Efficient Query Evaluation on Distributed Graph with Hadoop Environment

Le-Duc Tung, Quyet Nguyen-Van, Zhenjiang Hu

Graph has emerged as a powerful data structure to describe the variety of 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 which is a rooted, edge-labeled graph, the algorithms are proved to be efficient in term of communication steps and data transferring during the evaluation. However, the disadvantage is that communication data are collected to one single site, this 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 algorithm is to significantly reduce a large amount of redundant data in the evaluation, named *one-step evaluation*, and the other is to resolve the bottleneck with the amount of data transferring via network being linear. We also design an efficient implementation with only one MapReduce job for our algorithms in Hadoop environment by utilizing highlight features of Hadoop file system. Experiments on cloud system show that our algorithms can detect and remove 50% of data being redundant in the evaluation process on YouTube and DBLP datasets. Besides, experimental results also show that our algorithms is running without the bottleneck even when we double the size of input data.

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 [3]. Social network graph consists of edges denoting many complex relationships [7] [13]. Despite of the popular of graph, there are still not so many research studies focusing on query evaluation on distributed graph whose nodes and edges are distributed on different sites.

To see the difficulties of query evaluation on distributed graph, consider simple graph queries defined in terms of regular expressions [6]. The process of evaluating a query is to find paths in the graph that satisfies the regular expression, and return nodes and edges relating to that paths. Therefore, 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. The reason is that on each site we just have the local information of the graph, hence sites have to communicate and exchange a lot of information with others.

Dan Suciu [12] has proposed an efficient algorithm to evaluate query on distributed graph with distinguished properties as follows:

- The number of communication steps is four (independent on the data or query).

Le-Duc Tung, 総合研究大学院大学（総研大), Dept. of Informatics, The Graduate University for Advanced Studies (SOKENDAI).

Quyet Nguyen-Van, Hung Yen University of Technology and Education, Vietnam.

Zhenjiang Hu, 国立情報学研究所, Programming Research Laboratory, Information Systems Architecture Research Division, National Institute of Informatics (NII).

- 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. This results in a bottleneck in the algorithm when evaluating on social graphs where the number of cross-links is quickly increasing [8]. In this paper, we show an approach to overcoming this bottleneck with following contributions:

- We improve Dan Suciu's algorithm so that we can remove a large number of redundant nodes and edges in the evaluation (Session 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 state in automaton, $|P|$ the number of partitions (Session 3. 2).
- We propose an efficient implementation for our algorithms using MapReduce programming model in the Hadoop environment. The implementation utilizes the features of Hadoop distributed file system (HDFS) to make program correct and efficient. Details will be mentioned in Section 3. 3.

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

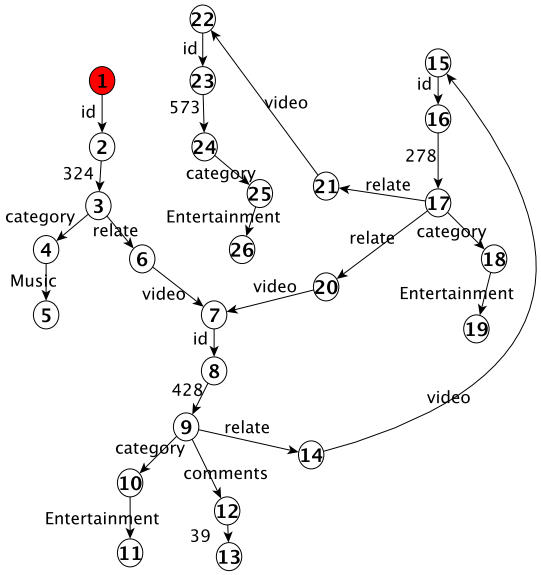

Figure 1 A graph of Youtube's data. The red node is the root node.

2 Query evaluation on distributed graph

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 $u \overset{label}{\rightarrow} v$. Label could be atomic values such as: Int, Long, String, Bool, Image, ...

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 distributed graph. An edge $u \rightarrow v$ is a cross-link if u and v are stored on different sites.

