Fast Top-K Path-based Relevance Query on Massive Graphs

Samamon Khemmarat [1], Lixin Gao [2]

Department of Electrical and Computer Engineering
University of Massachusetts Amherst
Amherst, USA
[1] khemmarat@ecs.umass.edu
[2] lgao@ecs.umass.edu

***Abstract*—The task of obtaining the items highly-relevant to a given set of query items is a basis for various applications, such as recommendation and prediction. A family of path-based relevance metrics, which quantify item relevance based on the paths in a given item graph, have been shown to be effective in capturing the relevance in many applications. Despite their effectiveness, path-based relevance normally requires time-consuming iterative computation. We propose an approach to obtain the top-k most relevant items for a given query item set quickly. Our approach can obtain the top-k items without having to compute converged scores. The approach is designed for a distributed environment, which makes it scale for massive graphs having hundreds of millions of nodes. Our experimental results show that the proposed approach can produce the result 20 to 50 times faster than a previously proposed approach and can scale well with both the size of input and the number of machines used in the computation.**

I. Introduction

The task of selecting the items that are highly relevant to a given set of items, or a *query item set*, is a key component in many applications. One well-known example of such applications is a personalized recommendation system, whose goal is to present the items that will interest a user. This can be achieved by selecting the items that are the most relevant to those that the user has previously shown interest in. For marketing, it is useful to know a set of people that will highly influence a targeted set of customers. Other examples are a personalized search system, which tries to produce the results that match both a given search query and a known user preference, and a search query suggestion system, which assists a user in searching by offering relevant search queries to the user's past queries.

To obtain a set of highly relevant items, a relevance metric is needed so that each item can be assigned a score based on their relevance to the query item set. There are many ways to quantify item relevance. Among them, there is a family of relevance metrics that define the relevance between items based on the structure of a graph induced from explicit relationships between items. Using the item relationship graph, in which each node is an item and each edge represents a relationship between a pair of item, these relevance metrics consider all the paths connecting between items in order to quantify their relevance. We refer to this type of relevance metrics as *path-based relevance metrics*. Scores computed from well-known algorithms such as Personalized PageRank [1] and Adsorption [2] can be classified as path-based metrics. These path-based metrics have been shown that they can effectively capture the relevance between items and provide high-quality results for recommendation and other applications.

Despite their effectiveness, path-based relevance scores can be time-consuming to compute. Since the metrics rely on the path structure in an input graph, their computation usually requires several iterations of graph processing. With many queries to process and the interactive nature of many applications, this can hinder the use of path-based metrics in real applications, especially when item graphs are very large as commonly found in today's applications. One solution is to precompute and store all the path-based scores between each pair of items, but this approach needs at least $O(n^2)$ storage space where n is the number of items, which is infeasible when n is large. As a result, the computation needs to be performed as each query is issued.

Our goal in this paper is to provide an approach to obtain the set of highly relevant items quickly when a path-based metric is used. We focus on the specific problem of finding the top-k items that are the most relevant to a given query item set. We propose an approach that detects the emergence of the top-k items during the computation. The proposed top-k emergence detection utilizes the lower bound and upper bound scores of items computed from intermediate values during the computation. We provide the lower bound and upper bounds generalized for path-based metrics. In contrast to existing work, which focuses on a single machine computation, our proposed top-k emergence detection works in a distributed environment, which makes it scale for massive graphs.

Our main contributions are as follows:

- We propose an approach for determining the emergence of the top-k most relevant items during the computation of path-based scores, thus eliminating the need to compute converged scores, resulting in faster response to top item queries.
- We propose generalized lower bounds and upper bounds for path-based scores to be used with the proposed emergence detection.
- We demonstrate that the proposed approach can be applied to distributed computation, thus allowing scalability

for massive graphs.

- We implement the proposed emergence detection and perform an evaluation of its performance on a single machine and in a distributed environment. The results show that with our approach the results for the top-k item queries can be obtained 20 to 50 times faster than an approach previously proposed for a single machine. A scalability test demonstrates that our approach works with massive graphs having several hundreds of millions of nodes and several billions of edges.

The rest of the paper is organized as follows. Section II introduces path-based metrics and formally defines the top-k query problem. In Section III, the top-k emergence detection mechanism is described, followed by generalized bounds for path-based metrics in Section IV. Section V addresses the implementation of the emergence detection in a distributed environment. The evaluation of our approach is presented in Section VI. We discuss related work in Section VII and conclude our work in Section VIII.

II. Top-k Query on Path-based Relevance Metrics

In the following we introduce path-based relevance metrics and formally define the top-k query problem.

A. Path-based Relevance Metrics

In many applications, items can naturally be represented as a graph where each node in the graph represents an item and each edge represents a relationship between a pair of items. For example, for a personalized news recommendation, a news graph can be created with edges connecting the news that are viewed by the same users. In online social networks, a user graph can be created with edges connecting the users that have added each other as friends. Path-based relevance metrics quantify the relevance between items based on the structure of the given item graph. A path-based metric defines a score of each path in the graph, and the relevance between nodes is defined as the summation of the scores of all the paths connecting the nodes. In the following, we present well known examples of path-based metrics and then provide the generalized form of the path-based metrics.

1) Example of Path-based Relevance Metrics: First, we introduce the notations that will be used throughout the paper. The goal of a relevance metric is to quantify the relevance of each item to a given query item set S. Each item in the query set may be associated with a preference value, reflecting its importance or the bias towards the item. We represent the query set S as a query vector s, where $s[i]$ is a positive real value corresponding to the preference value of item i. If i is not in the query set, then $s[i] = 0$.

Path-based metrics are defined over a given item graph with a set of nodes V and a set of edges E. An adjacency matrix A is a $|V| \times |V|$ matrix in which $A[i, j] = 1$ if there is an edge from node j to node i, otherwise $A[i, j] = 0$. A path of length l is a sequence of $l + 1$ nodes, $(p_1, p_2, ...p_{l+1})$, such that there is an edge between node p_i and p_{i+1} for all $1 \leq i \leq l$. p_1 is referred to as the source node of the path, and p_{l+1} is referred to as the destination node. For a query set S, $path(S, i)$ denotes the set of all paths having node $j \in S$ as the source node and node i as the destination node. We denote a relevance score vector computed for a given query vector s as v_s. Next, we describe three path-based metrics.

Personalized PageRank: Personalized PageRank is one of the most well-known algorithms for providing personalized recommendation. In a recommendation system's context, a query vector is formed based on known user preferences, e.g., the news that a user read or the products bought by a user. Given the query set S, Personalized PageRank (PPR) defines the relevance of each item as the stationary distribution of a random walk on a graph, in which in each step with a probability d randomly moves to an out-neighbor of the current node, and with a probability $1 - d$ jumps to a node in the query set chosen with the probability proportional to their preferences.

The PPR score of node i can be defined in terms of the sum of the score of all paths from the nodes in S to node i [3]. Let s be the query vector corresponding to the query set S. The PPR score of node i can be given by:

$$v_s[i] = s[i](1-d) + \sum_{\substack{(p_1,..,p_{l+1}) \\ \in path(S,i)}} s[p_1](1-d) \prod_{k=1}^{l} \frac{d}{out(p_k)},$$

where $out(i)$ is the out-degree of node i.

Adsorption: The Adsorption algorithm [2] is a label propagation algorithm proposed for personalized recommendation. From the random-walk interpretation of the algorithm given in [2], node j's label distribution is the convex combination of the labels of other nodes. The weight of label i at node j depends on the probability that a random walk on the reverse input graph starting from node j reaches node i and takes node i's label. In each step of the random walk at node i, with the injection probability p_{inj} it will take the label of node i, and with the continue probability $p_{cont}[i]$, it continues walking to a randomly selected neighbor of node i's. The relevance between a query set S and node i can be quantified by the sum of the amount of label i in the label distributions of the query nodes.

