DKWS: A Distributed System for Keyword Search on Massive Graphs

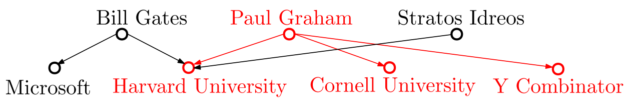
Jiaxin Jiang, Byron Choi, Xin Huang, Jianliang Xu, *Senior Member, IEEE*, and Sourav S Bhowmick

***Abstract*—Due to the unstructuredness and the lack of schemas of graphs, such as knowledge graphs, social networks, and RDF graphs, keyword search for querying such graphs has been proposed. As graphs have become voluminous, large-scale distributed processing has attracted much interest from the database research community. While there have been several distributed systems, distributed querying techniques for keyword search are still limited. This paper proposes a novel distributed keyword search system called DKWS. First, we present a *monotonic* property with keyword search algorithms that guarantees correct parallelization. Second, we present a keyword search algorithm as monotonic backward and forward search phases. Moreover, we propose new tight bounds for pruning nodes being searched. Third, we propose a *notify-push* paradigm and PINE *programming model* of DKWS. The notify-push paradigm allows *asynchronously* exchanging the upper bounds of matches across the workers and the coordinator in DKWS. The PINE programming model naturally fits keyword search algorithms, as they have distinguished phases, to allow *preemptive* searches to mitigate staleness in a distributed system. Finally, we investigate the performance and effectiveness of DKWS through experiments using real-world datasets. We find that DKWS is up to two orders of magnitude faster than related techniques, and its communication costs are 7.6 times smaller than those of other techniques.**

***Index Terms*—.**

I. Introduction

Knowledge graphs, social networks, and RDF graphs have a wide variety of emerging applications, including semantic query processing [48], information summarization [40], community search [14], collaboration and activity organization [36], and user-friendly query facilities [45]. Such graphs often lack useful schema information for users to formulate their queries. To make querying such data easy, *keyword search* has been proposed. Users can retrieve information without the knowledge of the schema or query language. In a nutshell, users only specify a set of keywords Q as their query on a data graph G. In recent years, there have been well-known projects that build graph-structured databases and allow querying with simply a set of keywords, e.g., BioCyc[1] and Google's knowledge graph search API.[2]



Query description: "Who owns *Y Combinator* and graduated from *Havard University* and *Cornell University*?"
Query keywords: {"*Y Combinator*", "*Havard University*", "*Cornell University*"}
Answer: the red substree rooted at "*Paul Graham*"

Fig. 1. Example of keyword search on a knowledge graph

The answer of keyword search semantics (cf. [5], [9], [16], [21], [31], [34], [47]) is generally a set of matches, where each match is a rooted subtree of G such that query keywords belong to the labels of leaf vertices. These semantics differ mainly in the score function of the matches. Interested readers may refer to comprehensive surveys on the keyword search semantics for more information [39], [43], [46]. For example, consider a partial knowledge graph shown in Fig. 1, where a node is an entity and an edge is a relation between entities. Assume that a user is identifying "who owns *Y Combinator* and graduated from *Havard University* and *Cornell University*?". He/She may simply provide the keywords Q={*Y Combinator*, *Havard University*, *Cornell University*} as his/her query. If users apply the keyword search to the knowledge graph, a substree rooted at *Paul Graham* can be returned as an answer (e.g., [9], [16], [31]).

Nowadays, graphs with billions of vertices or edges are common, and their sizes continue to increase. For example, WebUK [3], a large Web graph, contains 106 million nodes and 3.7 billion edges. Keyword search often involves numerous traversals of such massive graphs, which are computationally costly. Indexes (e.g., for shortest distance computations) on such graphs are often large, e.g., $O(|V|^2)$ in the worst case, where $|V|$ is the number of the vertices. Still, it is infeasible to load the index into the main memory, e.g., [16], [22]. As a result, distributed graph processing systems are a competitive solution. In this paper, we aim to propose a *distributed system* to answer the top-k keyword search on distributed graphs. Intuitively, each worker computes local top-k matches on a graph partition and

Manuscript received 29 December 2022; revised 14 July 2023; accepted 26 August 2023. Date of publication 25 September 2023; date of current version 5 April 2024. This work was supported by HKRGC GRF under Grants 12203123, 12201119, 12200022, 12202221, and C2004-21GF. Recommended for acceptance by B. C. M. Fung. *(Corresponding author: Byron Choi.)*

Jiaxin Jiang is with the School of Computing, National University of Singapore, Singapore 119077 (e-mail: jxjiang@nus.edu.sg).