For any cross-link $u \overset{\alpha}{\rightarrow} v$ from site α to site β, we replace it with $u \overset{\alpha}{\rightarrow} v' \overset{\epsilon}{\rightarrow} v$, where v' is just a copy of v and resides on the site α, ϵ is a special label denoting an empty label. Now, $v' \overset{\epsilon}{\rightarrow} v$ becomes a

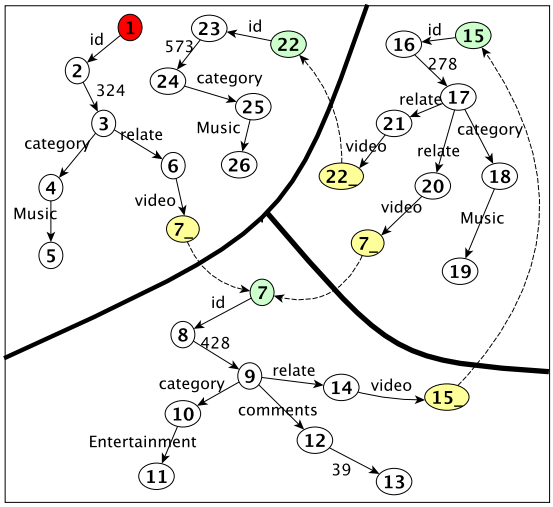

Figure 2 A distributed graph for Youtube's data. The red node is the root. Green nodes are output ones. Yellow nodes are input ones.

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 [4]. 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 include 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 [10] [12] 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, we compute a partial result P_α as in Algorithm 1. 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, 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.

Algorithm 1: Partial result computation

input : A fragment DG_α, an automaton $\mathcal{A}$

output: A partial result P_α

```
1  begin
2    P_α ← DG_α ;        /* copy nodes and
     edges from DG_α to P_α */
3    foreach u ∈ InputNodes(DG_α) do
4      foreach s ∈ States(A) do
5        S ← visit(s, u);
6        foreach p ∈ S do
7          adding an edge (s, u) --ε--> p to
           P_α;
8        end
9      end
10   end
11 end
```

Algorithm 2: Visiting algorithm

input : A node s, a state s

output: A set of node

```
1  visit(s, u) begin
2      if (s, u) ∈ visited then
3          return global_matches[(s,u)];
4      end
5      visited ← (s,u);
6      matches ← {};
7      if u is output node then
8          matches ← (s,u);
9          global_matches[(s,u)] ← matches;
10     else if s is terminal state then
11         matches ← u;
12         global_matches[(s,u)] ← matches;
13
14     foreach u -a-> v in DG_α do
15         foreach s -x-> s' satisfies x == a do
16             matches ← matches ∪
               visit(s', u);
17             global_matches[(s,u)] ←
               matches;
18         end
19     end
20     return matches;
21 end
```

Step 3: Using P_α, we compute a local accessible graph LAG_α. LAG_α contains all input and output nodes from partial result and an edge from an input node to an output node if there exists a path between them in the partial result.

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

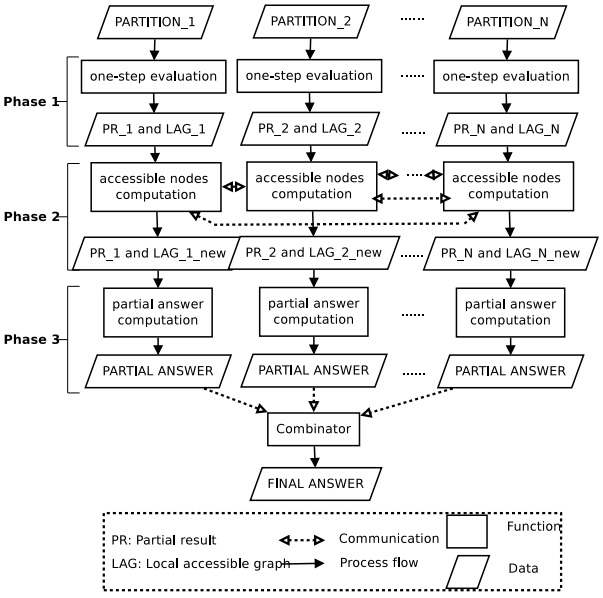

Figure 3 System overview

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

Step 6: Client will combine partial answers together 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. 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 $O((|N| \times |S|)^2) + O(r))$ [12] [6].

3 Efficient evaluation with Hadoop environment

In this section we focus on describing our framework that efficiently evaluate queries on distributed graph.

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 Session 3. 2. After that, we get new LAGs at each site. These new LAGs contain oaccessible nodes from the root node (s_{inital}, 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 DFS for each accessble nodes over 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-step evaluation

3. 1. 1 Observations

Before going to the detail of our one-step 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 the 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 matches the automaton starting from node 1 or node 2 with state $s2$, therefore there is no edge emanating from these nodes.

