Minimizing Data Transfers for Regular Reachability Queries on Distributed Graphs

Quyet Nguyen-Van
Hung Yen University of Technology and Education
Khoai Chau, Hung Yen
quyetict@utehy.edu.vn

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

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

ABSTRACT

Nowadays, there is an explosion of Internet information, which is normally distributed on different sites. Hence, efficient finding information becomes difficult. Efficient query evaluation on distributed graphs is an important research topic since it can be used in real applications such as: social network analysis, web mining, ontology matching, etc. A widely-used query on distributed graphs is the regular reachability query (RRQ). A RRQ verifies whether a node can reach another node by a path satisfying a regular expression. Traditionally RRQs are evaluated by distributed depth-first search or distributed breadth-first search methods. However, these methods are restricted by the total network traffic and the response time on large graphs. Recently, Wenfei Fan et al. proposed an approach for improving reachability queries by visiting each site only once, but it has a communication bottleneck problem when assembling all distributed partial query results.

In this paper, we propose two algorithms in order to improve Wenfei Fan's algorithm for RRQs. The first algorithm filters and removes redundant nodes/edges on each local site, in parallel. The second algorithm limits the data transfers by local contraction of the partial result. We extensively evaluated our algorithms on MapReduce using YouTube and DBLP datasets. The experimental results show that our method reduces unnecessary data transfers at most 60%, this solves the communication bottleneck problem.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Performance

Keywords

Reachability Queries, Distributed Graphs, MapReduce

1. INTRODUCTION

Graph data processing has been becoming increasingly important. There are many applications of reachability queries, such as friends recommendations in social networks [14], detecting signal pathways in protein interaction networks [17] and querying XML documents [10]. However, the data may be stored in different locations for some applications such as: web mining, social network analysis, etc. It is complicated to effectively exploit huge information in this distributed form. In this paper, we focus on optimizations for the problem of answering reachability queries with a regular expression on distributed graphs. We show how to minimize the data transfers needed for checking whether there exists a path between two vertices in a distributed graph such that it satisfies a regular expression.

A number of algorithms have been proposed to process graph reachability queries [6, 12, 11, 4, 5]. However, these methods only process general graphs and require a large of memory. Moreover, none of them support distributed processing. Although graph reachability is a popular problem, there are still not so many research studies focusing on query processing on distributed graphs. Wenfei Fan et al. [9] propose several algorithms based on partial evaluation that support three classes of reachability queries. Therein, a regular expression can be answered with time complexity $O(|F_m||R|^2 + |R|^2|V_f|^2)$ and a total network traffic of $O(|R|^2|V_f|^2)$, where $|F_m|$ is the size of the largest fragment in the distributed graph G, |R| is the size of regular expression R, $|V_f|$ is the number of nodes that have edges across different subgraph in G. In fact, databases could store extremely large graphs (i.e. hundreds of millions nodes and edges on Twitter social network [23], billions nodes/edges on Friendster social network [24]). Therefore, the number of cross-edges could be very large leading to a bottleneck when processing queries on distributed graphs. In this paper, we target on solving this problem by minimizing the data transfers via network.

The basic idea of our approach is to filter redundant data and locally contract partial results. We first find all nodes in a sub dependency graph of a local site not connected through an edge from any nodes in other sub dependency graphs or the nodes that can not reach any nodes. This can reduce unnecessary data transfers via network to the coordinator site. In Section 3.3, we show that the amount of redundant data is quite large. Moreover, we can also reduce total network traffic by contracting the size of the sub dependency graphs. The techniques are combining together for efficiently answering RRQs on distributed graphs.

Our work as described in this paper makes the following contributions.

- We propose an efficient algorithm to filter redundant data for local evaluation, in parallel (Section 4.1). It removes large amount of unnecessary data when assembling all distributed partial query results. Additionally, our method is much simple to apply.
- We present a *local contract* algorithm (Section 4.2), to reduce the size of partial result on each local site. It makes partial result equivalent to what is generated by Wenfei Fan's algorithm for local evaluation.
- We demonstrate that our method delivers excellent performance, with the amount of data shipped to coordinator site with real-life graphs. Indeed, we compare our implementation with Wenfei Fan's re-implementation [9] and show that, for real-life datasets YouTube and DBLP, our method can remove up to 60% of data being redundant in the dependency graph.

The rest of this paper is organized as follows. In Section 2, we introduce the distributed graphs and regular reachability queries to answer RRQs. In Section 3, we analyze three steps of the algorithm for answering RRQs based on partial evaluation in [9] and expose several discussions related to the redundant data. We describe the improvements of our two algorithms in Section 4. In Section 5, we give an implementation model based on MapReduce framework and experimental results using real-life graphs. Section 6 shows an overview on related work and conclude in Section 7.

2. DISTRIBUTED GRAPHS AND REGULAR REACHCHABILITY QUERIES

This section shows terms and definitions on distributed graphs and regular reachability queries.

2.1 Distributed Graphs

We consider the problem of efficient answering reachability queries on node-labeled, directed distributed graph. A graph G is a tuple $G = (V, E, L)$, where V is a finite set of nodes; E is a finite set of edges; L is a function which defines on V such that for each node v in V, $L(v)$ is a label in a set of labels Σ.