Byron Choi, Xin Huang, and Jianliang Xu are with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong (e-mail: bchoi@comp.hkbu.edu.hk; xinhuang@comp.hkbu.edu.hk; xujl@comp.hkbu.edu.hk).

Sourav S Bhowmick is with the School of Computer Engineering, Nanyang Technological University, Singapore 639798 (e-mail: assourav@ntu.edu.sg).

Digital Object Identifier 10.1109/TKDE.2023.3313726

[1]http://biocyc.org
[2]https://developers.google.com/knowledge-graph/

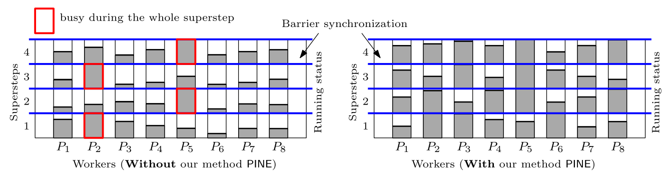

busy during the whole superstep
Barrier synchronization
Supersteps
Running status
Workers (Without our method PINE)
Workers (With our method PINE)

Fig. 2. Illustration of stragglers of distributed keyword search (grey denotes the worker P_i is busy, whereas white color denotes the worker P_i is idle.)

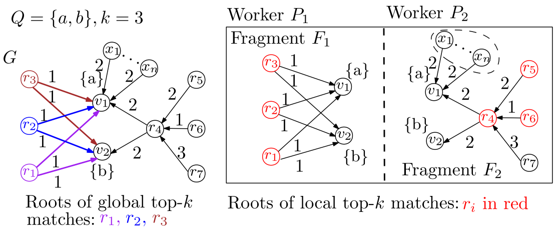

Q = {a, b}, k = 3
Worker P1
Worker P2
Fragment F1
Fragment F2
Roots of global top-k matches: r1, r2, r3
Roots of local top-k matches: ri in red

Fig. 3. Example of distributed keyword search

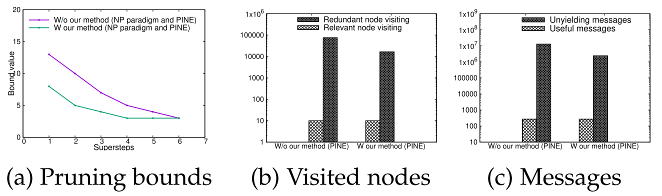


Fig. 4. Illustration of the potential of bound refinements, and messages of distributed keyword search

the global top-k matches are generated from such local matches. However, several major technical challenges of keyword search have not been addressed by existing generic distributed processing systems, e.g., [4], [13], [41].

Challenge 1. Straggler problem: Some workers in a distributed system may take substantially longer than others. We show the working status of supersteps 1-4 of a cluster with 8 workers (P_1 to P_8) as shown in Fig. 2. In each superstep, there are concurrent computation and barrier synchronization. The workers start concurrent computations and become busy (if there is some work) until a barrier synchronization. In the first superstep, worker P_2 take longer than the other 7 workers. They keep waiting idly and the computing power is not used until P_2 completes its tasks. This is known as the *straggler problem*. Similarly, P_2 is also the straggler in third superstep and P_5 is the straggler of second and fourth supersteps. This problem can be caused by either workload imbalance or graph characteristics. Previous studies have frequently focused on rebalancing partitions [8] or predicting machine workloads [10] during runtime. Nevertheless, these approaches come with additional costs, including the expenses associated with data transfer. Moreover, setting up the training model for keyword search is a non-trivial task.

Challenge 2. Lack of pruning techniques: Another challenge is that existing sequential keyword search works often utilize the global graph information (e.g., the upper bound of the score of the top-k matches) to develop some pruning techniques, e.g., [16], to avoid exhaustive traversals on the graph. Consider a graph G in Fig. 3, the upper bound of the score of top-k matches is 2 when the subtrees of G rooted at r_i $(i \in \{1,2,3\})$ are retrieved. r_i $(i \in \{4,5,6,7\})$ and x_i $(i \in [1,n])$ are not traversed. However, in a distributed graph system, such pruning techniques can be hardly directly applied since each machine only maintains a graph fragment. In Fig. 3, the workers P_1 and P_2 process the graph fragments F_1 and F_2, respectively. There are two local upper bounds $S_1 = 2$ and $S_2 = 8$ generated on F_1 and F_2, respectively. The search on F_2 can only be pruned by S_2. We show the refinement of bounds after each superstep in Fig. 4(a). The bound values tighten faster with our techniques. With tighter bounds, unnecessary node visits are significantly reduced, and false matches are pruned early, as shown in Fig. 4(b). Existing research studies such as [16], [22] often rely on indexing distance information for pruning. However, these types of indexing methods are typically designed for single-machine algorithms. Each machine lacks global information, which significantly limits the potential for pruning.