Formally, given a graph with a weight matrix W, where $W[i, j]$ is the weight of an edge from node j to node i, the relevance score of node i with respect to S can be given by:

$$v_s[i] = s[i]p_{inj} + \sum_{\substack{(p_1,..,p_{l+1}) \\ \in path^R(S,i)}} s[p_1]p_{inj} \prod_{k=1}^{l} \frac{p_{cont}[p_k] \cdot W^T[p_{k+1}, p_k]}{\sum_{u \in V} W^T[u, p_k]},$$

where $path^R(S, i)$ is the set of all paths from node $j \in S$ to node i in the reverse input graph.

Katz Metric: In the Katz metric [4], the relevance is quantified based on the number of paths between two nodes, where shorter paths are considered to be more important. The equation for computing the Katz proximity is $K = \sum_{j=1}^{\infty} \beta^j A^j$, where $K[j, i]$ is the proximity from node i to node j and $0 < \beta < 1$. For a given query vector s, the relevance score of

TABLE I: A list of path-based metrics with their c and H

Algorithm	$c[i]$	$H[j,i]$
PPR	$1-d$	$d \cdot \frac{1}{out(i)} \cdot A[j,i]$
Adsorption	p_{inj}	$p_{cont}[i] \cdot \frac{W^T[j,i]}{\sum_{u \in V} W^T[u,i]}$
Katz	1	$\beta \cdot A[j,i]$

node i is:

$$v_s[i] = s[i] + \sum_{\substack{(p_1,..,p_{l+1}) \\ \in path(S,i)}} s[p_1]\beta^l.$$

2) Generalized Form of Path-based Metrics: From the three examples, we identify a generalized form of path-based metrics as follows:

$$v_s[i] = s[i]c[i] + \sum_{\substack{(p_1,..,p_{l+1}) \\ \in path(S,i)}} s[p_1]c[p_1] \prod_{k=1}^{l} H[p_{k+1}, p_k] \quad (1)$$

where $s[i]$ is the preference level of node i, $c[i]$ is a constant associated with node i, and $H[j,i]$ is a non-negative real value quantifying the influence from node i to node j. An equivalent vector-matrix equation can be given as $v_s = \sum_{l=0}^{\infty} H^l c^*$, where $c^*[i] = s[i]c[i]$. Accordingly, to ensure convergence matrix H must satisfy $\rho(H) < 1$, where $\rho(H)$ is the spectral radius of matrix H. The values of c and H for the three example path-based metrics are given in Table I.

B. Top-k Query Problem

The problem of our interests is to find the items that are highly relevant to a query item set when a path-based metric is used to quantify item relevance. Specifically, the question can be stated as "what are the k items that are the most relevant to a given query set?". For example, in online news recommendations, there are k slots for news items on a web page, so the system needs to select the k most relevant news for a user. Our goal is to provide a quick answer to the top-k query. We formally define our problem as follows.

Top-k Query Problem: Given an item graph and a query vector s as an input, find the set of k items that have the highest relevance scores with respect to the query vector.

To answer a top-k query, a relevance score of every node needs to be computed and compared. The computation of path-based relevance scores is usually performed iteratively. It can take a large amount of time for the scores to converge to their final values, resulting in slow response time to the top-k query. Our observation is that in fact the final result for the top-k query does not require precise relevance scores. This provides an opportunity to improve the response time. If the correct result can be determined based on the intermediate scores during computation, the computation can terminate early and provide a response to the query sooner. The important question is how to check whether the result obtained with the intermediate scores at a given time is correct. In the rest of this paper, we provide the solution and demonstrate that our approach helps to decrease the computation time for the top-k query.

III. Determining the Emergence of Top-K Nodes with Bounds

In this section, we describe the approach for determining whether the correct top-k items can be obtained at a given point of time during computation. We refer to this process as the *top-k emergence test*. When the test determines that the top-k items have emerged, the computation can be terminated. The proposed top-k emergence test uses the upper bounds and lower bounds of node scores. This section assumes that the upper bound scores and lower bound scores of nodes at time t are known and explains how the upper bounds and lower bounds are used in the emergence test. Later, in Section IV we will address how to compute these bounds.

Our approach works by maintaining a set of *candidate nodes*, i.e., nodes that can potentially be in the top-k set, and using the upper bound scores and lower bound scores to prune the nodes that cannot be the top-k items from the candidate set. Assuming that these upper bounds and lower bounds become closer to the real scores as the computation progresses, the size of the candidate set will keep decreasing. When only k nodes remain in the candidate set, the computation can be terminated since all the remaining candidates must be the top-k nodes.

Now we describe how the bounds are used to determine the candidate status. Let $u_i^{(t)}$ and $l_i^{(t)}$ denote the upper bound score and the lower bound score of node i at time t, respectively. Let $i_1, i_2, ..., i_n$ be the list of nodes sorted by their lower bound scores at time t in a descending order. We know that the top-k items *must* have a score more than or equal to the lower bound score of node i_k. We refer to the lower bound score of node i_k as a *threshold score*. Based on this fact, the upper bound score of a node is used to check whether a node can potentially have a score higher than or equal to the threshold score. Any node with an upper bound score lower than the threshold score can be pruned off the candidate set. From this idea, we provide the following condition as a necessary condition for a node to be in the top-k set.

Candidate condition: A node i in the top-k set must satisfy $u_i^{(t)} >= l_{i_k}^{(t)}$

The real top-k nodes always satisfy the candidate condition, while the other nodes may or may not, depending on the tightness of the bounds. The pruning is performed periodically until the number of remaining candidates is k, which means all the remaining candidates are the real top-k nodes.

The computation time can be further decreased by allowing some inaccuracy in the results as a trade-off. As mentioned in [5], in many situations it is acceptable to output a few more than k items to trade-off for faster computation time. When that is the case, we allow the computation to terminate when the number of candidates is between k and $\bar{k}$, where $\bar{k}$ is a parameter specified by a user and $\bar{k} \geq k$. The number of output items will be between k and $\bar{k}$, and the real top-k items are guaranteed to be among the output items. As can be seen, the lower bounds and upper bounds of node scores are the essential components of the emergence test. In the next section, we describe how the bounds can be obtained.

IV. Score Bounds for Asynchronous Accumulative Computation

The top-k emergence test described in the previous section can be used with any types of computation of path-based scores as long as the upper bounds and lower bounds of the scores can be computed during the computation. We propose to use the emergence detection mechanism with asynchronous accumulative computation, with which path-based scores can be computed efficiently and the bounds of node scores can be easily obtained. In the following, we introduce asynchronous accumulative computation for path-based metrics and then present the upper bounds and lower bounds of node scores that can be computed during the computation.

A. Asynchronous Accumulative Computation

From Section II-A2, the generalized form of path-based metrics in a vector-matrix equation form is $v_s = \sum_{l=0}^{\infty} H^l c^*$, where $c^*[i] = s[i]c[i]$. For clarity, from this point on we denote the converged relevance score as v_s^{∞}, while v_s denotes the relevance score at any time during the computation. From the equation, path-based metrics can be computed iteratively. In each iteration k, $v_s^{(k)}$ is computed as $v_s^{(k)} = Hv_s^{(k-1)} + c^*$ where $v_s^{(0)} = c^*$. Alternatively, the iterative computation can be computed accumulatively, where the changes of values in each iteration, denoted by $\Delta v_s^{(k)}$, are computed and used to update the values in the next iteration [6]. The accumulative computation is computed with the following equations:

$$v_s^{(k)} = v_s^{(k-1)} + \Delta v_s^{(k-1)} \quad (2)$$

$$\text{and } \Delta v_s^{(k)} = H\Delta v_s^{(k-1)}, \quad (3)$$

where $v_s^{(0)} = 0$ and $\Delta v_s^{(0)} = c^*$.

