# Towards Efficient Subgraph Search in Cloud Computing Environments*

Yifeng Luo[1], Jihong Guan[2], and Shuigeng Zhou[1]

[1] School of Computer Science, and Shanghai Key Lab of Intelligent Information Processing, Fudan University, Shanghai, China
{luoyf,sgzhou}@fudan.edu.cn

[2] Dept. of Computer Science & Technology, Tongji University, Shanghai, China
jhguan@tongji.edu.cn

**Abstract.** This paper proposes an efficient approach to subgraph search over a large graph database under the MapReduce framework. The main idea is first to build inverted edge indexes for graphs in the database, and then to retrieve data only related to the query subgraph by using the built indexes to answer the query. Experimental results show that the proposed approach has good performance and scalability.

**Keywords:** Graph database; Subgraph search; Cloud computing; MapReduce; Inverted index.

## 1 Introduction

Graph is now an important data structure, which models objects as vertices and the pairwise relationships between objects as edges. Graph-based modeling is employed in more and more applications [1], including pattern recognition [5–7], social networks, chem-informatics [3], graph-structured XML query processing [4] and so on. When a database is used to manage the data of objects that are represented by graphs, this database is referred to as a graph database. Usually, a graph database falls into two categories [2]: graph-transaction setting where a graph database consists of a large number of relatively small graphs, and single-graph setting where a graph database contains only one large graph.

This paper deals with subgraph search in a large graph database containing many graphs. Subgraph search is one of the fundamental problems in many practical graph-related applications such as chemical compound search, community detection in social networks, motif finding in biological networks, and graph-structured XML query processing. The problem of subgraph search or query can be described formally as follows: given a graph database $D = \{g_1, g_2, \cdots, g_n\}$ and a graph query $q$, to answer the query is to find all graphs that contain $q$ in $D$. These resulting graphs are supergraphs of $q$, and $q$ is one of their subgraphs.

* This work was supported by National Natural Science Foundation of China under grants No. 60873040 and No. 60873070. Jihong Guan was also supported by the Shuguang Scholar Program of Shanghai Education Development Foundation under grant No. 09SG23.

Efficiently answering subgraph queries in a large graph database is absolutely not a trivial problem. Obviously, to scan the graph database sequentially and check whether a graph in the database is a supergraph of the query graph is prohibitively time-consuming. Existing subgraph search algorithms usually explore graph indexes to boost the processing of subgraph queries, and various indexing strategies have been proposed [8–12]. However, existing approaches are mainly based on centralized computing systems and evaluated on relatively small databases, each of which contains tens of thousands of small graphs with dozen of nodes and edges. Due to the combinatory issue, these approaches will face the scalability problem when dealing with large-scale graph databases, which consist of tens of millions of relatively large graphs with hundreds of nodes and edges. What is more, storage also faces challenge as the data amount of the graph database and its indexes is very huge. So how to scale up subgraph search algorithms to massive graph databases is an urgent and significant research issue, as the graph data amount is expanding drastically.

Cloud computing [17] is emerging as a new computing paradigm that has many merits. A cloud usually contains a cluster of nodes, each of which has computing and storage resources of its own. These resources are shared across the cluster at the cloud's disposal. Fault-tolerance is an intrinsic property of cloud computing. When a job is issued to a cloud, it will automatically split the relatively large job into small tasks, which will be scheduled to different nodes for execution. If some nodes on which some tasks are executing fail, these tasks will be re-assigned to some other running nodes for re-execution. Another important property of cloud computing is load-balancing. A cloud will monitor each node in the cluster and ensure that each node will get almost equal number of tasks to execute. Overall, cloud computing is efficient in handling both CPU intensive and I/O intensive jobs. Considering that subgraph search is both CPU intensive and I/O intensive, so implementing subgraph search on cloud computing platforms to exploit their advantages of scalability and elasticity is a natural choice.

This paper studies subgraph search on large-scale graph databases in cloud computing environments. Concretely, we propose and implement an efficient subgraph search approach under the MapReduce [18] framework that is a typical cloud-oriented parallel programming model. We also conduct experiments to validate the proposed approach. The rest of the paper is organized as follows: Section 2 reviews the related work. Section 3 presents an overview of our subgraph search approach. Section 4 introduces the implementation details of our approach. Section 5 gives the experimental evaluation on the proposed approach. Section 6 concludes the paper and pinpoints some future works.

## 2 Related Work

Due to space limit, here we give a brief review on the related work, including graph indexing/search and MapReduce-based computing.

