Keyword Search over Distributed Graphs with Compressed Signature

Ye Yuan, Xiang Lian, Lei Chen, Jeffery Xu Yu, Guoren Wang, Yongjiao Sun, *Member, IEEE,*

Abstract—Graph keyword search has drawn many research interests, since graph models can generally represent both structured and unstructured databases and keyword searches can extract valuable information for users without the knowledge of the underlying schema and query language. In practice, data graphs can be extremely large, e.g., a Web-scale graph containing billions of vertices. The state-of-the-art approaches employ centralized algorithms to process graph keyword searches, and thus they are infeasible for such large graphs, due to the limited computational power and storage space of a centralized server. To address this problem, we investigate keyword search for Web-scale graphs deployed in a distributed environment. We first give a naive search algorithm to answer the query efficiently. However, the naive search algorithm uses a flooding search strategy that incurs large time and network overhead. To remedy this shortcoming, we then propose a signature-based search algorithm. Specifically, we design a vertex signature that encodes the shortest-path distance from a vertex to any given keyword in the graph. As a result, we can find query answers by exploring fewer paths, so that the time and communication costs are low. Moreover, we reorganize the graph data in the cluster after its initial random partitioning so that the signature-based techniques are more effective. Finally, our experimental results demonstrate the feasibility of our proposed approach in performing keyword searches over Web-scale graph data.

Index Terms—Keyword Search, Distributed Graph

1 Introduction

1.1 Motivation

Keyword search on graphs, such as RDF graphs, social networks, graph-structured relational data or XML, has attracted a lot of interests in recent years [1], [2], [3], [4], [5], [6], since it can bring valuable information to users without the knowledge of the underlying schema and query language.

For example, Fig. 1 shows a publication graph where vertices represent “author” and “paper”, and “write” relationships between them are represented as edges. As shown in the figure, Daniel J. Abadi writes the paper “Aurora...” and hence there is an edge from J. Abadi to Aurora. Assume that a keyword query against the graph is like: “who is the common author of the papers Aurora and C-store?”. Due to the database normalization, it is possible that no individual vertices contain both keywords, Aurora and C-store. Thus, traditional search engines cannot return meaningful results containing keywords. In contrast, the graph search can find an answer like “Michael Stonebraker writes papers Aurora and C-store”, which is exactly what the user wants or could be interested in.

The result in this example is a tree structure (consisting of the root Michael Stonebraker and two leaf nodes Aurora and C-store in Fig. 1), which has been widely adopted to explain the answer semantics of keyword search over graphs. Therefore, in this paper, we focus on the answers which are tree structures, called *answer trees*. Clearly, given a set of keywords, there exist many possible answer trees. We are interested in the top-k answer trees that have the highest scores. The score of an answer tree T inversely

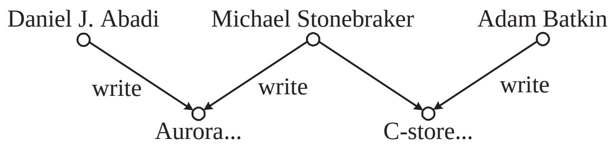


Fig. 1: Example of keyword search over a publication graph.

relates to the total path length from T's root to T's leaf nodes [2], [6].

Large-scale graph datasets with billions of vertices are becoming common in the context of social networks and Web-scale datasets. For example, the World Wide Web now contains more than 50 billion Web pages and more than one trillion unique URLs [7]. The friendship network of Facebook contains 1.3 billion nodes and over 140 billion links [8]. LinkedData now consists of 31 billion RDF triples and 504 million RDF links [9]. To support such huge graphs, many parallel and distributed graph computing systems have been developed, including Pregel [10] and its open-source alternatives [11], [12], GraphLab [13] and Trinity [14]. To handle the keyword search over Web-scale graphs, in this paper, we study the distributed graph keyword search. Here, a big graph is distributed across a cluster composed of slave servers. A master server forwards the query to the cluster of slave servers to conduct keyword search in parallel. The query results are then returned to the master. As an example, suppose that the data graph G in Fig. 2(a) is distributed across three slaves: SL_1, SL_2

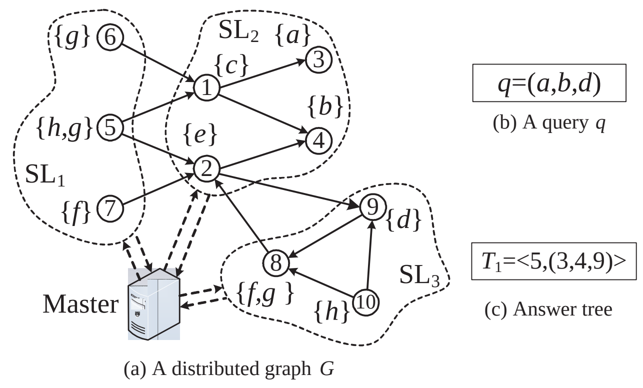


Fig. 2: Distributed graph keyword search.

and SL_3. For the query q in Fig. 2(b), the three slaves will perform the keyword search in parallel, to compute the answer tree as shown in Fig. 2(c).

1.2 Naive Solution

To find top-k *answer trees* in a distributed graph G, a naive solution consists in adapting a centralized search algorithm (i.e., backward search) to the distributed scenario, which works as the following three steps.

First, master M_q simply broadcasts a list of query keywords q to all the participating slaves. Second, upon receiving q, all slaves compute *answer trees* in parallel by performing the backward search. A backward search starts from keyword nodes (nodes containing keywords in q) and performs a breadth-first search (BFS) from each keyword node. An answer tree is discovered if its root node is found by the BFSs from all keyword nodes. Once an answer tree is discovered, it is sent back to M_q. Third, M_q selects top-k answer trees with the highest scores, from all its received answer trees, as final query answers. M_q ceases the backward search in each slave until no answer tree with a larger score is discovered. Note that, in the second phase, a backward search may traverse different slaves to discover answer trees across different slaves.

1.3 Our Contributions

The naive solution has the shortcoming of very large network overhead. Indeed, the backward search is just a flooding search that tries to discover the root of an answer tree. Large real-world graphs exhibit a skewed power-law distribution, and thus a slave may have many boundary nodes (i.e., many remote neighbors in other slaves) [10], [13]. As a consequence, the backward search from boundary nodes will incur immense network overhead.

To solve this shortcoming, we propose a signature-based search algorithm, in which we design a novel data structure called *vertex signature* that encodes keyword information in a highly compressed manner, and allows for efficient propagation. The amount of keyword information in a vertex signature (the number of bits used to store keywords) decreases rapidly with distance to the keyword. In the signature-based search algorithm, vertices with the maximum amount of information of query keywords are the root nodes of answer trees. Thus, each keyword node uses hints obtained from vertex signatures to perform the backward search, so that the search can reach root nodes efficiently. The use of root hints provides a significant advantage over the completely blind nature of the naive solution, translating into large reductions in time and network costs.

Through rigorous mathematical analysis, we give the time and communication complexities of the naive and signature-based search algorithms. From the result, we will see that either time cost or communication cost of the naive search is in the exponential of the search steps, whereas those of the signature-based search algorithm obtain the linear costs of the search steps. Moreover, we reorganize the distributed graph after its initial hash partitioning to further reduce the time and communication costs.

To summarize, we make the following contributions.

(1) We conduct the first study on keyword search over distributed graphs, and formally define this problem.

(2) We propose a signature-based search algorithm that provides root hints for the backward search so that it greatly reduces time and communication costs, compared to the naive search algorithm.

(3) We reorganize graph data in the cluster after its initial hash partitioning so that we can further speed up the query processing.

(4) Experimental results on real and synthetic datasets show our algorithms achieve high performance and significantly outperform the naive solution by many times.

The rest of the paper is organized as follows. Section 2 defines the problem. Sections 3 and 4 introduce the framework and details of the signature-based search algorithm. Section 5 discusses some issues to handle for the signature-based search algorithm. Section 6 gives the scheme of reorganizing the distributed graph. Section 7 shows an extensive experimental evaluation on real and synthetic datasets. Finally, Section 8 overviews the related work, and Section 9 concludes the paper.

2 PROBLEM DEFINITION

We summarize the abbreviations and notations of this paper in Table 1.

Following the convention [1], [3], [15], in this paper, we model a graph G as a directed graph $G = (V, E)$, where each vertex $v \in V$ is labeled with some keywords, denoted by $KW(v)$, and an edge $e = (u, v) \in E$ represents the relationship from u to v. For example, in the graph shown in Fig. 2(a), vertex 5 contains two keywords g and h.

For the directed graph G, if G has an edge (u, v), we call v is an *out-neighbor* of u and u is an *in-neighbor* of v. Below, we use $V(G)$ and $E(G)$ to denote the set of vertices and the set of edges for a graph G.

The graph $G = (V, E)$ will be distributed into m computational machines (called slaves) as an m-element $\mathcal{F} = (F_1, ..., F_m)$, where the fragment F_i is maintained in the i-th slave and is specified as follows. F_i maintains a subgraph $G_i = (V_i, E_i)$, which is an induced subgraph

TABLE 1: Abbreviation and Notation.

Symbol	Description
G	the distributed graph
M_q	the master
SL_i	the ith slave
F_i	the subgraph of G storied in SL_i
$q = (b_1, ..., b_w)$	the query keywords
$T = \langle r, (v_1, ..., v_w)\rangle$	the answer tree T with the root r and leaf nodes $(v_1, ..., v_w)$
h	the upper bound of the path length from r to any leaf node
l_w	the length of the longest path from r to v_i
KFS	keyword forwarding signature
$KW(u)$	the set of keywords contained in node u
$KS(u)$	the KFS of node u
$\theta(kw)$	the number of 1's in the KFS of keyword kw
$A[n]$	the bitstring of a KFS
c	the number of hash function
f	the linear decay factor
KFS^1	the KFS obtained in the Decay Rule
KFS^2	the KFS obtained in the Diffusion Rule
$\Theta^1(q)$	the largest indicator for q based on the Decay Rule
$\Theta^2(q)$	the largest indicator for q based on the Diff. Rule

of G over V_i. Here, $V = \bigcup_{i=1}^{k} V_i$ and $V_i \cap V_j = \emptyset$ for any $i \neq j$. There are edges across different fragments, which we call cross-edges. F_i maintains a set of cross-edges, $\{(u, v)\}$, denoted as $F_i.C$, where either u or v exists in a different fragment F_j. Below, we use $F_i.O$ to denote the set of vertices in $F_i.C$ that do not exist in G_i and use $F_i.B$ to denote the set of vertices (boundary vertices) in $F_i.C$ that exist in G_i.