Accumulative computation can be performed asynchronously to achieve better performance. In asynchronous computation, v_s and Δv_s of each node are updated independently and the most recent value of Δv_s of a node is always used, as opposed to using the value from the previous iteration as in the synchronous model. When node i is selected to be updated, it accumulates $\Delta v_s[i]$ to $v_s[i]$ and triggers the other nodes to update their Δv_s according to $\Delta v_s[i] \times H[j,i]$. The intuition for the correctness of the asynchronous computation is as follows. From Equation 2 and 3, it can be seen that the change of the value of each node (Δv_s) in iteration k is computed based on the changes of the other nodes in iteration $k-1$ and then accumulated to v_s. This can be viewed as a change of value in each node being propagated to the other nodes, affecting their values. Regardless of when the changes are propagated, as long as all of them are propagated, the values of the nodes will approach the correct values. The formal proof of the equivalence between asynchronous and synchronous computation is provided in [7].

Asynchronous accumulative computation provides an efficient way to compute the path-based scores. Moreover, its accumulative nature provides a good basis for deriving the upper bounds and lower bounds of node scores. Since the scores of nodes keep increasing and approaching their real values, the lower bounds can be naturally obtained. Also, since the scores of the nodes are accumulated based on Δv_s, the upper bounds can be derived from Δv_s. In the next section, we show precisely how the upper bounds and lower bounds of node scores can be computed based on v_s and Δv_s.

TABLE II: Notations

Notation	Definition/Description
$\|\|M\|\|_1$	$\max_j \sum_i M[i,j]$ (L1 norm of matrix M)
$\|\|M\|\|_\infty$	$\max_i \sum_j M[i,j]$ (Infinity norm of matrix M)
$\|\|v\|\|_1$	$\sum_i v[i]$ (L1 norm of vector v)
$\|\|v\|\|_\infty$	$\max_i v[i]$ (Infinity norm of vector v)

B. Score Bounds

During asynchronous accumulative computation, node i is associated with two values, $v_s[i]$ and $\Delta v_s[i]$. For simplicity, we simplify the notations to $v[i]$ and $\Delta v[i]$, respectively. Given vector v and Δv from the computation at time t, denoted by $v^{(t)}$ and $\Delta v^{(t)}$, we present a lower bound and different upper bounds that can be computed from these intermediate values.

1) Lower bound: In asynchronous accumulative computation, we repeatedly compute the change in a node score (Δv) based on the changes in its neighbors's scores, and then *accumulate* the change to the node score (v). An entry $H[j,i]$ in the influence matrix indicates the factor of how much a change of score of node i affects the score of of j (as in Equation 3). Since H is a non-negative matrix, the change in a node score (Δv) is always positive, which means a node's score is non-decreasing. Therefore, we can simply use $v^{(t)}$ as the lower bound score of a node as stated in Theorem 1.

Theorem 1 (Lower bound)**.** $v^{(t)}[i] <= v^{(\infty)}[i]$*, where* $v^{(\infty)}[i]$ *is the converged score of node* i*.*

2) Upper bound: To obtain the upper bound of the score of node i, we have to quantify how much more Δv node i will receive after time t if the computation continues until convergence. We denote this amount by $d^{(t)}[i]$. In other words, $d^{(t)}[i]$ is the distance from $v^{(t)}[i]$ to its converged value $v^{(\infty)}[i]$, i.e., $d^{(t)}[i] = v^{(\infty)}[i] - v^{(t)}[i]$. To derive $d^{(t)}[i]$, we suppose that after time t every node is updated synchronously in iterations. Then, $d^{(t)}[i]$ is the sum of Δv that node i receives in iteration $0, 1, 2, ...$, which can be computed as in Equation 3. Lemma 1 provides a formal equation for computing $d^{(t)}[i]$. The proof of the lemma can be found in Appendix A.

Lemma 1. $d^{(t)}[i] = \sum_{k=0}^{\infty}(H^k \Delta v^{(t)})[i]$.

To obtain the upper bound of node score, we have to derive the upper bound of $d^{(t)}[i]$, denoted by $\bar{d}^{(t)}[i]$. The upper bound of node score is in the form of $v^{(t)}[i] + \bar{d}^{(t)}[i]$. In the following, we make use of several vector and matrix norms. The definitions of the norms are provided in Table II.

Naive upper bounds

The first upper bound is derived using the facts that $d^{(t)}[i] \leq ||d^{(t)}||_1$ and that $d^{(t)}[i] \leq ||d^{(t)}||_\infty$. Then, using the properties of the norms we can derive the upper bound of $||d^{(t)}||_1$ and $||d^{(t)}||_\infty$ and use them as an upper bound of $d^{(t)}[i]$ for every

node i. We refer to these bounds as the *naive bounds*. The upper bounds are given in Theorem 1. The proof of the theorem can be found in Appendix B.

Theorem 1 (Naive upper bound)**.**
If $||H||_\infty < 1$*, then* $v^{(\infty)}[i] \leq v^{(t)}[i] + \frac{||\Delta v^{(t)}||_\infty}{1-||H||_\infty}$.
If $||H||_1 < 1$*, then* $v^{(\infty)}[i] \leq v^{(t)}[i] + \frac{||\Delta v^{(t)}||_1}{1-||H||_1}$.

The bounds given are applicable when either $||H||_\infty < 1$ or $||H||_1 < 1$ is true. This is usually satisfied as several path-based metrics are based on a normalized adjacency matrix of an input graph, such as Personalized PageRank and Adsorption. A metric based on an unnormalized adjacency matrix can also satisfy these conditions if its damping factor is sufficiently small. For example, for the Katz metric, the conditions will be met if β is less than the reciprocal of the maximum in-degree or maximum out-degree of nodes in the input graph. In the case that matrix $||H||$ does not have such properties, our third upper bound presented later can be used instead.

l-hop-precise upper bounds

In the naive bounds, the derived upper bound of $d^{(t)}[i]$ is the same for every node as no specific information of node i is used in computing the bounds. The naive bounds can be improved by computing the exact amount of Δv that a node will receive in the next l iterations and deriving the upper bound of Δv that a node will receive in the later iterations. This is equivalent to computing the exact values of the first $l+1$ terms of $d^{(t)}[i]$ (as given in Lemma 1) where $l \geq 0$ and deriving the upper bounds of the remaining terms. The improved bounds are given in Theorem 2. The proof of the theorem can be found in Appendix C. We refer to these bounds as l-hop-precise upper bounds.

Theorem 2 (l-hop-precise upper bound)**.**
If $||H||_\infty < 1$*, then* $v^{(\infty)}[i] \leq v^{(t)}[i] + \sum_{k=0}^{l}(H^k \Delta v^{(t)})[i] + ||\Delta v^{(t)}||_\infty \frac{||H||_\infty^{l+1}}{1-||H||_\infty}$.
If $||H||_1 < 1$*, then* $v^{(\infty)}[i] \leq v^{(t)}[i] + \sum_{k=0}^{l}(H^k \Delta v^{(t)})[i] + ||\Delta v^{(t)}||_1 \frac{||H||_1^{l+1}}{1-||H||_1}$.

Computing the l-hop-precise upper bounds of every node requires $O(l|E|)$ work where $|E|$ is the number of edges in the graph. To limit the overhead of computing the bounds we let $l = 1$, resulting in the 1-hop-precise bound shown in Corollary 1.