## 2.1 Graph Search and Indexing

In the past years, graph search has been extensively studied as a centralized computing issue in database area. For improving processing efficiency, indexes are widely used. Up to now, various indexing strategies for graph search have been proposed [8–12]. To reduce space overhead, usually only significant graph elements are indexed. So this is a feature-filtering based indexing scheme. The process is like this: indexes are built on the graph database using a set of selected feature $F = \{f_1, f_2, \cdots, f_m\}$, each feature may be an edge, a node, or a keyword appearing in the labels of edges and/or nodes. When a query graph $q$ is issued, the graph database will be checked by the following rule (maybe with a subsequent verification phase): for each graph $g \in G$, if $\exists f \in F$ such that $f \subseteq q$ and $f \nsubseteq g$, then $q \nsubseteq g$, $g$ is filtered out.

Graphgrep [12] builds the indexes by enumerating all paths with length up to $L$, of all graphs in the database, and filters graphs by paths when doing search. Although it is fast to index paths with length limit and the index size can be kept small, structural information of graphs is lost and the filtering power of paths is limited, which will lead to a large candidate set and subsequently high verification cost. [8] uses subgraphs to keep structural information and to improve filtering power. The feature set consists of discriminative frequent subgraphs mined from the graph database. Though better filtering power is achieved, the verification cost is still high. TreePi [13] builds feature set by mining discriminative frequent subtrees from the graph database, and uses the subtree feature set to filter database graphs. Still, the filtering power is limited and verification cost is relatively high. There are other indexing schemes, either closure-based [11] or coding-based [15, 16]. We will not go into any further detail about graph indexing, interested readers can refer to [1].

In this paper, we also use indexes to enhance graph search efficiency. We build indexes directly using graph edges. Considering the elasticity of resources in cloud platforms, we do not try to filter edges for indexing.

## 2.2 MapReduce-Based Computing

In this paper, we propose a cloud-based subgraph query approach to decomposing the relatively huge subgraph search job into multiple relatively small tasks, and then run these small tasks on a Hadoop [22] cluster for parallel execution. The Hadoop cluster consists of HDFS [23] and MapReduce [18]. HDFS is a scalable distributed file system that is capable of storing massive data, and it is the open-source implementation of Google's GFS [19]. GFS provides fault tolerance while running on inexpensive commodity hardware, and is capable of delivering high aggregate performance to a large number of clients.

MapReduce is a parallel programming model, it has now become a typical and popular cloud computing framework for data-intensive parallel computation in shared-nothing clusters. A MapReduce job consists of a Map phase and a Reduce phase. In Map phase, workers (computing nodes) called Mappers invoke user-defined map function(s) to process key/value pairs to generate a set of intermediate key/value pairs. In Reduce phase, workers called Reducers invoke user-defined reduce function(s) to merge all intermediate key/value pairs associated with the same intermediate key value. A large input file is first partitioned into several splits, each of which is fed to a Mapper as input. Input splits of different Mappers can be processed in parallel, the results are forwarded to the Reducers for merging.

A number of high-level applications have been developed on MapReduce because of its ease of use for parallel execution. In addition to data management applications [20, 21], there are also some graph-related works on cloud platforms. Kang et al. [25] created a software library using Hadoop that performs typical graph mining tasks in big graphs, including degree distributions and PageRank, diameter estimation [26], connected components and triangle counting. Recently, Gu et al. [24] implemented a breath first search (BFS) algorithm in graph analysis using Sector/Sphere, a MapReduce-like cloud computing programming model. However, none of them deals with subgraph search in large graph databases.

## 3 Cloud-Based Subgraph Search: An Overview

In this section, we give an overview of our subgraph search approach implemented in the MapReduce framework. The main idea is to build inverted edge indexes for the graphs in the database, and when processing queries, only the data related to the queries is checked, from which the final results are obtained. Fig. 1 shows the overview of the cloud-based subgraph search approach.

Our approach consists of two phases: the off-line index building phase and the online subgraph query processing phase. In the first phase, we build inverted indexes by two MapReduce jobs: one is responsible for building inverted indexes for each unique edge in the graph database, the other is responsible for building indexes over the inverted indexes for each unique edge built in the first phase, which is to construct the mappings between edges and their offsets in the inverted index files. In the second phase, subgraph queries are processed. When a

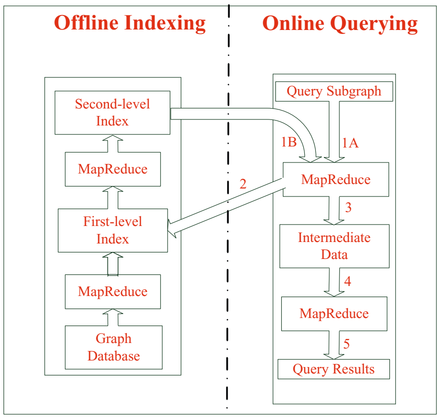


**Fig. 1.** An overview of cloud-based subgraph search

subgraph query is issued, two MapReduce jobs are launched. The first MapReduce job is to retrieve the candidate results by using indexing information, the second MapReduce job is to evaluate the final query results by employing set intersection operations.