In particular, the graph G is often partitioned into k different sites, where each site is a subgraph. The distributed graph G is defined by a set of subgraphs including $G_1, G_2, ..., G_k$ and a *cross-graph* G_c. Here, a subgraph G_i is denoted by (V_i, E_i, L_i), where $V_i \subseteq V$; $E_i \subseteq E$; $\forall v \in V_i \rightarrow L_i(v) = L(v)$. The cross-graph $G_c = (V_c, E_c)$, where E_c is a set of edges that connect subgraphs, called cross-edges, V_c is a set of nodes that have cross-edges to or from subgraphs.
(1) $V_c = \cup_{i=1}^{k}(V_i.in \cup V_i.out)$, where (a) $V_i.in$ is a set of input nodes of G_i with each node $v \in V_i.in$ existing at least one edge from node u in other subgraphs and $V_i.in \subseteq V_i$, (b) $V_i.out$ is a set of output nodes of G_i with each node $v' \in V_i.out$ in other subgraphs existing at least one edge from a node v in G_i connecting to v'.
(2) $E_c = \cup_{i=1}^{k} cE_i$, where cE_i is a set of all cross-edges of the subgraph G_i, its edges *(v, u)* are determined by the node v in G_i and the node u in another subgraph. We can see the relationship of graph G, the set of subgraphs and cross-graph G_c as following: $V = \cup_{i=1}^{k} V_i$ and $V_i \cap V_j = \emptyset$ if i $\neq$ j; $E = E_c \cup (\cup_{i=1}^{k} E_i)$ and $E_i \cap E_j = \emptyset$ if i $\neq$ j. In fact, data at each site include a subgraph G_i, a set of output nodes $V_i.out$ of G_i and a set of cross-edges cE_i of G_i. Overall data in a single site is a *fragment* of graph G that is denoted by $F_i = (V_i \cup V_i.out, E_i \cup cE_i, L_i)$. Therefore, query processing on each site means we are evaluating on a fragment of graph G.

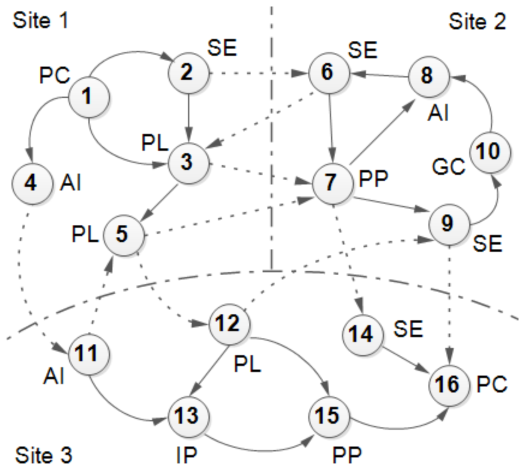


Figure 1: A distributed graph as researchers network

Table 1: Determining input and output nodes G

F_i	$V_i.in$	$V_i.out$
F_1	{3, 5}	{6, 7, 11, 12}
F_2	{6, 7, 9}	{3, 14, 16}
F_3	{11, 12, 14, 16}	{5, 9}

Example 1: Figure 1 illustrates a graph G of researchers network, where each node denotes a researcher with identity number (e.g., 1, 2, 3) and one research interest (e.g., **P**arallel **C**omputing (PC), **P**rogramming **L**anguages (PL), **P**arallel **P**rogramming (PP), **S**oftware **E**ngineering (SE)), each directed edge $u \rightarrow v$ denotes (u, v), it means that researcher u can contact to researcher v. In this figure, the graph G is partitioned into three subgraphs G_1, G_2, G_3 and stored on three sites S_1, S_2, S_3, respectively. Each subgraph stores information of researchers from three different countries. Therein, a lot of researchers in a country may contact one or many researchers in another country. The graph G is called a distributed graph. Here, we indicate the input and output nodes on each fragment F_1, F_2, F_3 as in Table 1. Besides, a cross-graph G_c of the graph G is defined by a set of nodes $V_c = \{2, 3, 4, 5, 6, 7, 9, 11, 12, 14, 16\}$ and a set of cross-edges E_c that consists of all edges connect among other fragments (e.g., (2, 6), (3, 7)...), here, a cross-edge is drawn with a dashed arrow.

2.2 Regular Reachability Queries

In reality, a regular expression is often used for evaluating graph queries. A regular reachability query is a 3-tuple denoted by $q_r(s, t, R)$, where s is a start node, t is a terminal node and R is a regular expression. RRQs check whether there exists a path ρ from node s to node t in G such that it satisfies a regular expression R, which is denoted $s \xrightarrow{\rho} t$. The result of this kind of query is *True* if there exists at least one path ρ from start node s to terminal node t in the graph G, where ρ is a value that satisfies R, otherwise returns *False*. Here, R is a regular expression over Σ:

$$\mathrm{R} = \epsilon \mid \mathrm{a} \mid \mathrm{RR} \mid \mathrm{R} \cup \mathrm{R} \mid \mathrm{R}^*,$$

where ϵ is an empty value, a is a label in Σ and RR, R $\cup$ R and R* denote alternation, concatenation and the Kleene closure, respectively.

3. DISTRIBUTED REGULAR REACHABILITY QUERIES

In this section, we describe an approach to answering RRQs on distributed graphs that was first shown in [9].

3.1 Partial Evaluation

An introduction to partial evaluation is given in [13]. This technique shows several different types of program optimization by specialization. The major motivation for doing partial evaluation is to increase the speed of processing. Processing program p is divided into k parts $(p_1, p_2, ..., p_k)$ that execute individually and guarantee to behave in the same way. The result of p_i is a subset of p's result. Normally, a program p_i runs faster than p. By employing these advantages of partial evaluation technique, [3, 8, 9] have provided efficient algorithms for query evaluation on distributed graphs.

3.2 Query Automaton

A regular expression can be converted into an automaton before using it to match paths. We use a non-deterministic automata (NFA) to represent query where the definition NFA as in [2]. In [2], an automaton M is converted in linear time from a regular expression R. A query automaton q_r is defined by a start node s, a terminal node t and an automaton M. We denote $q_r = (s, t, M)$, where M is a 5-tuple, as follows:

$$M = \{Q, T, \mu, q_s, q_t\},$$

where Q is a finite set of states, T is a set of transitions between two states, μ is a function that assigns each state a label in R, q_s is an initial (or start) state and $q_s \in Q$, q_t is a terminal state and $q_t \in Q$.