Corollary 1 (1-hop-precise upper bound)**.**
If $||H||_\infty < 1$*, then* $v^{(\infty)}[i] \leq v^{(t)}[i] + \Delta v^{(t)}[i] + H\Delta v^{(t)}[i] + ||\Delta v^{(t)}||_\infty \frac{||H||_\infty^2}{1-||H||_\infty}$.
If $||H||_1 < 1$*, then* $v^{(\infty)}[i] \leq v^{(t)}[i] + \Delta v^{(t)}[i] + H\Delta v^{(t)}[i] + ||\Delta v^{(t)}||_1 \frac{||H||_1^2}{1-||H||_1}$.

Global-score-based upper bound

Although the l-hop-precise upper bounds provide improvement over the naive bounds, both of these bounds do not take into account the structure of the influence between nodes embedded in H. Our third derivation of the upper bounds utilizes such the structure.

For this third upper bound, each node records a single precomputed value of Δv it receives when *every* node is initialized with Δv of 1. We refer to this value as a *global score* of a node. Given a snapshot of a computation at time t, to obtain the upper bound scores, we can assume every node has the same Δv equal to $||\Delta v^{(t)}||_\infty$, i.e., the maximum Δv. Then, we can obtain the upper bound of Δv a node will receive by scaling the node's global score according to $||\Delta v^{(t)}||_\infty$. We refer to this bound as *global-score-based upper bound.*

A formal definition of the global-score-based upper bound is as followed. Let $\vec{1}$ be a vector where every entry is equal to 1. Define a global score of node i to be the score of node i when $\Delta v^{(0)}$ is initialized as $\vec{1}$, that is, $v_{global}[i] = (\sum_{k=0}^{\infty} H^k \vec{1})[i]$. Theorem 3 provides an upper bound score of a node in terms of its global score. The formal proof of the theorem can be found in Appendix D.

Theorem 3 (Global-score-based upper bound)**.**
$v^{(\infty)}[i] \leq v^{(t)}[i] + \Delta v^{(t)}[i] + ||\Delta v^{(t)}||_\infty (v_{global}[i] - 1)$

To use the global-score-based upper bound, v_{global} should be precomputed and stored. v_{global} can be computed within reasonable time, as shown in our experiment in Section VI-A5. The space requirement for storing the global scores is $O(n)$. Although it is clear that the 1-hop-precise bound is tighter than the naive bound, there is no clear winner between the global-score-based bound and the 1-hop-precise bound. Our experiments in Section VI compare the efficiency of the top-k emergence detection when each of these bounds is used.

C. Optimization with Prioritized Node Update Scheduling

As mentioned in Section IV-A, nodes are selected to update their values independently in asynchronous accumulative computation. There are different ways to determine how the nodes should be selected for an update. One basic scheme is round-robin scheduling, in which the nodes are selected in a circular order. Alternatively, prioritized scheduling can be used, in which nodes with high "priority" are selected to be updated first. Intuitively, for the best performance, the nodes that will contribute the most to the completion of the computation should have higher priority.

For top-k query computation, the computation completes when the emergence test returns true. The success of the emergence test depends on the tightness of the upper bound and lower bound of the scores. When the bounds are tight, more nodes can be marked off as a non-candidate. Consider the bounds proposed in the previous section. Each node's upper bound and lower bound depend on its individual values, which are $v[i]$ and $\Delta v[i]$, and the globally aggregated values, which are $||\Delta v||_1$ and $||\Delta v||_\infty$. Since we want to make the bounds as tight as possible for *every* node, we should make $||\Delta v||_1$ and $||\Delta v||_\infty$ small because these values are commonly used to compute the upper bounds for every node. Accordingly, nodes with the highest Δv should be updated first so that $||\Delta v||_1$ and $||\Delta v||_\infty$ will decrease at a faster rate. Therefore, for top-k query computation, the priority of node i for prioritized

scheduling should be $\Delta v[i]$. The benefit from using prioritized update scheduling is evaluated in Section VI.

V. Detecting Top-k Emergence in Distributed Computation

This section discusses the implementation of the top-k emergence detection mechanism in a distributed setting. First, we describe how to perform asynchronous accumulative computation in a distributed setting. Then, we present the bounds modified from Section IV to be better suited for distributed computation. Finally, we explain the details of the implementation of distributed top-k emergence test.

A. Distributed asynchronous accumulative computation

There are multiple processors in distributed computation. We designate one processor as a master which controls the flow of the computation. The other processors are referred to as workers. To perform distributed asynchronous accumulative computation, nodes from an input graph are partitioned so that each worker is responsible for a subset of nodes of equal size. In this paper, we assume the nodes are partitioned by hashing. Better partitioning can be applied for better performance, but it is not in the scope of this paper. Each worker stores the information about the nodes it is responsible for, referred to as its *local nodes*. The information associated with node i includes $v[i]$, $\Delta v[i]$, and the influence from node i to other nodes according to matrix H (as defined in Section II-A2).

During the computation, each worker selects its local nodes to update based on an update scheduling policy. If an update of a local node affects the value of a non-local node, the corresponding Δv is sent to the worker responsible for that node. The master periodically collects the progress statistics from the workers and determines the termination of the computation. If the scores are to be computed until convergence, the master determines the termination by computing $||\Delta v||_1$ from the local statistics sent from the workers. When the value $||\Delta v||_1$ falls below the user-defined threshold, the computation will be terminated. When the top-k emergence test is used, the master determines the termination based on the number of remaining candidates sent from the workers. The details of this mechanism are provided in Section V-C.

From the above description, it can be seen that the use of non-local nodes' information at a worker will incur communication overhead and should be avoided. Next, we analyze the bounds proposed in Section IV and propose a modification to reduce such an overhead.

B. Bounds for distributed computation

In the top-k emergence test, the upper bound score of each node needs to be computed to determine whether it is a candidate node. We classify the values used for computing the bounds of node i into three types as follows.

1) Local values. These values includes all the values specific to node i, stored the the worker responsible for node i.
2) Remote values. These are the local values of the other nodes $j \neq i$.
3) Global statistics. These are the values computed using the local values of every node, e.g., the sum of Δv.

Each worker performs the upper bound score computations for its local nodes. It is preferable that the upper bound computation of a node depends only on the node's local values to avoid communication overhead. If the local values are insufficient, the use of global statistics is more preferable than the remote values. While the global statistics have to be computed using the values of every node, it turns out that the computation of the global statistics can be done by first computing local statistics on each worker and then aggregating the local statistics, resulting in low communication overhead. Additionally, a single transfer of the global statistics can be used for computing the bounds for every local node on a worker.

We first consider the naive bound and the global-score-based bound. Both of these bounds rely only on a node's local values and global statistics. Therefore, no further modification is needed for these two bounds.

Next, consider the 1-hop-precise bound (Corollary 2). The term $H\Delta v^{(t)}[i]$ requires Δv of the in-neighbors of node i. This can incur large communication overhead; therefore, we provide a relaxed version of the 1-hop-precise bound, which allows it to be computed more efficiently in a distributed setting.

The modified bound is obtained by using an upper bound for the term $(H\Delta v^{(t)})[i]$, instead of using its exact value. Let us first consider that $(H\Delta v^{(t)})[i]$ is computed by multiplying Δv of each in-neighbor j of node i to $H[i,j]$, and then summing the products from all the in-neighbors. With the goal of using only the local values and global statistics for the bound computation, there are two alternative upper bounds for this amount.

First, we can assume that Δv at every in-neighbors of i is the current maximum Δv, i.e., $||\Delta v^{(t)}||_\infty$. The global statistics $||\Delta v^{(t)}||_\infty$ is then used in place of the in-neighbors' Δv, eliminating the need for the in-neighbor's information. This is given formally in Lemma 2. The proof of the lemma can be found in Appendix E.

Lemma 2. $(H\Delta v^{(t)})[i] <= ||\Delta v^{(t)}||_\infty \sum_j H[i,j]$.

