LETTER

Scalable and Adaptive Graph Querying with MapReduce

Song-Hyon KIM[†a)], Kyong-Ha LEE[††], Inchul SONG[†††], *Nonmembers*, Hyebong CHOI[††††], *Student Member*, *and* Yoon-Joon LEE[††††], *Nonmember*

SUMMARY We address the problem of processing graph pattern matching queries over a massive set of data graphs in this letter. As the number of data graphs is growing rapidly, it is often hard to process such queries with serial algorithms in a timely manner. We propose a distributed graph querying algorithm, which employs feature-based comparison and a filter-and-verify scheme working on the MapReduce framework. Moreover, we devise an efficient scheme that adaptively tunes a proper feature size at runtime by sampling data graphs. With various experiments, we show that the proposed method outperforms conventional algorithms in terms of scalability and efficiency.

key words: *graph query, parallel processing, MapReduce, adaptive tuning*

1. Introduction

Graphs are widely used to model complex structures such as protein-to-protein interactions and Web data [1]. Graph data are growing rapidly and massively, e.g., PubChem [2] now serves more than 40 million graphs that represent chemical compounds. The overall storage size of the graphs hits tens of terabytes [2]. A graph pattern matching query is to find all data graphs that contain a given query graph from a set of data graphs. A graph pattern matching query requires to process expensive subgraph isomorphism test, which is known to be NP-complete [3]. Thus, many studies have been reported in the literature to cut down the number of subgraph isomorphism tests [4], [5]. Although the algorithms are prominent, they are not scalable enough to handle a massive set of data graphs like PubChem since they all are serial, designed to work on a single machine.

Meanwhile, it is important to set a proper feature size in a filter-and-verify scheme since it dominantly determines the overall performance of query processing (refer to Sect. 6). However, it is hard to find a proper feature size before query processing since it depends on the statistics about both query graphs and data graphs. If the feature size is too large, the overhead of extracting and storing features increases sharply. On the other hand, many false positives are produced at filtering stage if the feature size is too small, involving many obsolete subgraph isomorphism tests.

To tackle the problems, we propose MR-Graph, a distributed graph querying algorithm based on MapReduce [6]. MR-Graph tunes the feature size at runtime. It determines the most appropriate feature size by sampling some portions of data graphs. MR-Graph processes multiple queries over a massive set of a data graph simultaneously. In addition, MR-Graph does not require any index building and the modification of MapReduce internals. MR-Graph is sophisticatedly tailored to work in harmony with MapReduce's programming model.

Manuscript received April 8, 2013.

[†]The author is with Korea Air Force Academy, Korea.
[††]The author is with ETRI, Korea.
[†††]The author is with SAIT, Samsung Electronics, Korea.
[††††]The authors are with CS Dept., KAIST, Korea.
a) E-mail: kim.songhyon@gmail.com

DOI: 10.1587/transinf.E96.D.2126

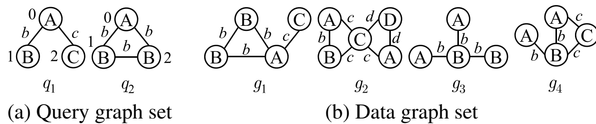
2. Related Work

A major issue in the graph pattern matching problem is to reduce the number of pairwise subgraph isomorphism testing. Most approaches to the issue are based on a filter-and-verify scheme with index techniques [4], [5]. They extract substructures called features from data graphs and then filter irrelevant graphs before actual subgraph isomorphism testing. It may greatly reduce the number of actual subgraph isomorphism tests. However, all the algorithms are serial and thus not scalable enough to process a massive set of data graphs. MapReduce is a programming model that enables to process a massive volume of data in parallel with a large cluster of commodity PCs [6]. It provides a simple but powerful method that enables automatic parallelization and distribution of large-scale computation. Recently, Luo et al. [7] proposed an index-based method for parallel graph pattern matching on MapReduce. They build an edge index over data graphs, which maps each distinct edge to the list of data graphs that include the edge. However, their approach has a restricted assumption that every edge in a query graph must be uniquely identified by labels of its endpoints and itself. Thus it requires a subsequent verification phase to process general graph pattern matching queries. There are some studies about processing graphs with MapReduce. Lin et al. [8] introduce several design patterns suitable for iterative graph algorithms such as PageRank. Pregel [9] and Pegasus [10] are systems devised for processing large-scale graphs. However, they are different from our work in that they focused on processing a single large graph such as Web data or social network.