## 4 Implementation Techniques

In this section we describe the details of the implementation of the cloud-based subgraph search approach.

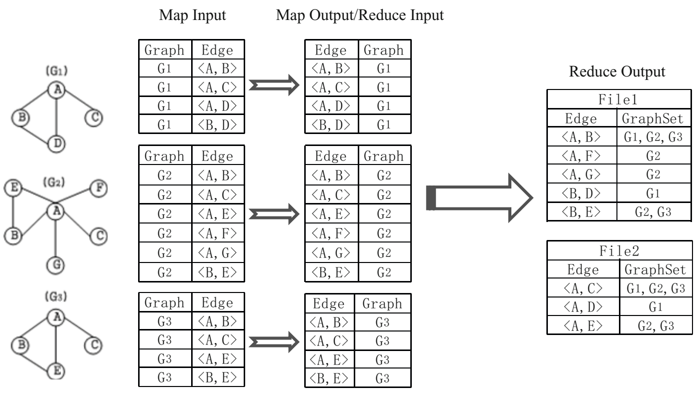
### 4.1 Index Building

Two MapReduce jobs are used to build the indexes over the edge set of the graph database. The logics of building index are presented in Algorithm 1. The first MapReduce job takes the graph database file(s) as input and builds inverted indexes over the edge set, as the first-level index. In the graph database, each edge of a graph is represented as a *GraphNo / EdgeLabel* pair where *GraphNo* represents the graph containing the indexed edge, *EdgeLabel* consists of the labels of the edge's two end vertices. If a graph consists of ten edges, then ten *GraphNo / EdgeLabel* pairs will be stored in the graph database, with identical *GraphNo*. The data of graph database is divided into a number of splits, which are distributed over different nodes of the cloud platform.

**Algorithm 1.** Offline-Index-Building

```
    INPUT: the graph database $D$
    OUTPUT: the bi-level index
1:  First-level Indexing Job
2:    Map Task
3:    for each graph labeled with $G_i$ in graph database $D$
4:      for each edge labeled with $EdgeLabel_i$
5:        output ($EdgeLabel_i$,$G_i$)
6:    Reduce Task
7:    for each edge labeled with $EdgeLabel_i$
8:      concat all graph labels to generate a GraphSet string $GS_i$: $G_{i1}$,$G_{i2}$,$\cdots$,$G_{in}$
9:      output an entry: ($EdgeLabel_i$,$GS_i$) in a first-level index file $t_j$
10:
11: Second-level Indexing Job
12:   Map Task
13:   for each entry: ($EdgeLabel_i$,$GS_i$) in file $t$
14:     get file name of $t$ : $fileName_i$, and offset of this entry in $t$ : $offSet_i$
15:     output an entry: ($EdgeLabel_i$,$fileName_i$,$offSet_i$) in a second-level index file $t'_j$
```

The logics in the Map and Reduce functions are very simple. Each record of the input splits is reversed by the Map function to output a *EdgeLabel / GraphNo* pair. A Hash function is used to repartition all outputs of the Mappers, by hashing *EdgeLabel* of each Mapper output pair. Those pairs with equal hash value share the same partition. A partition is a part of the input of a Reducer, which invokes the Reduce function. In the Reduce function, *EdgeLabel / GraphNo* pairs



Map Input

| Graph | Edge |
|---|---|
| G1 | ⟨A, B⟩ |
| G1 | ⟨A, C⟩ |
| G1 | ⟨A, D⟩ |
| G1 | ⟨B, D⟩ |

| Graph | Edge |
|---|---|
| G2 | ⟨A, B⟩ |
| G2 | ⟨A, C⟩ |
| G2 | ⟨A, E⟩ |
| G2 | ⟨A, F⟩ |
| G2 | ⟨A, G⟩ |
| G2 | ⟨B, E⟩ |

| Graph | Edge |
|---|---|
| G3 | ⟨A, B⟩ |
| G3 | ⟨A, C⟩ |
| G3 | ⟨A, E⟩ |
| G3 | ⟨B, E⟩ |