A local accessible graph contains all input and output nodes from a partial result and an ϵ-edge from an input node to an output node if there exists a path between them. Figure 5 is an example of a local accessible graph for the parital result in figure 4.

We have some important observations on LAGs as follows:

Observation 1: *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 $(s, u) \rightarrow u$ (Alogrithm 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 sastifies the automaton. Besides, we can not go further along the 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 $u_{in} \xrightarrow{\epsilon} v_{out}$ is 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 $w \to (s_{terminal}, u')$, 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 $w \to (s_{terminal}, u')$, then there is no path coming to node $(s_{terminal}, u)$, we can remove the edge $(s_{terminal}, u) \to u$.

3.1.2 One-step 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 nodes having states (these nodes and edges are computed from visiting algorithm); and the remaining part consists of edges from an input node having state to other non-state output node, 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 result and LAGs without non-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, this 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 $(s, u) \xrightarrow{\epsilon} v_{out}$.

Algorithm 3: One step algorithm

input : A fragment DG_α, an automaton $\mathcal{A}$

output: A partial result P_α, a LAG_α

```
1  begin
2      P_α ← DG_α ;        /* copy nodes and
       edges from DG_α to P_α */
3      foreach u ∈ InputNodes(DG_α) do
4          IO ← FindOutNodes(u);
5          foreach v ∈ IO do
6              LAG_α ← (u --ε--> v);
7          end
8          foreach s ∈ States(A\s_terminal) do
9              S ← visit(s, u);
10             foreach p ∈ S do
11                 if u has a state then
12                     adding an edge (s, u) --ε--> p
                       to P_α;
13                     adding an edge (s, u) --ε--> p
                       to LAG_α;
14                 else
15                     IO ←
                       FindOutNodes(u);
16                     foreach v ∈ IO do
17                         LAG_α ← (u --ε--> v);
18                     end
19
20             end
21         end
22     end
23 end
```

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 then the

Algorithm 4: A modified visiting algorithm

```
   input  : A node s, a state s
   output: A set of node
1  visit(s, u) begin
2  |  ⋮
3  |  if s is terminal state then
4  |  |  matches ← u;
5  |  |  global_matches[(s,u)] ← matches;
6  |  else if u is output node then
7  |  |  matches ← (s,u);
8  |  |  global_matches[(s,u)] ← matches;
9  |
10 |  ⋮
11 end
```

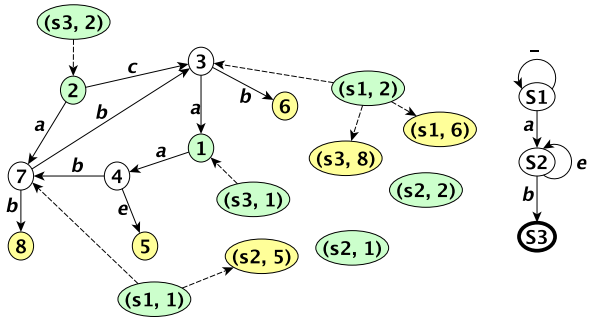

Figure 4 Partial result

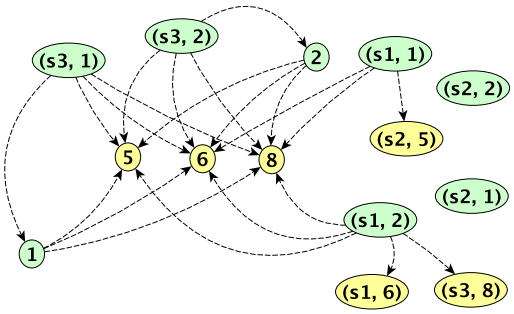

Figure 5 Local accessible graph (LAG)

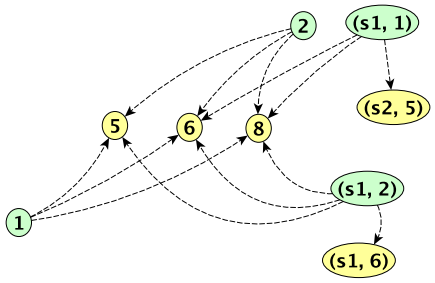

Figure 6 LAG after reducing redundant vertices

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 final answer because we use accessible nodes to extract final answer from partial results. Therefore sending out all nodes and edges in LAGs is not necessary.

Borrowing the idea of distributed BFS [9], we propose an efficient iterative algorithm to compute accessible nodes, named *iter_acc* algorithm. In the computation process, each node in *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 LAGs by adjacent lists, in which for each input node we have a set of output node linked to it. Let a list of adjacent nodes of u be $adj(u)$. Marking all nodes in LAGs as an OR node.