Fig. 2 shows a distributed graph G maintained in three fragments on three slaves, SL_1, SL_2, and SL_3. Take SL_1 as an example, where fragment F_1 is maintained. F_1 maintains a subgraph G_1 as indicated by the dot circle. F_1 also maintains the set of cross edges: $F_1.C = \{(6, 1), (5, 1), (5, 2), (7, 2)\}$. The boundary vertices denoted by $F_1.B = \{5, 6, 7\}$, and $F_1.O = \{1, 2\}$.

Initially, we distribute a graph G to slaves by hashing the vertex ids, which cannot result in an optimal graph partitioning, but it incurs the least cost.

In Section 6, we will propose a scheme of reorganizing graph data to assure a load balance, which can fix the issues of the initial random partitioning.

A keyword search query q consists of a list of *query keywords* $(b_1, ..., b_w)$. We formally define an answer to q as follows:

Definition 2.1: (**Answer Tree**) Given a query $q = (b_1, ..., b_w)$ and a graph G, an answer to q is a tree structure denoted by $\langle r, (v_1, ..., v_w)\rangle$, where r and v_i's are vertices of G satisfying: (1) For every i, node v_i contains keyword b_i (called *keyword node*); (2) For every i, there exists a directed path of G from r to b_i.

Note that a keyword node can contain multiple keywords, and the root can also be a leaf node. Fig. 2(c) gives an answer tree of query $q = (a, b, d)$ over graph G in Fig. 2(a). For clear presentation, we use term "node" for tree, and "vertex" for graph.

Answers can also be graph structures as introduced in [6], [15]. Our solution to the tree structure is simply extended to support the graph semantics as given in Section 5.

Definition 2.2: (**Score of Answer Tree** [2], [6]) Given an answer tree $T = \langle r, (v_1, ..., v_w)\rangle$, the score of T is defined as $f(\sum_{i=1}^{w} D(r, v_i))$, where $D(r, v_i)$ is the graph distance from r to v_i. The score function $f()$ inversely depends on the total path length from the root to leaf nodes, and we prefer the length as small as possible. Thus, a small path length obtains a good score, and the minimum Steiner tree induced by the root and leaf nodes corresponds to the best score. It is not meaningful for users to obtain an answer tree with a very large path length. Thus, similar to [2], [6], we use an integer h to bound the path length, i.e., the path length from r to v_i is no larger than h.

Since G contains massive answer trees, we are interested in top-k answer trees with the highest scores.

Problem Statement. Find top-k answer trees with the highest scores for a user-given query $q = (b_1, ..., b_w)$ in the distributed graph G over m slaves, where the i-th slave maintains the fragment F_i of graph G.

Fig. 2(c) shows the top-1 answer tree of q in G.

The problem of computing a top-1 answer tree is the Group Steiner Tree problem which is NP-complete [16]. Thus, in this paper, we will propose a c-approximation search algorithm to find the answer trees.

```
Framework Keyword Search(G, q, k)
Input: A distributed graph $G$, a list of query keywords $q$ and the number of answers $k$;
Output: $k$ answer trees with largest scores
Procedure Compute_LocalAnswerTree() // Phase 1;
1. $M_q$ sends $q$ to all the participating slaves;
2. Each slave $SL_i$ computes top-$k$ local answer trees and sends them to $M_q$;
3. $M_q$ selects $k$ trees with largest scores as the candidate answer trees $\{T_g\}$;
Procedure Compute_CrossingAnswerTree($\{T_g\}$) // Phase 2
//At master $M_q$
1. If (the "halting condition" is not satisfied) // The halting condition "$s > l_w$" is based on the $k$th tree in $\{T_g\}$.
2.    Upon receiving crossing answer trees do
3.       Update the candidate top-$k$ answer trees ;
4.       Update the lower bound $l$ based on the $k$th tree in $\{T_g\}$;
5. Return the current candidates as the final top-$k$ answer trees;
// At slave $SL_i$
1. Determine the partial trees $\{T_p\}$;
2. $\{T_c\}$ =Procedure Signature-based_Search ($q$, $\{T_p\}$, $F_i$);
3. Send crossing answer trees $\{T_c\}$ to $M_q$;
```

Fig. 3: Framework for the keyword search over a distributed graph.

3 Algorithmic Framework

Fig. 3 shows the framework for the keyword search algorithm over a distributed graph that consists of two phases. The first phase is to compute *local answer trees*, and the second phase is to determine *crossing answer trees*. Note that a local answer tree has all nodes in the same slave, while a crossing answer trees has at least two nodes in different slaves.

The first phase includes three steps: 1. Master M_q simply broadcasts a list of query keywords q to all the participating slaves. 2. Upon receiving q, each slave SL_i computes top-k local answer trees in it and sends them to M_q.

Many centralized search algorithms [6] can be employed to determine the top-k local answer trees. 3. From all received local answer trees, M_q selects k answer trees with the highest scores as the candidate answer trees.

Crossing answer trees determined in the second phase are used to update the top-k candidate answer trees obtained in the first phase. Specifically, the second phase works as follows: Every slave performs signature-based search algorithm to discover crossing answer trees $\{T_c\}$ in parallel. Once a T_c is found, it is sent back to the master M_q. Based on T_c, M_q updates the top-k candidate answer trees $\{T_g\}$, if T_c has a larger score than a T_g. We repeat the steps above until a "halting condition" is satisfied. Finally, the current candidate answer trees are the *query answers*.

The signature-based search algorithm is fed with *partial trees* that consist of the paths from boundary nodes to keyword nodes. Since each slave computes local answer trees using the backward search that starts from keyword nodes until meets the root. In this process, the backward search also finds the partial trees, because the backward search stops at boundary nodes of the slave if it cannot find an local answer tree. The signature-based search starts from partial trees $\{T_p\}$ and tries to discover complete answer trees across several slaves. The detailed steps will be introduced in Algorithm 2.

The halting condition is determined as follows: Let $T_k = \langle r_k, (v_1, ..., v_w)\rangle$ denote the kth candidate answer tree, where r_k is the root node of T_k and $v_1, ..., v_w$ are leaf nodes of T_k. Let l_w be the length of the longest path from r_k to v_i ($1 \leq i \leq w$), and let s be the number of steps performed by the signature-based search in each slave. Then the "halting condition" is $s > l_w$. In the second phase, M_q constantly updates T_k based on new found crossing answer trees, and thus the value of l_w is also updated accordingly. With more crossing answer trees are found, the value of l_w becomes more tight, so that the backward search converges efficiently.

The design of the signature-based algorithm (Lines 1–3 of phase 2 performed at each slave SL_i) is the core part of the algorithmic framework. We will defer a detailed description to the next section.

4 Signature-based Search Algorithm

In this section, we first design *keyword forwarding signature* (KFS) for each vertex of G, and then propose the search algorithm based on KFS.

4.1 Overview of the Search Algorithm

The key idea of our signature-based search scheme is captured in Fig. 4 which shows an answer tree $\langle r_1, (v_1, v_2, v_3, v_4)\rangle$ to query $q = (b_1, b_2, b_3, b_4)$. In this figure, v_1, v_2, v_3 and v_4 are keyword nodes, and r_1 is the root of the answer tree. The signature-based scheme propagates awareness about keywords hosted at a node in the neighborhood of the node, with the amount of information decreasing linearly with the distance from the keyword node. The information is the hash values of keywords contained in the node. In Fig. 4, partial information of b_1 (say I_1) is propagated to node r_2, and then partial information of I_1 is propagated to node r_1 from r_2. Similarly, information of b_2, b_3 and b_4 is propagated to nodes r_2 and r_3, and then all the information is propagated to node r_1. Since the total path length from r_1 to keyword nodes v_1, v_2, v_3 and v_4 is the smallest, the decrease information of keywords $\{b_1, b_2, b_3, b_4\}$ to r_1 along the paths is the *smallest*. In other words, r_1 has the *largest* remaining information of $\{b_1, b_2, b_3, b_4\}$ among all nodes of G. As a consequence, nodes close to the root have strong information about the query keywords and the root node has the largest strength of this information.

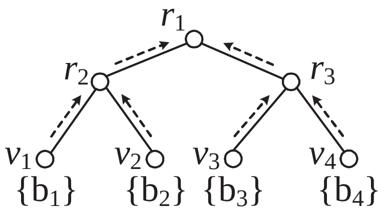


Fig. 4: Overview of our solution; Dashed lines denote the propagation of the keyword information.

With the strongest information (say I_s) about keywords, the signature-based search greedily finds the root along a path from a keyword node to the root. For example, in Fig. 4, I_s of $q = (b_1, b_2, b_3, b_4)$ guides the backward search from keyword node b_1 as the following steps: The search first reaches r_2, since r_2 has strong information about I_s. After that, the search reaches r_1 from r_2, since r_1 has stronger information about I_s. The search terminates at r_1, since no other node has stronger information about I_s than r_1. Similarly, other paths from keyword nodes to r_1 can be discovered using the greedy search algorithm.

When nodes of an answer tree are in different slaves, this search strategy will incur much less communication costs than the naive scheme that inherently is a flooding search.

4.2 Algorithm Design

We propose a *keyword forwarding signature* (KFS) to encode keywords of each vertex in G. In particular, KFS encodes the approximate hop-count distance from the vertex of G to any given keyword in the graph. In the following, we give a detailed introduction to KFS.

