Efficient Query Evaluation on Distributed Graphs with Hadoop Environment

Le-Duc Tung
The Graduate University for Advanced Studies
Japan
tung@nii.ac.jp

Quyet Nguyen-Van
Hung Yen University of Technology and Education
Vietnam
quyetict@gmail.com

Zhenjiang Hu
National Institute of Informatics
Japan
hu@nii.ac.jp

ABSTRACT

Graph has emerged as a powerful data structure to describe various data. Query evaluation on distributed graphs takes much cost due to the complexity of links among sites. Dan Suciu has proposed algorithms for query evaluation on semi-structured data that is a rooted, edge-labeled graph, and algorithms are proved to be efficient in terms of communication steps and data transferring during the evaluation. However, one disadvantage is that communication data are collected to one single site, which leads to a bottleneck in the evaluation for real-life data. In this paper, we propose two algorithms to improve Dan Suciu's algorithms: *one-pass* algorithm is to significantly reduce a large amount of redundant data in the evaluation, and *iter_acc* algorithm is to resolve the bottleneck. Then, we design an efficient implementation with only one MapReduce job for our algorithms in Hadoop environment by utilizing features of Hadoop file system. Experiments on cloud system show that *one-pass* algorithm can detect and remove 50% of data being redundant in the evaluation process on YouTube and DBLP datasets, and *iter_acc* algorithm is running without the bottleneck even when we double the size of input data.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System—*Query processing, Distributed databases*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming*

General Terms

Algorithms, Performance

Keywords

Graph databases, MapReduce, Big data

SoICT 13, December 05 - 06 2013, Danang, Viet Nam

ACM 978-1-4503-2454-0/13/12 ...$15.00
http://dx.doi.org/10.1145/2542050.2542086 .

1. INTRODUCTION

Currently, graphs are ubiquitous. The world wide web is a graph whose nodes correspond to static pages, and whose edges correspond to links between these pages [4]. Social network graph consists of edges denoting many complex relationships [12, 23]. Despite of the popularity of graph, there are still not so many research studies that focus on efficient evaluation for queries on distributed graphs whose nodes and edges are distributed on different sites.

To see the difficulties of query evaluation on distributed graphs, consider simple graph queries defined in terms of regular expressions [9]. The process of evaluating a query is to find paths in the graph that satisfy the regular expression, and return nodes and edges related to those paths. Query evaluation on a centralized graph is efficiently performed by DFS/BFS algorithms. However, with a distributed graph, query evaluation is more difficult in the sense that the amount of data transferred via network in the evaluation process is large. Because each site just has the local/partial information of the graph, sites have to communicate and exchange a lot of information with others.

Dan Suciu [21] has proposed an efficient algorithm to evaluate query on distributed graphs with distinguished properties as follows:

- The number of communication steps is four (independent of the data or query).
- The total amount of data exchanged during communications has size of $O(n^2) + O(r)$, where n denotes the number of cross-links (edges between two nodes at two different sites) in distributed database, r the size of the result of query.

However, in Dan Suciu's algorithms the amount of data $O(n^2)$ are sent to only one site to process, which results in a bottleneck in the algorithm when evaluating on social graphs where the number of cross-links is quickly increasing [13]. In this paper, we show an approach to overcoming this bottleneck with the following contributions:

- We improve the Dan Suciu's algorithm so that we can remove a large number of redundant nodes and edges in the evaluation (Section 3.1), and reduce the total amount of data transferring via network to $O(|N| \times |S| \times |P|)$, where $|N|$ denotes the number of input and output nodes, $|S|$ the number of states in automaton, and $|P|$ the number of partitions (Section 3.2).
- We propose an efficient implementation for our algorithms using MapReduce programming model in the

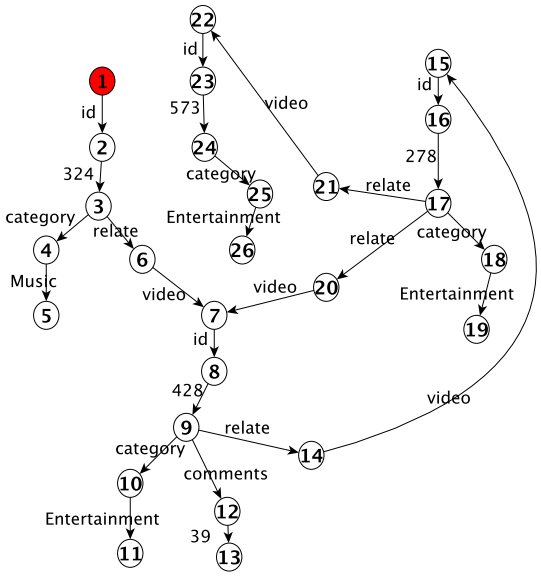

Figure 1: A rooted, edge-labeled graph of Youtube's data. The red node is the root node.

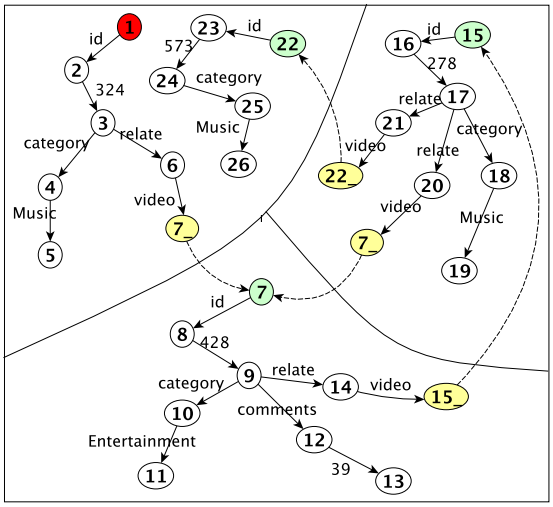

Figure 2: A distributed graph for Youtube's data with three fragments. The red node is the root. Green nodes are input ones. Yellow nodes are output ones.