Map Output/Reduce Input

| Edge | Graph |
|---|---|
| ⟨A, B⟩ | G1 |
| ⟨A, C⟩ | G1 |
| ⟨A, D⟩ | G1 |
| ⟨B, D⟩ | G1 |

| Edge | Graph |
|---|---|
| ⟨A, B⟩ | G2 |
| ⟨A, C⟩ | G2 |
| ⟨A, E⟩ | G2 |
| ⟨A, F⟩ | G2 |
| ⟨A, G⟩ | G2 |
| ⟨B, E⟩ | G2 |

| Edge | Graph |
|---|---|
| ⟨A, B⟩ | G3 |
| ⟨A, C⟩ | G3 |
| ⟨A, E⟩ | G3 |
| ⟨B, E⟩ | G3 |

Reduce Output

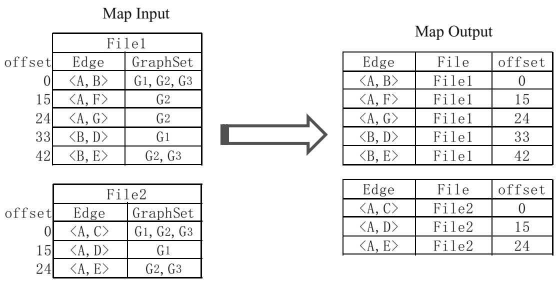
| File1 | |
|---|---|
| Edge | GraphSet |
| ⟨A, B⟩ | G1, G2, G3 |
| ⟨A, F⟩ | G2 |
| ⟨A, G⟩ | G2 |
| ⟨B, D⟩ | G1 |
| ⟨B, E⟩ | G2, G3 |

| File2 | |
|---|---|
| Edge | GraphSet |
| ⟨A, C⟩ | G1, G2, G3 |
| ⟨A, D⟩ | G1 |
| ⟨A, E⟩ | G2, G3 |

**Fig. 2.** Building the first-level index

with identical *EdgeLabel* are aggregated together, and the Reduce function generates an *EdgeLabel* / *GraphSet* pair where the *GraphSet* consists of *GraphNos* with similar *EdgeLabel*. When all Reducers finish execution, the first-level index is built, and multiple first-level index files are created. Each unique edge of the graph database appears only once in all these first-level index files. Which of the first-level index files a certain inverted edge index entry belongs to is determined by the Hash function. The process of the above MapReduce job is illustrated in Fig. 2.

When the first MapReduce job finishes, a second MapReduce job is launched to build the second-level index. The second-level index constructs the mappings between edges and their offsets in the first-level index files. Each of the first-level index files is treated as a whole input split for a Mapper of the second MapReduce job. No Reducers are initiated here. The logic in this Map function is also simple. When a *EdgeLabel* / *GraphSet* pair in any of the first-level index files is processed by the Map function, the file name and the offset of the current inverted index entry in the first-level index files are recorded. Then the Map function outputs an *EdgeLabel* / ⟨FileName, *Offset*⟩ pair, which corresponds to an entry in the second-level index. The whole process is presented in Fig. 3.



Map Input

| | File1 | |
|---|---|---|
| offset | Edge | GraphSet |
| 0 | ⟨A, B⟩ | G1, G2, G3 |
| 15 | ⟨A, F⟩ | G2 |
| 24 | ⟨A, G⟩ | G2 |
| 33 | ⟨B, D⟩ | G1 |
| 42 | ⟨B, E⟩ | G2, G3 |

| | File2 | |
|---|---|---|
| offset | Edge | GraphSet |
| 0 | ⟨A, C⟩ | G1, G2, G3 |
| 15 | ⟨A, D⟩ | G1 |
| 24 | ⟨A, E⟩ | G2, G3 |

Map Output

| Edge | File | offset |
|---|---|---|
| ⟨A, B⟩ | File1 | 0 |
| ⟨A, F⟩ | File1 | 15 |
| ⟨A, G⟩ | File1 | 24 |
| ⟨B, D⟩ | File1 | 33 |
| ⟨B, E⟩ | File1 | 42 |

| Edge | File | offset |
|---|---|---|
| ⟨A, C⟩ | File2 | 0 |
| ⟨A, D⟩ | File2 | 15 |
| ⟨A, E⟩ | File2 | 24 |