Given a vertex u of G, the keyword forwarding signature of u (denoted by $KS(u)$) is a bitstring obtained by hashing keywords in $KW(u)$. Let A represent a bitstring, where $|A| = n$. From $KW(u)$, $KS(u)$ is generated as follows: Using c different hash functions $h_1(.)$,...,$h_c(.)$, we set c out of n bits in A to be 1. Specifically, for each $kw \in KW(u)$, we set the bits in A indexed by $h_i(kw)$, $i = 1, 2, ..., c$ to be 1. However, given a query for a keyword kw, KFS simply returns $\theta(kw) = |\{i|A[h_i(kw)] = 1, i = 1, 2, ..., c\}|$, the quantity of 1 bits in the signature. Note that $\theta(kw)$ is approximately c bits[1]. In the algorithmic design,

1. The approximation is due to the collision of different hash functions

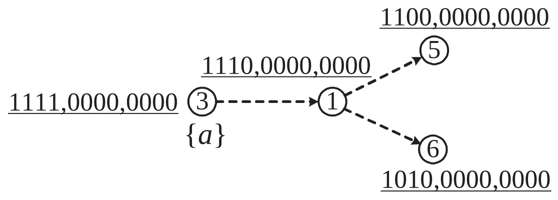


Fig. 5: Decaying process for vertex 3 of graph G in Fig. 2(a).

the number of bits $\theta(kw)$ decays linearly (at a constant rate f) with its distance (in terms of hops) from the vertex where the keyword is contained. In other words, at a vertex one hop away from the keyword, $\theta(kw)$ is approximately $c-f$ bits, at a vertex two hops away, $\theta(kw)$ is approximately $c-2f$, and so on. While spreading keyword information to downstream neighbors, vertices reset each of the bits in the KFSs received from upstream neighbors with a probability $(1-f/\theta(kw))$, thus achieving a linear decay by a factor of f.

Formally, we have a decay rule for KFS.

Definition 4.1: (**Decay Rule.**) Given hash functions $h_1, ..., h_c$, we build a KFS for each vertex u of G, and for each keyword $kw \in KW(u)$, KFS sets the bits $A[h_1(kw)]$,...,$A[h_c(kw)]$ to 1. When u decays its KFS to a one-hop in-neighbor (say v), for each $A[j]=1$ $(1 \leq j \leq n)$, we reset $A[j]=1$ with probability $(1-f/\theta(kw))$ and $A[j]=0$ with probability $f/\theta(kw)$. The probability assures that $\theta(kw)$ decays by a linear factor of f. When vertex v receives multiple KFSs from upstream neighbors, these KFSs are merged into a single KFS through a bitwise-OR operation. Given the path length limit h, we decay the KFS to a range of h-hop in-neighbor vertices away from u. Note that we should have $c > fh$ to assure a valid decay for c bits.

In Section 5.1, we will discuss how to set the parameters h, f and c in query processing.

Example 1: Fig. 5 illustrates the decaying process for vertex 3 of graph G in Fig. 2(a). Assume that the length of A is $n=12$, four hash functions are used (i.e., $c=4$), the decay factor is $f=1$, and the number of decay hops is $h=2$. As shown in Fig. 5, the initial hash value over keyword a is $1111,0000,0000$, and then one "1" bit of the hash value becomes 0 after one hop decayed to vertex 1. Assume that the decayed hash value is $1110,0000,0000$ at vertex 1. Using a similar process, we obtain the decayed hash values of other vertices as shown in the figure.

The signature-based search uses the KFS of each vertex to approach the root node of an answer tree in the backward search. The following theorem provides the root information corresponding to keyword nodes of q.

Theorem 1: Given a list of keywords $q=(b_1,...,b_w)$ and a distributed graph G, let $T=\langle r,(v_1,...,v_r)\rangle$[2] denote an answer tree of q over G. Given a KFS with c hash functions (h_1,...,h_c) and an array of bits A ($|A|=n$), each vertex of G constructs a KFS and decays the KFS as the Decay Rule. For given keyword nodes $v_1,...,v_r$, if T has the highest score, the KFS $A[]$ of the root r of T has the *largest* indicator $\Theta(q)=\sum_{j=1}^{w}|\{i|A[h_i(b_j)]=1, 1\leq i \leq c\}|$ to leaf nodes v_1,...,v_r.

Proof: We give the proof by contradiction. Assume that there is another tree $T'=\langle r',(v_1,...,v_r)\rangle$ (with root r' different from r, but with the same leaf nodes $\{v_1,...,v_r\}$ in which the root node r' has a larger indicator $(\Theta(q)')$ than r. Based on the Decay Rule, each node v_i $(1 \leq i \leq r)$ constructs its local KFS, $KS(b_i)$, for keyword b_i and then decays $KS(b_i)$ by a linear factor of f. Let the number of 1's of the initial KFS for v_i be θ_i, and let leaf node v_i be n_i hops from root r. After $KS(b_i)$ is decayed to node r, the number of 1's of the decayed $KS(b_i)$ becomes $\theta_i - n_i \times f$. Similarly, we can know the number of 1's of $KS(b_i)$ decayed to node r' is $\theta_i - n'_i \times f$, where n'_i is the number of hops from v_i to r'. Since we use bitwise-OR operation for KFSs, we obtain $\Theta(q)=\sum_{i=1}^{r}(\theta_i - n_i \times f) = \sum_{i=1}^{r}\theta_i - f\sum_{i=1}^{w}n_i$ and $\Theta(q)'=\sum_{i=1}^{r}\theta_i - f\sum_{i=1}^{r}n'_i$. Based on the assumption, we have $\Theta(q)' > \Theta(q)$ which leads to $\sum_{i=1}^{r}n_i > \sum_{i=1}^{r}n'_i$. According to Definition 2.2, T' has a higher score than T, which is a contradiction. □

Theorem 1 only provides the quantity information of the root signature but not the exact signature of the root node. Therefore, Theorem 1 cannot be applied to the backward search directly, and we add one more operation to the Decay Rule to remedy this shortcoming: *After each vertex u performs the Decay Rule, u sends its KFS to all out-neighbors within the range of h-hops from u.* Since h upper bounds the distance from the root to any leaf node, every node of an answer tree obtains the same KFS as its root node. In other words, given leaf nodes $v_1,...,v_w$, any node of the answer tree T has a larger $\Theta(q)$ than any node of other trees with $v_1,...,v_w$ as leaf nodes. For instance, in Fig. 2, the KFS of vertex 5 is $1100,1100,1100$ after keywords $\{b,d,f\}$ are decayed to vertex 5. Then, vertex 5 spreads the KFS to all vertices within $h=2$ hops away from it, i.e. vertices 1, 2, 3, 4 and 9. As a consequence, vertex 1 also has the KFS $1100,1100,1100$ and has a larger $\Theta(q)$ than vertex 6 which is not the node of the answer tree $T=\langle 5,(3,4,9)\rangle$.

Now, we can design a distributed backward search that starts from keyword nodes and visits in-neighbor nodes with the largest $\Theta(q)$. If a node r is visited by the backward search from every keyword node, r is the root of the answer tree.

Example 2: Fig. 6 illustrates the idea of the signature-based search algorithm. In this figure, two trees T_1 and T_2 have v_1 and v_2 as their leaf nodes, and the roots of T_1 and T_2 are r_1 and r_2. Assume that $D(r_1,v_1)+D(r_1,v_2) < D(r_2,v_1)+D(r_2,v_2)$, where $D(a,b)$ denotes the graph distance from node a to b. Assume that v_1 and v_2 are keyword nodes containing query keywords $q=(b_1,b_2)$. Based on the Decay Rule, r_1 has a larger $\Theta(q)$ than r_2, and thus all nodes of T_1 have larger $\Theta(q)$ than all nodes of T_2. We can design the backward search as follows: The search starts from v_1 and visits v_3 instead of v_4, since v_3

2. Note that $r \leq w$, since a keyword node may contain multiple keywords in q.

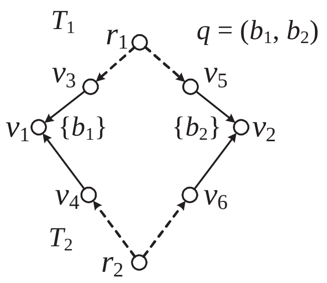


Fig. 6: Idea of the signature-based search algorithm.

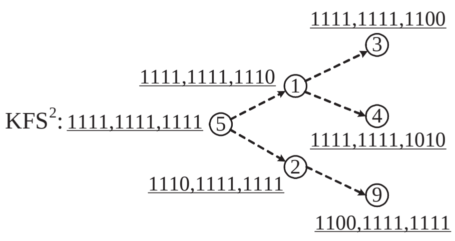


Fig. 7: The diffusion process for vertex 5 in Fig. 2.

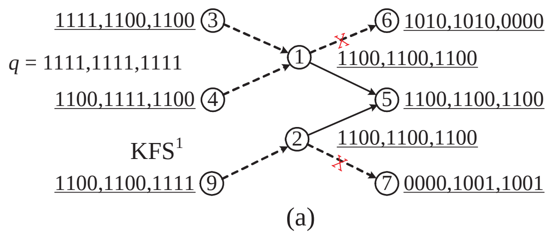
has a larger $\Theta(q)$ than v_4. After that the search goes along the path from v_1 to r_1. Similarly, another search starts v_2 and goes along the path from v_2 to r_1. Both searches will reach r_1, and hence r_1 is the root of an answer tree (i.e., T_1).

In Example 2, we observe that, when a node u is visited, the search selects to visit u's in-neighbor v with the largest $\Theta(q)$ among all u's in-neighbors. This scheme assures that the backward search always visits nodes of the answer tree instead of other trees. However, node v may have a smaller $\Theta(q)$ than u, since u's KFS is *decreased* to v. For example, in Fig. 2, vertex $u = 3$ has a KFS of $1111, 1100, 1100$ that has more "1" bits than $1100, 1100, 1100$ that is the KFS of vertex $v = 1$. In this case, the search will not visit v from u. To solve this issue, we propose another KFS for each vertex of G and this KFS is obtained by a Diffusion Rule given as follows:

Definition 4.2: (**Diffusion Rule.**) Given hash functions $h_1, ..., h_c$, we build a KFS for each vertex u of G, and for each keyword $kw \in KW(u)$, KFS sets the bits $B[h_1(kw)]$,...,$B[h_c(kw)]$ to 1, where $B[n]$ is an array of bits. Vertex u diffuses its KFS (without decay) to a range of h-hop out-neighbors away from u. When vertex v receives multiple KFSs from upstream neighbors, these KFSs are merged into a single KFS through a bitwise-OR operation. After this operation, given $q = (b_1, ..., b_w)$, the root r of an answer tree obtains all bits of q. Finally, each vertex of G decays its merged KFS to out-neighbors within h-hops away from it as the *Decay Rule*.