Hadoop environment (Section 3.3). The implementation utilizes the features of Hadoop distributed file system (HDFS) to make the implementation correct and efficient.

The organization of this paper is as follows. In Section 2 we briefly review the key features of a well-known method for query evaluation on distributed graphs. The main contributions are in Section 3 where we show improvements on Dan Suciu's algorithm and propose an efficient implementation in Hadoop environment. Section 4 shows experimental results of our algorithms with data from Youtube and DBLP. Related works and Conclusion will be mentioned in Sections 5 and 6.

2. QUERY EVALUATION ON DISTRIBUTED GRAPH

In this section, we shall give a brief review of the existing work on query evaluation for distributed graphs [17, 21].

2.1 Distributed graph

We will represent a database by a rooted, edge-labeled directed graph. It is a graph with a unique root. Each vertex (in this paper we call it a node) has its unique identity. Each edge from a node u to node v has a label, and is denoted by $u \xrightarrow{label} v$. Label could be atomic values such as: Int, Long, String, Bool, Image, ... This data model enables us to develop simple query languages with an underlying optimized algebras for querying and graph transformation [6, 5, 7].

A distributed graph DG is a graph whose nodes are partitioned into m sets located on m sites. At each site α, nodes and correspondent edges make a fragment DG_α of the distributed graph. An edge $u \to v$ is a cross-link if u and v are stored on different sites.

For any cross-link $u \xrightarrow{a} v$ from site α to site β, we replace it with a sequence of edges $u \xrightarrow{a} v' \xrightarrow{\epsilon} v$, where v' is just a copy of v and resides on the site α, ϵ is a special label denoting an empty label. Now, $v' \xrightarrow{\epsilon} v$ becomes a cross-link and $u \xrightarrow{a} v'$ is an edge in fragment DB_α. We call $u \xrightarrow{a} v'$ an ϵ-edge. v' is called an output node of fragment DB_α, and v is an input node of fragment DB_β.

Figures 1 and 2 are examples of a rooted edge-labeled directed graph whose root is the node with value "1", and dotted edges are cross-links with ϵ label. Input nodes are in green and output nodes are in yellow.

2.2 Query evaluation

A query Q on a graph DG is a select-where query that is a subset of UnQL query language [6, 7]. For simplicity, we just consider a query with one regular path expression:

select t
where $R \Rightarrow t$ **in** DG

Here t is a variable that stores records returned, R is a regular path expression:

$$R = a \mid \_ \mid R|R \mid R \Rightarrow R \mid R* \mid R$$

where a is a label including ϵ label. _ denotes any label, $R \mid R$ is an alternation, $R \Rightarrow R$ denotes concatenation, and $R*$ is the Kleene closure.

Evaluation of a query Q on a graph DG is denoted by $DG(Q)$.

A problem of query evaluation on a distributed graph is stated as follows: Given a distributed graph $DG = \bigcup_{\alpha=1,m} DG_\alpha$, and a query Q, compute $DG(Q)$.

A well-known method for query evaluation [17, 21] based on partial evaluation technique [10] consists of 6 steps:

Step 1: Client computes an automaton $\mathcal{A}$ for the regular expression R in the query Q, then sends $\mathcal{A}$ to each sites.

Step 2: At each site α, compute a partial result P_α as in Algorithm 1. Note that if a node u is a root node then (1) it is also an input node, (2) a node (s, u) is an input node,

Algorithm 1: Partial result computation

input : A fragment DG_α, an automaton $\mathcal{A}$
output: A partial result P_α

begin
  $P_\alpha \leftarrow DG_\alpha$; `/* copy nodes and edges from` DG_α `to` P_α `*/`
  foreach $u \in InputNodes(DG_\alpha)$ **do**
    foreach $s \in States(\mathcal{A})$ **do**
      $S \leftarrow visit(s, u)$;
      foreach $p \in S$ **do**
        adding an edge $(s, u) \xrightarrow{\epsilon} p$ to P_α;
      end
    end
  end
end

where s is a state in the automaton $\mathcal{A}$. If u is an input or output node, then (s, u) is an input or output respectively.

Step 3: Compute a local accessible graph LAG_α from P_α. LAG_α contains all input and output nodes from P_α and an edge from an input node to an output node if there exists a path between them in P_α.

Step 4: Each site α sends its LAG_α to client where $LAGs$ will be combined together, then be added cross-links to get a global accessible graph GAG. Starting from the node $(s_{initial}, u_{root})$ where $s_{initial}$ is the initial state in $\mathcal{A}$, u_{root} is the root node of DG, we find all nodes accessible in GAG, then send the set of accessible nodes back to each site.

Step 5: At each site α, compute a partial answer as follows. With each element in the set of accessible nodes, we find its successors in P_α and add them to the partial answer. Finally, send the partial answer to client.

Step 6: Client will combine partial answers together, and add cross-links to get a final answer of the query. The final answer is the result of $DG(Q)$.