Challenge 3. Message passing: Since keyword search on graphs often involves numerous traversals, keyword search on distributed graphs might cause massive message passing. For instance, in previous studies [47] and [28], the local matches rooted at r_i $(i \in [1,2,3])$ (resp. r_i $(i \in [4,5,6])$) are sent from F_1 (resp. F_2) to the coordinator for verification. The matches on F_2 are not among the final top-k matches, i.e., most of the computation on F_2 does not lead to matches. As shown in Fig. 4(c), the messages not yielding final matches are significantly reduced in our system. Previous works such as [24], [29] have often utilized partitioning strategies to reduce message overhead. However, these studies are designed for general purposes, where the partitioning is primarily based on the graph's structure. In a distributed environment, the message overhead in keyword search often depends on the distribution of the query keywords. This dependency makes these approaches less effective in such scenarios.

Contributions: In this paper, we propose a system for answering top-k keyword search called DKWS and show that all three challenges can be addressed. We investigate keyword search algorithms in a distributed environment and the techniques for DKWS as opposed to individual keyword search semantics (or algorithms).

1) We present the *monotonic* property with the keyword search algorithm which leads to correct parallelization. We show that *a sequential keyword search algorithm* can be rewritten into two main phases, (a) backward keyword search (bkws), and (b) forward keyword search (fkws). We propose new lower and upper bounds for pruning in fkws. We prove that bkws and fkws implemented in DKWS are monotonic.
2) We propose a *notify-push* paradigm for DKWS: (a) each worker *asynchronously notifies* the coordinator when the local upper bound is refined; (b) the coordinator maintains a global bound. When it receives the notification from workers, it refines the global upper bound and *asynchronously pushes* it to all workers. This incurs a small communication overhead, but the refined global bounds

TABLE I
FREQUENTLY USED NOTATIONS

Notation	Meaning
Q	a set of query keywords $Q = \{q_1, q_2, \dots q_l\}$
τ	the threshold of the distance between a distinct root and its leaf nodes
T	a match to query (Q, τ)
$\mathsf{scr}(u)$	the score of a match T rooted at u
bfkws/bkws/fkws	Sequential keyword search/backward search/forward search
$\mathsf{mat}_u/\mathsf{mat}_u^b/\mathsf{mat}_u^f$	the (partial) match found by bfkws/bkws/fkws
$\mathsf{dist}(u, v)$	the shortest distance between u and v
$\mathcal{A}$	the answer, which contains top-k matches
P_0, P_i	P_0: the coordinator; P_i: workers, where $i \in [1, m]$
Par	graph partition strategy
$\mathcal{F}$	fragmentation (*a.k.a.* partition) $\{F_1, \dots, F_m\}$
M_i	messages designated to worker P_i

provide global information to workers to prune some search locally.

3) We propose a PINE *programming model* that naturally fits the keyword search algorithm that has distinguished search phases. DKWS launches a *preemptive execution* of the searches. Hence, keyword searches are no longer one blocking operation in the distributed environment. We propose staleness indicators and a lightweight cost model that mitigate the straggler problem.
4) Using real-life graphs, we empirically compare the performance of DKWS and two baselines. We verify that (a) DKWS speeds up the query performance of top-k keyword search up to two orders of magnitude; (b) The communication cost of DKWS is 7.6 times smaller than that of baseline; and (c) DKWS using all optimizations is on average 1.64 times faster than DKWS without them.
5) Due to space limitations, we put the proofs, some optimizations, and more experiments in a technical report [17].

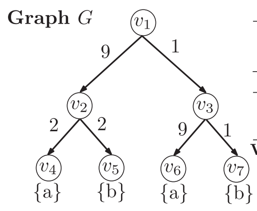
Organization: Section II provides some background and the problem statement. In Section III, we illustrate an efficient monotonic sequential keyword search algorithm. In Section IV, we propose DKWS and its two novel ideas, namely the notify-push paradigm and the PINE model. Section V reports experimental results. Section VI, presents the related work. We conclude the paper and present the future works in Section VII.

II. Preliminaries and Problem Statement

This section presents some background and the problem statement. Some frequently used notations are summarized in Table I.

Graphs: We consider a *labeled, weighted, directed graph* modeled as $G = (V, E, L, \Sigma, w)$, where (a) V is a set of vertices; (b) E ($\subseteq V \times V$) is a set of edges; (c) Σ is a set of keywords; (d) $L : V \to \Sigma$ is a label mapping function such that for each vertex $v \in V$, $L(v)$ maps v to a subset of labels/keywords in Σ; and (e) $w(e)$ is a positive weight of an edge $e = (u, v) \in E$. For simplicity, we may omit L, Σ and w when they are irrelevant to the discussions. The size of the graph is denoted by $|G| = |V| + |E|$.

