A Distributed Approach for Top-k Star Queries on Massive Information Networks

Jiahui Jin[†‡], Samamon Khemmarat[‡], Lixin Gao[‡], Junzhou Luo[†]

[†]*School of Computer Science and Engineering, Southeast University, China*
[‡]*Department of Electrical and Computer Engineering, University of Massachusetts Amherst, USA*
{*jhjin, jluo*}*@seu.edu.cn;* {*khemmarat, lgao*}*@ecs.umass.edu*

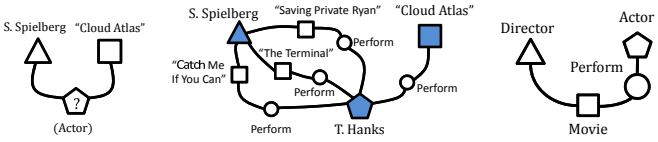
***Abstract*—Massive information networks, such as the knowledge graph by Google, contain billions of labeled entities. Star queries, which aim to identify an entity, given a set of related entities, are common on such networks. Answering star queries can be modeled as a graph pattern matching problem. Traditional approaches apply graph indices to accelerate the query processing. Unfortunately, it is so costly that it is nearly infeasible to build indices on billion node graphs since the time or storage complexity of most indexing techniques is super-linear to the graph size. In this paper, we propose an algorithm to identify the top-k best answers for a star query. Instead of using expensive indices, our algorithm utilizes novel bounding techniques to derive the top-k best answers efficiently. Further, the algorithm can be implemented in a distributed manner scaling to billions of entities and hundreds of machines. We demonstrate the effectiveness and the efficiency of our approach through a series of experiments on real-world information networks.**

***Index Terms*—Top-k Query, Distributed System, Star Query, Information Network, Billion-Node Graph, Big Data**

I. Introduction

Massive information networks, such as Linked Open Data [1] and Freebase [2], contain billions of labeled entities. For example, Linked Open Data, a collection of interrelated datasets on the Web, contains more than 0.52 billion entities (nodes) and 1.77 billion RDF triples (edges). Subgraph queries are commonly used to discover information in networks. However, in information networks the traditional solutions for subgraph isomorphism may not be readily applied to answering queries, since these information networks, consisting of community-curated databases of people, places, and things, can be incomplete. Further, in many cases user queries may not conform to the schemas of the graphs due to the lack of acquaintance with the schema on the user's side. For this purpose, *pattern match queries* have been studied in recent years [3–6].

To accelerate the pattern matching, various indexing techniques were proposed [3, 7–9]. However, these indexing techniques are so costly that they are nearly intractable for billion node graphs, since the time and storage costs of indexing are super-linear to the graph size [10]. In this paper, we propose an efficient distributed approach for answering pattern match queries in the massive information networks without requiring the expensive indices. Specifically, we investigate the problem of identifying the best k answers for a *star query*, a common but important pattern match query.



(a) Star Query (b) Answer (c) Schema

Fig. 1. An Example of Star Query

A star query aims to identify an entity, given a set of related entities. Fig. 1 shows an example of a star query on Freebase, a real-life knowledge graph. Fig. 1(a) shows a query that contains two given entities, *Steven Spielberg* and *"Cloud Atlas"*, and searches for *an actor in the movie "Cloud Atlas" who has worked with the director Steven Spielberg*. In Fig. 1(b) *Tom Hanks* is an answer to the query because *Tom Hanks* acted in *"Cloud Atlas"* and worked with *Steven Spielberg* in three movies. Note that the solutions for subgraph isomorphism cannot find the answer because the query directly connects the actor node to the director and the movie nodes, which does not conform to the schema of Freebase (Fig. 1(c)).

Our goal is to find the best k answers for a given star query. The answers are ranked by match scores quantifying the answers' relevance to the given entities. Instead of computing exact match scores, our algorithm utilizes a novel bounding technique to accelerate query processing. As a result, our approach can find the top-k answers efficiently without expensive indices. The key contributions of this paper are summarized as follows:

- We propose an approach to answer star queries in noisy and incomplete information networks. The effectiveness of the approach is evaluated using two real-world datasets. Our answers are more accurate than those found by the index-based approaches.
- We present a novel bounding technique to accelerate query processing without using indices. Comparing with computing exact match scores, the bounding technique reduces the running time by more than two orders of magnitudes.
- Our algorithm is implemented in a distributed manner,

allowing it to support graphs with billions of nodes. The algorithm can answer star queries on a one-billion-node graph within 10 seconds when deployed on a 100 virtual-machine cluster.

The rest of this paper is organized as follows. Section II provides the formal definition of the match score and the top-k star query problem. We describe our bound based algorithm in Section III. Section IV presents our distributed solution. In Section V, we present the evaluation of our algorithm. We discuss related work in Section VI and conclude the paper in Section VII.

II. Problem Description

A massive information network is modeled as a *target graph*, denoted by $G(V, E, T)$, where V is a set of nodes, E is a set of undirected edges, and T is a set of node types. We denote the type of node v by *type*(v). For simplicity, there are no weights on the edges, and there is at most one edge between two nodes. However, the proposed approach could be extended for graphs with weighted edges.

A *star query* describes a *specific node* set and the type of the query node. For example, in Fig. 1 *Cloud Atlas* and *Steven Spielberg* are the specific nodes and *actor* is the type of the query node. We denote a star query by $Q(V_Q^S, \tau)$, where V_Q^S is the specific node set, and $\tau \in T$ is the node type. Since the specific nodes correspond to the entities given by users, for each $v^s \in V_Q^S$, there is *an anchor node* $\phi(v^s)$ in the target graph which v^s can be mapped to. Since entities usually have unique names, we assume $\phi(v^s)$ is unique, given a specific node v^s.