In step 4, the total amount of data transferring is $O((|N| \times |S|)^2)$ where $|N|$ is the number of cross-links, $|S|$ is the number of state in the automaton. Note that these data are aggregated into client. In step 5, that is $O(r)$, where r is the size of query answer. Therefore, the total data in communication for the evaluation is $O((|N| \times |S|)^2) + O(r))$. [21, 9].

3. EFFICIENT EVALUATION WITH HADOOP ENVIRONMENT

In this section we focus on describing our framework that efficiently evaluates queries on distributed graphs.

Figure 3 shows the overview of the framework that consists of three phases. In the first phase, we compute both a partial result and a local accessible graph in one step. This computation is performed locally at each site. In the second phase, we compute accessible nodes from LAGs in the distributed way. Each site will communicate with others to update its LAG. The amount of data in communication will be computed in detail in Section 3.2. After that, we get new LAGs at each site. These new LAGs contain nodes accessible from the root node $(s_{initial}, u_{root})$. Note that our partial results are unchanged in the phase 2. Finally, we extract accessible nodes from new LAGs locally, and construct partial answers by doing a BFS for each accessible node over

Algorithm 2: Visiting algorithm

input : A node s, a state s
output: A set of node
`visit(`*s, u*`)` **begin**
  if $(s, u) \in$ *visited* **then**
    return global_matches[(s,u)];
  end
  visited $\leftarrow$ (s,u);
  matches $\leftarrow$ {};
  if *u is output node* **then**
    matches $\leftarrow$ (s,u);
    global_matches[(s,u)] $\leftarrow$ matches;
  else if *s is terminal state* **then**
    matches $\leftarrow$ u;
    global_matches[(s,u)] $\leftarrow$ matches;
  foreach $u \xrightarrow{a} v$ *in* DG_α **do**
    foreach $s \xrightarrow{x} s'$ *satisfies* $x == a$ **do**
      matches $\leftarrow$ matches $\cup$ $visit(s', u)$;
      global_matches[(s,u)] $\leftarrow$ matches;
    end
  end
  return matches;
end

partial results. Partial answers are fragments of the query's final answer. A combinator is used to gather partial answers and add cross-links between them to make the final answer.

3.1 One-pass evaluation

3.1.1 Observations

Before going to the detail of our *one-pass* evaluation, we briefly review the form of partial results and local accessible graphs.

A partial result consists of the whole nodes and edges in a fragment. Besides, it has additional nodes in the form of (s, u), where s is a state in the automaton, u an input or output node. An example of a partial result is shown in Figure 4. The input fragment for that partial result includes nodes indexed from 1 to 8 and edges between them. Here we have two input nodes: node 1 and node 2; three output nodes: node 5, node 6, and node 8. The edge $(s1, 1) \xrightarrow{\epsilon} (s2, 5)$ is there because, starting from node 1 with state $s1$, we can reach node 5 at state $s2$ by following transitions in the automaton. Since $s3$ is the terminal state, we have edges $(s3, 1) \xrightarrow{\epsilon} 1$, $(s3, 2) \xrightarrow{\epsilon} 2$. We can not find any node that matches the automaton starting from node 1 or node 2 with state $s2$, therefore there is no edge emanating from these nodes.

A LAG contains all input and output nodes from a partial result and ϵ-edges from input nodes to output nodes if there exists a path between them. Figure 5 is an example of a LAG for the partial result in Figure 4.

We have some important observations on LAGs as follows.

Observation 1: *In a partial result, if an input node* (s, u) *has no links to other nodes, then we remove it from LAG.* We prove this by considering the state s: if s is a terminal state, then there always exists an edge $(s, u) \rightarrow u$ (Algorithm 2), therefore in this case, s is not a terminal state. Because s is not a terminal state, u is not the final node yet that satisfies the automaton. Besides, we can not go further along the

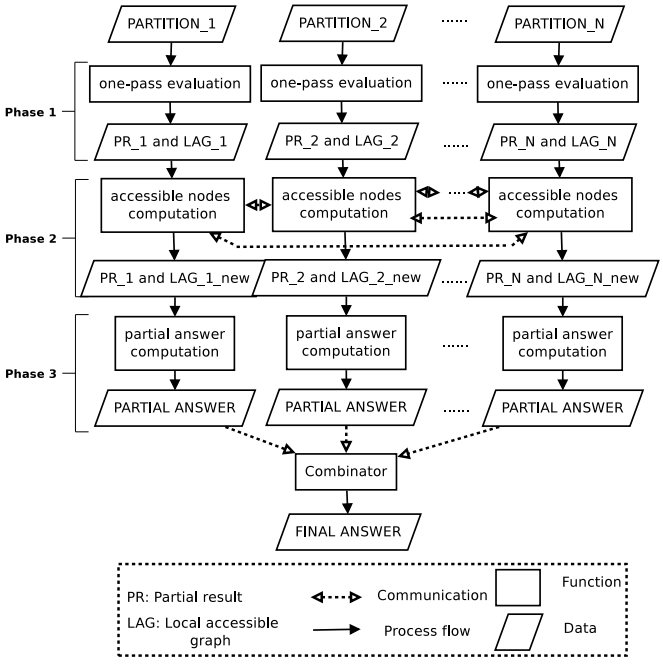

Figure 3: Architecture of our query evaluation framework.

automaton, so the node u does not contribute to the final result, it means that we can remove the node (s, u).