To distinguish the KFS in the Decay Rule, we denote KFS^1 by the KFS in the Decay Rule and KFS^2 by the KFS in the Diffusion Rule. Similarly, we denote $\Theta^1(q)$ and $\Theta^2(q)$ by the number of bits of q in the Decay Rule and Diffusion Rule, respectively.

Based on the Diffusion Rule, given $q = (b_1, ..., b_w)$, the root r of an answer tree T has the largest $\Theta^2(q)$ among all nodes of T. In T, nodes close to r have larger $\Theta^2(q)$ than nodes close to leaf nodes, where $\Theta^2(q) = \sum_{j=1}^{w} |\{i|B[h_i(b_j)] = 1, 1 \leq i \leq c\}|$. For example, Fig. 7 shows the process of the Diffusion Rule from vertex 5. In this example, $h = 2$ and the KFS^2 of vertex 5 is $1111, 1111, 1111$ after the diffusion from keywords $\{a, b, d\}$ with bits equal to $\{1^4 0^4 0^4, 0^4 1^4 0^4, 0^4 0^4 1^4\}$ through the OR operation. Then, the KFS^2 of vertex 5 is decayed to vertices 1, 2, 3, 4 and 9 as shown in Fig. 7.



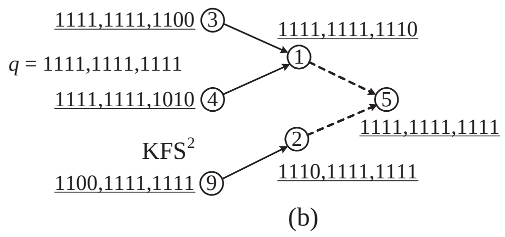


Fig. 8: Query processing for query $q = (a, b, d)$ over the distributed graph G in Fig. 2(a).

Now, we can remedy the backward search algorithm based on the Diffusion Rule as follows:

For the current node u in the backward search, assume that node v is the u's in-neighbor with the largest $\Theta^1(q)$ among all u's in-neighbors. If v has a smaller $\Theta^1(q)$ than u, we test if v has a larger $\Theta^2(q)$ than u. If the answer is yes, the search visits the in-neighbor v, and otherwise the backward search terminates at node u.

Example 3: Fig. 8 shows the query processing for query $q = (a, b, d)$ over the distributed graph G in Fig. 2(a). The signature-based algorithm uses the hash value, Hash(a)∨Hash(b)∨Hash(d)= $1^4, 1^4, 1^4$, to guide the backward search. We take the search from node 9 as an example: The backward search first tries to select node 2 to forward by examining KFS^1 of nodes 2 and 9, but it fails, since node 9 has a larger $\Theta^1(q)$ (eight 1's) than node 2 (six 1's as shown in Fig. 8(a)). In this case, the search examines KFS^2 of nodes 2 and 9 and finds that the $\Theta^2(q)$ (eleven 1's) of node 2 is larger than that of node 9 (ten 1's). Thus, the search visits node 2 from node 9 as given in Fig. 8(b). After that, the backward search chooses node 5 to forward, since node 5 has the largest $\Theta^1(q)$ among all neighbors of node 2 and the $\Theta^2(q)$ of node 5 is not smaller than that of node 2 (as shown in Figure 8(a)). Similarly, backward searches from nodes 3 and 4 also visit node 5. Thus, node 5 is a candidate root and we obtain the answer tree $T = \langle 5, (3, 4, 9) \rangle$.

Below, we give the construction algorithm for KFS^1 and KFS^2 based on the Decay and Diffusion rules. We provide the signature-based search algorithm based on KFS^1 and

```
Algorithm 1 Construct_KFS (v, h)
Input: A vertex v and a range of decaying h;
Output: KFS^1 A^1 and KFS^2 A^2 at v
Procedure Decay_KFS (v, h)
//Create local KFS A
1.  For(each keyword kw of KW(v))
2.      Set bits $A[h_1(kw)], \dots, A[h_c(kw)]$ to 1;
3.  A.TTL=h and send KS(A) to v's BFS in-neighbors;
4.  Create an empty KFS C with all bits set to 0;
5.  For(each KFS B received from v's out-neighbors)
6.      For(r=1; r<n; r++)
7.          If(B[r]==1)
8.              With probability $1 - f/\theta(B)$, C[r]=1;
9.      KS(A)=KS(A)∪ KS(C);
10.     If(B.TTL≠0)
11.       C.TTL=B.TTL-1;
12.       Send C to v's BFS in-neighbors;
13. If(Decay Rule has finished)
14.     v sends KS(A) to a range of h-hop out-neighbors away from v;
15.     For(each KFS B received from in-neighbors of v)
16.         KS(A^1)=KS(A)∪ KS(B);
Procedure Diffuse_KFS (v, h)
1.  Set bits $A[h_1(kw)], \dots, A[h_c(kw)]$ to 1;
2.  v sends KS(A) to a range of h-hop in-neighbors away from v;
3.  For(each KFS B received from out-neighbors of v)
4.      KS(A)=KS(A)∪ KS(B);
5.  KS(A^2)=Decay_KFS (v, h);
```

Fig. 9: Construction of keyword forwarding signatures.

```
Algorithm 2 Signature-based_Search( q, {T_p}, F_i )
Input: Query keywords q, parital trees {T_p}, framegent F_i at slave SL_i;
Output: Crossing answer trees
// At slave SL_i
1.  Begin {synchronization}
2.  If (the "halting condition" is not satisfied)
3.      Determine valid partial trees in {T_p};
4.      For (each valid partial tree T_p)
5.          Let u be the boundary node of T_p in F_i;
6.          v=Forward_Neighbor (u, q); //When u is a boundary node, v is its remote
                                        //in-neighbor at a different slave.
7.          If ($\theta^1_q(v) \geq \theta^1_q(u)$)
8.              Recursively perform Forward_Neighbor (v, q);
9.          Else if ($\theta^2_q(v) \geq \theta^2_q(u)$)
10.             Recursively perform Forward_Neighbor (v, q);
11.         Else
12.             Terminate the backward search;
13.             Node u is a candidate root;
14.             Send the discovered crossing trees to the master;
15.  End {synchronization}
Procedure Forward_Neighbor (u, q)
Input: A node u, a list of keywords q;
Output: Next forwarding node v
1.  $\Theta = \phi$;
2.  For (each in-neighbor x of u)
3.      For (each keyword $b_i$ of q)
4.          $\theta_x = KS^1_x[h_1(b_i)] + \cdots + KS^1_x[h_c(b_i)]$;
5.       $\Theta = \Theta \cup \{\theta_x\}$;
6.  Pick the in-neighbor $x_M$ such that $\theta_x$ = MAX($\Theta$);
7.  $v = x_M$;
```

Fig. 10: Signature-based search algorithm.

KFS^2.

Algorithm 1 gives the procedure of constructing two keyword forwarding signatures for each vertex, i.e., Decay_KFS() and Diffuse_KFS() (Fig. 9). The procedure Decay_KFS() constructs KFS^1 based on the Decay Rule and consists of four phases. In the first phase (Lines 1-3), we create a local KFS A, for each vertex v of G, that takes in the list of keywords hosted by v. Then, we start decaying A with a range of h rounds (where h is the path length limit.). In the second phase (Lines 5-9), we process the KFSs received from neighbors of v. Specifically, for each received KFS B, the bits of B are randomly reset to zero so that only $1 - f/\theta(B)$ of the bits survive this decay operation. Then, we aggregate the decayed KFSs $\{B\}$ with A by merging all KFSs into a single KFS through a bitwise-OR operation and maintain A at vertex v. In the third phase (Lines 10-12), we send the decayed KFSs $\{C\}$ to v's BFS neighbors. In the last phase (Lines 13-16), the signature of v is sent to the vertices within h-hops away from v, after the Decay Rule is finished for each vertex of G. For each received signature at v, we merge them with v's signature, and return KFS^1 of v finally. The procedure Decay_KFS() constructs KFS^2 based on the Diffusion Rule and consists of two phases. The first phase is to send the local signature of v to vertices h-hops away from v and merge the received signatures at v. The second phase is to decay the signature of v obtained in the first phase as the Decay Rule. Finally we can return the KFS^2 of v.

Each vertex of G has two signatures with the same size n. The signatures are highly compressed though they encode a great deal of keyword information of G. Fig. 5 illustrates the process of constructing KFS^1 and Fig. 7 shows the process of building KFS^2.

Algorithm 2 gives the procedure of the signature-based search algorithm (Fig. 10). As introduced in the framework, this procedure is performed at each slave and is to compute answer trees across different slaves. Recall that the signature-based search starts from partial trees $\{T_p\}$ and tries to discover complete crossing answer trees. Specifically, we first determine *valid* partial trees that should comply with the Decay and Diffusion Rules (Line 3). The number of valid partial trees is much smaller than that of $\{T_p\}$, since valid partial trees consists of the paths with the maximum $\Theta^1(q)$ and $\Theta^2(q)$ from keyword nodes, whereas $\{T_p\}$ are determined using the flooding search. Consequently, valid partial trees will incur much less traffic costs than $\{T_p\}$ that is used in the naive search algorithm.

After that, the signature-based search starts from the boundary node of each valid partial tree and tries to discover crossing answer trees (Line 5). During each superstep, the current node u forwards the search to *one* in-neighbor v with two different conditions (Lines 7–10). When u is a boundary node, v is its remote in-neighbor. If both conditions are not satisfied, node u can be a candidate root if its answer tree is crossing different slaves. The two conditions are based on the Decay and Diffusion Rules as discussed above. Note that we should *synchronize* the distributed search at all slaves, so that we can obtain the same number of supersteps s for different slaves to examine the halting condition $s > l_w$. With the aware neighborhood extending throughout the graph, and a linear increasing number of bits pointing towards the root of an answer tree, the search will follow the shortest path to the root node.