An answer to the star query is referred to as a *match*. Given a target graph $G(V, E, T)$ and a star query $Q(V_Q^S, \tau)$, an *exact match* of Q in G is a node $u \in V$ that satisfies the following two constraints: (1) *type*$(u) = \tau$ and (2) $\forall v^s \in V_Q^S$, edge $(u, \phi(v^s)) \in E$. Traditional query processing techniques based on subgraph isomorphism focus on how to find exact matches. However, *approximate matches* are more suitable for answering real-life queries because target graphs are usually incomplete. Approximate matching allows edge mismatch such that constraint (2) is relaxed.

In this work, we consider *the top-k star query problem*, which returns the best k approximate matches according to a *match score*. The match score measures the closeness between a match and the anchor nodes. An approximate match that has a higher match score should be closer to all the anchor nodes.

The closeness between a pair of nodes should have the following properties. First, if the distance between two nodes, *i.e.,* the length of the shortest path, is shorter, these nodes are closer. Second, if two pairs of nodes have the same distance, the node pair with more shortest paths are closer. Therefore, our *closeness score* function takes into account the length and the number of the shortest paths. The closeness score between node u and v, denoted by $\varphi(u, v)$, is defined by:

$$\varphi(u,v) = \begin{cases} 1, & u = v \\ \min\{n_{u,v}\alpha^{l_{u,v}}, N\alpha^{l_{u,v}}\}, & \text{otherwise} \end{cases} \quad (1)$$

where $l_{u,v}$ and $n_{u,v}$ are the length and the number of the shortest paths between u and v; α is a constant having the value between 0 and 1. To ensure that the pairs of nodes having shorter distance have the higher scores, we introduce a constant N. When $N < \frac{1}{\alpha}$, $\min\{n_{u,v}\alpha^{l_{u,v}}, N\alpha^{l_{u,v}}\}$ is less than $\alpha^{l_{u,v}-1}$. Therefore, we can guarantee that the closeness score between l-hop nodes is always greater than that between $(l+1)$-hop nodes. To extend our approach to the graphs with weighted edges, the graphs can first be converted by replacing each weighted edge with multiple-hop paths according to its weight. Then, our closeness score function can be applied to the converted graphs.

By summing up the closeness scores between a match and all the anchor nodes, the match score of a match is defined as follows.

Match score. Given a target graph G and a star query Q, the match score of Q's match u is defined as:

$$\mathcal{S}(u) = \sum_{v^s \in V_Q^S} \varphi(u, \phi(v^s)) \quad (2)$$

Our match score has the following properties: (1) For any exact match u, $\mathcal{S}(u) = \alpha|V_Q^S|$; (2) For any match u that is inexact, $\mathcal{S}(u) < \alpha|V_Q^S|$.

Problem definition. Given a target graph G and a star query Q, the top-k star query problem is to identify k matches of Q that have the largest k match scores.

Table I lists the notations that are frequently used in this paper.

Notation	Description
$G(V, E, T)$	a data graph
$Q(V_Q^S, \tau)$	a query graph
$\phi(v^s)$	the anchor node of specific node v^s
$\varphi(u, v)$	the closeness score between u and v
$\mathcal{S}(u)$	the match score of u
Ψ	a set of candidate matches (defined in Section III-B)
$\mathcal{R}^s$	the closeness propagation starting from node s (defined in Section III-C)

TABLE I
Frequently Used Notations

III. The Algorithm For Identifying Top-k Matches

In this section, we describe our approach for identifying the top-k matches for a given star query. A straightforward approach is to rank the matches by their match scores. To accelerate the query processing, we compute the bounds of the match scores instead of the exact match scores and use the bounds to identify the top-k matches.

A. Identifying Top-k Matches with Bounds

We illustrate how our top-k identification works in the following example.

Example 1. *In Fig. 2, v_3 and v_4 are the top-2 matches of*

Q because v_3 and v_4 are closer to anchor node v_1 than v_5. To find v_3 and v_4, we perform breadth-first search from v_1 and v_2. When the search starting form v_1 reaches v_3 and v_4, we know that $\varphi(v_1, v_3)$ and $\varphi(v_1, v_4)$ are larger than $\varphi(v_1, v_5)$. Therefore, the top-2 matches can be identified without computing the exact closeness score between v_1 and v_5.

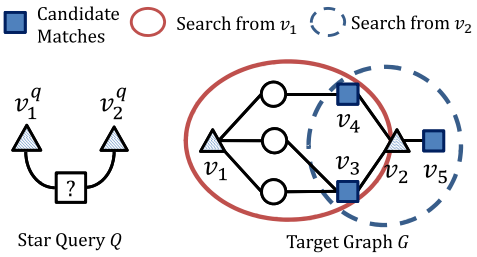


Fig. 2. Identifying the Top-2 Matches (v_1 and v_2 are the anchor nodes of v_1^q and v_2^q, respectively)

Now we outline our approach for the top-k identification. We maintain a set of candidate matches, the matches that can potentially be the top-k. The set is shrunk by evaluating the bounds of match scores for each candidate match. The bounds are iteratively refined. The answer for the query is found when the number of candidate matches is k. In the rest of this section, we first introduce the top-k emergence test, which is used to check whether the top-k matches are found (Section III-B). Then, the derivation of the bounds of match scores and the proposed algorithm are described (Section III-C).