Second, instead of focusing on what each node receives from incoming neighbors, we can compute the maximum possible contribution from each node j to any other node. Then, the sum of the maximum contribution from every node is the upper bound of how much a node can receive from the other nodes. The sum of the maximum contribution from every node is considered to be one of the global statistics since it is computed once from the values of every node and then used by every node. Lemma 3 provides a formal statement for this upper bound. The proof of the lemma can be found in Appendix F.

Lemma 3. $(H\Delta v^{(t)})[i] <= \sum_j ||H_{*,j}||_\infty \Delta v^{(t)}[j]$, *where*

$H_{*,j}$ is the column j of matrix H.

Using the above lemmas, the term $H\Delta v^{(t)}[i]$ in the 1-hop-precise bound is replaced with the minimum between the two of its upper bounds. We refer to the resulting bound as the *1-hop-approx bound*.

C. Distributed Top-k Emergence Test

The top-k emergence test works by checking the number of candidate nodes periodically. In order to determine the candidates, each worker needs the threshold score and a set of global statistics used for computing the upper bound scores of its local nodes. All of the global statistics used in the proposed bounds can be computed in a distributed setting efficiently. The master only has to aggregate the local statistics from the workers and distribute the globally aggregated values to the workers. In the following, we describe the roles of the master and the workers in each step of the emergence test in details.

1) The master broadcasts the PREPARE message to the workers.
2) Upon receiving the PREPARE message, each worker pauses the update of the nodes and replies to the master with the READY message.
3) When the master receives the READY message from every worker, it broadcasts the START message.
4) Upon receiving the START message, each worker computes the local $\sum_j ||H_{*,j}||_\infty \Delta v^{(t)}[j]$, $||\Delta v^{(t)}||_\infty$, $||\Delta v^{(t)}||_1$, and the k highest current $v[i]$ from its local nodes and sends the values to the master.
5) The master aggregates the local values and broadcasts the global $\sum_j ||H_{*,j}||_\infty \Delta v^{(t)}[j]$, $||\Delta v^{(t)}||_\infty$, $||\Delta v^{(t)}||_1$, and the k^{th} highest score (threshold score) to the workers.
6) When each worker receives the global statistics, it determines the candidacy of its local nodes and reports the number of candidates back to the master. Then, it resumes the computation.
7) The master computes the total number of candidates and determines the termination.

The first three steps are needed to make sure that the local statistics at every worker are computed from the snapshot of the values at the same time. This is achieved by making all the workers pause the computation before starting to compute the local statistics. Step 4 and 5 involve computing the global statistics. Finally, step 6 and 7 compute the number of candidates to determine the emergence of the top-k nodes.

From these steps, some optimization can be applied to speed up the process. First, each worker should maintain the set of the current local candidate nodes, initially containing all the local nodes. Each time an emergence test is performed, the worker only needs to check the candidate status of the current candidate nodes. A node that fails the candidate test can be permanently removed from the candidate set. Since the candidate set gets smaller as the computation continues, the work needed for checking the candidates in the later emergence tests is reduced. Second, consider the fact that the master only needs to know whether the number of total candidates exceeds $\bar{k}$ to determine the termination, but not the exact number of candidates. In step 6, as soon as the worker finds that its number of local candidates exceeds $\bar{k}$, it can report the number of candidates as $\bar{k}+1$ without having to check all the nodes.

D. Complexity Analysis of Top-k Emergence Test

Here we discuss the complexity of the work introduced by using the top-k emergence test in comparison to performing a traditional convergence check. Let the number of workers in the system be w. Each worker is responsible for $|V|/w$ nodes. The traditional convergence check requires computing the sum of Δv of every node ($||\Delta v||_1$). This is executed by having each worker compute the sum of Δv of its local nodes and send the local sum to the master. The workload at the worker is $O(|V|/w)$ and the workload at the master is $O(w)$ for aggregating the local sums.

In a top-k emergence test, the first phrase is to compute all the global statistics needed. All the global values except the k^{th} highest current $v[i]$ (threshold score) can be computed in the same way as in the traditional convergence check. To compute the k^{th} highest $v[i]$, each worker first computes its local k highest $v[i]$'s. A QuickSelect algorithm can be used for this purpose, which requires an average work of $O(|V|/w)$ at each worker. Then the master can select the k^{th} highest values among the highest k values sent from every worker. This requires $O(kw)$ on an average case. The second phrase is to prune the candidates, in which each worker scans its local nodes and checks their candidate status. The work at each worker is $O(|V|/w)$.

In summary, the total work for performing each top-k emergence test is $O(kw)+O(|V|/w)$. The workload of $O(kw)$ at the master is an addition to the workload of the traditional convergence test. In practice $kw << |V|/w$; therefore, the work is dominated by the computation at the workers.

VI. Evaluation

We present the evaluation of our approach in this section. We compare the performance with a baseline approach and evaluate the scalability and accuracy of the results obtained from our approach.

A. Preliminary

First, we review the approaches in our evaluation and describe our experimental setup, the dataset, the performance metric, and the preprocessing in our experiments.

1) Baseline and evaluated approaches: We implement the proposed top-k emergence detection by modifying Maiter [7], a framework for distributed asynchronous accumulative computation. Maiter originally determines the termination of a computation by checking $||v||_1$ or $||\Delta v||_1$ against a predefined threshold value. We replace the termination check with the top-k emergence test as described in Section V-C.

In Section IV and V we propose a lower bound and three upper bounds of node scores for using with the top-k emergence detection. The three upper bounds are the naive bound,

the 1-hop-approx bound, and the global-score-based bound. We implement the top-k emergence test using each of these upper bounds to compare their performance. Additionally, we implement a combination bound which selects the minimum between the 1-hop-approx bound and the global-score-based bound, referred to as the combined bound. We refer to each version of the implementation as *M-naive, M-1hop, M-global,* and *M-combine*, respectively.

The baseline approach we compare with is the approach based on the Basic Push algorithm (BPA) proposed by Gupta et al. in [5] to compute the top-k items for Personalized PageRank. The approach is proposed in a context of a single machine computation. It uses the BPA algorithm to compute node scores and uses bounds to detect the top-k items' emergence. In contrast to Maiter, the computation is performed by updating the node with the maximum Δv one at a time. The upper bound used is different from ours. The proposed upper bound of node i is $v^{(t)}[i] + (1-d)||\Delta v^{(t)}||_\infty + \frac{d}{1-d}||\Delta v^{(t)}||_1$, where d is a damping factor in Personalized PageRank. (Due to the difference in the definition of values, we have converted the bound to its equivalent form in terms of $v^{(t)}$ and $\Delta v^{(t)}$.) We refer to this approach as *BPA*.

2) Experimental setup: Our experiments are performed on a 4-machine local cluster and Amazon EC2 cloud. Two machines in the local cluster have Intel Xeon E5607 2.27GHz CPUs and the other two machines have Intel Xeon E5504 2.00GHz CPUs. Each machine has 4GB memory. The machines are connected via a switch having a bandwidth of 1Gbps. The experiments on Amazon EC2 utilize the medium instances, each of which having 2 EC2 compute units and 3.75 GB memory. The number of instances used ranges from 20 to 70. Our configuration runs one worker process on each machine/instance. Nodes in an input graph are partitioned among workers using a hash-based partitioning.

3) Dataset: Our dataset consists of real graphs from different domains and synthetic graphs. The summary statistics of the graphs in our dataset are shown in Table III. The real graphs are unweighted graphs, which are used with Personalized PageRank. For Adsorption, which runs on weighted graphs, we generate a 2-million-node synthetic graph with a log-normal in-degree distribution with the parameters $\mu = -0.5$ and $\sigma = 2.3$, and the float weights for the edges are generated with the parameters $\mu = 0.32$ and $\sigma = 0.8$. The number of edges in the graph is 18 millions.