**Fig. 3.** Building the second-level index

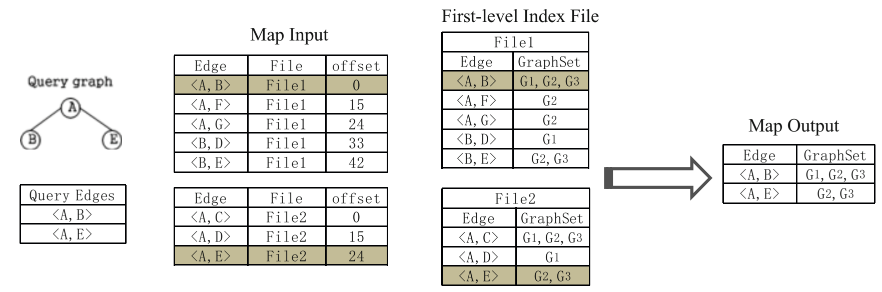
## 4.2 Subgraph Search

When a query is issued, two MapReduce jobs are initiated to process the subgraph query. The logics of executing graph query are presented in Algorithm 2. The first MapReduce job is used to retrieve the inverted edge index entries whose corresponding edges are contained in the query graph. The second MapReduce job performs a series of set intersection operations to generate the final query results. The query graph is represented as a set of *EdgeLabels*, each of which consists of the labels of the two end vertices.

**Algorithm 2.** Online-Graph-Querying

```
    INPUT: second-level index
    PARAMETER: query graph $q$ with $l$ edges
    OUTPUT: final query result
 1: First-level Index Entry Retrieval Job
 2:   Map Task
 3:   for each entry: $(EdgeLabel_i, fileName_i, offSet_i)$ in a second-level index file $t'_j$
 4:     if edge labeled with $EdgeLabel_i \in q$
 5:       $fstream\ in = $ open$(fileName_i)$
 6:       $in.seek(offSet_i)$
 7:       $in.read(EdgeLabel_i, GS_i)$
 8:       output an entry: $(EdgeLabel_i, GS_i)$ in an intermediate file $f$
 9:
10: Graph Set Intersection Job
11:   Map Task
12:   declare a set variable: $lqr_k$ to store the local intersection result
13:   for each entry: $(EdgeLabel_{ij}, GS_{ij})$ in file $f$
14:     if $j == 1$
15:       $lqr_k = GS_{ij}$
16:     else
17:       perform $lqr_k = lqr_k$ intersects with $GS_{ij}$
18:   output (null,$lqr_k$) in file $f'$
19:   Reduce Task
20:   declare a set variable: $gqr$ to store the final query result
21:   for each entry: (null,$lqr_{ki}$) in file $f'$
22:     if $i == 1$
23:       $gqr = lqr_{ki}$
24:     else
25:       perform $gqr = gqr$ intersects with $lqr_{ki}$
26:   output (null,$gqr$)
```

The first MapReduce job takes as input the second-level index files and the query graph. Each of the second-level index files is processed by a Mapper as a whole split. Each Mapper sequentially processes the records in the input splits assigned to it, and all Mappers do their works in parallel. When an *EdgeLabel* / ⟨ *FileName*, *Offset*⟩ pair is input to a Mapper, the Map function will check whether the *EdgeLabel* is contained in the query graph. If the *EdgeLabel* is contained in the query graph, the Mapper will open the *FileName* file and seek to *Offset* location, and then read a series of *GraphNos* from the *FileName* file. The outputs of Mappers are *EdgeLabel* / *GraphSet* pairs, here *EdgeLabels* are query edges. This process is shown in Fig. 4. No Reducer is initiated in this MapReduce job. The outputs of Mappers are merged into one file for the second MapReduce job to further process.



| Query Edges |
|---|
| <A, B> |
| <A, E> |

Map Input

| Edge | File | offset |
|---|---|---|
| <A, B> | File1 | 0 |
| <A, F> | File1 | 15 |
| <A, G> | File1 | 24 |
| <B, D> | File1 | 33 |
| <B, E> | File1 | 42 |

| Edge | File | offset |
|---|---|---|
| <A, C> | File2 | 0 |
| <A, D> | File2 | 15 |
| <A, E> | File2 | 24 |

First-level Index File

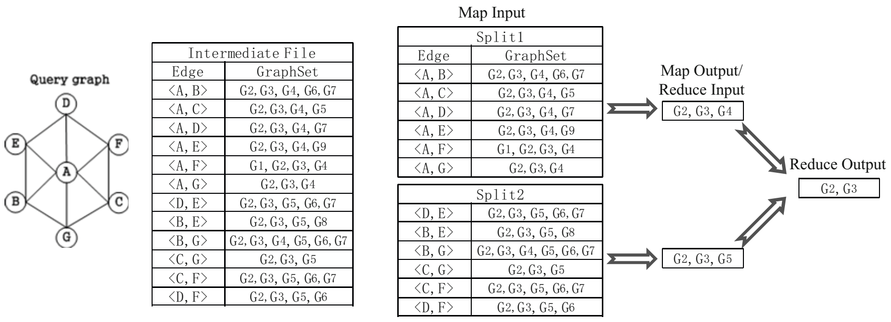
| File1 | |
|---|---|
| Edge | GraphSet |
| <A, B> | G1, G2, G3 |
| <A, F> | G2 |
| <A, G> | G2 |
| <B, D> | G1 |
| <B, E> | G2, G3 |

| File2 | |
|---|---|
| Edge | GraphSet |
| <A, C> | G1, G2, G3 |
| <A, D> | G1 |
| <A, E> | G2, G3 |