B. Top-k Emergence Test

The top-k emergence test determines whether the top-k matches can be identified. Assume for now that we have the upper bound and the lower bound of $\mathcal{S}(v)$ for each candidate match v, denoted by $\overline{\mathcal{S}}(v)$ and $\underline{\mathcal{S}}(v)$, respectively. Let Ψ denote the candidate match set. Initially Ψ contains all the nodes that have the same type as the query node.

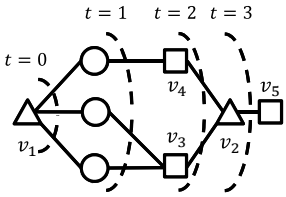
The top-k emergence test consists of two main steps.

- *Step (1)*: We find the k^{th} largest $\underline{\mathcal{S}}$ among the candidates, denoted by $\underline{\mathcal{S}}_k$. $\underline{\mathcal{S}}_k$ is the lower bound score of the top-k matches. In other words, the match score of every top-k match must be greater than or equal to $\underline{\mathcal{S}}_k$.
- *Step (2)*: The candidates with $\overline{\mathcal{S}}$ less than $\underline{\mathcal{S}}_k$ are removed from Ψ because it is certain that they cannot be the top-k matches.

The test is successful if $|\Psi|$ is equal to k. Otherwise, the test fails and the bounds should be further refined. The ideas similar to the top-k emergence test have been studied in [11, 12]. In this paper, our focus is on how to derive the bounds of match scores.

C. Bounds of Match Score

Now we describe how to compute and refine the bounds of match scores, $\overline{\mathcal{S}}$ and $\underline{\mathcal{S}}$. From the definition of the match score



Bounds of $\varphi(v_1, v_3)$

t	$\overline{\varphi}(v_1, v_3)$	$\underline{\varphi}(v_1, v_3)$
0	$N\alpha$	0
1	$N\alpha^2$	0
2	$2\alpha^2$	$2\alpha^2$
3	$2\alpha^2$	$2\alpha^2$

Fig. 3. An Example of Closeness Propagation

in Eq. (2), we have

$$\overline{\mathcal{S}}(u) = \sum_{v^s \in V_Q^S} \overline{\varphi}(\phi(v^s), u) \text{ and } \underline{\mathcal{S}}(u) = \sum_{v^s \in V_Q^S} \underline{\varphi}(\phi(v^s), u) \quad (3)$$

It can be seen that to compute $\overline{\mathcal{S}}$ and $\underline{\mathcal{S}}$ for *every* candidate match v, we need the bounds of closeness scores. To be precise, for each specific node v^s, we need $\overline{\varphi}(\phi(v^s), u)$ and $\underline{\varphi}(\phi(v^s), u)$ for every candidate match $u \in \Psi$. Next, we show how to refine the bounds of the closeness scores.

Bound refinement by closeness propagation. We refine the bounds of closeness scores with *closeness propagation*. Given a source node, *i.e.,* an anchor node, the *propagation* iteratively refines the bounds of the closeness scores between the source node and the other nodes. We denote the propagation starting from source node s by $\mathcal{R}^s$. The execution of $\mathcal{R}^s$ is as follows.

We denote the upper bound and the lower bound of the closeness score between source node s and node v in iteration t by $\overline{\varphi}^t(s, v)$ and $\underline{\varphi}^t(s, v)$, respectively. The initial bounds (for $t = 0$) of the source node can be derived directly from the definition of the closeness score.

$$\overline{\varphi}^0(s, s) = 1 \text{ and } \underline{\varphi}^0(s, s) = 1 \quad (4)$$

For the other nodes $v \neq s$, we know that v is at least 1-hop away from s. Therefore, we have

$$\overline{\varphi}^0(s, v) = N\alpha \text{ and } \underline{\varphi}^0(s, v) = 0 \quad (5)$$

In each iteration, every node v whose lower bound is 0 updates its lower bound using the information from its neighbors. Let $\mathbb{N}(v)$ denote the set of neighbors of node v. In iteration t, the lower bound is computed as follows:

$$\underline{\varphi}^t(s, v) = \begin{cases} \underline{\varphi}^{t-1}(s, v) & \underline{\varphi}^{t-1}(s, v) > 0 \\ \min\{N\alpha^t, & \text{otherwise} \\ \quad \alpha \cdot \sum_{u \in \mathbb{N}(v)} \underline{\varphi}^{t-1}(s, u)\} & \end{cases} \quad (6)$$

For the upper bound, consider that for a node v, if $\underline{\varphi}^t(s, v) = 0$, then v is at least $t + 1$ hops away from s. Based on this observation, the upper bound in iteration t is computed as follows:

$$\overline{\varphi}^t(s, v) = \begin{cases} \underline{\varphi}^t(s, v) & \underline{\varphi}^t(s, v) > 0 \\ N\alpha^{t+1} & \text{otherwise} \end{cases} \quad (7)$$

Example 2. *Fig. 3 shows the propagation starting from v_1. Let us consider the bounds of node v_3. When $t < 2$, $\underline{\varphi}^t(v_1, v_3)$ is 0 and $\overline{\varphi}^t(v_1, v_3)$ decreases from $N\alpha$ to $N\alpha^2$. When $t = 2$, $\underline{\varphi}^t(v_1, v_3)$ is updated to $2\alpha^2$ and $\overline{\varphi}^t(v_1, v_3)$ equals $\underline{\varphi}^t(v_1, v_3)$. When $t > 2$, the bounds will not be changed.*

Coordinating multiple propagations. Since there are $|V_Q^S|$ anchor nodes, we perform $|V_Q^S|$ propagations to compute the match score bounds for the candidate matches. These propagations are performed concurrently. Namely, the propagations perform one iteration in turn. The order of the propagations to be performed is important since it affects the efficiency of the top-k candidate identification, as illustrated in the following example.