Observation 2: *For a terminal state $s_{terminal}$, if there is an ϵ-edge $(s_{terminal}, u_{in}) \xrightarrow{\epsilon} u_{in}$ in partial result, then in LAG only ϵ-edges $u_{in} \xrightarrow{\epsilon} v_{out}$ are enough, where v_{out} is an output node that is reachable from u_{in}.* It means that, we do not need to add edges $(s_{terminal}, u_{in}) \xrightarrow{\epsilon} v_{out}$ to LAG. This is easily proved by showing that if existing a path from $(s_{terminal}, u_{in})$ to v_{out}, then the path must contain u_{in}. According to Algorithm 2, when visiting node $(s_{terminal}, u_{in})$, because $s_{terminal}$ is a terminal state, we add an edge $(s_{terminal}, u_{in}) \xrightarrow{\epsilon} u_{in}$ to LAG and finish the visit of $(s_{terminal}, u_{in})$. Therefore, the only way to go from $(s_{terminal}, u_{in})$ to v_{out} is via u_{in}.

Observation 3: *Assuming that between fragment $\mathcal{F}_i$ and fragment $\mathcal{F}_j$ there is a cross-link $u' \rightarrow u$, $u' \in \mathcal{F}_i$, $u \in \mathcal{F}_j$. Then, there always exists edges $(s_{terminal}, u) \rightarrow u$ in LAG_j of $\mathcal{F}_j$. We prove that these edges are redundant.* In LAG_j of $\mathcal{F}_j$, we consider two cases: (1) If there exists an ϵ-edge $w \rightarrow (s_{terminal}, u')$, then we have a path as follows: $w \xrightarrow{\epsilon} (s_{terminal}, u') \xrightarrow{\epsilon} (s_{terminal}, u) \xrightarrow{\epsilon} u$. That path can be replaced by $w \xrightarrow{\epsilon} u' \xrightarrow{\epsilon} u$. (2) If there does not exist an ϵ-edge $w \rightarrow (s_{terminal}, u')$, then there is no path coming to node $(s_{terminal}, u)$, we can remove the edge $(s_{terminal}, u) \rightarrow u$.

3.1.2 One-pass Algorithm

Applying above observations, we can remove a large of redundant nodes and edges from LAGs (Example in Figure 6), leading to significantly decreasing the amount of data during the computation of accessible nodes. Also by the observations, we re-organize a local accessible graph into three parts: *the first part* contains only edges between non-state input nodes and non-state output nodes (nodes without state); *the second part* includes of only edges between

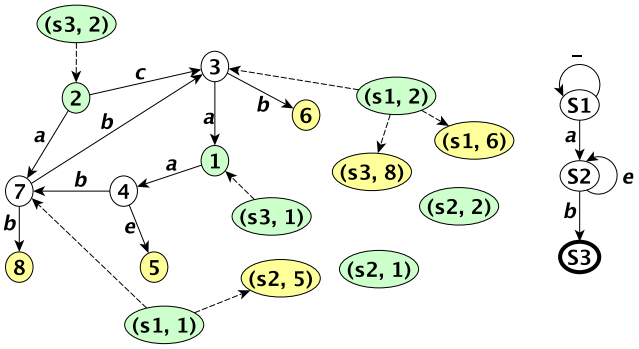

Figure 4: Partial result.

nodes having states (these nodes and edges are computed by the visiting algorithm); and *the remaining part* consists of edges from input nodes having state to other non-state output nodes, these edges show us that there is a strong probability that result of query can be appeared in current site.

We re-design algorithms in Section 2 so that it computes both partial results and LAGs without redundant data. The algorithm is shown in Algorithm 3. For each input node, firstly we compute output nodes that can be reached from that input node, which constructs the first part of LAG. The second part and third part of LAG is copied from result of the visiting algorithm for each node of (s, u_{in}), here we do not visit with terminal state in order to remove all redundant edges of $(s_{terminal}, u_{in}) \xrightarrow{\epsilon} u_{in}$. The visiting algorithm (Algorithm 4) is almost the same as the former one. There is only one difference that is we check whether a node reaches a terminal state or not before checking whether it is an output node, this will replace all edges of $(s, u) \xrightarrow{\epsilon} (s_{terminal}, v_{out})$ by $(s, u) \xrightarrow{\epsilon} v_{out}$.

3.2 Efficient computation of accessible nodes

As discussed in Section 2, to compute accessible nodes we have to combine all LAGs into a global accessible graph (GAG). The size of GAG is $O((|N| \times |S|)^2)$. This is also the amount of data we have to send over network. In social networks where the number of relationship is large, the number of cross-links is also increasing. This causes a bottleneck in evaluation of query. We see that the number of accessible nodes is not more than the number of node in the final answer because we use accessible nodes to extract the final answer from partial results. Therefore sending out all nodes and edges in LAGs is not necessary.

Borrowing the idea of distributed BFS [15], we propose an efficient iterative algorithm to compute accessible nodes, named *iter_acc* algorithm. In the computation process, each node in a *LAG* will have one of two values: either ACC node or OR node, hence we refer a LAG as an ACC/OR graph. Our algorithm is as follows.

Step 1: Represent a LAG by adjacent lists, in which for each input node there is a set of output nodes linked with it. Let a list of adjacent nodes of u be $adj(u)$. Initially, marking all nodes in LAGs as OR nodes.