Additionally, we generate synthetic graphs with different number of nodes to evaluate the scalability of our approach. The synthetic graphs are generated with a log-normal in-degree distribution with the parameters $\mu = -0.5$ and $\sigma = 2.3$. The algorithm and parameters used for graph generation are the same as in [7], in which the parameters are extracted from real graphs. The size of the synthetic graphs ranges from 100 million nodes to 500 million nodes. In all the synthetic graphs, the number of edges is approximately 8 to 9 times the number of nodes.

The queries used in our experiment are generated randomly. For each graph, we generate 30 queries containing 10 items ($|S| = 10$) and 30 queries containing 100 items ($|S| = 100$).

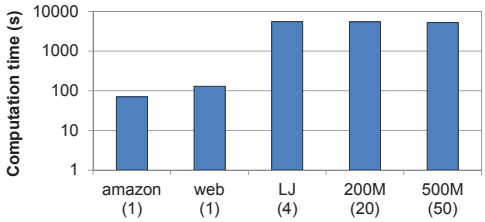

Fig. 1: Global score computation time. The numbers of workers used are shown in the parentheses.

TABLE III: Dataset

Graph	Nodes	Edges
Amazon co-purchasing network [8]	$400K$	$3.3M$
Web graph [9]	$870K$	$5.1M$
Live journal online social network [10]	$4.8M$	$68M$
Synthetic graphs	2M to 500M	18M to 4.5B

4) Performance metric: The performance metric used in our evaluation is the computation time until the emergence of the top-k items is detected. We refer to this time as *detection time*. Since different queries have varying detection time, the average detection time from the queries of the same size is used as representative detection time.

5) Graph preprocessing: Each graph in our experiment is preprocessed to compute the information needed to compute the upper bounds. For the 1-hop-approx bound, the total incoming influence for every node i ($\sum_j H[i,j]$) and the maximum outgoing influence ($||H_{*,j}||_\infty$) for every node j are computed. These can be obtained quickly by scanning the graphs once. To compute the global scores for the global-score-based bound, the original Maiter framework is used. Examples of the computation time along with the number of workers used in the computation are shown in Figure 1.

B. Performance Comparison

Our performance evaluation was performed on a 4-machine local cluster. For this experiment, $\bar{k}$ was set to be equal to k to obtain the exact result.

First, we compare the performance when different upper bounds were used. Figure 2 shows the detection time of the top-k items for Personalized PageRank and Adsorption when $|S| = 10$ and $k = 10$. The quality of the bounds are reflected in the results. In comparison to M-naive, M-1hop does not achieve much improvement in detection time, and sometimes it can perform slightly worse due to the increased overhead. In contrast, M-global has much smaller detection time. M-global runs 4 to 22 times faster than M-naive. This means the global-score-based bound is tighter than the other bounds. The performance of M-combine is slightly better than M-global, which follows from the fact that it selects the tighter bound between the 1-hop-approx bound and the global-score-based bound.

The performance comparison to BPA is shown in Figure 3. Since BPA is proposed in a single machine setting, this

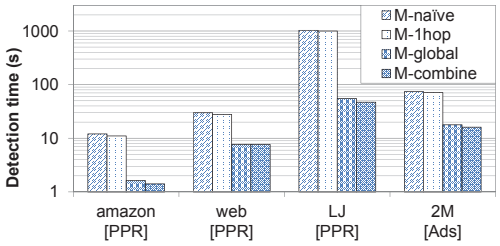

Fig. 2: Detection time on a 4-machine cluster

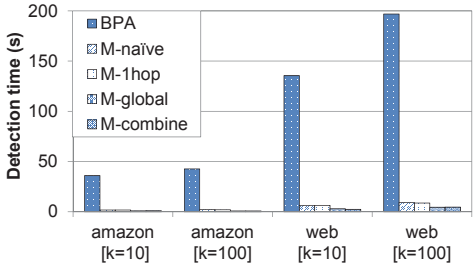

Fig. 3: Performance comparison with baseline approaches on a single machine

experiment was ran on a single machine and only the smaller graphs were used. It can be seen that our approach performs better than BPA. M-combine can achieve a speedup of 20 to 50 times over the BPA approach. There are two main reasons for the performance improvement. First, the BPA approach needs to select the node with the maximum Δv one at a time, which requires using a heap. The overhead for maintaining the heap is high since Δv of nodes keeps changing. This is especially costly when an input graph is very large. Second, the bound used in the BPA approach is slightly looser than our 1-hop-approx bound, which performs worse than the global-score-based bound as shown in the result above. In the following evaluation, we focus on M-combine since it yields the best performance among all the approaches.

C. Effect of $\bar{k}$ on Detection Time

The parameter $\bar{k}$ controls the maximum number of remaining candidate nodes allowed when the computation is terminated. A larger $\bar{k}$ helps to decrease the detection time in exchange for a less precise output, i.e., the output set is larger then k. Here we study how setting $\bar{k}$ to be larger than k affects the detection time. We set k to be 10, 50 and 100, and compare the detection time when $\bar{k} = k$ and $\bar{k} = k + 10$. Figure 4 shows the comparison of detection time for the two cases. The result shows that setting $\bar{k}$ larger than k can reduce the detection time significantly. We observe that some queries benefit more from setting $\bar{k} > k$. For these queries, there are nodes having scores very close to the scores of the real top-k nodes. These queries need more computation to determine the real top-k nodes. By setting $\bar{k} > k$, some of these nodes are allowed to be in the output set, so the computation can be terminated sooner. The speedup depends on the input graphs and k, ranging from 1.6 to 3.6 times faster.

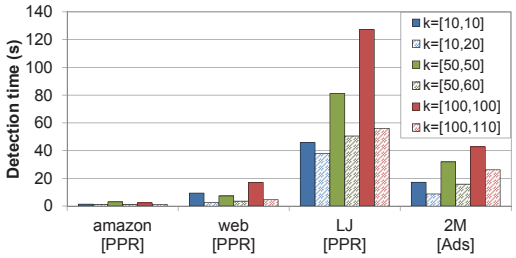

Fig. 4: Effect of $\bar{k}$ on detection time

D. Accuracy when $\bar{k} > k$

Setting $\bar{k} > k$ can result in an output set larger than the target k. While some applications are flexible in the output size, but there are some applications that strictly need a fixed number of k items. For example, in a video recommendation system, only a limited number of recommended videos can be shown to a user at a time. Here we examine whether we can leverage the benefit of setting $\bar{k}$ to be larger than k to decrease the detection time, while obtaining exactly k nodes in the output set with good precision. To select k nodes out of an output set, the most up-to-date scores of nodes at the time when the emergence detection is successful can be used. We evaluate whether the k nodes selected this way are the real top-k nodes.

In this experiment, we set $k = 100$ and set $\bar{k}$ to 120, 140, 160, 180 and 200. For each query, once the computation stops running, we sort the nodes by their most recent scores and obtain the set of top-k nodes, K_{out}. To obtain the real top-k nodes for each query as ground truth, denoted by K_{real}, we run the computation until $||\Delta v||_1 < 10^{10}$ and select the top-k nodes based on the final scores. The precision of K_{out} is computed as $|K_{out} \cap K_{real}|/|K_{out}|$. For all the values of $\bar{k}$ used, the precision obtained is 0.99 for all the cases. This suggests that in practice, the non-converged scores can be used to trim the output set to the desired size, while still achieving high accuracy.

E. Effectiveness of Prioritized Update Scheduling

To measure how much prioritized update scheduling affects the performance, we compare the detection time when the round-robin scheduling and the prioritized scheduling are used. The Maiter framework supports prioritized update scheduling by allowing assignment of each node's priority. Scheduling is performed in rounds at each worker. In each round, the top ρ fraction of nodes with the highest priority are scheduled for the update, where ρ is the computation parameter. When $\rho = 1$, the scheduling is performed in a round-robin fashion.