Example 3. *Consider the query in Fig. 2. To find the top-2 matches, at least $\mathcal{R}^{v_1}$ and $\mathcal{R}^{v_2}$ should be performed two iterations and one iteration, respectively. In this case, we would know v_3, v_4 and v_5 are 1-hop away from v_2, but v_3 and v_4 are closer to v_1 than v_5. Consider two propagation orders: (1) $\langle \mathcal{R}^{v_1}, \mathcal{R}^{v_1}, \mathcal{R}^{v_2} \rangle$ and (2) $\langle \mathcal{R}^{v_2}, \mathcal{R}^{v_1}, \mathcal{R}^{v_2}, \mathcal{R}^{v_1} \rangle$. The first order is more efficient because it only requires a total of three iterations, as opposed to four iterations in the latter case.*

From the example, after $\mathcal{R}^{v_2}$ is performed for one iteration, v_3, v_4, and v_5 have obtained the closeness scores from v_2. Since $\mathcal{R}^{v_2}$ cannot help us improve the bounds of the match score further, there is no need to continue $\mathcal{R}^{v_2}$. This suggests that to determine the propagation order, we should consider how each of the propagations would help to improve the bounds.

Our heuristic to determine the order among the propagations is as follows. In order to detect the top-k candidates fast, the bounds for the match scores of the candidates, *i.e.,* $\overline{\mathcal{S}}$ and $\underline{\mathcal{S}}$, should approach the real scores as quickly as possible. Therefore, we should perform the propagations that will reduce the gap between $\overline{\mathcal{S}}$ and $\underline{\mathcal{S}}$ of each candidate the most.

For a candidate match u, the gap between $\overline{\mathcal{S}}(u)$ and $\underline{\mathcal{S}}(u)$ is

$$\overline{\mathcal{S}}(u) - \underline{\mathcal{S}}(u) = \sum_{s \in V_Q^S} \{\overline{\varphi}(s,v) - \underline{\varphi}(s,v)\}. \quad (8)$$

Therefore, propagation $\mathcal{R}^s$ contributes $\overline{\varphi}(s,u) - \underline{\varphi}(s,u)$ to the bound gap of v. Because we want to reduce the gaps for all the candidates, the priority of propagation $\mathcal{R}^s$, denoted by $\mathcal{P}(\mathcal{R}^s)$, is computed as follows.

$$\mathcal{P}(\mathcal{R}^s) = \sum_{u \in \Psi} \{\overline{\varphi}(s,u) - \underline{\varphi}(s,u)\} \quad (9)$$

The propagation that has the highest $\mathcal{P}$, meaning it contributes the most to the bound gaps for the candidate matches, is selected to perform the refinement.

Our algorithm for the top-k identification is shown in Algorithm 1. Given a star query (V_Q^S, τ), the initial candidate match set Ψ contains all the nodes whose type is τ (line 1). We iteratively refine the bounds of the match score of each candidate match (line 2-5). At each iteration, we perform one iteration for the propagation that has the highest priority (line 8-10) and update the bounds of match scores (line 11-12). During the bound refinement, the top-k emergence test is executed periodically to perform the termination check. The test updates Ψ according to the bounds (line 14-17). The algorithm stops when there are k candidate matches in Ψ (line 18-21). Finally, the k candidate matches in Ψ are returned as the top-k matches.

Algorithm 1: Top-k Identification

Input: Graph $G(V,E,T)$; Star Query $Q(V_Q^S, \tau)$; Match Number k
Output: Top-k Match Set Ψ

1 $\Psi \leftarrow \{v | type(v) = \tau$ and $v \in V\}$
2 Initialize propagation $\mathcal{R}^s$ for every anchor node s by Eq. (4) and (5)
3 **repeat**
4 Perform *BoundRefinement*
5 **until** *TerminationCheck* returns *True*
6 **return** Ψ
7 **Procedure BoundRefinement**
8 Compute the priority score for every propagation by Eq. (9)
9 Select propagation $\mathcal{R}^s$ that has the highest priority
10 Perform one iteration for $\mathcal{R}^s$
11 Update $\overline{\varphi}(s,v)$ and $\underline{\varphi}(s,v)$ by Eq. (6) and (7)
12 Update $\overline{\mathcal{S}}(u)$ and $\underline{\mathcal{S}}(u)$ for every candidate match by Eq. (3)
13 **Procedure TerminationCheck**
14 Get $\underline{\mathcal{S}}_k$ by computing the k^{th} largest $\underline{\mathcal{S}}(u)$
15 **foreach** u *in* Ψ **do**
16 **if** $\overline{\mathcal{S}}(u) < \underline{\mathcal{S}}_k$ **then**
17 $\Psi \leftarrow \Psi - \{u\}$
18 **if** $|\Psi| = k$ **then**
19 **return** *True*
20 **else**
21 **return** *False*

IV. Design of Distributed System

To support massive information networks, our approach is implemented on a distributed system that contains multiple machines connected via high-speed Ethernet.

A. System Overview

Our system is developed based on Piccolo [13], a programming framework for in-memory graph computation. The system consists of one *master* task and multiple *worker* tasks. The tasks are executed on the computer cluster. Fig. 4 shows the interactions between the tasks when answering a star query. The master coordinates the workers by telling them which propagations should be performed and when the top-k emergence test should be performed. The workers use a *distributed in-memory state table* to store the target graph. They perform the top-k emergence test and the bound refinement in parallel. Next, we describe the distributed in-memory state table, which is used for storing the target graph and the closeness score bounds.