Example II.1: Consider a graph G in Fig. 5 , a) $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ is the vertex set, b) E is a set of edges, e.g., $(v_1, v_2) \in E$ is an edge, c) $\Sigma = \{a, b\}$ is a set of keywords, d) L maps each vertex in V to a subset of keywords in Σ, e.g., $L(v_4) = \{a\} \subseteq \Sigma$, and e) w maps each edge in E to a positive weight, e.g., $w(v_1, v_2) = 9$.



Vertex set	$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$
Edge set	$E = \{(v_1, v_2), (v_2, v_4), (v_2, v_5),$ $(v_1, v_3), (v_3, v_6), (v_3, v_7)\}$
Label set	$\Sigma = \{a, b\}$
Label function	$L(v_4) = \{a\}$ $L(v_5) = \{b\}$ $L(v_6) = \{a\}$ $L(v_7) = \{b\}$
Weight function	$w(v_1, v_2) = 9$ $w(v_1, v_3) = 1$ $w(v_2, v_4) = 2$ $w(v_2, v_5) = 2$ $w(v_3, v_6) = 9$ $w(v_3, v_7) = 1$

Fig. 5. Example of frequently used graph notations

TABLE II
REPRESENTATIVE KEYWORD SEARCH INVOLVED TRAVERSALS AND SHORTEST DISTANCE COMPUTATION OR ESTIMATION

Keyword search semantics	bkws	fkws	Indexing techniques	
			Exact index	Apx. index
Distinct root trees	[16], [47], [9], [31]	[47]	[16]	[19], [47]
Group Steiner trees	[5], [21]	[5], [21], [34]	[34]	–
Other semantics	[20], [44]	[20], [33]	[20], [22]	[30]

Partition strategy: Given a number m, a strategy Par partitions a graph G into *fragments*$\mathcal{F} = \{F_1, \dots, F_m\}$ such that each $F_i = (V_i, E_i, L_i)$ is a subgraph of G, $E = \bigcup_{i\in[1,m]} E_i$, $V = \bigcup_{i\in[1,m]} V_i$ and $L_i = L$, and F_i resides at worker P_i, where $i \in [1, m]$ is the fragment id. There are two special sets of nodes for each fragment.

- $F_i.I \subset V_i$: the set of nodes $v \in V_i$ such that there is an edge (v', v) *incoming* from a node v' in F_j $(i \neq j)$; and
- $F_i.O$: the set of nodes v' such that there exists an *outgoing* edge (v, v') in E, $v \in V_i$ and v' is in some F_j $(i \neq j)$.

In addition, we denote $\mathcal{F}.O = \bigcup_{i\in[1,m]} F_i.O$, and $\mathcal{F}.I = \bigcup_{i\in[1,m]} F_i.I$. We refer to the nodes in $F_i.I \cup F_i.O$ as the *border nodes* (a.k.a. *portal nodes*) of F_i w.r.t. Par. Partition strategies (e.g., [24]) are orthogonal to our work. In this paper, we utilize the edge-cut partitioning approach, where vertices are assigned to different partitions. As a result, edges may span across two partitions.

Platform: In this work, we propose our system, DKWS, built on top of the code-base of GRAPE [13]. GRAPE exemplifies a generic approach to parallel computations through a programming model that consists of three functions for implementing user-defined algorithms - PEval, IncEval, and Assemble. These functions together form the PIE program paradigm. GRAPE parallelizes the sequential algorithms (and minor revisions are required). GRAPE inherits all optimization strategies available for sequential algorithms and graphs, such as indexing. DKWS inherits the strengths of GRAPE while introducing a novel efficient paradigm PINE (detailed in Section IV) and novel optimizing such as indexing techniques for keyword search.

Semantics of keyword search (kws) *for graphs:* Several keyword query semantics have been proposed, e.g., [16], [22], [47]. They are driven by various interesting applications. We list some representative works of keyword search and their characteristics in Table II. Many of them involve backward search (bkws) and/or forward search (fkws). We consider the same query semantic of [9], [16], [31], which is the most popular semantic among the