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

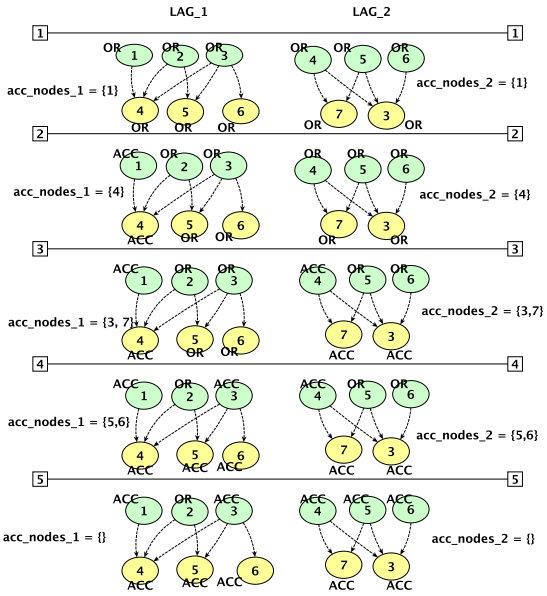

Figure 7 An example of computation of accessible nodes with five iterations. *acc_nodes* **contains a list of accessible nodes before each iteration.**

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

- If $u \in LAG_\alpha$, then (1) if u is an OR node, then update its 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, then add v to acc_nodes_α.
- If $u \notin LAG_\alpha$, do nothing.
- 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.

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 a hash data structure to store input nodes of 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 [1] is an open source framework for MapReduce programming model [5]. It is proved to be efficient and scalable. In MapReduce programming model, we only need to write two functions: Map and Reduce. Map functions produce a list of pair of (key, value). After that, Shuffle and Sorting phase will collect pairs with the same key and group them into pairs of (key, list of values), this phase is automatically done by Hadoop system. For each different key and its list of values, the system will invoke a reduce function to process. Reduce functions will emit results that are pairs of (key', value'). 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* algorithms can be performed by iterating over MapReduce jobs. In the first MapReduce job, each map task takes account into a partition and performs one-step 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 finish when there is no new accessible nodes 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, this 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 that: we keep LAGs and partial result in memory, 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 iteration**: How to know that, at each iteration i-th, map tasks only read files of accessible node 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 [2]. 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 is always 0 we set up the default value is the size of a file of 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 iteration 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 partition.
- At each iteration i, each map task for partition p does:
  - output a file containing new accessible nodes to HDFS, with the file name of 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.

4 Experimental results

In this session, we present experiments for our implementation using datasets of YouTube and

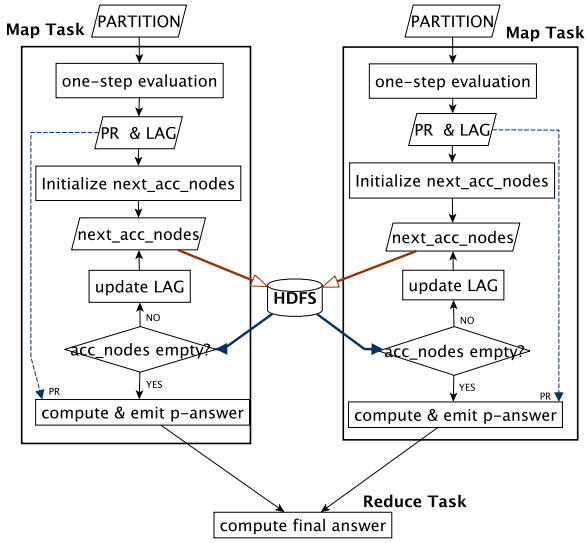

Figure 8 Hadoop-based implementation of evaluation system

DBLP.

4.1 Experimental environment

Our 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 these graphs into 32 partitions stored by 32 different files and put them on HDFS file system. When the algorithm is executed, each map task will read a file to process.

For each graphs, we randomly chose the node with id "3307" as the root node. We used two different queries with following regular expression for each dataset.

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 fine papers published in 2002.

4.2 Results

We compared the total size of all local accessible graphs generated by Dan Suciu's algorithm with that generated by our one-step 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-step algorithm to compute LAGs, however one program sends all LAGs to one reduce task to compute accessible nodes 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 milions, meanwhile the other can compute with the data that is two times larger. Besides, 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.