3. Preliminaries

We denote a graph by a tuple $g = (V, E, L, l)$, where V is a set of vertices and E is a set of undirected edges such that $E \subseteq V \times V$. L denotes a set of labels for both vertices and

Copyright © 2013 The Institute of Electronics, Information and Communication Engineers



(a) Query graph set (b) Data graph set

Fig. 1 An example of query and data graph set.

edges. l denotes a labeling function: $(V \cup E) \rightarrow L$. We also define the *size* of a graph g, denoted by $|g|$, as the number of edges in g, i.e., $|g| = |E(g)|$.

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

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

Problem statement Let $D = \{g_1, g_2, \cdots, g_n\}$ be a set of data graphs and $Q = \{q_1, q_2, \cdots, q_m\}$ be a set of query graphs, where $m << n$. For each query $q \in Q$, we find all graphs D_q such that $D_q = \{g_i \mid q \subseteq^s g_i, g_i \in D\}$.

Figure 1 shows an example which will be used throughout this paper. In the example, answers for two query graphs are $D_{q_1} = \{g_1, g_2, g_4\}$ and $D_{q_2} = \{g_1\}$, where D_{q_i} means a set of answers for q_i.

Graph representation In MapReduce, all input data are represented by key-value pairs. To parallelize graph query processing with MapReduce, we first encode each graph as a single key-value pair. A graph g is encoded as a pair of (`ID`, `CODE`) where `ID` is a unique identifier for each graph and `CODE` is a serialized version of g. `CODE` is composed of *the number of vertices*, *the number of edges*, *a list of vertex labels*, and *a list of edges*. In the format, each edge is represented by a triple that consists of *from-vertex ID*, *to-vertex ID*, and *edge label*. For example, query graph q_2 in Fig. 1 (a) is encoded as (q_2, '3,3,A,B,B,0,1,b,0,2,b,1,2,b').

Features in graph query processing A feature is a loosely defined term for a substructure of a graph. There are various kinds of features such as path, subtree, and subgraph [4], [5]. In this letter, we use a subgraph as feature because it exhibits the best filtering power among them [5]. However, any other substructures can be used in MR-Graph without loss of generality.

4. Basic Algorithm

A MapReduce-based algorithm works in two stages: *map* and *reduce*. In MapReduce, input data are first partitioned into equal-sized blocks and each block is assigned to a mapper at map stage and mapped outputs are then stored into local disks. Finally, the outputs are shuffled and pulled to reducers for further processing at reduce stage. Algorithm 1 is the basic algorithm of MR-Graph. Each mapper processes queries over data graphs and each reducer groups mapped outputs by query graph's `ID`.

Algorithm 1: Basic MR-Graph

input: D: a set of data graphs, Q: a set of query graphs
output: (ID_q, a list of ID_g) pairs

```
1  init ()                                  // initializing a mapper
2      F[] = ExtractFeatures (Q)
3  map (String ID_g, String CODE_g)         // mapper
4      F_g = ExtractFeatures (CODE_g)
5      foreach (ID_q, CODE_q) ∈ Q do
6          if CheckContainment (F[ID_q], F_g) then
7              if TestSubIso (CODE_q, CODE_g) then
8                  emit (ID_q, ID_g)
9  reduce (String ID_q, Iterator ID_g)      // reducer
10     answer = Enlist (ID_g)
11     emit (ID_q, answer)
```

At map stage, we extract features from a query graph and a data graph by different rules (line 2 and 4). Graph query processing is performed in a filter-and-verify scheme. For each data graph, MR-Graph tests whether the data graph contains all the features that compose a certain query graph in a query set Q (line 6). If it is true, the algorithm pairs data graph with the query graph and tests subgraph isomorphism (line 7). At reduce stage, result data graphs are grouped by a query graph ID and emitted. MR-Graph greatly reduces the number of graphs that must be tested subgraph isomorphism based on this filter-and-verify scheme.