Map Output

| Edge | GraphSet |
|---|---|
| <A, B> | G1, G2, G3 |
| <A, E> | G2, G3 |

**Fig. 4.** Retrieving the first-level index entries



| Intermediate File | |
|---|---|
| Edge | GraphSet |
| <A, B> | G2, G3, G4, G6, G7 |
| <A, C> | G2, G3, G4, G5 |
| <A, D> | G2, G3, G4, G7 |
| <A, E> | G2, G3, G4, G9 |
| <A, F> | G1, G2, G3, G4 |
| <A, G> | G2, G3, G4 |
| <D, E> | G2, G3, G5, G6, G7 |
| <B, E> | G2, G3, G5, G8 |
| <B, G> | G2, G3, G4, G5, G6, G7 |
| <C, G> | G2, G3, G5 |
| <C, F> | G2, G3, G5, G6, G7 |
| <D, F> | G2, G3, G5, G6 |

| Split1 | |
|---|---|
| Edge | GraphSet |
| <A, B> | G2, G3, G4, G6, G7 |
| <A, C> | G2, G3, G4, G5 |
| <A, D> | G2, G3, G4, G7 |
| <A, E> | G2, G3, G4, G9 |
| <A, F> | G1, G2, G3, G4 |
| <A, G> | G2, G3, G4 |

| Split2 | |
|---|---|
| <D, E> | G2, G3, G5, G6, G7 |
| <B, E> | G2, G3, G5, G8 |
| <B, G> | G2, G3, G4, G5, G6, G7 |
| <C, G> | G2, G3, G5 |
| <C, F> | G2, G3, G5, G6, G7 |
| <D, F> | G2, G3, G5, G6 |

**Fig. 5.** Evaluating graph sets intersection

The second MapReduce job performs a series of set intersection operations to generate the final query results. Initially, we have tried to do intersection operations on the graph sets output by the first MapReduce job on the coordinator of the MapReduce cluster, but we found that it took too long time for the coordinator to finish the intersection operations by itself. So, we finally decided to start a second MapReduce job to do graph sets intersection operations. The merged output file of the first MapReduce job is taken as input of the second MapReduce job. The input file is partitioned into several splits, each of which is processed by a Mapper. Each Mapper performs intersection operations on the graph sets contained in its input split. If a Mappers finishes its work, it outputs its local intersection results to a Reducer to process further. The Reducer performs intersection operations on the intersection results from different Mappers to generate the final query results. Only one of such Reducer is initiated. The execution process is illustrated in Fig. 5.

## 5 Experimental Evaluation

In this section, we present the experimental results of evaluating the efficiency of the cloud-based subgraph search approach.

### 5.1 Experimental Settings

We deploy a cluster with 10 nodes, each of which is a commodity PC. Each node has an Intel Duo Core2 2.93GHz CPU, 3GB memory, and Windows XP OS. We use Hadoop 0.19.2, and compile the source codes under JDK 1.6 in Eclipse 3.3.2. One of the ten nodes is used as coordinator, and the rest nine nodes are used as computing and storage nodes. As we have not found any large-scale graph database suitable for our experiment, we use synthetic datasets to evaluate the efficiency of the proposed cloud-based subgraph search approach. Experiments with real-life datasets are left for our future work. The synthetic datasets are generated according to the Erdos-Renyi random graph model. Three datasets are generated. Major statistics of the three datasets is presented in Table 1. Query graphs are randomly generated with 20 edges, 30 edges, 50 edges and 100 edges respectively.

We measure the query time to evaluate the efficiency of the cloud-based subgraph search approach. One point is worthy of being mentioned: in a distributed environment, the performance of a distributed algorithm or system is influenced by many factors, including network, computing model, resource scheduling, etc.

