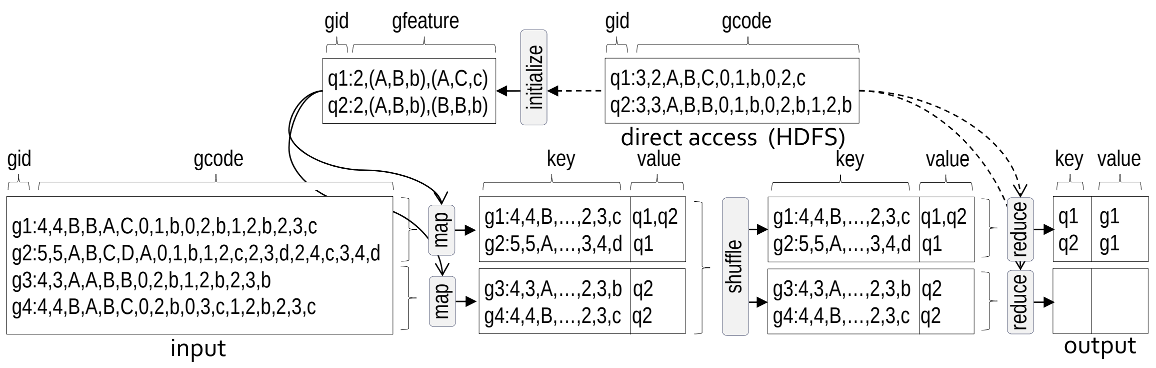


Figure 2. An illustration of parallel processing of graph queries shown in Algorithm 2 with MapReduce

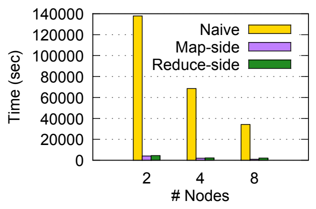


Figure 3. Methods (synthetic)

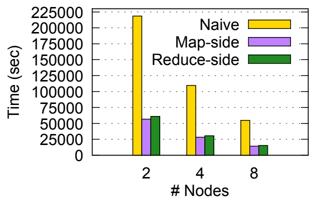


Figure 4. Methods (real)

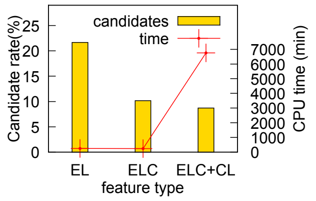


Figure 7. Candidate ratio with different feature types

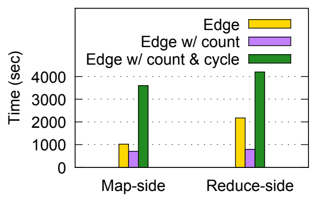


Figure 5. Feature type (synthetic)

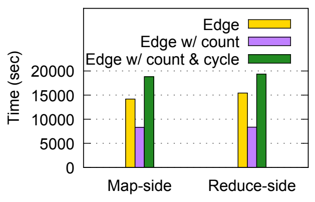
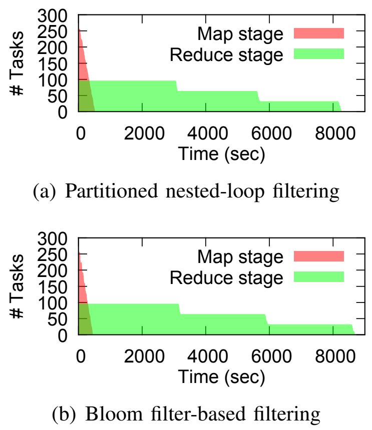


Figure 6. Feature type (real)