For prioritized update scheduling, we set ρ to 0.01. Each node is assigned a priority equal to its current Δv as discussed in Section IV-C. Figure 5 shows the detection time of the two cases. With prioritization, the detection time is significantly smaller than when nodes are updated in a round-robin manner. The results confirm that prioritized update scheduling improves the rate at which the bounds approach the real scores.

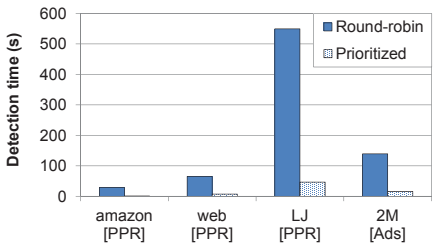

Fig. 5: Performance comparison between prioritized and round-robin scheduling

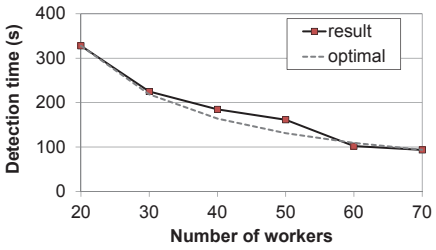

Fig. 6: Scalability with the input size

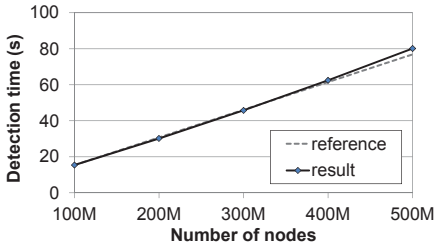

Fig. 7: Scalability with the number of workers ($|S| = 100$)

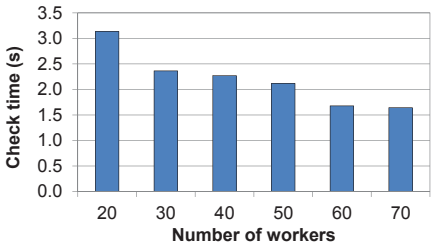

Fig. 8: Emergence check time vs. the number of workers

From our results, prioritized scheduling can yield the speedup of 3 to 6 times over round-robin scheduling.

F. Scalability

Large graphs are very common in current real world data, making scalability a very important property. In this section, we evaluate the scalability of our approach.

1) Scalability with input size: To study the scalability with the input size, we measure the detection time on the synthetic graphs of varying sizes, ranging from 100 to 500 millions nodes using 50 instances on Amazon EC2 cloud. In each graph, the number of edges are approximately 9 times the number of nodes. We set $|S| = 10$, $k = 10$ and $\bar{k} = 20$. Figure 6 shows the scaling performance of our approach with different input sizes. The figure also shows a linear scaling reference line having a constant factor of 1. It can be seen that the detection time grows almost linearly with the size of the graphs. This result also demonstrates that our approach can be applied to a large graph with billion edges. For the 500-million node graph having 4.5 billion edges, the average detection time is 83 seconds when $|S| = 10$, and 400 seconds when $|S|$ = 100.

2) Scalability with the number of workers: In the ideal situation, when the number of processors used is w, the running time should be T/w, where T is the running time when a single processor is used. However, in practice this is not the case since there are overheads introduced when multiple workers are used such as communication overhead. We test the scalability of our approach when different numbers of machines are used. In this experiment, we run the computation on Amazon EC2 cloud. The input graph is a 200-million-node graph having 1.8 billion edges. The number of workers is varied from 20 to 70 workers.

Figure 7 shows the average detection time for each number of workers. The ideal detection times computed based on the running time when the number of workers is 20 are also shown in the figures as a reference. It can be seen that using more machines decreases the detection time, and the measured detection time is not far from the ideal. Note that there is a case that our result is a little better than the ideal running time. This is possible as the ideal running time is computed based on the running time of 20 workers and Maiter's implementation has a nondeterministic factor introduced by the sampling-based method used in selecting the high priority nodes [7].

Additionally, we measure the average time used for each round of the emergence test as the number of workers increases as shown in Figure 8. The result agrees with our analysis in Section V-D. Since the work for the emergence test is dominated by the candidate status check and local statistics computation at each worker, with the complexity of $O(|V|/w)$, we can see that the average test time decreases as the number of workers increases.

VII. Related Work

Path-based relevance metrics have been demonstrated to be effective in quantifying the relevance between items and able to provide satisfactory performance in several applications, such as recommendation system [2], [11], [12], link prediction [13], [14], and word sense disambiguation [15].

Despite their effectiveness, computing path-based relevance scores can be time-consuming. Among the family of path-based relevance metrics, Personalized PageRank has received the most attention from research community. Several existing pieces of work offer a solution for accelerating Personalized PageRank computation. For example, techniques proposed by Forgaras et al. [16] and Bahmani et al. [17] compute an approximation of Personalized PageRank scores based on stored short random walks. An approach proposed by Jeh and Widom in [3] stores partial Personalized PageRank scores of a subset of nodes, referred to a hub set, allowing the

reconstruction of the full scores to be performed online faster. This approach provides accurate scores only when the query nodes belong to the hub set. A few other approaches are based on approximation of the input graph [18], [19]. Although these approaches allow the Personalized PageRank scores to be computed faster, it is not clear how the approximation affects the accuracy of the results when the scores are used for top-k queries.

A few pieces of work focus on obtaining the top-k items for Personalized PageRank. The work from Fujiwara et al. focuses on obtaining the exact top-k nodes for a random walk with restart, which is equivalent to Personalized PageRank with a single query node [20]. Following this work, they proposed an approach for Personalized PageRank, in which there can be multiple query nodes [21]. Both of these work utilize node permutation and matrix decomposition to decrease the space needed to store the precomputed scores to a feasible size. The scores can be obtained fast because there is no iterative computation. In their approaches, to obtain the top-k nodes, the scores of nodes are computed one by one. They propose a pruning method so that the computation can be stopped when reaching the nodes that cannot belong to the top-k set, which greatly helps to speed up the computation. Another work that focuses on top-k queries is from Gupta et al. [5]. In their approach, top-k emergence test is used with the Basic-Push algorithm [22] to allow early termination of computing Personalized PageRank scores. Their emergence test criteria and the bounds used are different from the ones we proposed. The recent work from Fang et al. [23] proposed a new graph proximity metric based on round trip paths between nodes, in which path scores in the forward and backward trips are computed like in Personalized PageRank. They proposed a computation framework using the score bounds along with a distributed solution for scalability.

These existing works provide efficient solutions for obtaining the top-k items for Personalized PageRank and its variants. However, they focus only on a specific path-based metric and mostly address the computation in a single machine setting. While the work from Fang et al. provided a distributed solution, the computation is performed on a single machine and the other machines acts as a distributed graph storage. This solution may work well with a small query set, but some applications with larger graphs and larger query sets can benefit more from using multiple processors. In this work, our goal is to provide a generic and scalable approach that can be applied to speed up the computation of the top-k queries for any path-based metrics. Our distributed platform provides good scalability, allowing computation on large graphs with billion of edges.

There are several frameworks proposed for computing iterative algorithms efficiently in a distributed environment, which allows the computation to scale for very large input, such as Haloop [24], GraphLab [25], and Maiter [26]. These frameworks can be used to compute path-based relevance scores on large graphs efficiently. The recently proposed Maiter framework uses asynchronous accumulative updates to achieve fast convergence for iterative algorithms. In this paper, we take advantage of the efficient asynchronous accumulative computation model to compute path-based scores. Coupling with our proposed top-k emergence test, designed for top-k query computation, the system has demonstrated good performance in computing the top-k items for path-based metrics.