Example 2: Suppose we have a graph G as described in Example 1. An actual situation as the following: a researcher has identity number 1 (id = 1, is called *researcher 1*). Currently, *researcher 1* has a science project and wants to collaborate with *researcher 16* in another country. Therefore, *researcher 1* needs to find whether there exists a communication with *researcher 16* through a few other researchers in other fields. A regular expression that describes the requirement of *researcher 1* is the following: R = ((PL)* PP) $\cup$ (SE)*). It means that *researcher 1* wants to contact *researcher 16* through a chain of researchers in the **P**rogramming **L**anguages (PL) field then be via a researcher in the **P**arallel **P**rogramming (PP) field. *Researcher 1* also accepts the contacts through the list of researchers in the field of **S**oftware **E**ngineering (SE). Here, the query automaton q_r = (1, 16, ((PL)* PP)) $\cup$ (SE)*) is illustrated in Figure 2. This query will be answered in the examples later.

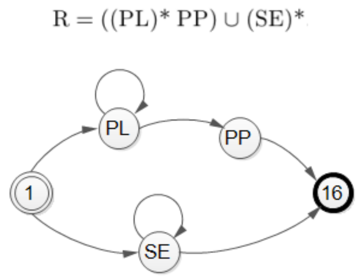


Figure 2: The illustration a regular expression is represented by automaton

3.3 Answering RRQs based on Partial Evaluation

Answering RRQs on the graph G is equivalent to finding of existence paths accepted by the automaton M. Here, P(R) denotes the path, which satisfies R in the graph G. Let $P(R) = \{v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_n\}$, where $v_i \in V$ (i = 1, 2,..., n). The automaton M accepts P(R) if a sequence of states $q_1, q_2, ..., q_n$ exists in Q with the following conditions: $v_1 = s, v_n = t, L(v_i) = \mu(q_i)$ for (i = 1, ..., n), where s is a start node and t is a terminal node in the query q_r . We say that node v_i is a match of a state q_i in M.

Algorithm 1 Using procedure ***localEval*** to get partial result

Input: A fragment F_i and query automaton $q_r(s, t, M)$
Output: Partial answer $P_i(R)$ on fragment F_i

```
P_i ← ∅;
if s ∈ V_i then
    V_i.in ← V_i.in ∪ {s};
end if
if t ∈ V_i.out then
    V_i.out ← V_i.out ∪ {t};
end if
for each node v ∈ V_i do
    v.visit ← false;
end for
for each node v ∈ V_i.out do
    v.rvec ← ∅;
    for each state q ∈ Q do
        if v = t and q = q_t then
            v.rvec[q] ← true;
        else
            if L(v) = μ(q) then
                v.rvec[q] ← X_(v,q);
            else
                v.rvec[q] ← false;
            end if
        end if
    end for
    v.visit ← true;
end for
for each node v ∈ V_i.in do
    v.rvec ← getRvec(v, F_i, q_r);
    P_i ← P_i ∪ v.rvec;
end for
Send P_i to the coordinator site S_c
```

Algorithm 2 Procedure ***getRvec*** computes the vector v.rvec for a node v

Input: A node v, a fragment F_i, a query automaton $q_r(s, t, M)$
Output: The vector v.rvec

```
if v.visit = true then
    return v.rvec;
end if
for each state q ∈ Q do
    v.rvec[q] ← false;
end for
for each node u ∈ ChildrenNodes(v, F_i.V) do
    if u.visit = false then
        u.rvec ← getRvec(u, F_i, q_r);
    end if
    for each state q ∈ Q do
        if L(v) = μ(q) then
            v.rvec[q] ← v.rvec[q] ∪ getVec(u, u.rvec, q, q_r);
        end if
    end for
    v.visit ← true;
end for
return v.rvec;
```

The approach to answering RRQs based on partial evaluation and using query automaton presented in [9] consists of 3 steps, as follows:

Step 1: Construct the query automaton M from regular expression R at coordinator site, and then send it to other sites.

Step 2: After receiving the query from S_c, each site S_i performs local evaluation to get partial result P_i, in parallel. Each partial result indicates that the nodes might be related to the final answer of query q_r on graph G. Those relationships are among input and output nodes. In P_i, the input node v can reach output node u, but it does not know if u can reach terminal node t or not. Therefore, a *Boolean variable* is used to associate with output nodes. Now, P_i can be represented as a set of vectors of *Boolean Formulas* associated with nodes in $V_i.in$. This step is illustrated by Algorithm 1 and Algorithm 2.

Step 3: Combine the partial results into a dependency graph G_d on coordinator site S_c. Then a breadth-first search (BFS) algorithm is used to check whether start node s can reach terminal node t on G_d.

Next we look into the details of the two core algorithms in this approach, which would help to understand its problems and our later improvement.

Algorithm 1 computes a partial result to query q_r. Firstly, (1) initializes P_i is an empty set of vectors, where $v.\text{rvec} \in P_i$ to be a vector of O(|Q|) entries, where Q is the set of states in M, the entry v.rvec[q] is a *Boolean Formula* that indicates whether node v matches state q in Q; gets and sets the input nodes in $V_i.in$ and the output nodes in $V_i.out$. (2) It then computes the vector v.rvec for each output node v in $V_i.out$, as follows. If node v is a terminal node then *True* value is set to v.rvec[q_t]. Otherwise, if v matches state $q \in$ Q in M ($L(v) = \mu(q)$) then it sets a *Boolean variable* $X_{(v,q)}$ to v.rvec[q], if not, value v.rvec[q] is set to *false*. Finally, (3) it computes the vector for each input node $v \in V_i.in$ by calling procedure *getRvec*.

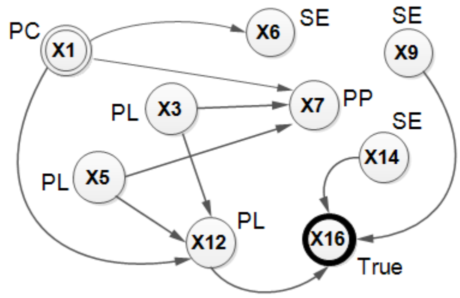


Figure 3: Dependency graph G_d

Algorithm 2 computes and returns a vector of *Boolean Formulas* for each input node in F_i. It is a recursive procedure, the vector *rvec* of each input node is computed via its children nodes. This procedure calls a simple procedure *getVec*. Here, *getVec* gets the value of *rvec* of the input node u at state q, which was computed before.