When it comes to feature extraction, we extract $min(k, |q|)$-sized features for a query graph q where $|q|$ is the size of query graph q. Note that we do not use features whose sizes are less than $min(k, |q|)$. We explain a rationale behind this. Suppose that a query graph q has two features f_x and f_y, where $f_x \subseteq^s f_y$ and the size of feature f_y, denoted by $|f_y|$, is set to $min(k, |q|)$. It is straightforward that a data graph g contains f_x if it contains f_y. Thus, testing if g contains f_x is unnecessary when g is confirmed to contain f_y. Meanwhile, a data graph is processed with a set of various sized query graphs. For a data graph g, we extract features whose sizes are up to $min(k, |g|)$ each. In other words, the size of feature f for a data graph g must satisfy the condition, $1 \leq |f| \leq min(k, |g|)$. The `ExtractFeatures` function in Algorithm 1 extracts features from both query graphs and data graphs with the feature size following the rules described thus far.

5. AdaptiveTune: Tuning the Feature Size at Runtime

Note that the feature size k in Algorithm 1 is determined by a user before runtime. However, feature size decision is not easy since the best feature size varies according to the characteristics of query graphs and data graphs. To address the issue, we devise an adaptive feature size tuning technique. AdaptiveTune examines a set of trial values for k, denoted by V_k, in the first few mappers, called *trial mappers*, then determines a proper feature size. Note that the number of

Algorithm 2: AdaptiveTune

input: k_{init}: initial feature size, V_k: trial values for k, α: number of trials for each feature size in V_k
output: k: the best appropriate feature size

1. The initial feature size k_{init} is set into DCS.
2. If a mapper is a trial mapper, choose one value from V_k. Otherwise, use the feature size stored in DCS.
3. The trial mapper updates DCS.
4. DCS notifies its update to all running mappers.
5. For each running mapper,
   a. Estimate the remaining time left till the mapper ends, denoted by $T_{current}$, and the remaining time expected if the mapper uses the best feature size, denoted by T_{best}.
   b. If $T_{best} < T_{current}$, change the feature size with one in DCS.
   c. Process the remaining data graphs in the input split.

trial mappers is set to be α times of $|V_k|$. The larger α, the more data graphs are tested to determine k. This tuning process requires information about other mappers to determine the best appropriate feature size and the remaining time of each mapper. We format the information as a 4-tuple relation (k, T, N, O), where each tuple item represents 'a feature size', 'elapsed time', 'the number of data graphs processed', and 'time overhead for feature extraction (lines 1–2 in Algorithm 1)', respectively. For example, $(3, 10, 5, 4)$ means that some mapper processes 5 data graphs in 10 seconds by using feature size 3 and the time taken to extract features from query graphs is 4 seconds. We exploit a distributed coordination system (DCS) to share the information among mappers.

Algorithm 2 describes AdaptiveTune. Note that this algorithm begins with feature size k_{init}, initially set into DCS by a user. After third step in Algorithm 2, DCS stores 4-tuple relation (k, T_B, N_B, O_B) that means some trial mapper finishes its input split that contains N_B data graphs in T_B time, while it takes O_B time to extract features from a query graph set. The $T_{current}$ and T_{best} in the algorithm are estimated as follows:

$$T_{current} = \frac{T_C}{N_C} \times (N_B - N_C) \quad (1)$$

$$T_{best} = \frac{T_B}{N_B} \times (N_B - N_C) + O_B \quad (2)$$

where T_C is the elapsed time when with a given k and T_B is the elapsed time when k is the best. N_C is the number of data graphs processed with the current feature size. We estimate the remaining input size of a mapper by $N_B - N_C$. Note that since N_B is computed when a mapper ends, it also represents the total number of data graphs in the mapper's input split. As AdaptiveTune runs α times with a trial value, N_B is expected to converge to the average number of data graphs in an input split. We also estimate execution time per data graph by $\frac{T_C}{N_C}$ or $\frac{T_B}{N_B}$.

Example 1. Figure 2 shows an example of AdaptiveTune that works with 4 mappers. Suppose that $V_k = \{k_1, k_2, k_3\}$ and $\alpha = 1$, then the number of trial mappers is 3. T1, T2, and T3 are trial mappers and they use one of V_k, while T4 uses the feature size stored in DCS, k_{init}. After finishing its input split, T2 updates DCS with '$(k_2, \ldots)$' at time t_1. DCS notifies this tuple to the other mappers. Then the mappers determine whether they keep or change their feature size, and process remaining input splits with the feature size. Finally, T1 and T4 change the current feature size to k_2.