Step 2: Create a list to keep accessible nodes during the computation process, named acc_nodes_α for each site α. Initially, acc_nodes_α contains only the root node (s, r) (s is

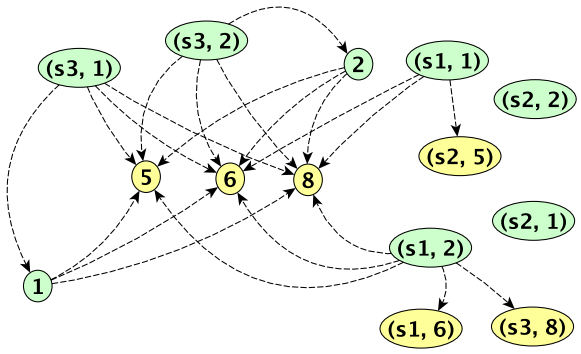

Figure 5: Local accessible graph (LAG).

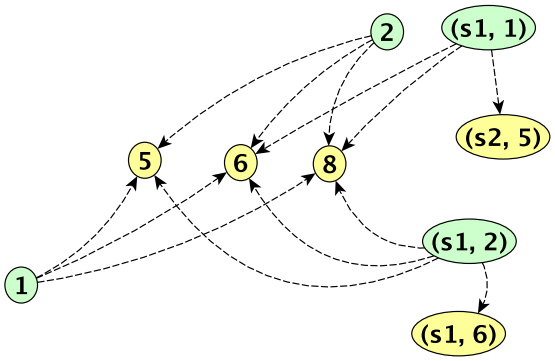

Figure 6: LAG after reducing redundant vertices.

Algorithm 3: *One-pass* algorithm

input : A fragment DG_α, an automaton $\mathcal{A}$
output: A partial result P_α, a LAG_α

```
begin
    P_α ← DG_α ; /* copy nodes and edges from DG_α
    to P_α */
    foreach u ∈ InputNodes(DG_α) do
        IO ← FindOutNodes(u);
        foreach v ∈ IO do
            LAG_α ← (u -ε-> v);
        end
        foreach s ∈ States(A)\{s_terminal} do
            S ← visit(s, u);
            foreach p ∈ S do
                if u has a state then
                    adding an edge (s, u) -ε-> p to P_α;
                    adding an edge (s, u) -ε-> p to LAG_α;
                else
                    IO ← FindOutNodes(u);
                    foreach v ∈ IO do
                        LAG_α ← (u -ε-> v);
                    end
            end
        end
    end
end
```

an initial state of the automaton, r is the root node of the input graph DG).

Step 3: Each site α sends its acc_nodes_α to all other sites.

Step 4: Each site α receives acc_nodes_i, $i = 1..n, i \neq \alpha$, from others and combines them into its acc_nodes_α, then removes duplicate nodes.

Step 5: Each site α uses its acc_nodes_α to update its LAG_α. The update process is as follows: For each node $u \in acc\_nodes_\alpha$:

- If $u \in LAG_\alpha$, then (1) if u is an OR node, then update it to an ACC node. (2) For each node v in $adj(u)$, if v is an OR node, then update v to an ACC node and add v to acc_nodes_α.
- If $u \notin LAG_\alpha$, do nothing.

Algorithm 4: A modified visiting algorithm

input : A node u, a state s
output: A set of node

```
visit(s, u) begin
    ⋮
    if s is a terminal state then
        matches ← u;
        global_matches[(s,u)] ← matches;
    else if u is an output node then
        matches ← (s,u);
        global_matches[(s,u)] ← matches;
    ⋮
end
```

- Removing u from acc_nodes_α.

Step 6: Repeat Step 3 until all acc_nodes_i, $i = 1..n$, is empty.

Step 7: At each site α, extract all ACC nodes from LAG_α, we have a set of accessible nodes for its fragment $\mathcal{F}_\alpha$.

An example of the above algorithm is shown in Figure 7 for two LAGs distributed on two different sites.

The amount of data transferring over network in our algorithm is $O(|N| \times |S| \times |P|)$, where $|N|$ denotes the number of input and output nodes, $|S|$ the number of state in automaton, $|P|$ the number of partitions. By using hash data structure to store LAGs, we check whether u in LAG_α or not in the time complexity of $O(1)$. Because the organization of LAGs is simple in the sense that it just contains edges between input and output nodes, the number of levels to explore in each iteration is only one. This will significantly reduce the overhead between iterations.

3.3 Efficient Hadoop-based implementation

3.3.1 Hadoop environment

Hadoop [2] is an open source framework for MapReduce programming model [8]. It is proved to be efficient and scalable. In MapReduce programming model, we only need to write two functions: Map and Reduce. Map functions accept a pair of *(key1, value1)* as its input and produce a list

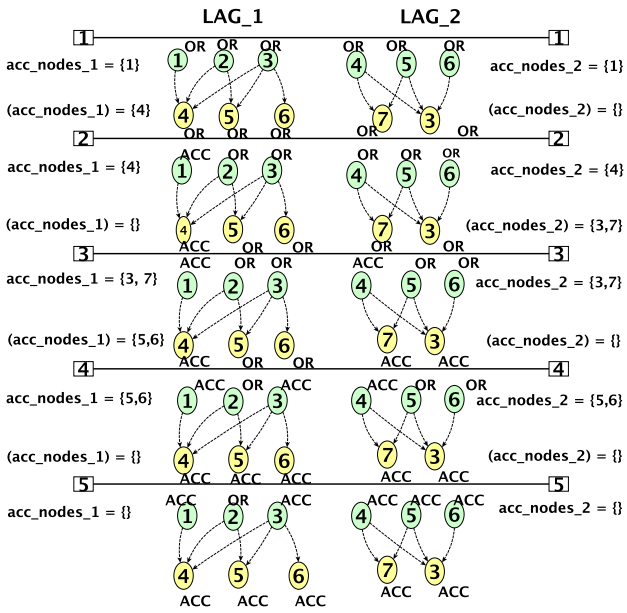

Figure 7: An example of computation of accessible nodes with five iterations. *acc_nodes_i* **denotes value of** *acc_nodes* **in site** i **before each iteration.** (*acc_nodes_i*) **denotes value of** *acc_nodes* **in site** i **after each iteration. Numbers in square boxes denote the iteration number.**