TABLE III
IMPORTANT VERTEX SETS FOR THE DISCUSSION OF bkws AND fkws

Datasets	$\|V\|$	$\|E\|$	avg. # of keywords per node
YAGO3	2,635,317	5,260,573	3.79
DBpedia	5,795,123	15,752,299	3.72
DBLP	2,221,139	5,432,667	10
WebUK	133,633,040	5,507,679,822	1

others.[3] A keyword query is a binary tuple (Q, τ) which contains a set of keywords $Q=\{q_1, \ldots, q_l\}$ and a distance threshold τ. Given a graph $G = (V, E)$, a match of Q in G is a subgraph of G, denoted by $T = \{u, \langle v_1, \ldots v_l\rangle\}$, such that (i) T is a tree rooted at u; (ii) $\forall i \in [1, l]$, v_i is a leaf vertex of T and $q_i \in L(v_i)$; and (iii) $\mathsf{dist}(u, v_i) \leq \tau$, where $\mathsf{dist}(u, v_i)$ is the shortest distance between u and v_i. Existing works design indexes for distance estimation/computations. However, the indexes for computing exact matches are large on massive graphs and also non-trivial to be adapted in a distributed environment. On the other hand, the indexes for approximate match computation return bounds for pruning false matches. This work proposes new bounds for pruning false matches and adopts a lightweight index.

Keyword searches can yield numerous matches, particularly within a massive graph. However, users are often concerned with interpreting most compact matches. As such, our focus is on the top-k query that determines the *top-k matches* as the query answers. To facilitate this approach for top-k queries, each vertex can serve as the root match only once. The more compact, the higher the rank. Accordingly, we augment the query structure from (Q, τ) to (Q, τ, k), where τ is the distance threshold between the root vertex and the leaf vertices.

It is well-received that an ideal match is a compact structure that contains all keywords. Hence, existing studies assign a score to each match T, using the root u as a basis. This score is denoted as $\mathsf{scr}(u)$. In this context, a lower score for T signifies a more compact match, considered preferable. Specifically, we employ the same score function as presented in [9], [16], [31]. This function is defined as follows.

Definition II.1 (Score function $\mathsf{scr}(u)$*):* Given a match, $T = \{u, \langle v_1, \ldots v_l\rangle\}$, to the query (Q, τ, k), the *score of* T is denoted by $\mathsf{scr}(u) = \sum_{i\in[1,l]} \mathsf{dist}(u, v_i)$, where $\mathsf{dist}(u, v_i)$ is the shortest distance between u and v_i.

Problem statement: Given a graph G, a keyword query (Q, τ, k), we investigate a distributed system to compute the top-k matches $\mathcal{A}$ (i.e., the answer) of the query on G.

III. BACKWARD AND FORWARD KEYWORD SEARCH

In this section, we discuss the monotonic property of keyword search (kws) algorithms, which is crucial for its correct parallelization [13]. Specifically, backward and forward keyword search (bfkws) consists of two phases, namely, backward keyword search (bkws) and forward keyword search (fkws). Intuitively, bkws starts from the vertices that contain the query keywords and performs a backward search to identify potential vertices that might serve as the roots of a match. fkws initiates its search from these identified roots and proceeds forward. The objective of fkws is to discover any missing keywords within the subtrees that consider these vertices as roots. We prove that both bkws and fkws have the monotonic property (detailed at the end of Sections III-B and III-C, respectively). The monotonic property of a few popular keyword search algorithms, such as [9], [16], [31], can be analyzed similarly, which is omitted.

A. Monotonic Algorithms for Keyword Search

This subsection presents how the keyword search algorithm has the monotonic property. More specifically, the monotonic property is defined with a *partial order* of *match variables* from a *finite domain*. Intuitively, the shortest distance between the root u and a query keyword $q \in Q$, denoted as $\mathsf{dist}(u, q)$ (i.e., $\mathsf{dist}(u, q) = \min\{\mathsf{dist}(u, v)|q \in L(v)\}$), is of a finite domain. When the monotonic property holds, its value decreases or remains unchanged during query processing and converges to the *exact* shortest distance after query processing ends. Before providing further details, we present the structure of the match variable, mat_u, which maintains the substree rooted at u.

Definition III.1 (Match mat_u*):* For a given graph G and a query (Q, τ, k), a match mat_u with its root at u represents a *map*. $\forall q \in Q$, if $\mathsf{dist}(u, q) \leq \tau$, $\mathsf{mat}_u[q]$ is set to $\mathsf{dist}(u, q)$. Otherwise, $\mathsf{mat}_u[q]$ is set to null.

Complete matches and partial matches: $\mathsf{mat}_u[q]$ is initialized to null. Throughout the search process, certain keywords for all u may be discovered within mat_u and $\mathsf{mat}_u[q]$ is set to $\mathsf{dist}(u, q)$, while others may remain null. Formally, a match mat_u is referred to as a *partial match* if and only if $\exists q \in Q$, $\mathsf{mat}_u[q]$ is null (i.e., some keyword is not matched). Otherwise, mat_u is a *complete match*.