The neighbor forwarding scheme (given in Neighbor_Forward()) can be seen as a simple greedy strategy that is processed as follows. For each in-neighbor x of current node u, query keywords in $q = \{b_i\}$ are looked up in the KFS associated with x (Lines 2-5). The result of this lookup is an indicator θ_x, which is the total number of bits set to 1 in locations indexed by $h_j(b_i)$, $j \in 1...c$ (Line 4). The search is forwarded to the in-neighbor with the highest indicator value (Lines 6-7). In every forwarding, we send the search to only one in-neighbor so that the search will reach root node through one path. Obviously, Algorithm 2 avoids the flooding search as in the naive search algorithm. Therefore, our solution is more efficient and cost-effective than the naive search algorithm.

4.3 Algorithm Analysis

Approximation ratio: The studied problem in this paper is the Group Steiner Tree problem which is NP-complete [16]. Thus, we first give the approximation ratio of our solution. As introduced above, the signature-based search is a backward search that runs shortest-path algorithms from keyword nodes to discover the root. Therefore, the **approximation ratio** of our solution is $O(|q|)$ [17].

Next, we give time and communication complexities of our solution, respectively.

Let d be the average vertex degree of G, Δ be the maximum vertex degree, $T_k = \langle r_k, (v_1, ..., v_w)\rangle$ be kth query answer tree and l_w be the length of the longest path from r_k to v_i ($1 \leq i \leq w$). Let λ_i be the number of keyword nodes for keyword b_i where $1 \leq i \leq w$. Assume that G is randomly distributed to m slaves. Below we give time and communication complexities, respectively.

Time Complexity: Since the signature-based search is performed in parallel and synchronized at different slaves, the longest superstep performed by each slave is l_w. In every superstep, the time cost taken in a slave depends on the number of nodes visited in this slave during the signature-based search. Let A_i be the number of nodes that perform the signature-based search in ith superstep and $A = \sum_{i=1}^{l_w} A_i$, i.e., A is the total number of nodes performing backward search in l_w steps of Algorithm 2. We first analyze how many nodes of A_i are distributed to a slave SL. This follows easily from a direct Chernoff bound application. Since each vertex of G is mapped independently and uniformly to the set of m slaves, the expected number of nodes mapped to a slave is $O(A_i/m)$ w.h.p. The concentration follows from a standard Chernoff bound [18]. Then, for every i, each slave incurs $O(A_i/h)$ time costs. Since Algorithm 2 takes at most l_w supersteps, it follows that Algorithm 2 takes $O(\sum_{i=1}^{l_w}\lceil A_i/h\rceil) = O(\sum_{i=1}^{l_w}(A_i/m + 1)) = O(A/m + l_w)$ time costs. Let $\Lambda = \sum_{i=1}^{w} \lambda_i$. During Algorithm 2, a search starts from a keyword node and visits other nodes along *only one path* until Algorithm 2 is finished. Thus, at most Λ nodes are expanded in ith superstep and at most $\Lambda \times l_w$ nodes are visited during the whole search, i.e., $A = l_w \times \Lambda$. To this end, the time complexity of the **signature-based search algorithm** is $O(\frac{l_w \times \Lambda}{m} + l_w)$. Similarly, we obtain that the time complexity of the **naive search algorithm** is $O(\frac{d^{l_w} \times \Lambda}{m} + l_w)$.

Communication Complexity: To analyze communication costs, we consider the number of crossing edges of a slave, since a slave receives messages from crossing edges. Fix a slave SL. Let A be the set of nodes performing the signature-based search in ith superstep and B be the set of edges induced by the nodes of A. Assume that $|A| = a$ and $|B| = b$. Let random variable X_i^s be defined as follows: $X_i^s = d(v_i)$ ($d(v_i)$ is the degree of node v_i in A) if v_i is assigned to slave SL, otherwise $X_i^s = 0$. Let $X^s = \sum_{i=1}^{a} X_i^s$ denote the total degree of the nodes assigned to slave SL; in other words, it is the total number of edges that have at least one end-node assigned to slave SL. We have $E[X_i^s] = d(v_i)/m$ and $E[X^s] = \sum_{i=1}^{a} E[X_i^s] = \sum_{i=1}^{a} d(v_i)/m = 2b/m$. Furthermore, $Var(X_i^s) = E[(X_i^s)^2] - E[X_i^s]^2 = \frac{1}{m}(d(v_i))^2 - (\frac{d(v_i)}{m})^2 = \frac{(d(v_i))^2}{m}(1 - 1/m)$ and hence $Var(X^s) = \sum_{i=1}^{a} Var(X_i^s) = \frac{1}{m}(1 - \frac{1}{m})\sum_{i=1}^{a}(d(v_i))^2 \leq \frac{1}{m}(1 - \frac{1}{m})\sum_{i=1}^{a} \Delta d(v_i) = \frac{1}{m}(1 - \frac{1}{m}\Delta b)$. By putting these results into Bernstein's inequality, we have $X_p < \Phi$ with probability at least $1 - 1/a^2$, where $\Phi = c(b/m + \Delta)$ for some large constant $c > 0$. Let random variable Z^s denote the number of crossing edges of SL. We have $E[Z^s|X^s] \leq \lceil \frac{X^p}{m-1} \rceil$. The edges mapped to SL are crossing edges independently, and hence we can apply a standard Chernoff bound for 0-1 random variables [18]: $Pr(Z^s < c'\Phi/m + 1) \leq Pr(X^s > \Phi) + Pr(Z^s < c'\Phi/m + 1|X^s < \Phi) \leq 1/a^2 + 2^{-c'\Phi/m+1} \leq 1/a^2 + 2^{-3loga+\Omega(1)} = O(1/a^2)$, for a sufficiently large constant $c' > 0$.

Thus, whp, the number of crossing edges assigned to SL is $c'\Phi/m = O(b/m^2 + \Delta/m + 1)$. Similar to the time cost, b is a linear of l_w and thus the communication cost of the **signature-based search algorithm** is $O(l_w \times \Lambda/m^2 + \Delta/m + 1)$. Similarly, we obtain that the communication complexity of the **naive search algorithm** is $O(d^{l_w} \times \Lambda/m^2 + \Delta/m + 1)$.

Compared to the exponential time and traffic costs of the naive algorithm, the time and traffic costs of the signature-based algorithm are greatly improved to be linear costs.

5 Discussion

5.1 Determining Parameters

The signature-based algorithm uses a number of parameters. In this subsection, we discuss how to set up these parameters to improve the query quality.

Given a vertex u of G, we encode the keywords of $KW(u)$ into a bitstring with length n and set c out of n bits to be 1. In the encoding, we can use string hash functions, such as BKDR and AP [18], [19]. Based on such transformation, we obtain u's initial keyword forwarding signature (KFS). Then, we can obtain KFS^1 and KFS^2 by the Decay and Diffusion Rules. In the process of decaying and diffusing KFS, it produced false positives for KFS caused by the collisions of different hash functions. Clearly, the parameters n and c are related to the false positive, of a KFS, for which it suggests that a keyword is in the KFS even though it is not. The reason is that, when inserting a new keyword in the KFS or merging with other KFSs, all bits were previously set to 1 by other keywords.

For a vertex u of G, let $|KW(u)| = s$. Below we calculate the false positive probabilities of KFS^1 and KFS^2, respectively.

As shown in the Decay Rule, KFS^1 is obtained by constructing and decaying a KFS from u, respectively.

For constructing a KFS, let p be the probability that a random bit of the KFS is 0, and assume that s keywords have been added to the KFS. Then, $p = (1 - 1/n)^{s\times c} \approx e^{-s\times c/n}$ as $c\times s$ bits are randomly selected, with probability $1/n$ in the process of adding each keyword. We use α_i to denote the false positive probability caused by the new keyword insertion, and we have the expression:

$$\alpha_i = (1-p)^c \approx (1 - e^{s\times c/n})^c. \tag{1}$$

According to the Decay Rule, after decaying a KFS to one-hop neighbor, the "1" bits of the KFS are reset to "0" bits with a probability $f/\theta(KS)$. Let θ_i denote the number of "1" bits in the KFS decayed to vertex (v) i hops from u. Then, $\theta_i = \theta(KS) - i \times f$. The false positive probability of the KFS decayed from u to i-hops neighbor v is,

$$\beta_i = \alpha_i \prod_{j=1}^{i} (1 - \frac{f}{\theta(KS) - j \times f}) \tag{2}$$

The received KFSs by vertex u are recorded as KS_i^l where $1 \leq i \leq h$ and $1 \leq l \leq (d-1)^{i-1}$. Similarly, we can denote the false positive probability given in Equ. 2 as β_i^l, where $1 \leq i \leq h$ and $1 \leq l \leq (d-1)^{i-1}$. Thus, the false positive probability over the merged KS_i^l signatures can be computed as:

$$\gamma_u^1 = 1 - \prod_{i=1}^{h} \prod_{l=1}^{(d-1)^{i-1}} (1 - \beta_i^l) \tag{3}$$

By substituting Equ. 2 into Equ. 3, we obtain the false positive probability of KFS^1 of vertex u:

$$\gamma_u^1 = 1 - \prod_{i=1}^{h} \prod_{l=1}^{(d-1)^{i-1}} [1 - (\alpha_i^l \prod_{j=1}^{i} (1 - \frac{f}{\theta(KS_i^l) - j \times f})] \tag{4}$$

As shown in the Diffusion Rule, KFS^2 is obtained by diffusing and decaying a KFS from u, respectively.

In the diffusing, every vertex diffuses its local signature within h-hops. Due to the symmetry, a vertex u receives KS_i^l signatures, where $1 \leq i \leq h$ and $1 \leq l \leq (d-1)^{i-1}$. Denote α_i^l by the false positive probability of KS_i^l for every vertex, where α_i^l is given by Equ 1. The false positive probability of the merged KS_i^l signatures at vertex u is:

$$x_u = 1 - \prod_{i=1}^{h} \prod_{l=1}^{(d-1)^{i-1}} (1 - (\alpha_i^l) \tag{5}$$

In the decaying, similar to KFS^1, the false positive probability of the signature decayed from u to a i-hops neighbor becomes,

$$y_i = x_u \prod_{j=1}^{i} (1 - \frac{f}{\theta(KS) - j \times f}) \tag{6}$$

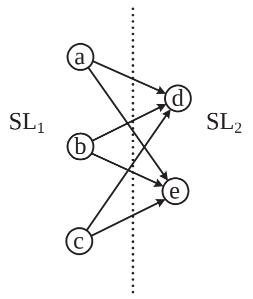


Fig. 11: Idea of the graph clustering-based algorithm.