Example 3: In this example, we use the query automaton described in Example 2. According to the algorithms above, to answer query q_r = (1, 16, ((PL)* PP) ∪ (SE)*), we need to perform a local evaluation on three sites. We then collect the partial results P_1, P_2, P_3 to construct a dependency graph and get the final answer. Here, we only illustrate the local evaluation on fragment F_1. Initially, $P_1 = \emptyset; V_1.in = \{3, 5\}; V_1.out = \{6, 7, 12\}$. In fragment F_1, node *id = 1* is a start node, so *node 1* is put into $V_1.in$. The vector *Boolean Formulas* of each node in F_1 consists of five entries, corresponding to the states (1, PL, PP, SE, 16) in the query automaton. For the output nodes, (false, false, false, $X_{(6,SE)}$, false), (false, false, $X_{(7,PP)}$, false, false), and (false, $X_{(12,PL)}$, false, false, false) are corresponding to each node in sets of {6, 7, 12}. The vector *rvec* for each input node in $V_1.in$ is computed by procedure *getRvec*, the results are as follows: ($X_{(6,SE)} \vee X_{(7,PP)} \vee X_{(12,PL)}$, false, false, false, false), (false, $X_{(7,PP)} \vee X_{(12,PL)}$, false, false, false), (false, $X_{(7,PP)} \vee X_{(12,PL)}$, false, false, false), corresponding to each node in sets of {1, 3, 5}. It is also the result of partial result P_1 which will be sent to coordinator site. Similarly, we can compute the partial results P_2 and P_3 of two fragments F_2 and F_3, respectively.

A dependency graph G_d is constructed as shown in Figure 3. Here, a *Boolean variable* $X_{(v,q)}$ in the partial result is a node in G_d denoted as Xv (e.g., X1, X2). After that, the procedure *evalRRQ* (not shown) is called, which uses BFS algorithm to search for the final answer. Here, there exists a path on G_d from start node to terminal node as follows: X1 → X12 → X16. In this example, a path in graph G which satisfies query q_r is: $\rho = 1 \rightarrow 3 \rightarrow 5 \rightarrow 12 \rightarrow 15 \rightarrow 16$. Therefore, the final answer to query q_r in this example is *True*.

Discussion. The approach in [9] improves for answering RRQs on distributed graphs by visiting each site only once. However, it is difficult to implement such approach on large graphs (e.g., several hundred million nodes) due to the amount of data transfers to one machine (coordinator site S_c). The bottleneck occurs on S_c when both the input and output nodes increase and the number of paths satisfying a query raises.

We detected a lot of redundant nodes/edges in partial results which sent to coordinator site. These nodes/edges are

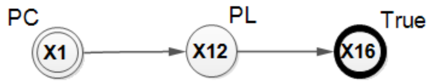


Figure 4: Dependency graph G_d after use *Local Filter* algorithm

unnecessarily finding answer to query q_r. Indeed, we can see dependency graph in Figure 3, the set of nodes {X3, X5, X6, X7, X9, X14} and the set of edges {(X1, X6), (X1, X7), (X3, X7), (X3, X12), (X5, X7), (X5, X12), (X9, X16), (X14, X16)} is unnecessary for finding the final answer. We can not find any path satisfying the query q_r via these nodes/edges. Therefore, our work proposes method to find and remove redundant nodes/edges. The effectiveness of our work avoids the communication bottleneck when query processing on large graphs.

4. OUR APPROACH

In this section, we present two algorithms that solve communication bottleneck on the coordinator site while assembling and searching the final answer for RRQs. We remove the amount of redundant data at each local site in parallel before it is sent to coordinator site. First, an effective filtering technique is given in Section 4.1. Second, we perform a modification of the local evaluation that is shown in Algorithm 1. This algorithm minimizes the size of dependency graph G_d, so we named it *Local Contract*. It is presented in Section 4.2.

4.1 Local Filtering

In this section, we present how to reduce the amount of redundant data in dependency graph $G_d = (V_d, E_d, L_d)$. Here, the problem is: *How to detect redundant nodes/edges on each local site?* We remove all redundant nodes/edges on each local site as soon as they are detected. To detect the redundant nodes/edges we give two definitions as follows:

Definition 1 (Node Redundant). A node u is called redundant in dependency graph G_d if and only if it does not have any node $v \in V_d$ connecting to u or from u to v, where $u \neq v$ and u is not a start or terminal node.

Definition 2 (Edge Redundant). An edge (u, v) is called redundant in dependency graph G_d if and only if node u or v is a redundant node, where $u \neq v$ and u, $v \in V_d$

Based on Definitions 1 and 2, we develop an algorithm to find and remove redundant nodes/edges on each local site, in parallel. We now call it *Local Filter* (LF) algorithm. It is described in Algorithm 3.

Example 4: Consider again query q_r(1, 16, R) on graph G as described in Example 3. We perform a local filter from the partial results shown in previous example. The filtering of redundant nodes/edges on F_1 is as follows: (1) a sub dependency graph G_s is made from P_1, here, $V_s.in$ = {1, 3, 5}, $V_s.out$ = {6, 7, 12} and cE_s = {(1, 6), (1, 7), (1, 12), (3, 7), (3, 12), (5, 7), (5, 12)}; (2) a set of reachable input nodes from three fragments *reachInputs* = {1, 3, 5, 9, 12} and the set of reachable output nodes *reachOutputs* = {6, 7, 12, 16}; (3) find and remove: for the nodes in $V_s.in$, input node 1 appears in *reachOutputs* $\cup$ {1}, but nodes 6, 7 do not appear in *reachInputs* $\cup$ {16} , therefore, the set of edges {(1, 6), (1, 7)} is removed from cE_s. The input node 3 and 5 do not appear in *reachOutputs* $\cup$ {1}, remove {3, 5} from $V_s.in$ and remove a set of edges {(3, 7), (3, 12), (5, 7), (5, 12)} from cE_s. Now, $V_s.in$ = {1}, $V_s.out$ = {6, 7, 12}, cE_s = (1, 12). (4) For the output nodes in $V_s.out$, node 6 and node 7 do not appear in any edges $\in cE_s$, therefore {6, 7} is removed from $V_s.out$. Finally, a new sub dependency graph $G_{d1} = (V_{d1}, E_{d1})$ is the result of the filter, where V_{d1} = {1, 12} and E_{d1} = {(1, 12)}.