Distributed in-memory state table. The distributed in-memory state table is maintained by the workers. Each worker has a partition of the table. The rows of the table are indexed by the keys of the nodes. Each row contains a node's metadata including the node's type and adjacent neighbors. We use hash-partitioning to assign the nodes to different workers. However, users can implement other state-of-the-art partitioning strategies [10] in our system. We also store the lower bounds of closeness scores in the in-memory state table. Only the non-zero lower bounds are stored. Additionally, each worker maintains the iteration number of each propagation for

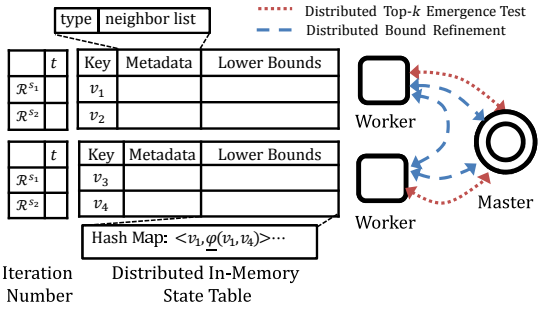


Fig. 4. System Architecture

computing the upper bounds. All the needed upper and lower bounds can be derived from the non-zero lower bounds and the iteration numbers as in Eq. (6) and (7).

Having the distributed in-memory state table, the distributed bound refinement (Section IV-B) and the distributed top-k emergence test (Section IV-C) can be performed in the distributed system.

B. Distributed Bound Refinement

To improve parallelism, the distributed bound refinement performs multiple propagations in one *super-iteration*. During each super-iteration, as shown in Fig. 5, the master (1) aggregates the priority scores of the propagations that are computed by the workers and (2) produces an *execution list* containing the propagations with the highest priorities. The workers (3) fetch the execution list from the master and (4) perform one iteration for each propagation in the list. The workers are synchronized when a super-iteration is finished. There are two main issues to be addressed in the above design: *how to produce an execution list* and *how to perform propagations in a distributed manner*.

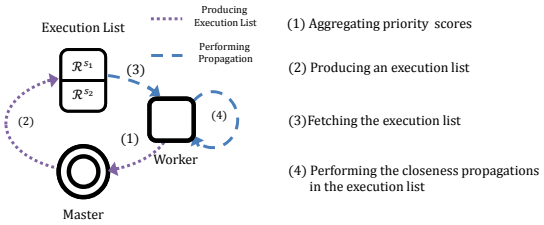


Fig. 5. A Super-Iteration of Distributed Bound Refinement

Producing execution list. The master produces the execution list at the beginning of each super-iteration. To get the execution list, we need to compute the priority score for each propagation. The priority scores are locally computed by the workers and aggregated by the master. Then, the master puts the propagations with the highest priorities into the execution list.

The size of the execution list is critical in determining the efficiency of our approach. A shorter list introduces more super-iterations, while a longer list may include the propagations with low priority. Both of the cases may increase the query's processing time. For determining the size of the execution list, we analyze the running time of each super-iteration. The running time of a super-iteration is $T_{iter} = T_{update} + T_{pri} + T_{syn}$, where T_{update} is the time for updating the closeness score bounds; T_{pri} is the time for computing priority scores; and T_{syn} is the time for worker synchronization. T_{syn} and T_{pri} are static for every super-iteration, while T_{update} is a monotonically increasing function of the number of the visited nodes. When the number of the visited nodes is small, T_{syn} and T_{pri} would dominate T_{iter}. Therefore, we need to select the size of the execution list such that there are sufficiently large number of nodes visited in one super-iteration.

Based on the analysis, we determine the size of the execution list using *frontier counting*. Given a propagation, its *frontier* is the set of nodes whose lower bounds are updated in the latest iteration. A larger frontier indicates that more nodes will be visited in the next iteration. The list size is determined by a predefined threshold, *MaxFrontier*. The propagations are added into the execution list as many as possible until the total frontier size of the propagations in the list is larger than *MaxFrontier*. In our system, we set *MaxFrontier* to $10000 \cdot w$, where w is the number of workers.

Performing propagations. Given an execution list, all the propagations in the list will be performed in the next super-iteration. Each iteration of a propagation $\mathcal{R}^s$ is performed as follows. Let v be a node in $\mathcal{R}^s$'s frontier. The worker containing v sends message $\langle s, \underline{\varphi}(s,v)\rangle$ to the workers containing v's neighbors. When a worker containing v's neighbor, u, receives the message, it updates the closeness score lower bound of u according to the update function in Eq. (6). The iteration of $\mathcal{R}^s$ finishes when all the nodes in $\mathcal{R}^s$'s frontier are processed.

C. Distributed Top-k Emergence Test

The distributed top-k emergence test is based on a divide-and-merge strategy. Each worker performs a local test and obtains a local result. The local results are aggregated on the master. The master announces whether the distributed test is successful. In our system, the test is performed every 0.1 seconds.

There are two steps in the distributed test, which correspond to the two steps in the centralized test. Step (1): find the global k^{th} largest lower bound. Each worker computes the largest k match score lower bounds for the local candidate matches. Then, the workers send the local top-k lower bounds to the master. The master produces the global k^{th} largest lower bound from the local bounds. Step (2): find the qualified candidates. The master sends the global k^{th} largest lower bound to the workers. Each worker evaluates the local candidates' upper bounds and identifies the qualified candidates. The master aggregates the numbers of qualified candidates from all the workers and determines whether the test is successful.