of pairs of *(key2, value2)*. After that, Shuffle and Sorting phase will collect pairs with the same key and group them into pairs of *(key2, list of values)*, this phase is automatically done by Hadoop system. For each different key and a list of its values, the system will invoke a reduce function to process. Reduce functions will emit results that are pairs of *(key3, value3)*. Data, which are used during computation of a MapReduce job, are stored in a high performance distributed file system named HDFS. Map tasks and reduce tasks will be efficiently scheduled so that they are "nearest" to its input data. "Nearest" in the sense that tasks and its input data are in the same machine or in the same subset of network.

3.3.2 An intuitive implementation

Intuitively, our *iter_acc* algorithm can be performed by iterating over MapReduce jobs. In the first MapReduce job, each map task takes account into a partition and performs *one-pass* evaluation algorithm. There is no reduce task in the first job, so map task will directly write its result to HDFS. Next, we have a loop of MapReduce jobs for *iter_acc* algorithm. Each loop has one MapReduce job. The loop will finish when there is no new accessible node found. In each iteration, Map task will compute new accessible nodes following the algorithm described in Section 3.2, with each accessible node it emits a pair *(1, accessible_node)*. Because all pairs emitted by map phase has the same key 1, every accessible nodes will come to the same Reduce task where we will remove duplicate nodes and emit a list of new accessible nodes, this list will be used for the next MapReduce job. The final MapReduce job will compute partial answers in its Map tasks and send them to a reduce task to get the final answer.

However, we recognize that the time for initializing a MapReduce job is not small, which leads to an inefficient program as it iterates over many jobs. Besides, after each job finished, we have to write results (here, LAGs and accessible nodes) to HDFS and read them back for next jobs. Because the size of LAGs is large, the algorithm takes much time. The problem is that how to design an implementation with only one MapReduce job and avoiding as much as possible reading/writing so much data from/to HDFS.

3.3.3 An efficient implementation

Our idea is as follows. We keep LAGs and partial result in memory to maximize the performance of the algorithm, and only send out new accessible nodes. In Figure 8, we propose an implementation for that idea using HDFS. At each iteration, a map task writes its new accessible nodes to a file in HDFS, then it reads back all other files what have written by other map tasks, combines them together, and checks whether the whole list of accessible nodes is empty or not in order to decide if we should finish the loop or not. There are two problems we have to handle in that process:

- **Consistency**: A task can not read the content of a file in HDFS before other task completely finish writing it to HDFS.
- **Synchronization between iterations**: How to know that, at each iteration i-th, map tasks only read files of accessible nodes from the previous iteration $(i-1)$-th.

As of consistency, the problem is that how to know when HDFS finished writing to a file. HDFS uses the length of a file to do its magic [3]. The length of a file stays at 0 while the first block is being written to. After the first block is fully written, the length of file will be updated to the length of data written so far, this update continues until all blocks of data written to the file. The default value of a block is 64MB. Because in our model the size of files is not greater than the size of a file that contains all input and output nodes, therefore, to keep the length of a file always to be 0 we set up the default value to be the size of the file containing all input and output nodes. Now, we just check the length of a file to see whether other task has finished writing it or not.

To ensure the synchronization between iterations we name files by using two parameters: *iteration identity* and *partition identity*. The algorithm is as follows:

- Each partition has a unique identity p, $p = 1..n$, n the number of partitions.
- At each iteration i, each map task for a partition p does:
  - output a file containing new accessible nodes to HDFS, with the file name of the form of : *i_p.txt*
  - read back n files *i_p.txt* from HDFS, $p = 1..n$
  - check whether it completely finished reading all files *i_p.txt* or not, $p = 1..n$. If finished, increasing the value of i. If not, continuing to read.

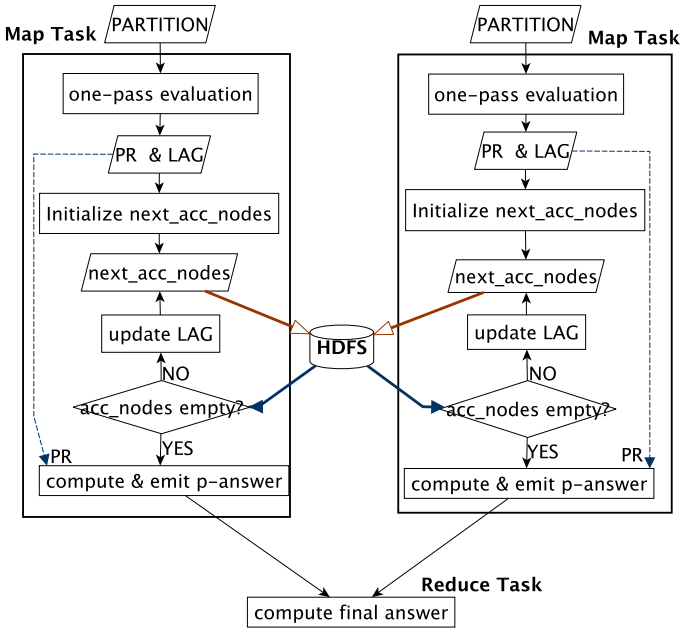

Figure 8: Hadoop-based implementation of evaluation system.