It is similar to filter on F_2 and F_3. Figure 4 is a dependency graph after using our algorithm. It shows an efficient optimization for answering RRQs on distributed graphs.

Algorithm 3 Using procedure ***LocalFilter*** to find and remove redundant nodes and edges from partial answer P_i;

Input: The partial answer P_i; ***reachInputs*** is a set of reachable input nodes from all fragments; ***reachOutputs*** is a set of reachable output nodes from all fragments; ***s*** is a start node; ***t*** is a terminal node.
Output: The new sub dependency graph G_{si}

```
Construct sub dependency graph G_s = (V_s.in ∪ V_s.out, cE_s, L_s) from partial answer P_i;
for each node v ∈ V_s.in do
    if node v ∉ {reachOutputs ∪ {s}} then
        V_s.in ← V_s.in/{v};
        cE_s ← cE_s/{v.edges};
    else
        for each edge (v, u) ∈ v.edges do
            if node u ∉ {reachInputs ∪ {t}} then
                cE_s ← cE_s/{(v, u)};
            end if
        end for
    end if
end for
for each node u ∈ V_s.out do
    if has no edge coming node u then
        V_s.out = V_s.out/{u}
    end if
end for
Send new sub dependency graph G_s to coordinator site;
```

4.2 Local Contraction

In this section we improve the local evaluation in Algorithm 2 to limit the redundant nodes/edges for computing the partial result.

Our idea is described as in Algorithm 4. Here, a *Boolean variable* can be set between two input nodes (line 7-14). If an input node v can reach another input node u, v matches state q and u matches state q' then $X_{(u,q')}$ is put into v.rvec[q] instead of a *Boolean Formulas* of u at state q', where u, $v \in V_i.in$ and q, $q' \in$ Q in M. In case node v is an input node, after v.rvec is computed (line 21), we continue to filter and remove the *Boolean variable* in v.rvec occurring simultaneously in v.rvec and v'.rvec, where v' is also an input node and v is reachable v' (line 22-32).

Example 5: In this example, we focus on computing partial result P_1 for query q_r(1, 16, R) on fragment F_1 using procedures: *localEval* and *getRvecEq* above. The computing of P_1 in this example is different to Example 3 by computing vectors of *Boolean Formulas* for input nodes. Therefore, we show the results of computing vector *rvec* for the input nodes $\in V_1.in$ = {1, 3, 5}. Figure 5 shows two sub dependency graphs on F_1 with two different computation ways. Intuitively, we can see that the size of G_{sd1} using *Lo-*

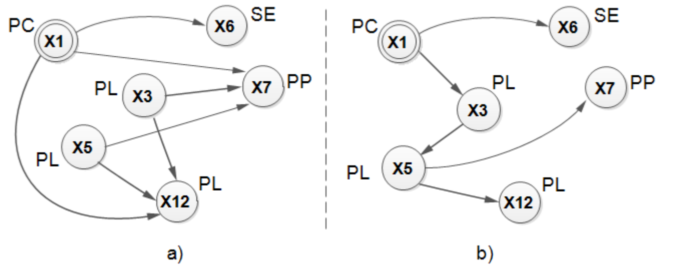


Figure 5: Sub dependency graphs on fragment F_1, a) Compute the using *Local Evaluation* in Algorithm 1, b) Compute the using our modifying algorithm with *Local Contraction*

cal Contract method is smaller than only using Wenfei Fan's algorithm.

5. IMPLEMENTATION AND EXPERIMENTS

In this section, we present our implementation using MapReduce and show experimental results. We used real-life datasets as well as created graphs for the evaluation. We also compared our algorithms with Wenfei Fan's re-implementations available and show results for graphs with different size and the number of partitions..

5.1 MapReduce Overview

MapReduce [7] is a programming model and an associated implementation for processing and generating large datasets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Written programs in this functional style are automatically parallelized and executed on a large cluster of commodity machines. A MapReduce job usually splits the input dataset into independent chunks which are processed by the map tasks. The outputs of the maps are sorted then input to the reduce tasks. Typically both the input and the output of the job are stored in the Hadoop Distributed File System (HDFS) [1]. The job is finished when all map and reduce tasks are completed.

The mechanism of MapReduce is consistent for implementation algorithms using partial evaluation technique as shown in Section 3.1. We also utilize this advantage to perform efficiently the optimization performance of answering query on distributed graphs (see Section 4).

5.2 Implementation

We present techniques using MapReduce to implement our algorithms. The answering RRQ model is shown in Figure 6. We use two MapReduce jobs in our implementation.

In the first job, we perform a local evaluation in parallel by calling procedure ***LocalEval*** as in Algorithm 5. Each map task is assigned a different key to send its partial result to second job. Specially, we do not need to use a reducer in the first job, as used in the usual pattern of MapReduce framework. Time is reduced by skipping the shuffle step between Mapper and Reducer. The result of each map task is stored in HDFS as follows: (1) extract input nodes in partial result and save into a HDFS file in a folder ***Iset*** with file name format *<key>.txt*; similarly in the output nodes it is stored in the folder ***Oset***; (2) the partial result is written

Algorithm 4 Using procedure ***getRvecEq*** to modify algorithm to minimal equivalent sub dependency graph q_r