(a) Partitioned nested-loop filtering

(b) Bloom filter-based filtering

Figure 8. MapReduce job execution with real dataset

B. Performance Analysis

We first compared three methods, i.e., naïve, map-side verification, and reduce-side verification, of processing graph queries with both real and synthetic datasets. Our results are shown in Figure 3 and 4. The map-side verification method showed the best performance in both cases. In addition, overall execution time decreases linearly with the increasing number of nodes in a cluster. However, The map-side verification could not outperform the reduce-side verification method. The reason is that the size of our intermediate results is marginal, since we gave our best effort to reduce I/Os while delivering intermediate results to reducers. We also compared three feature types as shown in Figure 5 and 6. The feature type of *edge label with count* showed the best performance. Although the most complex feature type, i.e., *edge label with count & cycle* (ELC+CL), has the least number of average candidates, it was not the best since it spent much time in extracting features and testing set-inclusion as shown in Figure 7 (in the figure, CPU time in the right axis means summation of all the elapsed time of mappers in Algorithm 2). Next, we tested partitioned nested-loop and Bloom filter-based filtering schemes. We built a 160-bit length Bloom filter for each $g \in D$ with 4 hash functions. Since $|E(g)|$, $g \in D$ is 25.76 in average, the probability of false positives that the Bloom filter has is 5.2%. Figure 8 explains time taken by map and reduce stages in two filtering schemes running on Hadoop. In nested-loop filtering scheme, map stage took 529 seconds and 8,288 seconds in reduce stage. Bloom filter-based filtering improves the filtering time with 462 seconds. However, overall time was rather extended to 8,723 seconds. The reason is that as the Bloom filter-based filtering scheme generates more candidates, thus it requires more subgraph isomorphism test that is a dominant factor in execution time. Finally, we tested the scalability of our methods on MapReduce, varying the number of data graphs

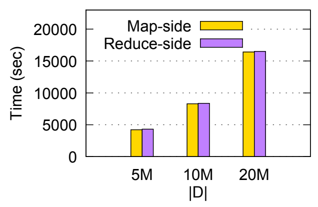


Figure 9. Varying # of data graphs (real)

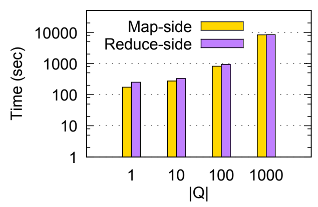


Figure 10. Varying # of queries (real)

and query graphs. As shown in Figure 9 and 10, filter-and-verify scheme on MapReduce scales linearly with both the number of data graphs and query graphs.

VII. Conclusion

In this paper, we applied the MapReduce framework to process multiple graph queries over a large graph data set. Lessons learned in this paper are as follows. Filter-and-verify scheme is better than the naïve approach as expected. Complex features help our system improve its execution time, but there is a tradeoff that feature extraction and comparison overhead may harm the overall execution time. Lastly, reduction of the number of candidates is more crucial for the overall execution time than reduction of the filtering time.

Acknowledgment

This work was partly supported by NRF grant funded by Korea Government (No. 2011-0016282).

References

[1] C. Aggarwal and H. Wang, "Managing and mining graph data," *Advances in Database Systems*, vol. 40, 2010.

[2] "The pubchem project," http://pubchem.ncbi.nlm.nih.gov, (accessed November 20, 2012).

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] K. Lee, Y. Lee, H. Choi, Y. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *ACM SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[6] "Apache hadoop," http://hadoop.apache.org, (accessed November 20, 2012).

[7] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph*," *Data mining and knowledge discovery*, vol. 11, no. 3, pp. 243–271, 2005.

[8] S. Blanas, J. Patel, V. Ercegovac, J. Rao, E. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *Proceedings of the 2010 international conference on Management of data*. ACM, 2010, pp. 975–986.

[9] R. Vernica, M. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *Proceedings of the 2010 international conference on Management of data*. ACM, 2010, pp. 495–506.

[10] Y. Luo, J. Guan, and S. Zhou, "Towards efficient subgraph search in cloud computing environments," *Database Systems for Adanced Applications*, pp. 2–13, 2011.

[11] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. ACM, 2010, pp. 78–85.

[12] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," in *Stanford InfoLab*, 1999.

[13] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*. ACM, 2010, pp. 135–146.

[14] U. Kang, C. Tsourakakis, and C. Faloutsos, "Pegasus: mining peta-scale graphs," *Knowledge and information systems*, vol. 27, no. 2, pp. 303–325, 2011.

[15] W. Han, J. Lee, M. Pham, and J. Yu, "igraph: a framework for comparisons of disk-based graph indexing techniques," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 449–459, 2010.

[16] X. Yan, P. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 335–346.

[17] P. Zhao, J. Yu, and P. Yu, "Graph indexing: tree+ delta >= graph," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 938–949.

[18] H. Lu, K. Tan, and B. Ooi, *Query processing in parallel relational database systems*. IEEE Computer Society Press, 1994.

[19] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[20] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 international conference on Management of Data*. ACM, 2012, pp. 25–36.

[21] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 10, pp. 1367–1372, 2004.