D. Fault Tolerance

In our system, the fault tolerance strategy is implemented by using a set of backup tasks. Each a master/worker task has a backup task that periodically copies the in-memory table

from the master/worker task and replaces the master/worker task when it fails. The copying is performed at the end of each super-iteration. Only the closeness score lower bounds updated in the last super-iteration are copied. When failure in one or more master/worker tasks is detected, the current super-iteration is re-executed, where the backup tasks are activated to perform the computation instead of the faulty tasks.

V. Evaluation

A. Experiment Settings

Environment and settings. We implemented the distributed system in about 5,300 lines of C++ and performed the experiments on two computer clusters, a local cluster and an Amazon EC2 cluster. The local cluster consisted of four machines connected by a 1Gb Ethernet switch. Each machine had 16GB RAM and one 1.86GHz Intel Xeon E5502 CPU with four cores. The Amazon EC2 cluster consisted of 100 m3.large instances, each of which had 7.5GB RAM and two vCPUs. In each experiment, we set N to 9 and α to 0.1.

Graph datasets. Two real-life graphs and several synthetic graphs, listed in Table II, were used in our experiments. (1) *DBLP.* The DBLP graph [14] is a bibliography network containing three types of nodes: author, publication, and venue. (2) *Freebase.* Freebase is a collaborative knowledge base. A subset of Freebase containing movie information was used in our experiments. The movie dataset has five types of nodes: movie, director, actor, perform, and role. (3) *Synthetic Graphs.* Synthetic graphs were generated by duplicating the DBLP graph. Each node in DBLP has several copies in the synthetic graphs. We varied the duplication factor from 30 to 300 to produce the synthetic graphs with different sizes.

Graph	Node Number	Edge Number	Data Size
Freebase	1.3M	1.8M	68.49MB
DBLP	3.4M	9.4M	231.25MB
Synthetic Graphs	105M - 1B	285M - 2.8B	7.21GB - 70GB

TABLE II
Statistics of The Datasets

B. Comparing Our Approach with Index-based Approaches

We compared our approach with *h*-hop neighborhood indexing. The *h*-hop neighborhood indexing has been studied in [8, 9]. When measuring closeness scores, the *h*-hop neighborhood indexing bounds the shortest path length by h. Namely, given nodes u and v, if the shortest distance $l_{u,v} > h$, then the closeness score between u and v is 0. We implemented two index-based approaches that indexed the closeness scores of 2-hop neighbors (*2-Hop*) and 3-hop neighbors (*3-Hop*) for each node, respectively.

Effectiveness of our approach Comparing with the index-based approaches, our approach is more accurate since it does not limit the value of h. To illustrate the effectiveness of our approach, we use the real-life star query shown in Fig. 1 as an example. The query searches for *an actor in the movie Cloud Atlas that has worked with the director Steven Spielberg*. The top-5 answers for this query are shown in Table III.

Rank	Ours	*3-Hop*	*2-Hop*
1	Tom Hanks†‡	Tom Hanks†‡	Zhou Xun
2	Keith David†‡	Keith David†‡	James D'Arcy
3	Jim Broadbent†‡	Jim Broadbent†‡	Keith David†‡
4	Hugo Weaving‡	Zhou Xun	Hugo Weaving‡
5	Susan Sarandon‡	Hugo Weaving‡	Tom Hanks†‡

† This actor has worked with *Steven Spielberg* according to Freebase.
‡ This actor has worked with *Steven Spielberg* in the real life.

TABLE III
Top-5 Answers for The Real-Life Star Query

Our approach found the answers that exist in real life but are not shown in Freebase. According to Freebase, *Tom Hanks*, *Keith David*, and *Jim Broadbent* worked with *Steven Spielberg*. Interestingly, *Hugo Weaving* and *Susan Sarandon* are also found by our approach. Since Freebase is incomplete, although *Hugo Weaving* and *Susan Sarandon* collaborated with *Steven Spielberg* in real life (*Transformers*[1] for *Hugo Weaving* and *Little Women/Hook* for *Susan Sarandon*), Freebase shows that there is no collaboration between them. The reason of the successful identification by our approach is as follows. Although there is no collaboration, *Hugo Weaving* and *Susan Sarandon* are indirectly related to *Steven Spielberg* in Freebase. For example, *Susan Sarandon* was in *George Miller*'s movie, and *George Miller* worked with *Steven Spielberg*. Our approach takes into account the indirect relationships, so these actors are in the set of the top-5 answers.

Our approach's result is more accurate than the result given by *3-Hop* and *2-Hop*. The fourth best answer of *3-Hop* is *Xun Zhou*, but *Xun Zhou* did not work with *Steven Spielberg* in real life. Since *3-Hop* bounds the shortest path length by 3, the closeness score between *Hugo Weaving* (or *Susan Sarandon*) and *Steven Spielberg* is 0, as these nodes are more than 3 hops away in Freebase. Therefore, *Hugo Weaving* and *Susan Sarandon* have the same match scores as *Xun Zhou*, and *3-Hop* cannot successfully identify the better match. Similarly, *2-Hop* only finds the actors that acted in *Cloud Atlas*, but the ranking is inaccurate.

Overhead of indexing. We measured the overhead of indexing DBLP on a single machine and estimated the overhead of indexing a one-billion-node information network (1B-Network). The time and storage costs of indexing, shown Table IV, increase significantly when the indexing depth increases from 2-hop to 3-hop. Unfortunately, as discussed earlier, *3-Hop* still cannot provide accurate answers. To increase the accuracy, we have to increase the indexing depth further. As a result, the indexing costs would be unaffordable. In contrast, our approach is index-free; it only requires an adjacency list to represent the target graph and can obtain more accurate answers. Next, we will show that our approach is also efficient in answering the queries.