Definition III.2 (Monotonic kws *algorithm):* Given a graph $G = (V, E)$, where each node $u \in V$ is associated with mat_u. A *monotonic keyword search algorithm* kws satisfies the following conditions:

1) mat_u of all vertices are in a finite domain; and
2) there exists a partial order $\preceq$ on mat_u such that, $\forall u \in V$, kws updates mat_u in the order of $\preceq$.

We next illustrate the details of the monotonic property of kws in relation to a finite domain and a partial order on the matches.

(1) Finite domain of kws*:* To illustrate a finite domain of match variables, we encode null with a constant large value $+\infty$ larger than $\sum_{e_i\in E} w(e_i)$. Consider the value of $\mathsf{mat}_u[q]$. $\mathsf{mat}_u[q] \in \left\{\sum_{e_i\in E'} w(e_i)|E' \subseteq E\right\} \cup \{+\infty\}$.

(2) Partial order of kws*:* We propose the partial order $\preceq$ on mat_u which is defined as follows. Suppose kws updates the (partial or complete) matches by following an order $\preceq$. If $\mathsf{mat}'_u \preceq \mathsf{mat}_u$, $\mathsf{mat}'_u[q] \leq \mathsf{mat}_u[q]$ or $\mathsf{mat}_u[q] = \mathsf{null}$, then $\preceq$ is a partial order of kws. Intuitively, kws follows the partial order and keeps *refining* the distances between the roots of the matches and the query keywords to obtain the top-k complete matches.

Remarks: A keyword search algorithm kws can be parallelized and terminated with the correct answer (a.k.a. the top-k matches) in a distributed environment if kws is correct for the query Q on a single machine and has a monotonic property. We follow the proof pipeline of Theorem 1 in [13].

(i) Termination: In each superstep, at least one mat_u has to be updated. Given a graph G, the number of distinct values to

[3] According to Google scholar in Jun 2023, the total number of citations of the query semantic [16] received 718 citations.

update mat_u is bounded since all mat_u are in a finite domain and updates follow the partial order $\preceq$. Therefore, the number of supersteps is bounded.

(ii) Correctness: Given that kws is correct for query Q, at the superstep $R = 1$, kws returns a set of correct local matches with roots in each fragment F_i. Matches with roots on portal nodes are passed to their copies in other fragments (if any) at the end of each superstep. At the superstep $R = s$, each node u contains its local match mat_u from the superstep $R = (s-1)$ and the matches mat'_u which are rooted at its copies in other fragments. Therefore, kws can compute the correct match for the current superstep for each node. The correctness of the final matches is thus established by induction on the supersteps.

*B. Monotonic Backward Search (*bkws*)*

We next present the major steps of backward search for keyword search (bkws), which is essential to many previous studies, e.g., [9], [16], [31], [47]. The detailed pseudocode is illustrated with *Lines* 2-22 of Algorithm 2, to be discussed with PINE in Section IV-B. Given a keyword query, bkws goes through three key steps. First, bkws initializes a set of search origins. Second, bkws expands the search origins backward. Third, a complete match is found once the node u is expanded by all search origins. We elaborate the details below.

Answer $\mathcal{A}$: The answer to the query is a set of the top-k matches at the end of the query algorithm. If mat_u is a top-k match, $\mathsf{mat}_u \in \mathcal{A}$. We use S to denote the score of the current k-th match in $\mathcal{A}$. Hence, S is the *upper bound* of the score of any match in $\mathcal{A}$. Given a complete match mat_u, if $\mathsf{scr}(u) > S$, we say that mat_u is a *candidate match*, i.e., it is not among the current top-k matches. Candidate matches may be refined and added to $\mathcal{A}$ by traversing adjacent fragments.

Maintenance of answer: bfkws maintains the top-k matches $\mathcal{A}$ in a priority queue of a fixed size k and is ordered in descending order according to the scores of the matches. The match at the head of $\mathcal{A}$ has the highest priority to be removed as it has the least compact structure. S is initialized to $+\infty$. It remains unchanged when $|\mathcal{A}| < k$. Otherwise, it is always set to the score of the match at the head of $\mathcal{A}$. S will be refined once there is a match found with a score which is smaller than S. Formally, when a candidate match mat_u is refined, the following are checked:

1) if $|\mathcal{A}| < k$, mat_u is inserted into $\mathcal{A}$ directly; and
2) if $|\mathcal{A}| = k$ and $\mathsf{scr}(u) < S$, the match at the head of $\mathcal{A}$ is removed and mat_u is inserted into $\mathcal{A}$.

(Step 1) Initialization: Consider a set of query keywords $Q = \{q_1, q_2, \ldots, q_l\}$. We denote the set of vertices that contain the keyword $q \in Q$ as O_q (a.k.a. search origin), and the set of vertices that could reach q (i.e., one of the vertices in O_q) as V_q.