**Table 1.** Statistics of synthetic datasets

| Statistics | DataSet1 | DataSet2 | DataSet3 |
|---|---|---|---|
| #graphs | 50,000 | 70,000 | 100,000 |
| #vertices per graph | 30-40 | 40-70 | 50-100 |
| #edges per graph | 300-400 | 600-800 | 800-1000 |

### 5.2 Experimental Results

We first measure the query time for querying four randomly-generated graphs over the three datasets. The results are presented in Fig. 6, Fig. 7 and Fig. 8. The results seem not what we have expected. As we initially imagined, if the number of query edges increases, the query time will increase, since more time will be spent on checking whether the *EdgeLabels* of an entry in the second-level index is contained in the query graph. However, as we can see from these three figures, the query time sometimes decreases as the number of query edges increases. The reason behind this is that the computing cost spent on checking is minor, compared with that of data distribution and the subsequent graph set intersection operations. As for the MapReduce job that performs graph set intersection operations, the majority of the execution time is spent on the Reducer. The Reducer performs intersection operations on all the outputs of the preceding Mappers. So the input of Reducer may be relatively of large size, and thus it takes the Reducer much time to finish the intersection operation on the graph sets.

When the query graph contains more edges, more Mappers will be initiated to perform local graph set intersection operations, and each Mapper gets more graph sets as inputs. So the outputs of the Mappers and thus the input of the Reducer may become smaller, compared with that of query graphs with fewer

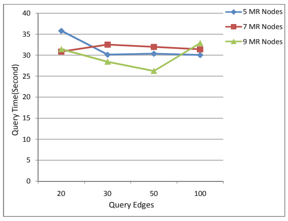


**Fig. 6.** Results over DataSet1

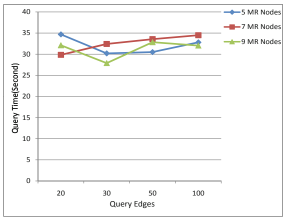


**Fig. 7.** Results over DataSet2

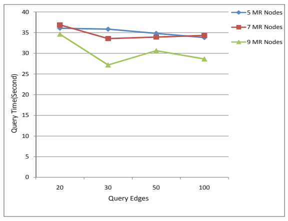


**Fig. 8.** Results over DataSet3

edges. In our experiments, the query edges affect query time little, and the query time must not decrease as the number of query edges increases. This may be not a general conclusion. For a general conclusion, more experiments with larger databases on larger-size clusters are requested, which is also left for our future work.

We also measure the average query time on each database graph. For DataSet1, the average query time spent on each database graph is 0.5 ms to 0.7 ms, for DataSet2, the average query time spent on each database graph is 0.4 ms to 0.5 ms, and for DataSet3, the average query time spent on each database graph is 0.25 ms to 0.4 ms. It seems that the average time spent on a database graph decreases as the size of the graph database increases. The reason is that the cost of network, data distribution and MapReduce job startup is shared by all database graphs. The more database graphs are, the less the shared cost by each graph is. This phenomenon validates the scalability of our subgraph searching approach, that is, our approach can scales up to massive large-scale graph databases.

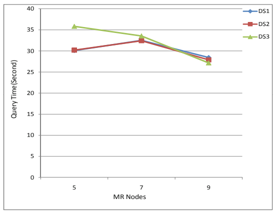


**Fig. 9.** Results for 30-edge queries

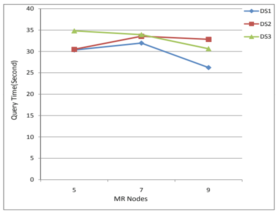


**Fig. 10.** Results for 50-edge queries

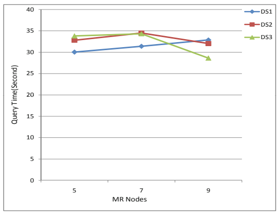


**Fig. 11.** Results for 100-edge queries

We then measure the query time when changing the size (the number of MapReduce nodes) of the cloud cluster. The experimental results are presented in Fig. 9, Fig. 10 and Fig. 11. As we can see from these three figures, the query time of a given database does not follow a monotone trend, as the number of MapReduce (MR) nodes increases. When the number of MR nodes increase to seven, the query time turns longer, compared with that of five-node cluster. But the query time decreases when the number of MR nodes increase to nine. Our explanation on this phenomenon is as follows:

For a small-scale graph database, a centralized system may be adequate to process the queries efficiently. In such cases, processing queries in a distributed system must not be advantageous over in a centralized system. For a given graph database, the query time is not determined only by the number of nodes. Adding more nodes must not yield more performance gain, because data distribution and synchronization cost may counteract the benefit of using more nodes. For all the three synthetic datasets we use, a cluster of five computing and storage nodes is adequate. When the number of nodes increases to seven, the query time increases, compared with that of query with a five-node cluster. However, when nine nodes are employed to execute queries, the query time decreases. This is because the benefit of using more nodes surpasses the overhead of data distribution and synchronization as well as the initialization of MapReduce jobs. It is not difficult to predict that the query time will increase again as the number of MR nodes continues to increase. So our conclusion from the experimental results is: it is suitable to employ a large-scale cluster to implement subgraph query for large-scale databases, and a middle-scale cluster should be employed for middle-scale graph databases.