After merging the decayed signatures at u, the false positive probability of KFS^2 at u is computed as:

$$\gamma_u^2 = 1 - \prod_{i=1}^{h} \prod_{l=1}^{(d-1)^{i-1}} (1 - y_i^l) \tag{7}$$

By combining Equ. 4, 7 and the condition $c > hf$ in the Decay Rule, we can determine the parameters. For example, if we wish the false positive probabilities of KFS^1 and KFS^2 to be no larger than 1×10^{-3}, the parameter setting can be tuned to ensure the hold: $\gamma_u^1 < 1 \times 10^{-3}$, $\gamma_u^2 < 1 \times 10^{-3}$ and $c > hf$. For the DBLP dataset shown in the experiment, we set $c = 32$, $f = 4$ and $h = 5$. In this case, we have $\gamma_u^1 < 1 \times 10^{-3}$ and $\gamma_u^2 < 1 \times 10^{-3}$.

5.2 Extension on Answer Graph Semantics

This paper uses a tree structure to define an answer. However, some works give answers which are graph structures, since the graph-based answers provide more compact information than the tree-based answers [6], [15]. Due to the limited space, in the Appendix, we discuss how to extend the proposed solutions to the answer graph semantics.

5.3 Updating Signature

Keywords contained by the graph data are not static and are updated regularly. Due to the limited space, the detail of the updating scheme and the experiment about the scheme are given in the Appendix.

6 Distributed Graph Clustering

As introduced in the Decay and Diffusion Rules, every vertex of G performs a breadth-first search (BFS) by h-hops. Since G is very big and distributed, such massive BFSs will incur an extremely large number of communication costs. In this section, we propose a graph clustering-based algorithm to greatly reduce the network overhead.

Below, we give an example to show the main idea of the graph clustering-based algorithm.

Example 4: Fig. 11 shows a bipartite graph $B = (X, Y)$ distributed two slaves SL_1 and SL_2. One part X of B (i.e., nodes a, b and c) is at SL_1 and the other part Y of B (i.e., nodes d and e) is at SL_2. In the distributed B, a, b and c have common remote neighbors d and e. In the Decay and Diffusion rules by Algorithm 1, a sends its signature to d and e through two messages. Similarly, either b or c sends its signature to d and e by two messages. Thus, there are total six messages incurred between SL_1 and SL_2. In the graph clustering-based algorithm, all nodes a, b, c, d and e are grouped into one cluster C, i.e., B is viewed as a cluster. After that, in the Decay and Diffusion Rules, X sends Y one message including all the signatures of a, b and c. This example demonstrates that the graph clustering-based algorithm needs only *one* message instead of *six* messages as needed in Algorithm 1.

From the example, we observe one general principle of the graph clustering-based algorithm: *Vertices sharing similar remote neighbors should be processed together in the Decay and Diffusion Rules*.

To achieve the principle, we first define an augmented graph for each slave, and then propose the graph clustering-based algorithm based on the augmented graph. Given a graph $G = (\mathcal{V}, \mathcal{E})$ distributed slaves, we define the augmented graph of a slave as follows.

***Definition** 6.1 (Augmented Graph):* An augmented graph on a slave SL is a bipartite graph $B(V, E)$, where $V = X \cup Y$ and X are local vertices in SL and Y are remote neighbors of X (more precisely, $Y = \bigcup_{v \in X} Neg(v) - X$), and $E = X \times Y \cap \mathcal{E}$.

Given an augmented graph $B(X \cup Y, E)$ and $X' \subseteq X, Y' \subseteq Y$, we use $e(X', Y')$ to denote the number of edges between X' and Y'. Then, we can define the graph clustering based on the augmented graph.

***Definition** 6.2 (Graph Clustering (GC)):* Given an augmented graph $B(X \cup Y, E)$, we aim to find a partitioning $C_1, C_2, ..., C_k$ on B, such that

- (1) each $C_i = (X_i \cup Y_i, E_i)$ is a bipartite graph with $X_i \subseteq X$, $Y_i \subseteq Y$, and
- (2) $\sum_i [e(X_i, Y_i) - |Y_i|]$ is maximized.

In the GC-based algorithm, we hope to maximize the number of remote accesses that is saved for the Decay and Diffusion Rules. When we process each X_i, Y_i needs to be retrieved at least one time and overall $[e(X_i, Y_i) - |Y_i|]$ communications are saved. Since $\sum_i |Y_i| = |Y|$ is fixed for a slave, the objective is changed to maximize $\sum_i e(X_i, Y_i)$.

Thus, in Definition 6.2, we observe that graph clustering is related to the graph partitioning problem. More precisely, it is the *weighted balanced graph partitioning* problem defined as follows.

***Definition** 6.3 (Weighted balanced graph partitioning):* (BGP) Given a vertex-weighted graph $G(V, E, w)$ and an integer k, where $w(v)$ ($v \in V$) is the vertex weight, find a partitioning $S_1, S_2, ..., S_k$ on V such that

- (1) Balance: for any $i \neq j$, $\sum_{v \in S_i} w(v) = \sum_{u \in S_j} w(u)$;
- (2) Quality: $\sum_{i,j} |e_{ij}|$ is minimized, where e_{ij} is the number of edges between S_i and S_j.

To connect BGP to GC, we use the balance constraint of BGP for an augmented graph $B = (X, Y)$. Specifically, for each vertex in X and Y, we set its weight as zero. Then, the balance constraint of BGP can be neglected in GC. The quality constraint of BGP is to minimize crossing edges between clusters, which equals to maximize the inner edges of a cluster, i.e., the second condition of GC.

Now we can solve the GC problem by a solution to BGP. The BGP problem is a well-known NP-hard problem [16]. Many effective and efficient solutions have been proposed. Among them, METIS is the state-of-the-art solution [20]. METIS can produce good partitioning results and can scale up to large graphs with billions of nodes. Thus, we adopt METIS to solve the GC problem. For each obtained cluster $C_i = (X_i, Y_i)$, we send *one message* from vertices of X_i to their common remote neighbors of Y_i in the Decay and Diffusion Rules.

Finally, we are left an issue that the number of partition in GC (i.e., k) should be pre-determined. We solve the issue by setting $k = |X|/M$, where M is the memory size of the slave containing X. We set the value $k = |X|/M$ due to the following considerations. Even after distribution of G, on each slave the local graph may be still too big to be completely resident in memory. Hence, in general the big graph resides on the disks and needs to be scheduled into memory when needed. When the graph is big enough, we can only schedule a part of local vertices X into memory for computation. As a result, local vertices X logically can be partitioned into disjoints parts $(X_1, X_2, ..., X_k)$ ($k = |X|/M$) to be scheduled into the memory.

TABLE 2: Real Datasets

Graph	DBLP	IMDB	Freebase	Friendster
Vertex Count	1,356K	1673K	40.3M	65.6M (million)
Edge Count	3158K	6075K	180M	1.8B (billion)
Keyword Count	1185K	1748K	72.6M	92.5M (million)
Query Keyword Count	1–6, **3**	1–6, **3**	1–8, **3**	1–10, **5**

TABLE 3: Synthetic Datasets

Vertex Count	4M	16M	**64M**	256M	1024M	4096M
Average Degree	10	20	**30**	40	50	60
Query Keyword Count	1–6, **3**	1–6, **3**	1–8, **3**	1–10, **4**	1–10, **4**	1–10, **4**

7 Performance Evaluation

In this section, we report the efficiency of test results of our newly proposed techniques. Our methods are implemented on a cluster of 17 machines in a high speed kilomega network, where one machine is selected as the Master and the remaining machines are selected as Slaves. The default number of slaves is 16. Each machine has 2 Intel Xeon E5345 CPUs, 32GB memory and is running CentOS Linux 5.6. All programs are coded in Java.

Experimental setting. In the experiments, we use real and synthetic datasets.

(1) Real dataset. We use four real life datasets: DBLP, the Internet Movie database (IMDB), Freebase and Friendster that have been widely used in the examination of keyword graph search [2], [21]. Table 2 gives the parameters of four graphs. As shown in the table, the size of the datasets varies widely: Friendster is more than 48 times larger than DBLP, and IMDB and Freebase lie in between.

(2) Synthetic dataset. We construct synthetic graphs using the R-MAT graph generator in GT-Graph software [22] which has been widely used to generate power-law, community-structured graphs [23]. The default values for vertex count and average degree are 64M and 30, respectively. We generate two groups of data graphs by varying these two parameters. (a) Varying vertex count: we generate

TABLE 4: Determining parameters of the signature-based search algorithm.

Parameters	h	f	c	n	$(\gamma_u^1, \gamma_u^2) \times 10^{-3}$
DBLP	5	4	32	2000K	(0.94, 0.93)
IMDB	5	4	32	3000K	(0.93, 0.91)
Freebase	5	8	128	50M (1000K)	(0.96, 0.93)
Friendster	5	8	128	50M (1000K)	(0.95, 0.93)

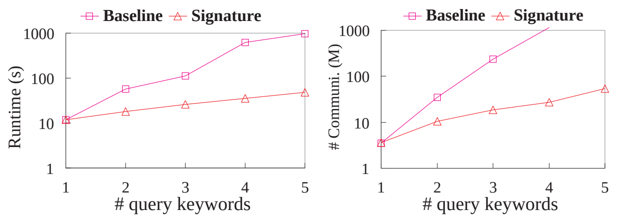
6 data graphs by varying the vertex count between 4M and 4096M. (b) Varying average degree: we generate 6 data graphs by varying the average degree between 10 and 60. For each graph, we use the tool in [24] to generate keywords and assign them uniformly to the vertices of the graph. Table 3 gives the detailed settings of the datasets.
(3) Query generator. We generate query workload from each dataset as the benchmark method in [6], [21]. Table 2 provides the range in number of query keywords for each real dataset and the default value (in bold). Table 3 shows the query settings for synthetic datasets. The value of top-k is set to 10–50, and the default one is 30. According to statistics in [6], [21], $h = 5$ is large enough to obtain satisfactory answers in real applications. Thus, we set $h = 5$. In each experiment, we run 50 queries and report the average result.
(4) Algorithms. We evaluate the following two algorithms. (a) Baseline: The naive search algorithm introduced in Section 1; that is, a distributed backward search using a flooding strategy. (b) Signature: The signature-based search algorithm proposed in Section 4; that is, a distributed backward search guided by vertex signatures. To compare with existing approaches, we implement the algorithms KWS [25] and RDF [2], respectively. KWS is a keyword search algorithm that adopts two atomic operations designed for MapReduce [25]. RDF is a keyword search algorithm over RDF graphs and supported by the path index designed for MapReduce [2].