[1]Steven Spielberg is the executive producer of Transformers.

	Indexing Time(Hour)			Size of Graph and Index(GB)		
	Ours	*2-Hop*	*3-Hop*	Ours	*2-Hop*	*3-Hop*
DBLP	0	1.25	24.16	0.23	11.06	305.25
1B-Network	0	375	7,248	70	3,300	100,000

TABLE IV
TIME AND STORAGE COSTS OF INDEXING

C. Running Time

We evaluated the running time of our approach using a series of synthetic queries on DBLP and Freebase. The evaluation was performed on the local cluster with four workers. To generate a synthetic query $Q = (V_Q^S, \tau)$, we first randomly selected a node with type τ. Then, the specific nodes were randomly selected from the node's 3-hop neighborhood. In every experiment, we generated 20 queries on DBLP and Freebase and evaluated the average running time.

We compare our approach (the bound-based approach with priority) with two algorithms: (1) the baseline approach that computes exact match scores for all matches and (2) the bound-based approach that coordinates the propagations in a round-robin manner (no priority).

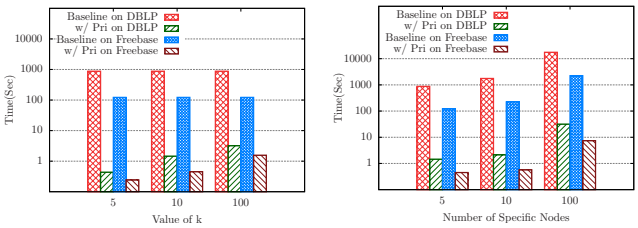
Comparing with baseline approach. We first set $|V_Q^S| = 5$ and set k to 1, 10, and 100. The running time is shown in Fig. 6(a). From the figure, the speedup of the bound-based approach over the baseline approach ranges from 610 to 2037 times faster on DBLP and from 162 to 508 times faster on Freebase for all the values of k. When $k = 1$, our approach can answer the queries in 0.43 seconds on DBLP and 0.24 seconds on Freebase. When k becomes larger, the running time increases slightly, but most queries can still be answered within 2 seconds.

To evaluate the effect of the query size, we set $k = 10$ and set $|V_Q^S|$ to 5, 10, and 100. Fig. 6(b) shows that the running time of every approach increases with the query size. When $|V_Q^S| = 10$, our approach can answer the queries within 2 seconds and 0.5 seconds on DBLP and Freebase, respectively. Since most of the real-life queries are small, this result shows that our approach can answer the small queries interactively. Although our approach needs 30 seconds to answer the large queries with 100 nodes, we can increase the worker number to decrease the running time, as will be shown in Section V-D.

Effects of prioritization policy. We varied the setting of k and the query size to evaluate the efficiency of the prioritization policy. Fig. 7 shows that in average, the approach with priority is about 3 times faster than the approach without priority. In Fig. 7(a), when k is larger, the benefit from using the prioritization policy is more significant. When $k = 1$, the approach with priority is as fast as that without priority, but when $k = 100$, the speedup increases to 6 times.

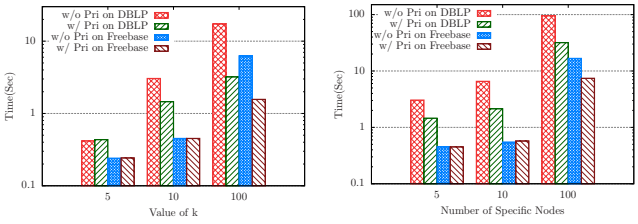
D. Scalability

The scalability experiments were performed on the Amazon EC2 cluster, and the synthetic graphs were used. The number of specific nodes $|V_Q^S|$ was set to 5, 10, 50 and 100.



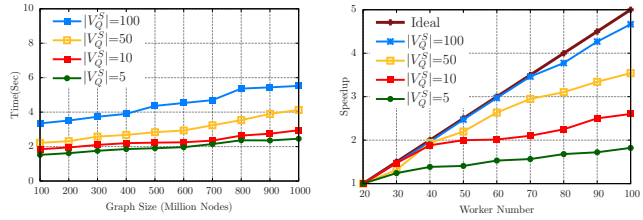
(a) Varying k (b) Varying Query Size

Fig. 6. Comparison between Baseline and Bound-with-Priority



(a) Varying k (b) Varying Query Size

Fig. 7. Comparison between Bound-without-Priority and Bound-with-Priority



(a) Increasing Graph Size (b) Increasing Worker Number

Fig. 8. Scalability of Distributed System

Scalability with graph size. To evaluate the scalability with the graph size, we utilized 100 workers in the Amazon EC2 cluster. The graph size was varied from 100 million to 1 billion nodes. Fig. 8(a) shows the running time for answering the query graph on different sizes of the target graph. The result shows that the graph size has no significant impact on the running time when the nodes' degrees are fixed.[2] For example, when $|V_Q^S| = 5$, the running time increases from 1.51 seconds to 2.45 seconds as the node number increases from 100 million to 1 billion. Therefore, the running time is mainly determined by the size of a query. This demonstrates the ability of our distributed system to scale for target graphs with millions or billions of nodes.