Input: A node v, a fragment F_i, a query automaton $q_r(s, t, M)$
Output: The vector v.rvec

if v.visit = true **then**
  return v.rvec;
end if
for each state $q \in Q$ **do**
  v.rvec[q] $\leftarrow$ false;
end for
for each node $u \in ChildrenNodes(v, F_i.V)$ **do**
  if u.visit = false **then**
    u.rvec $\leftarrow$ getRvec(u, F_i, q_r);
  end if
  for each state $q \in Q$ **do**
    if $L(v) = \mu(q)$ **then**
      if $u \in V_i.in$ and u *matches* $q' \in Q$ **then**
        v.rvec[q] $\leftarrow$ v.rvec[q] $\cup\{X_{(u,q')}\}$;
      else
        v.rvec[q] $\leftarrow$ v.rvec[q] $\cup$ getVec(u, u.rvec, q, q_r);
      end if
    end if
  end for
  v.visit $\leftarrow$ true;
end for
if $v \in V_i.in$ **then**
  for each state $q \in Q$ **do**
    for each $X_{(v',q)} \in v$.rvec[q] **do**
      if $v' \in V_i.in$ and $\exists X_{(u',q)} \in v'$.rvec[q] **then**
        if $X_{(u',q)} \in v$.rvec[q] **then**
          v.rvec[q] $\leftarrow$ v.rvec[q]$/\{X(u', q)\}$;
        end if
      end if
    end for
  end for
end if
return v.rvec;

Algorithm 5 Procedure ***LocalEvalMapper***

Input: A pair key/value $< i, F_i >$
Output: A pair key/value $< i, P_i >$

1: Get query automaton $q_r(s, t, M)$ from Distributed Cached File System
2: $P_i \leftarrow$ localEval(F_i, q_r);
3: Construct sub dependency graph $G_s = (V_s.in \cup V_s.out, cE_s, L_s)$ from P_i;
4: Write $V_s.in$ and $V_s.out$ to HDFS with two files, respectively
5: Send $< i, P_i >$ to mapper i in Job 2

Algorithm 6 Procedure ***LocalFilterMapper***

Input: A pair key/value $< i, P_i >$
Output: A pair key/value $< i, G_{di} >$

1: Get query automaton $q_r(s, t, M)$ from Distributed Cached File System
2: *reachInputs* $\leftarrow$ getAllReachInputs();
3: *reachOutputs* $\leftarrow$ getAllReachOutputs();
4: $G_{di} \leftarrow$ localFilter($P_i, reachInputs, reachOutputs, s, t$);
5: Send $< 1, G_{di} >$ to a reducer in Job 2

Algorithm 7 Procedure ***EvalReducer***

Input: A list of pair key/value $< 1, G_{di} >$
Output: The Boolean value finalAnswer for query q_r

Get query automaton $q_r(s,t,M)$ from Distributed Cached File System
for each pair <1, G_{di}> from client site **do**
  $G_d \leftarrow G_d \cup G_{di}$;
end for
finalAnswer $\leftarrow$ EvalRRQ(G_d, q_r);
return finalAnswer;

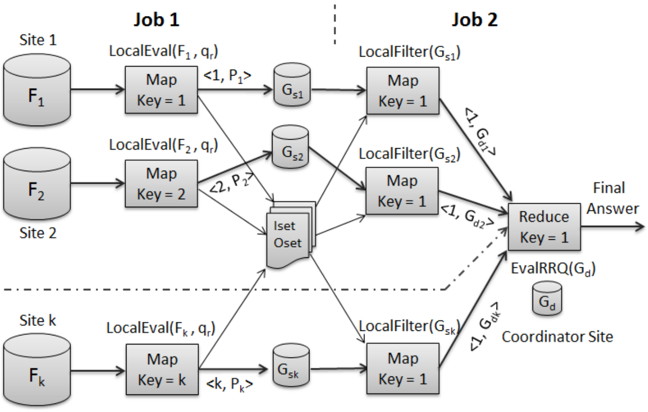


Figure 6: Model optimization for answers regular reachability query in MapReduce

by Mapper to HDFS with a corresponding key.

The second job collects the partial answers and searches to get final answer. It is shown in Algorithm 6 and Algorithm 7. Here, each map task in the second job (a) reads all input nodes in the folder ***Iset***, all output nodes in the folder ***Oset*** and removes duplicate values simultaneously; (b) gets the partial answer which is stored in HDFS by map task in the first job; (c) performs the filtering redundant nodes/edges by calling procedure ***LocalFilter***; (d) sends partial answer by only one key for all map tasks to the reducer. Further on, the reducer assembles all partial results and calls procedure ***EvalRRQ*** to get the final answer to query q_r.

Thus, our implementation model using MapReduce is different from the model in [9]. Here, we use two maps and one reduce function instead of one map and one reduce function as in [9]. Based on this model we can perform the filtering of redundant data.

5.3 Experimental Settings

The setting for the experiment used to execute algorithms in this paper is the following.

Environment setting. Our experiments were run 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. All algorithms are implemented in Java.

Real-life dataset. Table 2 lists the real graphs that we used. YouTube[2] a social network of videos, where each video is a node in the graph with attributes (e.g., category) and each edge indicates a recommendation, and whether a video is related to another video. DBLP[3] a citation network dataset [21] in which each node is a paper associated with attributes (e.g., publication venue), each edge shows if a paper is cited by another paper.

Table 2: Real-life datasets

Dataset	|V|	|E|	|L|
Youtube	914,300	2,285,709	13
DBLP	714,207	885,793	3,815

We have partitioned real-life graphs by using GraphLab[4] and synthetic graphs G into a set of partitions. Here, we first chose the number of partitions as 32. Then we increase this number by 64 to raise it over the number of cross-links among partitions.

Random Query sets. We also implement a random generator algorithm to create a set of queries from Σ. By fixing the number of states and transitions we generate a set of queries. To partition the graph by GraphLab, we converted *video ids* in YouTube dataset from *string* type to *integer*. We generated the query that checks whether there exists a path from a *video id* to another *video id* by limitation of the categories (e.g., Music, Entertainment). For DBLP dataset, the query determines whether a paper can reach another paper by limitation of the publication venues (e.g. PPoPP, VLDB).