## 6 Conclusion and Future Work

This paper studies subgraph search over large-scale graph databases in cloud computing environments. A cloud-based subgraph search approach is presented, which uses a bi-level indexing structure to boost the search efficiency. Inverted index over edge set of the graph database is built as the first-level index, and then a second-level index is built to construct the mappings between edges and offsets of their corresponding inverted index entries in the first-level index files. Experiments on synthetic datasets show that cloud-based subgraph search is efficient for large-scale graph databases.

Some optimization and improvements can be done on the proposed approach. On one hand, merging multiple queries into one can reduce I/Os and averaged startup cost of MapReduce jobs per query, and thus can improve query throughput. On the other hand, exploring more efficient lookup strategies over the second-level index will speed up the locating of query edges, as currently the second-level index is sequentially scanned.

## References

1. Aggarwal, C.C., Wang, H. (eds.): Managing and mining graph data. Kluwer Academic Publishers, Dordrecht (2010)
2. Kuramochi, M., Karypis, G.: Finding frequent patterns in a large sparse graph. In: Proceedings of SDM (2004)
3. Willett, P.: Chemical similarity searching. J. Chem. Inf. Comput. Sci. 38, 983–996 (1998)
4. Polyzotis, N., Garofalakis, M.: Statistical Synopses for Graph-Structured XML Databases. In: Proceedings of SIGMOD (2002)

5. Beretti, S., Bimbo, A., Vicario, E.: Efficient Matching and Indexing of Graph Models in Content Based Retrieval. IEEE Trans. on Pattern Analysis and Machine Intelligence 23, 1089–1105 (2001)
6. Messmer, B., Bunke, H.: A new algorithm for error-tolerant subgraph isomorphism detection. IEEE Trans. on Pattern Analysis and Machine Intelligence 20, 493–504 (1998)
7. Petrakis, E., Faloutsos, C.: Similarity searching in medical image databases. IEEE Trans. on Knowledge and Data Engineering 9(3), 435–447 (1997)
8. Yan, X., Yu, P., Han, J.: Graph Indexing Based on Discriminative Frequent Structure Analysis. ACM Transactions on Database Systems 30(4), 960–993 (2005)
9. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: Proceedings of SIGMOD (2007)
10. Williams, D.W., Huan, J., Wang, W.: Graph Database Indexing Using Structured Graph Decomposition. In: Proceedings of ICDE (2007)
11. He, H., Singh, A.K.: Closure-Tree.: An Index Structure for Graph Queries. In: Proceedings of ICDE (2006)
12. Giugno, R., Shasha, D.: Graphgrep: A fast and universal method for querying graphs. Proceedings of ICPR 2, 112–115 (2002)
13. Zhang, S., Hu, M., Yang, J.: TreePi: A Novel Graph Indexing Method. In: Proceedings of ICDE, pp. 181–192 (2007)
14. Ferro, A., Giugno, R., Mongiovi, M., et al.: GraphFind: enhancing graph searching by low support data mining techniques. BMC Bioinformatics 9 (2008)
15. Jiang, H., Wang, H., Yu, P., Zhou, S.: GString: A Novel Approach for Efficient Search in Graph Databases. In: Proceedings of ICDE (2007)
16. Zou, L., Chen, L., Jeffrey, Y.L.: A novel spectral coding in a large graph database. In: Proceedings of EDBT (2006)
17. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing. Technical Report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory (February 2009)
18. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large cluster. In: Proceedings of OSDI, pp. 137–150 (2004)
19. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. In: Proceedings of SOSP, pp. 29–43 (2003)
20. Olston, C., Reed, B., Srivastava, U., et al.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of SIGMOD, pp. 285–296 (2008)
21. Abouzeid, A., Pawlikowski, K.B., Abadi, D.J., et al.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In: Proceedings of VLDB, pp. 285–296 (2009)
22. `http://hadoop.apache.org`
23. `http://hadoop.apache.org/hdfs/`
24. Gu, Y., Lu, L., Grossman, R., Yoo, A.: Processing massive sized graphs using Sector/Sphere. In: Proceedings of the Workshop on Many-task Computing on Grids and Supercomputers (MTAGS 2010), co-located with SC 2010, New Orleans, LA (November 2010)
25. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations, In: Proceedings of ICDM 2009 (2009)
26. Kang, U., Tsourakakis, C.E., Appel, A., Faloutsos, C., Leskovec, J.: HADI: Fast diameter estimation and mining in massive graphs with Hadoop, CMU ML Tech Report CMU-ML-08-117 (2008)