Scalability with worker number. For this experiment, the 100-million-node synthetic graph was used, and the worker number was varied from 20 to 100. Ideally, when the worker number increases by x times, the running time should reduce by x times. We use the running time on 20 workers as a baseline. Fig. 8(b) shows the speedup[3] when the number of workers is increased. The figure shows the reference line that indicates the ideal linear speedup. We observe that when $|V_Q^S|$ is larger, the speedup is better. When $|V_Q^S| = 100$, the speedup of our system is close to the ideal linear speedup. This is because when more data nodes are visited during the closeness

[2]All the synthetic graphs are generated by duplicating the DBLP graph.
[3]The speedup of w workers is computed as $\frac{\text{Runing time on 20 workers}}{\text{Runing time on } w \text{ workers}}$.

propagation, there is more work for the parallelism and the workers have balanced workloads. As a result, our distributed system can scale to the clusters with tens or hundreds of workers.

VI. Related Work

Graph queries. Graph queries has been studied in various application domains. Usually, graph queries are answered by subgraph isomorphism [15]. To cope with incompleteness and noisiness of real-life graphs, algorithms for detecting similar matches have been proposed [3, 6, 8, 9]. Distance-Join [3] answers pattern match queries with a 2-hop labeling index. The time and storage complexity of the index is $O(|E|^{\frac{1}{2}})$ and $O(|V||E|^{\frac{1}{2}})$, respectively. Ness [8] and NeMa [9] utilize h-hop neighborhood indexing, but it is difficult to predefine a reasonable h for every query. Our approach does not have such constraints.

Top-k query processing. Two classic algorithms for top-k query processing are Fagin's Algorithm (FA) and Threshold Algorithm (TA) [16]. They use sorted lists to determine the top-k answers. The sorted lists are disk-based since they are storage-intensive. The top-k path-based relevance query on massive graphs is studied in [12], but the closeness score functions are different from ours. Our algorithm is more general and can be extended to support the closeness score functions in [12].

Distributed graph computation. A series of distributed frameworks [13, 17–19] were developed to support graph computation, such as PageRank and single source shortest path. Our algorithm can be implemented on any frameworks that support distributed breadth-first search. Recently, optimization techniques for distributed breadth-first-search have been proposed [20]. However, optimizing breadth-first search is not our focus; these optimization techniques can be applied in our system. Distributed graph isomorphism, which finds exact matches, is studied in [10]. In contrast, our distributed approach can find approximate matches.

VII. Conclusion

In this paper, we address the top-k star query problem for massive informative networks. We present an algorithm for finding the top-k best answers efficiently. The algorithm utilizes the bound-based top-k detection mechanism to obtain the answers quickly without indexing. A distributed system for the algorithm is developed to allow scalability. Our experiments demonstrated that the proposed approach can provide meaningful answers and can efficiently compute answers for a query on massive information networks with billions of nodes.

Acknowledgements

This work is partially supported by U.S. NSF grants CNS-1217284 and CCF-1018114, National Key Basic Research Program of China under Grants No. 2010CB328104, National Natural Science Foundation of China under Grants No. 61320106007, No. 61370207, No. 61202449, No. 61300024, National High-tech R&D Program of China (863 Program) under Grants No. 2013AA013503, China National Key Technology R&D Program under Grants No. 2010BAI88B03 and No. 2011BAK21B02, China Specialized Research Fund for the Doctoral Program of Higher Education under Grants No. 20110092130002, Jiangsu research prospective joint research project under Grants No. BY2012202, No.BY2013073-01, Jiangsu Provincial Key Laboratory of Network and Information Security under Grants No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grants No. 93K-9. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsor. Jiahui Jin was a visiting student at UMass Amherst, supported by China Scholarship Council, when this work was performed.

References

[1] "Linked Open Data," http://www.w3.org/wiki/SweoIG/Task Forces /Community Projects/LinkingOpenData.

[2] Google, "Freebase data dumps," https://developers.google.com /freebase/data.

[3] L. Zou, L. Chen, and M. T. Özsu, "Distancejoin: Pattern match query in a large graph database." *PVLDB*, pp. 886–897, 2009.

[4] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs." in *KDD*, 2007, pp. 737–746.

[5] J. Cheng, X. Zeng, and J. X. Yu, "Top-k graph pattern matching over large graphs." in *ICDE*, 2013, pp. 1033–1044.

[6] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: From intractable to polynomial time." *PVLDB*, pp. 264–275, 2010.

[7] Y. Tian and J. M. Patel, "Tale: A tool for approximate large graph matching." in *ICDE*, 2008, pp. 963–972.

[8] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao, "Neighborhood based fast graph search in large networks." in *SIGMOD*, 2011, pp. 901–912.

[9] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan, "Nema: Fast graph search with label similarity." *PVLDB*, pp. 181–192, 2013.

[10] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs." *PVLDB*, pp. 788–799, 2012.

[11] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems." *ACM Comput. Surv.*, 2008.

[12] S. Khemmarat and L. Gao, "Fast top-k path-based relevance query on massive graphs." in *ICDE*, 2014, pp. 316–327.

[13] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables." in *OSDI*, 2010, pp. 293–306.

[14] "DBLP," http://www.informatik.uni-trier.de/ ley/db/.

[15] J. R. Ullmann, "An algorithm for subgraph isomorphism." *J. ACM*, pp. 31–42, 1976.

[16] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware." *J. Comput. Syst. Sci.*, pp. 614–656, 2003.

[17] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation." *IEEE Trans. Parallel Distrib. Syst.*, pp. 2091–2100, 2014.

[18] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing." in *SIGMOD*, 2010, pp. 135–146.

[19] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritizing iterative computations." *IEEE Trans. Parallel Distrib. Syst.*, pp. 1884–1893, 2013.

[20] F. Checconi and F. Petrini, "Traversing trillions of edges in real-time: Graph exploration on large-scale parallel machines." in *IPDPS*, 2014.