4. EXPERIMENTAL RESULTS

In this section, we present experiments for our implementation using datasets of YouTube and DBLP.

4.1 Experimental environment

Experiments are performed on Edubase Cloud System [1], we built a Hadoop environment from five virtual machines on the Cloud: one machine for master node, and four others for compute nodes. Each compute node has 8 CPUs and 24GB of RAM.

We used real-life data from YouTube [2] and DBLP [3] to generate edge-labeled directed graphs with the sizes as follows.

Dataset	|V|	|E|	Size (|V|+ |E|)
YouTube	7,354,581	8,297,699	15,652,280
DBLP	3,976,588	4,303,895	8,280,483

To make distributed graphs, we used GraphLab library [4] to partition the centralized graphs into 32 partitions stored by 32 different files and put on HDFS file system. When the algorithm is executed, each map task will read a file to process. Using GraphLab library is to ensure the balance between sizes of partitions and to minimize the number of cross-links.

For each graphs, we randomly chose a node to be the root. We used two different queries with following regular expressions.

[1]http://edubase.jp/cloud
[2]http://netsg.cs.sfu.ca/youtubedata/
[3]http://arnetminer.org/citation
[4]http://graphlab.org/

Dataset	Regular expression
YouTube	"$_* => category => Music$"
DBLP	"$_* => year => 2002$"

A query for YouTube graph is to find videos that have Music as its category, and the one for DBLP to find papers published in 2002.

4.2 Results

We compared the total size of all local accessible graphs generated by Dan Suciu's algorithm with the one of that generated by our *one-pass* algorithm. Figure 9 and Figure 10 shows that we can reduce almost 50% of the size of LAGs for both YouTube and DBLP graphs.

To simulate a bottleneck, we decreased the heap size for a reduce task to 1024MB. We compared between two programs: both use the one-pass algorithm to compute LAGs, in which one program sends all LAGs to one reduce task to compute accessible nodes (Dan Suciu's algorithm) and the other uses our iterative algorithm. Figure 11 shows that the program without the iterative algorithm can not pass when evaluating the dataset of the size of 8 millions, meanwhile the other can compute with the data being two times larger. Furthermore, we can see that when the data size becomes larger, the difference in running time is also increased. Experiments with DBLP (Figure 12) also show the same performance.

Finally, we do a comparison to see the overhead of a loop of MapReduce jobs. As shown in the Figures 11 and 12, the algorithm using a loop of MapReduce jobs (MRLoop) is about 3 or 4 times slower than our iterative algorithm which used the in-memory technique. This overhead is proportional to the number of iterations in a loop.

5. RELATED WORKS

Evaluating regular path queries on distributed, rooted, edge-labeled directed graphs are studied by Dan Suciu in [21], and extended in [18] based on message passing. In [18], the algorithm will create a set of processes, each process starts by creating an initial task for itself and a table to store results during computation. Tasks then communicate with others to update their tables in an iterative way. This is different to our approach: (1) We only use the iterative way to compute accessible nodes, most computations of the algorithm are done locally; (2) The amount of data sending over the network in [18] is $O(n^2)$, where n is the number of cross-links between different sites.

In [9], Fan Wenfei proposed an algorithm for reachability problem with regular path expressions. The algorithm used Boolean formulas to keep local accessible nodes, and then combine them together to make a dependent graph. This algorithm only needs one visit for each site and therefore fits to the MapReduce model. Basically, the approach in [9] still follows the query evaluation model in [21], therefore the amount of data transferred is quadratic to the number of cross-links. Furthermore, our algorithm can evaluate more general queries of regular expression (queries that return data extracting from the input graph) other than reachability queries with True/False answer.

In 2010, Google developed Pregel [14], a distributed programming framework, focused on providing users with a natural API for programming graph algorithms while managing the details of distribution invisibly, including messaging

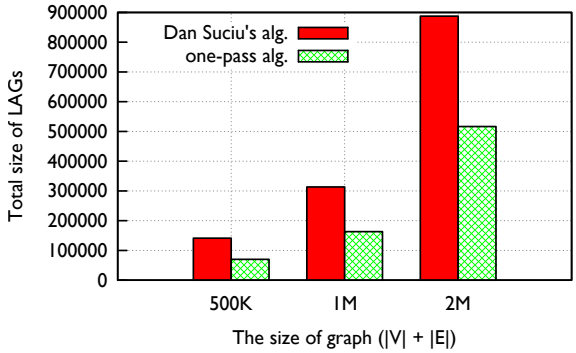

Figure 9: Reduction of redundant data (YouTube dataset).

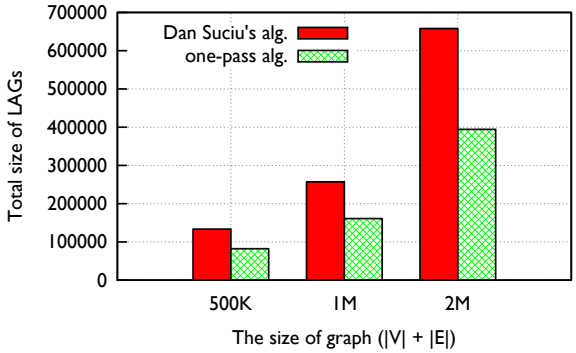

Figure 10: Reduction of redundant data (DBLP dataset).