5.4 Experimental Results

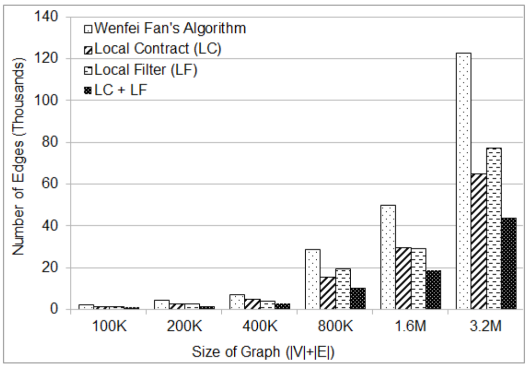
In this section, we present experimental results of our implementation using real-life datasets of YouTube and DBLP. Moreover, we also compare the total edges in the dependency graph with the result generated by Wenfei Fan's algorithm (called WFA from now on).

To evaluate the efficiency and scalability, we varied the graph size from 100K to 3.2M on YouTube dataset and from 100K to 1.6M on DBLP dataset. The comparison among algorithms is based on executions by the one query. Figure 7 shows size of reduction of the dependency graph with real-life datasets by varying the number of partitions.

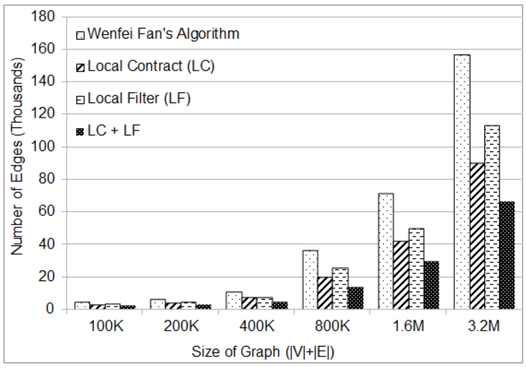
The first experiment is on YouTube data with 6 different graph sizes. We choose randomly a query with 4 states and 6 transitions to evaluate efficiency of our method. As shown in Figure 7(a), the size of the dependency graph in our algorithms is smaller than that in WFA. Indeed, we compute the average reduction in the number of edges in dependency graph and show that: (1) only *Local Contract* (called LC) is reduced by 39%; (2) only *Local Filter* (called LF) is reduced by 38%; (3) the combination LC + LF is reduced by 60%. In addition, we also evaluate the efficiency by increasing the number of partitions to 64. Hence, the total number of input and output nodes is raised. As result, the amount of data transfers via network grows. Similarly, we show the results in Figure 7(b). More specifically, we use the same query in the case of 32 partitions and the average reductions are 41%, 30%, and 58% corresponding to LC, LF, and (LC + LF).

Similarly, for experiments on the DBLP dataset, we use 5 variations in the graph size. In case of 32 partitions, the results are shown in Figure 7(c). Here, our improvements

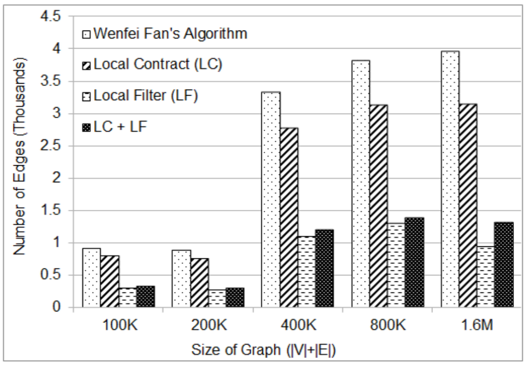
[1]http://edubase.jp/cloud
[2]http://netsg.cs.sfu.ca/youtubedata/
[3]http://arnetminer.org/citation
[4]http://graphlab.org/



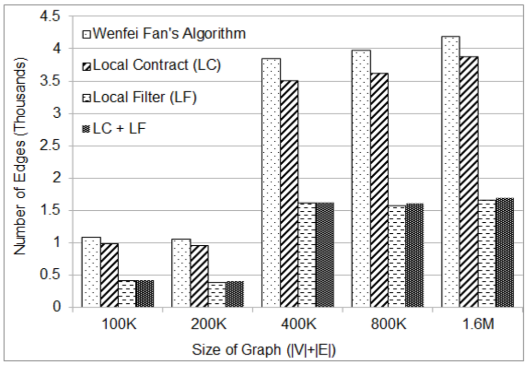
(a) YouTube dataset with 32 partitions



(b) YouTube dataset with 64 partitions



(c) DBLP dataset with 32 partitions



(d) DBLP dataset with 64 partitions

Figure 7: Size reduction of dependency graph by varying size of graph and the number of partitions

still reduce large amounts of redundant data. Therein, *Local Filter* algorithm is the most dominated with an average reduction of 69%. The combination of *Local Filter* and *Local Contract* is reduced by 64%. By using *Local Contract* in this dataset it is reduced by 16%. We found an efficiency reduction when increasing the number of partitions to 64. The corresponding results with LC, LF, and (LC + LF) are 8%, 60%, and 59%. Those results are illustrated in Figure 7(d).

Besides, the computation time is shown in Figure 8 for the experiment on YouTube data with 32 partitions. Here, time differences among experiments with different graph sizes is not much. It fluctuates from 24 seconds to 26 seconds with *Local Filter* and to 30 seconds with the combination of *Local Contract* and *Local Filter*. We also illustrate time of WFA in Figure 8.

Analysis. The experimentation on the two datasets above indicates that our method reduced large amount of redundant data when answering RRQs on distributed graphs. The efficiency of the techniques is different in each dataset. Indeed, *Local Contract* efficiently reduces the size of the dependency graph on YouTube dataset by approximately 40%. The combination of *Local Contract* and *Local Filter* is the most effective technique on the YouTube dataset with a reduction of 60% in the amount of redundant data.

However, for the DBLP dataset, *Local Contract* seems to be less effective reducing only 16% on 32 partitions and 8% on 64 partitions. In this case, *Local Filter* is the most effective technique with an amount of redundant data reduction of 69% and 60% corresponding to 32 and 64 partitions.