5 Related works

Evaluating regular path queries on rooted, edge-labeled directed graphs are studied by Dan Suciu in [12], and extended in [11] based on message passing. In [11], the algorithm will create a set of pro-

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

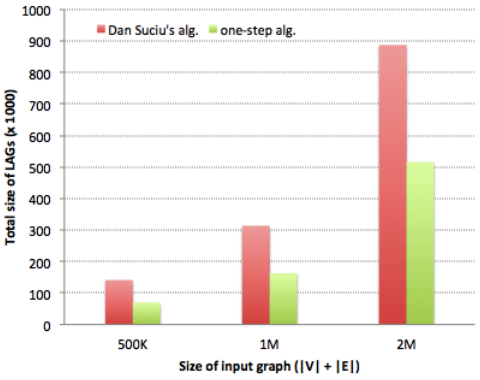

Figure 9 Reduction of redundant data (YouTube dataset).

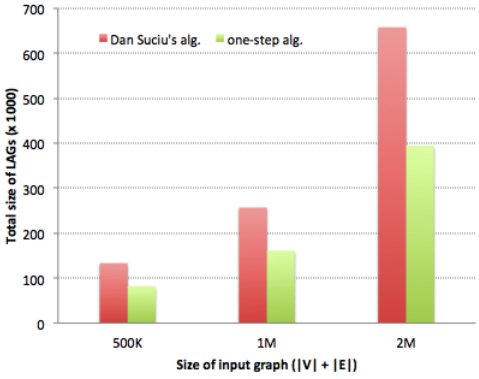

Figure 10 Reduction of redundant data (DBLP dataset).

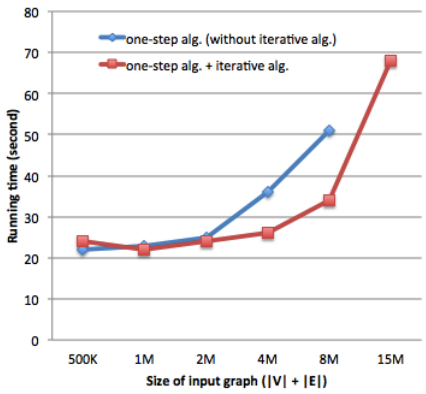

Figure 11 Running time with YouTube dataset.

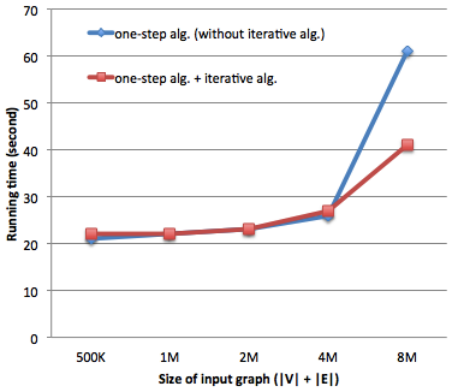

Figure 12 Running time with DBLP dataset.

cesses, 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, almost computations of the algorithm is done locally; (2) The amount of data sending over the network in [11] is $O(n^2)$, where n is the number of cross-links between different sites.

In [6], Fan Wenfei proposed an algorithm for reachability problem with regular path expressions. He 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 sites and therefore fits to the MapReduce model. Basically, the approach in [6] still follows the query evaluation model in [12], therefore the amount of data transferred is quadratic to the number of cross-links. Besides, 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.

6 Conclusion

We have proposed an improvement for query evaluation on distributed graph, 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 between iterations in the algorithm.

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.

References

[1] : Apache Hadoop, `http://hadoop.apache.org/`. 2013.

[2] Borthakur, D.: HDFS Architecture Guide, `http://hadoop.apache.org/docs/stable/hdfs_design.html`. Online, accessed: 2013-07-10.

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

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

[5] Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters, *Commun. ACM*, Vol. 51, No. 1(2008), pp. 107–113.

[6] Fan, W., Wang, X., and Wu, Y.: Performance guarantees for distributed reachability queries, *Proc. VLDB Endow.*, Vol. 5, No. 11(2012), pp. 1304–1316.

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

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

[9] Mark, S.: From prefix computation on PRAM for finding euler tours, `www.cs.hut.fi`.

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

[11] Shoaran, M. and Thomo, A.: Fault-tolerant computation of distributed regular path queries, *Theoretical Computer Science*, Vol. 410, No. 1(2009), pp. 62–77.

[12] Suciu, D.: Distributed query evaluation on semistructured data, *ACM Trans. Database Syst.*, Vol. 27, No. 1(2002), pp. 1–62.

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