In the experiments, we evaluate the *total processing time* and *communication costs* of these algorithms. We use the number of communication messages to measure communication costs. The total messages include messages between the master and the slaves, as well as the messages between the slaves.

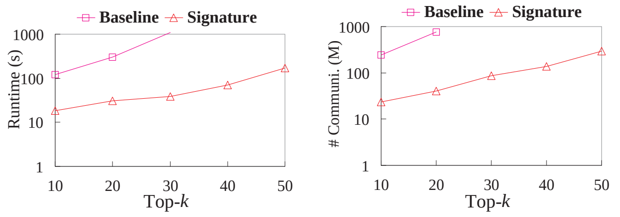
Determining parameters used in the algorithm. The signature-based search algorithm needs some parameters to determine in advance, i.e. length limit h, decay factor f, number of hash functions c and the size of each vertex signature n. As analyzed in Section 5.1, these parameters affect the query quality of Signature, i.e., the false positive probabilities γ_u^1 given by Equ. 4 and γ_u^2 given by Equ. 7. Theoretically, if we wish to upper bound the false positive, we can use Equ. 4 and 7 to determine the parameters. For example, we wish $\gamma_u^1 < 10^{-3}$ and $\gamma_u^2 < 10^{-3}$ in the experiments. Table 4 shows the parameters set to each real dataset, based on which the false positive probabilities are upper bounded by 10^{-3}.

Experimental results on real datasets.
Results w.r.t. Query keyword count: We examine the total query processing time and network costs with respect to query keyword count on the Freebase graph in Fig. 12. As shown in Fig. 12(a), the processing time of Signature grows slowly while Baseline increases very rapidly. The increase of Signature is proportional (linear) to the increase of query keyword count, but Baseline is not completely exponential to the keyword number. This is because: the flooding strategy adopted by Baseline leads to a more rapid search as the keyword number increases, but Signature explores only one path from each keyword node which incurs a linear time cost. An interesting observation is that Signature and Baseline has the same running time at the keyword count of 1, owning to that they both start the backward search from keyword nodes (the keyword count is 1).

Similar to the time costs, the communication costs (in Fig. 12(b)) of Signature and Baseline rise with the increasing of keyword number, but the trend of the rising of communication costs is more obvious than that of time costs. Baseline grows exponentially with respect to the increasing of keyword number, since the flooding nature of Baseline has a great influence on the communication costs. By comparison, the trend of the rising of Signature is even more smooth, due to that Signature incurs little network overhead by traversing few paths. The trends behaved in this experiment are consistent with the theoretical analysis results for Signature and Baseline.
Results w.r.t. Top-k: We examine the total query processing time and communication costs with respect to values of top-k on the Friendster graph in Fig. 13. As reported in Fig. 13(b), the time costs grow with the increase of k due to that more query answers need more computing time. Signature is very efficient, e.g., using 37 seconds to search the Friendster graph with 65 million nodes and 18 billion edges. This is reasonable because 1) the computation



(a) Query processing time (b) Communication costs
Fig. 12: Performance on the Freebase graph with respect to # query keyword.



(a) Query processing time (b) Communication costs
Fig. 13: Performance on the Friendster graph with respect to top-k.

complexity of Signature grows polynomially with k; 2) Signature processes the query with short time due to the help of vertex signatures, which saves much meaningless exploration.

The traffic costs (in Fig. 13(b)) of Signature increase smoothly as k gets larger, and Signature beats Baseline significantly at the network overhead. This is reasonable since more answers increase the graph exploration and since Baseline has to do much useless inter-slave message exchange which can be successfully avoided by the vertex signatures used in Signature. For example, in Fig. 13(b), when we return top-20 answers, Signature incurs 40M communication costs while the value of Baseline has approached 1000M.

Performance of the signature construction: We present the running and communication costs of signature construction with respect to decay and diffusion hops in Fig. 14. Either total running or traffic costs include those incurred in both Decay and Diffusion Rules. In the examination, we compare two algorithms, the graph clustering (GC) and the flooding (FL), at the Freebase and DBLP graphs. As shown Fig. 14, GC incurs much less running and communication costs than FL at both datasets. The main reason is that GC packets all the signatures of a cluster into one communication, whereas FL communicates with each remote neighbor by one packet. Thus, the superiority of GC PK FL at the network overhead is more obvious than that at the time overhead. On the other hand, in Fig. 14, we observe that both running time and communication costs grow dramatically with respect to decay and diffusion hops in the early stage, and begin to increase more and more slowly after the decay and diffusion hops are larger. This is because many vertices covered by a long decay or diffusion path have received signatures from vertices with short decay or diffusion paths.

Experimental results on syntectic datasets. Performance w.r.t. Vertex count: We demonstrate the total query processing time and communication costs with respect to vertex count on syntectic datasets. From the result of running time in Fig. 15(a), we can see that as the graph size increases, the processing time cost of Signature increases smoothly, which makes sense because in larger graphs longer time is needed during the search. The running time of Signature at the billion node graph is within 4 minutes, but Baseline cannot terminate at five hours. The similar experimental result in Fig. 15(b) is obtained at the network overhead of Baseline and Signature, since more partial trees are exchanged between slaves and Signature can handle the large volume exchanges easily. From the experiments we can conclude that our proposed approach has a good scalability for big graphs.

Performance w.r.t. Vertex degree: We show the total query processing time and traffic costs with respect to vertex degree on syntectic datasets. As vertex degree increases, the graph becomes more dense and query length is shorter. This is why the running time shown in Fig. 16(a) first increases and then becomes smooth. As shown in Fig. 16(b) for traffic costs, the curve of Baseline increases rapidly as vertex degree rises, which makes sense since more dense graphs have more paths traversed in the search algorithm. However, the signature-based solution grows linearly, thus Signature scales well as graph becomes more dense.

Performance w.r.t. Slave number: We test the total query processing time and traffic costs with respect to slave number. Generally, the query processing time is reduced by adding more slaves, as illustrated in Fig. 17(a). However, the decrease of time cost is not completely proportional to the increase of slave number. We find two reasons: 1) the loads of the search task are not balanced on all slaves, and 2) all global top-k answers are determined on the master. As illustrated in Fig. 17(b), the traffic cost increases with the slave number. The reason is that more synchronization operations will be incurred with more slaves.

Performance of the graph clustering: We examine the query processing time and traffic costs over the graph partition based on the hashing and clustering algorithms.

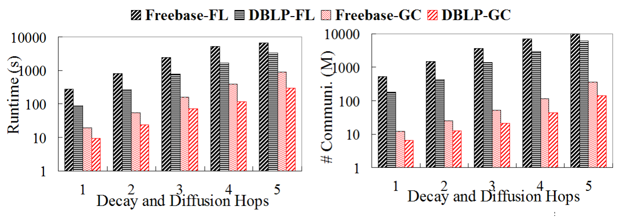
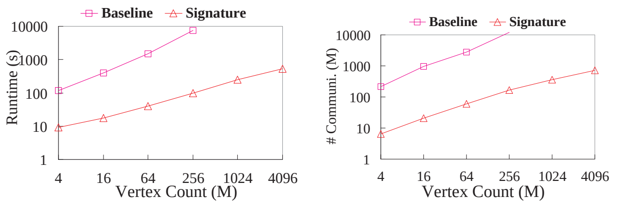
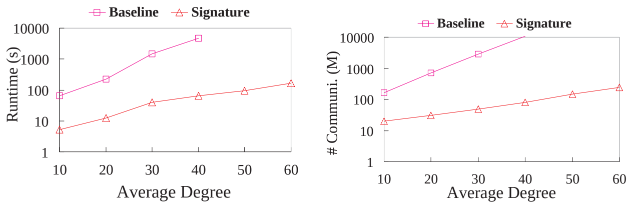


Fig. 14: Performance of the signature construction.



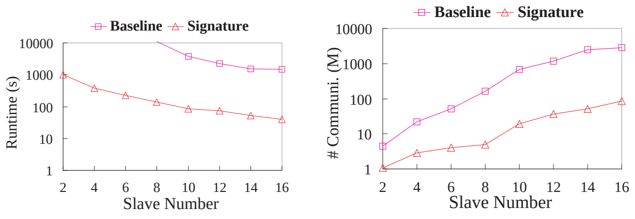
(a) Query processing time (b) Network overhead

Fig. 15: Scalability with respect to vertex count.



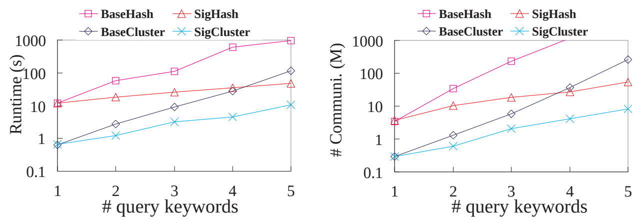
(a) Query processing time (b) Network overhead

Fig. 16: Scalability with respect to vertex degree.