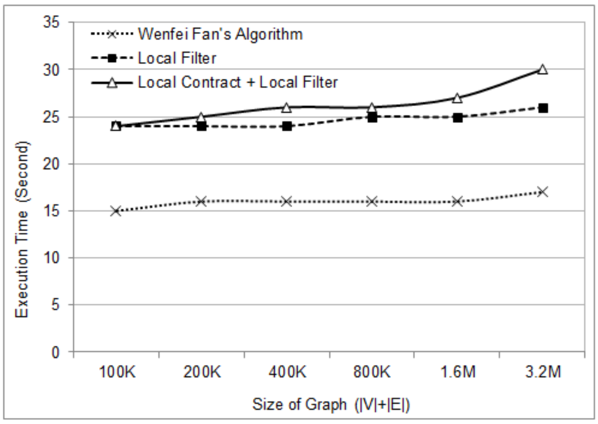


Figure 8: The average execution time in answering RRQs on YouTube dataset with 32 partitions

The difference is due to the size of label set in YouTube data (|L| = 13) dramatically smaller than DBLP data (|L| = 3,815) (see Table 2). On the other hand, the size of graph of YouTube data is larger than that of DBLP data. Therefore, the relationship among input nodes in each fragment is becoming denser for YouTube data. This is an advantage for *Local Contract*. Thus, we can choose the appropriate technique depending on the characteristics of the distributed graphs.

6. RELATED WORK

There are two approaches to answer a reachability query. It can be processed traversing from a node to another node using distributed breadth first search (DBFS) or distributed depth first search (DDFS) over the graph on demand (see [22]). However, the problem with DFS is difficult to parallelize [16]. Mark Sevalnev in [18] gave a DBFS algorithm on Hadoop-framework which had time efficiency of O(V + E) $*$ V $*$ *log*(V)), where V is the total number of nodes and E is the total number of edges in a graph. It takes nearly cubic time depending on graph size. The former requires too much time in querying and the latter requires too much space. Therefore, the two approaches might be infeasible.

Several approaches have been developed for evaluating queries on distributed graphs. Dan Suciu in [20] proposed an approach to evaluate queries on semistructured databases, and an extension is given in [19] based on message passing. It divides the query processing into a set of processes. Therein, each process will compute and deliver its results to other processes. It takes a bounded by $O(n^2)$ for the amount of data transfers via network, where n is the total of cross-edges. In recent years, several systems have been designed to support extremely large graphs such as Malewicz et al. with Pregel [15]. It is also based on message passing. Yildirim et al. proposed Grail system which stands for graph reachability indexing via randomized interval labeling (see [25]).

Recently, Wenfei Fan et al. in [9] proposed efficient algorithms for answering three classes of reachability queries on distributed graphs based on partial evaluation. Therein, both the total computation time and the total network traffic depend on the total of number of cross-edges and the number of states in the query automaton. However, it has a communication bottleneck problem when assembling all distributed partial query results. This is the problem that we solved in this paper. Here, the large amount of redundant data is detected and removed by our method, which is not found in the Wenfei Fan's algorithm.

7. CONCLUSION

We have proposed two algorithms to reduce data transfers for answering regular reachability queries on distributed graphs using MapReduce. Our algorithms contract partial results and filter large amount of redundant data, locally. Thus, the communication bottleneck is limited when assembling all distributed partial query results. More significantly, if the communication bottleneck problem is solved, it will increase scalability for query evaluating on distributed graphs. Hence, our method has a lot of potential applications in areas from social network analysis, web mining, to ontology matching. Our experimental results on real-life graphs demonstrated the effectiveness of our method.

In the future, we will apply our approach to optimize other queries on distributed environment with MapReduce. We are currently developing distributed evaluation algorithms on semistructured data. Another research direction is to extend our approach to efficiently answer *select-where* queries on distributed databases.

8. REFERENCES

[1] D. Borthakuro. Hdfs architecture guide. `http://hadoop.apache.org/docs/stable/hdfs_design.html`, June 2013.

[2] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[3] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 211–222. VLDB Endowment, 2006.

[4] J. Cheng, J. X. Yu, X. Lin, H. Wang, and S. Y. Philip. Fast computation of reachability labeling for large graphs. In *Advances in Database Technology-EDBT 2006*, pages 961–979. Springer, 2006.

[5] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 193–204. ACM, 2008.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 39–50. IEEE, 2011.

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

[10] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. D. Viglas, Y. Wang, J. F. Naughton, et al. Mixed mode xml query processing. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 225–236. VLDB Endowment, 2003.

[11] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 123–134. ACM, 2010.

[12] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 595–608. ACM, 2008.

[13] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

[14] I. Konstas, V. Stathopoulos, and J. M. Jose. On social networks and collaborative recommendation. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 195–202. ACM, 2009.

[15] 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*, pages 135–146. ACM, 2010.

[16] J. H. Reif. Depth-first search is inherently sequential.

Information Processing Letters, 20(5):229–234, 1985.

[17] J. Scott, T. Ideker, R. M. Karp, and R. Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. *Journal of Computational Biology*, 13(2):133–144, 2006.

[18] M. Sevalnev. From prefix computation on pram for finding euler tours to usage of hadoop-framework for distributed breadth first search, 2010.

[19] M. Shoaran and A. Thomo. Fault-tolerant computation of distributed regular path queries. *Theoretical Computer Science*, 410(1):62–77, 2009.

[20] D. Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems (TODS)*, 27(1):1–62, 2002.

[21] J. Tang, D. Zhang, and L. Yao. Social network extraction of academic researchers. In *ICDM'07*, pages 292–301, 2007.

[22] G. Tel. Distributed graph exploration. *Obtained from http://carol. wins. uva. nl/ delaat/ netwerken college/explo. pdf*, 1997.

[23] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 177–186. ACM, 2011.

[24] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 3. ACM, 2012.

[25] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.