(Step 2) Backward expansion: bkws expands the vertex set O_q backwardly. In each search step, bkws compares the next vertex to be expanded for each query keyword, and the vertex u with the smallest distance to the search origin is selected. In the expansion, u is added to V_q and mat_u is checked whether it is a complete match, where (u, v) is an incoming edge of v. If (a) $\sum_{q\in Q} \mathsf{dist}(u, q) > S$, where u is the nearest vertex of O_q and has not been expanded by query keyword q (i.e.,

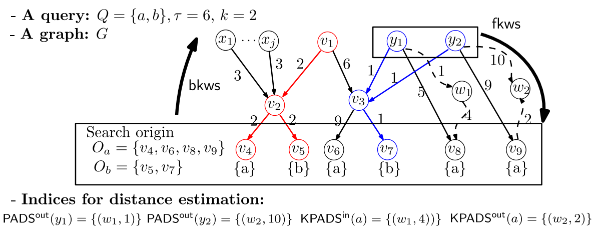

- A query: $Q = \{a, b\}, \tau = 6, k = 2$
- A graph: G
x_1
x_j
v_1
y_1
y_2
fkws
bkws
3
3
2
6
1
1
1
5
10
9
w_1
w_2
v_2
v_3
4
2
2
2
9
1
Search origin
$O_a = \{v_4, v_6, v_8, v_9\}$
$O_b = \{v_5, v_7\}$
v_4 {a}
v_5 {b}
v_6 {a}
v_7 {b}
v_8 {a}
v_9 {a}
- Indices for distance estimation:
$\mathsf{PADS}^{\mathsf{out}}(y_1) = \{(w_1, 1)\}$ $\mathsf{PADS}^{\mathsf{out}}(y_2) = \{(w_2, 10)\}$ $\mathsf{KPADS}^{\mathsf{in}}(a) = \{(w_1, 4))\}$ $\mathsf{KPADS}^{\mathsf{out}}(a) = \{(w_2, 2)\}$

Fig. 6. Query, a data graph (top) and indexes (bottom) for the illustration of the key steps of bkws and fkws

$u_i = \arg\min_u \mathsf{dist}(u, v)$, where $u \notin V_q$ and $v \in O_q$) or (b) all adjacent vertices of V_qs are expanded, the expansion stops. Otherwise, the backward expansion continues.

(Step 3) Match discovery: It discovers a complete match rooted at u such that u can reach at least one node that contains q, for each $q \in Q$, i.e., $u \in \bigcap_{q\in Q} V_q$.

Example III.1: Consider the graph in Fig. 6 . Given a keyword query (Q, τ, k), where $Q = \{q_1, q_2\}$ that is $q_1 = a$ and $q_2 = b$, $\tau = 6$ and $k = 2$. For brevity, Fig. 6 only shows the vertex labels relevant to Q. We illustrate the backward search with Step (a) in Fig. 7 . Initially, $O_a = \{v_4, v_6, v_8, v_9\}$ and $O_b = \{v_5, v_7\}$. The backward expansion iterates over V_a. The first seven vertices are $[v_4, v_6, v_8, v_9, v_2, v_1, w_1]$, which are ordered by the first time the vertices expanded. Similarly, the vertices of V_b can be expanded as follows: $[v_5, v_7, v_3, v_2, y_1, y_2, v_1]$. Two complete matches rooted at v_1 and v_2 are discovered. The score of mat_{v_1} (resp. mat_{v_2}) is $\mathsf{scr}(v_1) = 8$ (resp. $\mathsf{scr}(v_2) = 4$). Hence, the upper bound $S = 8$. The next vertex to expand for V_a is x_1. The next vertex to expand for V_b is x_1, too. $\mathsf{dist}(x_1, a) + \mathsf{dist}(x_1, b) = 5 + 5 = 10 > S$. The subsequent backward expansions, such as $x_i\,(i \in [1, j])$, are skipped since the termination condition is met.

Analysis of bkws*:* We show that bkws identifies all the partial and complete matches. We denote the union (resp. intersection) of V_q by $\mathbb{V} = \bigcup_{q\in Q} V_q$ (resp. $\mathcal{V} = \bigcap_{q\in Q} V_q$). We note that $u \in \mathbb{V} \setminus \mathcal{V}$ reaches some of the query keywords but not all of them, i.e., mat_u is a partial match. We denote the set of roots by $\bar{\mathcal{V}} = \mathbb{V} \setminus \mathcal{V}$ and have the following proposition.