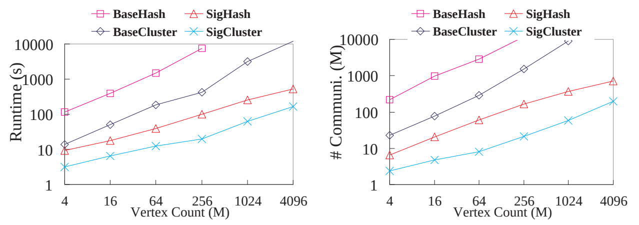
(a) Query processing time (b) Network overhead

Fig. 17: Scalability with respect to # slaves.



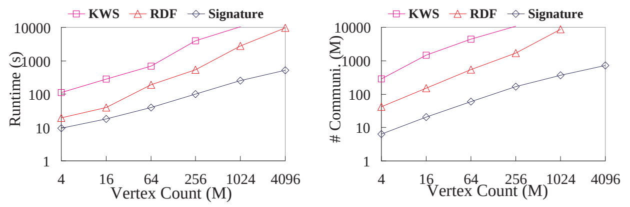
(a) Query processing time (b) Communication costs

Fig. 18: Comparison of the hashing-based and clustering-based graph partitioning on the real dataset.



(a) Query processing time (b) Communication costs

Fig. 19: Comparison of the hashing-based and clustering-based graph partitioning on the syntectic dataset.



(a) Query processing time (b) Communication costs

Fig. 20: Compared with the existing algorithms.

The baseline search algorithm over the hashing and clustering graph partitions are named BaseHash and BaseCluster, respectively. Similarly, the signature-based search algorithm is named SigHash and SigCluster, respectively. Fig. 18 gives the results on the Freebase graph with respect to keyword counts. As shown in the figure, both BaseCluster and SigCluster incur even less query time and traffic costs than BaseHash and SigHash. The reason is that the clustering-based graph partition is more balanced and has fewer crossing edges than the hashing-based graph partition.

Fig. 19 reports the results on the syntectic graphs with respect to vertex counts. After the graph clustering, the speedup is over 10 times compared to the search algorithm based on the graph hashing. The clustering algorithm beats the hashing algorithm, since the search algorithm packets all the information of a cluster into one communication, whereas the search algorithm over the hashing partition communicates with each remote neighbor by one packet.

Compared with the existing algorithms: Finally, we give a comparison of the query processing time and traffic costs among Signature, KWS and RDF. KWS and RDF are MapReduce-based search algorithms, and we deploy them on the same running environment as Signature. Fig. 20 reports the results of these algorithms regarding vertex count. From the results, we observe that Signature is the most efficient, KWS is the slowest and RDF lies in between. For example, Signature completes query processing within 3 minutes on the 1 billion node graph. However, RDF almost takes more than 50 minutes to process the 1 billion node graph and KWS did not terminate within 3 hours at the same size graph.

Essentially, KWS is just a breadth-first search and similar to the baseline algorithm, and hence it incurs a large volume of query time and traffic costs. Though RDF is supported by the path index constructed over MapReduce, it needs many MapReduce rounds to obtain query answers and in each round it produces a large number of intermediate results that are iterated into the next round.

8 Related Works

One most related work to ours is centralized keyword search over graphs, that is, given a set of keywords, we are required to return top-k subtrees, subgraphs or h-cliques that contain these keywords [1], [2], [4], [15]. In brief, finding the exact top-k subtrees is an instance of the group Steiner tree problem [16], which is NP-hard. To find top-k subtrees, [4] proposed backward search in BANKS-I, and [26] utilized the power of database and backward search to find answer trees. More detailed works can be found in a survey of graph keyword search [6]. [27] studied top-k spatial keyword queries on road networks, but with different query aims from this paper. The reason is the following: the query in [27] returns the k best objects ranked in terms of both spatial distance to the query location and textual relevance to the query keywords, whereas our studied query returns k answer trees with the smallest total distances.

There are some works to study the variants of graph keyword search [28], [29]. [28] aimed at finding top-k subtrees from a large uncertain graph. The filtering-and-verification framework is employed to answer the query efficiently. The problem definition and solutions in [28] focus on the probability computation of subtrees over *uncertain* graphs in a *centralized* manner, whereas those in this paper concentrate on the parallel and distributed computation of subtrees over *deterministic* graphs. [29] studied the top-k nearest keyword (k-NK) query over a graph. A k-NK query searches for k nearest answer nodes, each of which contains all the query keywords. In contrast, the query in this paper looks for subtrees, in which leaf nodes jointly contain all the query keywords. Therefore, the studied problems in [28], [29] are different from that in this paper, and their proposed techniques cannot be directed used for solving our problem in this paper. It seems that the reachability index [6] is related to our work. A reachability index is used to answer if two given nodes are reachable in a directed graph. In an answer tree, every leaf node is reachable from the root. However, given leaf nodes, we do not know which vertex of the graph is the root in advance. Therefore, we cannot simply use a reachability index to answer the keyword query.

Another most related works are graph keyword search over MapReduce [30], [31], [32], [33]. [30] studied scalable big graph processing based on two atomic join operations designed for MapReduce. The graph processing supports keyword search that employs the BFS-based backward search algorithm from each keyword node [30]. Thus, the search strategy in [30] is similar to the naive algorithm in this paper and is very inefficient. [32] studied keyword search over a RDF graph via MapReduce. This work supports keyword search by constructing an index consisted of the RDF paths. However, it needs many MapReduce rounds to obtain query answers and in each round it produces a large number of intermediate results that are iterated into the next round. Therefore, as shown in Fig. 20, our proposed solution is more efficient and cost effective than the algorithms in [30], [32]. The works in [31], [33] support the distributed graph keyword search by returning kNN locations and the smallest k-compact tree to the query. Therefore, the studied query in [31], [33] is different from our studied query and the solutions cannot be directed used for solving our problem in this paper.

9 Conclusion

Applications on big graph usually suffer from extremely high space and time complexities, and thus require solutions of the distributed and parallel computing. In this paper, we study the problem of the keyword search over a distributed graph, which returns top-k answer trees. To efficiently find query answers, we give a distributed backward search algorithm and synchronize it over the cluster to terminate the search as early as possible. However, this naive solution incurs massive time and communication costs, due to the flooding nature of the search strategy. To solve this issue, we design a signature-based search mechanism which explores the approach of spreading summary information about the vertex of hosted keywords in its neighborhood, and then uses this information for forwarding the distributed backward search. As a consequence, the signature-based search algorithm determines query answers by traversing only few paths of the graph, so that it costs very low time and network overhead. In addition, we reorganize the distributed graph to optimize the processes of the Decay and Diffusion Rules. Experiments show that our approach can improve the search efficiency by many times faster than the naive algorithm.

References

[1] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding top-k min-cost connected trees in databases," in *ICDE*, 2007.

[2] H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: ranked keyword searches on graphs," in *Proc. of SIGMOD*, 2007, pp. 305–316.

[3] J. Li, C. Liu, and M. S. Islam, "Keyword-based correlated network computation over large social media," in *Proc. of ICDE*, 2014.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using banks," in *Proc. of ICDE*, 2002, pp. 431–440.

[5] J. Li, C. Liu, and J. X. Yu, "Context-based diversification for keyword queries over xml data," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 27, no. 3, pp. 660–672, 2015.

[6] H. Wang and C. C. Aggarwal, "A survey of algorithms for keyword search on graph data," in *Managing and Mining Graph Data*. Springer, 2010, pp. 249–273.

[7] http://www.worldwidewebsize.com/.

[8] http://www.facebook.com/press/info.php?statistics.

[9] http://www.w3.org/.

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

[11] "http://hama.apache.org/."

[12] "http://incubator.apache.org/giraph/."

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

[14] "http://research.microsoft.com/en-us/projects/trinity/."

[15] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *Proc. of SIGMOD*, 2008, pp. 903–914.

[16] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*. W.H.Freeman, 1979.

[17] D. H. (ed.), *Approximation algorithms for NP-Hard problems*. PWS, 1997.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2001.

[19] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120–133, 2010.

[20] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.

[21] J. Coffman and A. C. Weaver, "An empirical performance evaluation of relational keyword search techniques," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 1, pp. 30–42, 2014.

[22] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining." in *SDM*, vol. 4, 2004, pp. 442–446.

[23] C. Aggarwal and H. Wang, *Managing and mining graph data*. Springer, 2010.

[24] R. T. Kasper, "A flexible interface for linking applications to penman's sentence generator," in *Proceedings of the workshop on Speech and Natural Language*, 1989, pp. 153–158.

[25] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan, "Keyword search on external memory data graphs," in *Proc. of VLDB*, 2008.

[26] Q. Lu, X. Y. Jeffrey, and C. Lijun, "Keyword search in databases: the power of rdbms," in *Proc. of SIGMOD*, 2009, pp. 681–694.

[27] J. B. Rocha-Junior and K. Nørvåg, "Top-k spatial keyword queries on road networks," in *Proc. of EDBT*, 2012, pp. 168–179.

[28] Y. Yuan, G. Wang, L. Chen, and H. Wang, "Efficient keyword search on uncertain graph data," *TKDE*, vol. 25, no. 12, pp. 2767–2779, 2013.

[29] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian, "Top-k nearest keyword search on large graphs," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 901–912, 2013.

[30] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in mapreduce," in *Proc. of SIGMOD*, 2014, pp. 827–838.

[31] S. Luo, Y. Luo, S. Zhou, G. Cong, J. Guan, and Z. Yong, "Distributed spatial keyword querying on road networks." in *EDBT*, 2014, pp. 235–246.

[32] R. De Virgilio and A. Maccioni, "Distributed keyword search over rdf via mapreduce," in *European Semantic Web Conference*, 2014, pp. 208–223.

[33] C. Liu, L. Yao, J. Li, R. Zhou, and Z. He, "Finding smallest k-compact tree set for keyword queries on graphs using mapreduce," *World Wide Web*, pp. 1–20, 2015.

Ye Yuan is now a professor at the Northeastern University, China.
Xiang Lian is now an assistant professor at the Kent State University
Lei Chen is now a professor at the Hong Kong University of Science and Technology.
Jeffrey Xu Yu is now a professor at the Chinese University of Hong Kong.
Guoren Wang is now a professor at the Northeastern University, China.
Yongjiao Sun is now an associate professor at the Northeastern University, China.