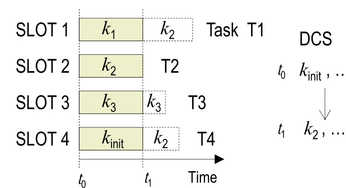


Fig. 2 An example of AdaptiveTune.

Table 1 Statistics of graph datasets.

Dataset	D/Q	#	|V|	|E|	|L|	size
Real	D	10M	23.98	25.76	78	2.31 GB
	Q	1000	14.26	14.00	11	131 KB
Synthetic	D	10M	14.23	30.53	47	2.62 GB
	Q	1000	10.25	12.00	47	120 KB

Table 2 Statistics of input splits.

	Real	Synthetic
# of input splits	256	256
avg. # of data graphs per input split	39062.5	39062.5
Standard deviation	6416.4	48.6

6. Experiments

Experimental setup We performed our experiments with Hadoop version 1.0.3 working on a 9-node cluster. Each data node is equipped with an Intel i5-2500 3.3 GHz processor, 8 GB memory, and a 7200 RPM HDD, running on CentOS 6.2. All nodes are connected via a Gigabit switching hub. The number of task slots is 32, which indicates how many tasks can run simultaneoulsy on Hadoop. The total number of map and reduce tasks were set to be 8 times (256) and 2 (64) times as many as the number of task slots. We tested two datasets introduced in [11], i.e., *PubChem* for a real-world dataset and *synthetic.E30.D3.L50* for a synthetic dataset. In addition, we varied the number of data graphs or query graphs. Table 1 presents the statistics of our datasets where the $|V|$, $|E|$, and $|L|$ values are in average. We set α to 1 and V_k to $\{1, 2, \ldots, 6\}$. The statistics of input splits is shown in Table 2.

We compared our method with a non-filtering approach, denoted by Basic, and Luo et al.'s approach, denoted by Luo [7]. All algorithms are written in C++ and the verification phase was attached into Luo for fair comparison (please refer to Sect. 2). In our system, we used `VF2` [12] for subgraph isomorphism test, `gSpan` [13] for feature extraction, and `ZooKeeper` [14] for distributed coordination.

Result analysis We first tested how a feature size k affects the overall performance of graph pattern matching. Figure 3 shows that the execution time, depicted by bars, is the fastest

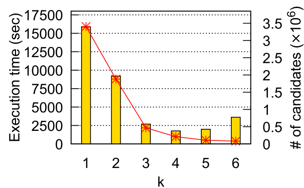


Fig. 3 Execution time (real).

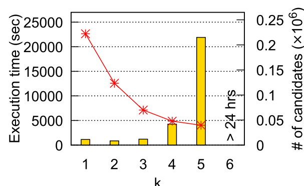


Fig. 4 Execution time (synthetic).

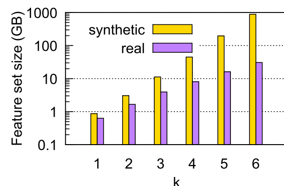


Fig. 5 The size of feature sets.

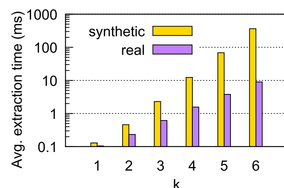


Fig. 6 Feature extraction time.

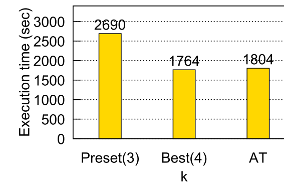


Fig. 7 AdaptiveTune (real).

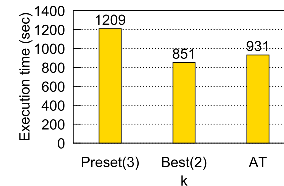


Fig. 8 AdaptiveTune (synthetic).

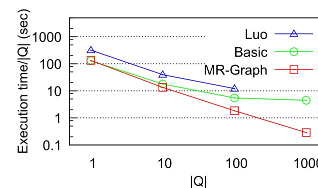


Fig. 9 Comparison of methods (real, 1M).