VIII. Conclusion

In this paper, we propose an approach to accelerate the computation time for the top-k query when path-based metrics are used. The approach works in conjunction with asynchronous accumulative computation by detecting the emergence of the top-k items during the computation. This eliminates the need to compute the converged scores of items, which results in faster computation time. The proposed approach is generalized for a family of path-based metrics and can be applied to both a single machine computation and a distributed computation, allowing it to support a very large input graph containing several hundreds of millions of nodes. Our experimental results show that the proposed method can significantly decrease the response time for the top-k queries and provide high scalability.

Acknowledgment

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work is supported by the NFS under Grant No. CNS-1217284 and CCF-1018114.

References

[1] T. Haveliwala, "Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 4, pp. 784–796, 2003.

[2] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for youtube: taking random walks through the view graph," in *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, pp. 895–904.

[3] G. Jeh and J. Widom, "Scaling personalized web search," in *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003, pp. 271–279.

[4] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[5] M. Gupta, A. Pathak, and S. Chakrabarti, "Fast algorithms for topk personalized pagerank queries," in *Proceeding of the 17th international conference on World Wide Web*. ACM, 2008, pp. 1225–1226.

[6] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: a distributed framework for prioritized iterative computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 13.

[7] "Maiter: A message-passing distributed framework for accumulative iterative computation," http://rio.ecs.umass.edu/~yzhang/maiter-full.pdf.

[8] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Trans. Web*, vol. 1, no. 1, May 2007. [Online]. Available: http://doi.acm.org/10.1145/1232722.1232727

[9] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *CoRR*, vol. abs/0810.1355, 2008.

[10] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 44–54. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150412

[11] M. Gori and A. Pucci, "Itemrank: a random-walk based scoring algorithm for recommender engines," in *Proceedings of the 20th international joint conference on Artifical intelligence*. Morgan Kaufmann Publishers Inc., 2007, pp. 2766–2771.

[12] 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*, ser. SIGIR '09. New York, NY, USA: ACM, 2009, pp. 195–202.

[13] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American society for information science and technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[14] Z. Yin, M. Gupta, T. Weninger, and J. Han, "A unified framework for link recommendation using random walks," in *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*. IEEE, 2010, pp. 152–159.

[15] E. Agirre and A. Soroa, "Personalizing pagerank for word sense disambiguation," in *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, 2009, pp. 33–41.

[16] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós, "Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments," *Internet Mathematics*, vol. 2, no. 3, pp. 333–358, 2005.

[17] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized pagerank," *Proceedings of the VLDB Endowment*, vol. 4, no. 3, pp. 173–184, 2010.

[18] H. Tong, C. Faloutsos, and J.-Y. Pan, "Fast random walk with restart and its applications," in *Proceedings of the Sixth International Conference on Data Mining*, ser. ICDM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 613–622.

[19] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos, "Neighborhood formation and anomaly detection in bipartite graphs," in *Data Mining, Fifth IEEE International Conference on*, nov. 2005, p. 8 pp.

[20] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa, "Fast and exact top-k search for random walk with restart," *Proc. VLDB Endow.*, vol. 5, no. 5, pp. 442–453, Jan. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2140436.2140441

[21] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka, "Efficient personalized pagerank with accuracy assurance," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 15–23. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339538

[22] P. Berkhin, "Bookmark-coloring algorithm for personalized pagerank computing," *Internet Mathematics*, vol. 3, no. 1, pp. 41–62, 2006.

[23] Y. Fang, K. C.-C. Chang, and H. W. LAUW, "Roundtriprank: Graph-based proximity with importance and specificity." IEEE International Conference on Data Engineering (ICDE), 2013.

[24] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[25] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[26] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Accelerate large-scale iterative computation through asynchronous accumulative updates," in *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*. ACM, 2012, pp. 13–22.

APPENDIX

A. Proof of Lemma 1

Proof: To derive $d^{(t)}[i]$, suppose that after time t the nodes are updated at the same time in iterations. According to Equation 2, in the k^{th} iteration node i will receive $(H^k \Delta v^{(t)})[i]$. At convergence, node i will receive a total of $\sum_{k=0}^{\infty}(H^k \Delta v^{(t)})[i]$. ■

B. Proof of Theorem 1

Proof: The derivation of the first part is as follows.

$$\begin{aligned} v^{(\infty)}[i] &= v^{(t)}[i] + d^{(t)}[i] \\ &<= v^{(t)}[i] + ||d^{(t)}||_\infty \\ &= v^{(t)}[i] + ||\sum_{k=0}^{\infty} H^k \Delta v^{(t)}||_\infty \\ &<= v^{(t)}[i] + \sum_{k=0}^{\infty} ||H||_\infty^k ||\Delta v^{(t)}||_\infty \\ &= v^{(t)}[i] + \frac{||\Delta v^{(t)}||_\infty}{1 - ||H||_\infty} \text{ if } ||H||_\infty < 1 \end{aligned}$$

For the second part, the derivation starts from $v^{(\infty)}[i] <= v^{(t)}[i] + ||d^{(t)}||_1$ and continues similarly to the above derivation using the L1 norm instead of the infinity norm. ■

C. Proof of Theroem 2

Proof: The derivation of the first part is as follows.

$$\begin{aligned} v^{(\infty)}[i] &= v^{(t)}[i] + \sum_{k=0}^{l}(H^k \Delta v^{(t)})[i] + \sum_{k=l+1}^{\infty}(H^k \Delta v^{(t)})[i] \\ &<= v^{(t)}[i] + \sum_{k=0}^{l}(H^k \Delta v^{(t)})[i] + ||\sum_{k=l+1}^{\infty}(H^k \Delta v^{(t)})[i]||_\infty \\ &<= v^{(t)}[i] + \sum_{k=0}^{l}(H^k \Delta v^{(t)})[i] + \sum_{k=l+1}^{\infty} ||H||_\infty^k ||\Delta v^{(t)}||_\infty \\ &= v^{(t)}[i] + \sum_{k=0}^{l}(H^k \Delta v^{(t)})[i] + ||\Delta v^{(t)}||_\infty \frac{||H||_\infty^{l+1}}{1 - ||H||_\infty} \\ &\text{if } ||H||_\infty < 1 \end{aligned}$$

The second part of the theorem can be derived similarly using the L1 norm instead of the infinity norm. ■

D. Proof of Theroem 3

Proof: From Lemma 1,

$$\begin{aligned} v^{(\infty)}[i] &= v^{(t)}[i] + \sum_{k=0}^{\infty}(H^k \Delta v^{(t)})[i] \\ &<= v^{(t)}[i] + \Delta v^{(t)}[i] + (\sum_{k=1}^{\infty} H^k \Delta v^{(t)})[i] \\ &<= v^{(t)}[i] + \Delta v^{(t)}[i] + (\sum_{k=1}^{\infty} H^k ||\Delta v^{(t)}||_\infty \vec{1})[i] \\ &= v^{(t)}[i] + \Delta v^{(t)}[i] + ||\Delta v^{(t)}||_\infty (\sum_{k=1}^{\infty} H^k \vec{1})[i] \\ &= v^{(t)}[i] + \Delta v^{(t)}[i] + ||\Delta v^{(t)}||_\infty (v_{global}[i] - 1) \end{aligned}$$ ■

E. Proof of Lemma 2

Proof:

$$(H\Delta v^{(t)})[i] = \sum_j H[i,j]\Delta v^{(t)}[j] <= \sum_j H[i,j] ||\Delta v^{(t)}||_\infty$$ ■

F. Proof of Lemma 3

Proof:

$$(H\Delta v^{(t)})[i] = \sum_j H[i,j]\Delta v^{(t)}[j] <= \sum_j ||H_{*,j}||_\infty \Delta v^{(t)}[j]$$ ■