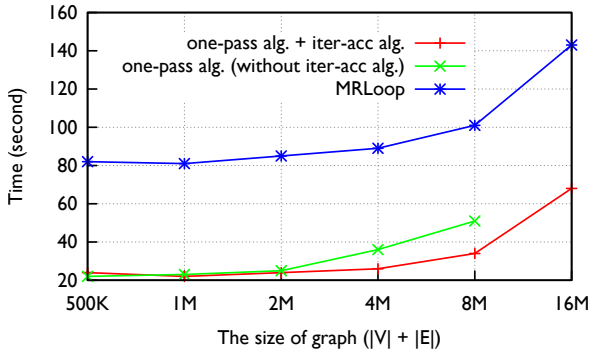

Figure 11: Running time with YouTube dataset.

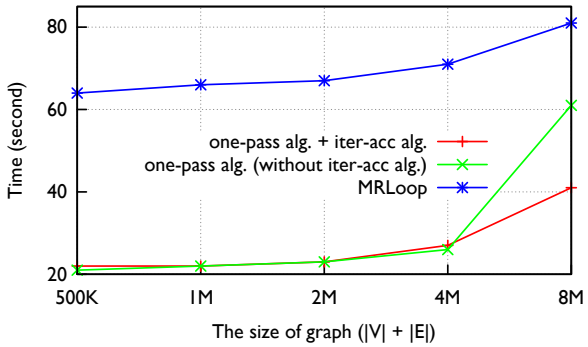

Figure 12: Running time with DBLP dataset.

and fault tolerance. Pregel was inspired by the Bulk Synchronous Parallel model [24], which provides its synchronous superstep model of computation and communication. Pregel concepts were cloned by several open source projects such as: Apache Hama [16], Giraph [1], Signal/Collect [20]. This model is different with ours, in which our model is mainly based on partial evaluation. We believe that Pregel can be used to compute accessible nodes in the *iter_acc* algorithm.

Ligra [19] is a lightweight interface for graph algorithms that is particularly well suited for graph traversal problems. This work is motivated by developing a very fast BFS for shared memory machines. Ligra consists of two simple interfaces: (1) EDGEMAP is to apply a function to all edges with source vertex in a subset of vertex and target vertex satisfying some condition, and (2) VERTEXMAP is to apply a function to every vertex in a subset of vertex. Nevertheless, the idea of applying Ligra to query evaluation was not mentioned in [19].

Pig [11] and Hive [22] are two popular high-level dataflow systems on top of MapReduce to analyze enormous datasets in spirit of SQL. Programs are compiled into sequences of Map-Reduce jobs and executed in Hadoop environment. However, they were not designed mainly to support scalable processing of graph-structured data.

6. CONCLUSION

We have proposed an improvement for query evaluation on distributed graphs, which reduces a large amount of redundant data and avoids the bottleneck during the evaluation. The total amount of data transferred over network of our algorithm is linear to the number of cross-links between different sites. We have also proposed an efficient implementation based on Hadoop file system HDFS and used HDFS to ensure the consistency of data and to synchronize iterations in the algorithm. With our knowledge, we see that our algorithm is the first approach trying to combine the partial evaluation with iteration approach to evaluate queries on distributed graphs.

In the future, we will apply our approach to deal with more other queries of UnQL query language, and to evaluate queries on multiple sources of data. Another direction is to extend our approach to incrementally maintain views to database.

7. REFERENCES

[1] Apache giraph. `http://giraph.apache.org//`. 2013.

[2] Apache hadoop. `http://hadoop.apache.org/`. 2013.

[3] D. Borthakur. Hdfs architecture guide. `http://hadoop.apache.org/docs/stable/hdfs_design.html`. Online, accessed: 2013-07-10.

[4] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 309–320, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[5] P. Buneman. Semistructured data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '97, pages 117–121, New York, NY, USA, 1997. ACM.

[6] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 505–516, New York, NY, USA, 1996. ACM.

[7] P. Buneman, M. Fernandez, and D. Suciu. Unql: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, Mar. 2000.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *Proc. VLDB Endow.*, 5(11):1304–1316, July 2012.

[10] Y. Futamura. Partial evaluation of computation process an approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, Dec. 1999.

[11] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, Aug. 2009.

[12] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[13] K. Lang. Finding good nearly balanced cuts in power law graphs. Technical Report YRL-2004-036, Yahoo! Research Labs, November 2004.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[15] S. Mark. From prefix computation on pram for finding euler tours. `www.cs.hut.fi`.

[16] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 721–726, Washington, DC, USA, 2010. IEEE Computer Society.

[17] A. Serge, B. Peter, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, California, 2000.

[18] M. Shoaran and A. Thomo. Fault-tolerant computation of distributed regular path queries. *Theor. Comput. Sci.*, 410(1):62–77, Jan. 2009.

[19] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[20] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: graph algorithms for the (semantic) web. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 764–780, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] D. Suciu. Distributed query evaluation on semistructured data. *ACM Trans. Database Syst.*, 27(1):1–62, Mar. 2002.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[23] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv: ...*, pages 1–17, 2011.

[24] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.