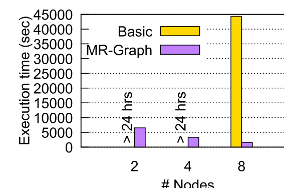


Fig. 10 Scalability (real).

when k is set to 4 whereas the average number of candidates, depicted by lines, is the least when k is 6 for the real dataset. The reason is that the cost of feature extraction and feature comparison offsets the benefit of fine filtering effect. The best value for k on the synthetic dataset is 2 as shown in Fig. 4. Note that we omitted some results which exceeded 24 hours in our figures. The results also confirmed that the best k can be different according to the characteristics of given data and query graph sets. Figures 5 and 6 show how a feature size impacted on the overall performance. As the feature size increases, both the total storage size of feature sets and the average feature extraction per data graph increase exponentially. This causes not only space overhead to store the features, but a lot of I/Os during query processing on Hadoop.

We now evaluate AdaptiveTune. Figures 7 and 8 show the experimental results. In the figures, the 'Preset' value is a fixed initial feature size whereas the 'Best' value is the best feature size computed after finishing all graph queries. In both cases, AdaptiveTune (AT) shows good execution time even compared to the best feature size. With the real dataset, AdaptiveTune was successful to reduce the execution time 52.5%.

We also compared our algorithm with conventional algorithms. We used 1 million real dataset rather than 10 million graphs in Table 1, since Luo exhibited limitation in scalability. As shown in Fig. 9, our algorithm is always better than the others in terms of execution time per query except in the case of processing a single graph query. As the number of query graphs becomes larger, MR-Graph outperforms two other algorithms more. We also noticed that the performance of Luo was even poorer than our Basic algorithm. The reason is that Luo suffered from frequent accesses to the storage of index files and a skewness problem; value distribution in the indexes is highly skewed [15]. Finally, we evaluate scalability of MR-Graph. As shown in Fig. 10, MR-Graph is scalable to the size of a cluster. Specifically, MR-Graph records 25.3 times faster than Basic in execution time when we use 8 data nodes.

7. Conclusions

We proposed an adaptive and parallel algorithm, MR-

Graph, which efficiently processes multiple query graphs over a massive set of data graphs based on MapReduce. AdaptiveTune improves query processing as it adaptively tunes a feature size at runtime, keeping the system in a balance between the effectiveness of graph filtering and the cost of feature extraction. Our experiments show that AdaptiveTune performs with very little overhead in execution time.

Acknowledgement

This work was partly supported by NRF grant funded by Korea Government (No. 2011-0016282) and partly funded by the MSIP, Korea in the ICT R&D program 2013.

References

[1] C. Aggarwal and H. Wang, Managing and mining graph data, Springer, 2010.
[2] "The pubchem project." http://pubchem.ncbi.nlm.nih.gov
[3] M.R. Garey and D.S. Johnson, Computers and Intractability; A Guide to the Theory of NP-Completeness, W.H. Freeman & Co., New York, NY, USA, 1990.
[4] X. Yan, P. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," Proc. SIGMOD, pp.335–346, 2004.
[5] P. Zhao et al., "Graph indexing: Tree + delta >= graph," Proc. VLDB, pp.938–949, VLDB Endowment, 2007.
[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Commun. ACM, vol.51, no.1, pp.107–113, 2008.
[7] Y. Luo et al., "Towards efficient subgraph search in cloud computing environments," DASFAA Workshops, pp.2–13, 2011.
[8] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," Proc. MLG, pp.78–85, 2010.
[9] G. Malewicz et al., "Pregel: A system for large-scale graph processing," Proc. SIGMOD, pp.135–146, 2010.
[10] U. Kang et al., "Pegasus: Mining peta-scale graphs," Knowl. Inf. Syst., vol.27, no.2, pp.303–325, 2011.
[11] W. Han et al., "iGraph: A framework for comparisons of disk-based graph indexing techniques," PVLDB, vol.3, no.1, pp.449–459, 2010.
[12] L. Cordella et al., "A (sub) graph isomorphism algorithm for matching large graphs," IEEE Trans. Pattern Anal. Mach. Intell., vol.26, no.10, pp.1367–1372, 2004.
[13] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," Proc. ICDM, pp.721–724, 2002.
[14] "Apache zookeeper." http://zookeeper.apache.org
[15] Y. Kwon et al., "SkewTune: Mitigating skew in mapreduce applications," Proc. SIGMOD, pp.25–36, 2012.