Proposition III.1: The node set visited by bkws, $\mathbb{V}$, has the following properties:

(1) $\forall u \notin \mathbb{V}$, $\mathsf{mat}_u \notin \mathcal{A}$; and (2) $\forall \mathsf{mat}_u \in \mathcal{A}$, $u \in \mathbb{V}$.

Proof: The proof is presented in Appendix A.1 of [17], available online.

Intuitively, if a vertex is not visited during the backward expansion of any query keyword, it cannot serve as the roots of the top-k matches. Proposition III.1 ensures that the roots of the top-k matches are in $\mathbb{V}$. Some vertices in $\mathbb{V}$ that are not roots of the top-k matches will be further pruned in fkws (Section III-C).

Example III.2: We illustrate the key steps of bkws with the graph in Fig. 7(a). $V_a = \{v_4, v_6, v_8, v_9, v_2, v_1, w_1\}$ and $V_b = \{v_5, v_7, v_3, v_2, y_1, y_2, v_1\}$ are expanded, after bkws. $\mathcal{V} = V_a \cap V_b = \{v_1, v_2\}$ are the roots of the complete matches, i.e., they can reach the vertices containing the query keywords $\{a, b\}$. $\mathbb{V} = \{v_4, v_6, v_8, v_9, v_2, v_1, w_1, v_5, v_7, v_3, y_1, y_2\}$ are the vertices which are traversed during bkws. $\bar{\mathcal{V}} = \mathbb{V} \setminus \mathcal{V} = \{v_4, v_6, v_8, v_9, w_1, v_5, v_7, v_3, y_1, y_2\}$ are the vertices that are not backward traversed by either keyword a or b.

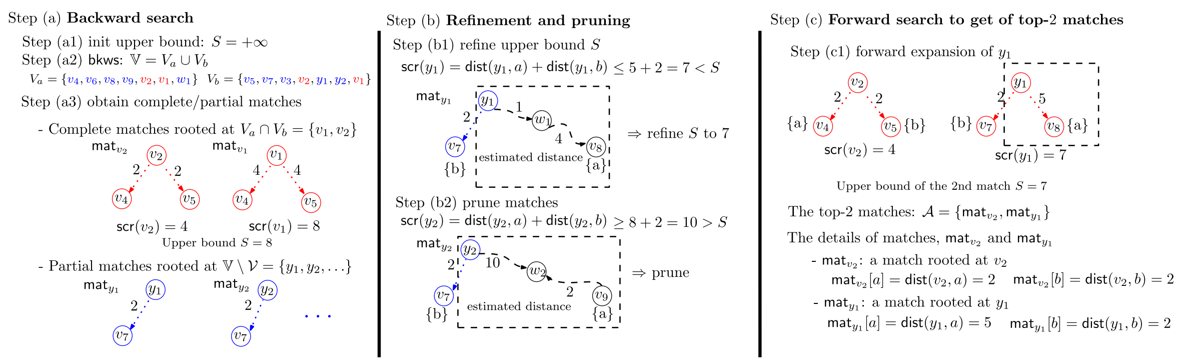

Step (a) Backward search
Step (a1) init upper bound: $S = +\infty$
Step (a2) bkws: $\mathbb{V} = V_a \cup V_b$
$V_a = \{v_4, v_6, v_8, v_9, v_2, v_1, w_1\}$ $V_b = \{v_5, v_7, v_3, v_2, y_1, y_2, v_1\}$
Step (a3) obtain complete/partial matches
- Complete matches rooted at $V_a \cap V_b = \{v_1, v_2\}$
$\mathsf{scr}(v_2) = 4$ $\mathsf{scr}(v_1) = 8$
Upper bound $S = 8$
- Partial matches rooted at $\mathbb{V} \setminus \mathcal{V} = \{y_1, y_2, \ldots\}$
Step (b) Refinement and pruning
Step (b1) refine upper bound S
$\mathsf{scr}(y_1) = \mathsf{dist}(y_1, a) + \mathsf{dist}(y_1, b) \leq 5 + 2 = 7 < S$
estimated distance
⇒ refine S to 7
Step (b2) prune matches
$\mathsf{scr}(y_2) = \mathsf{dist}(y_2, a) + \mathsf{dist}(y_2, b) \geq 8 + 2 = 10 > S$
estimated distance
⇒ prune
Step (c) Forward search to get of top-2 matches
Step (c1) forward expansion of y_1
$\mathsf{scr}(v_2) = 4$ $\mathsf{scr}(y_1) = 7$
Upper bound of the 2nd match $S = 7$
The top-2 matches: $\mathcal{A} = \{\mathsf{mat}_{v_2}, \mathsf{mat}_{y_1}\}$
The details of matches, mat_{v_2} and mat_{y_1}
- mat_{v_2}: a match rooted at v_2
$\mathsf{mat}_{v_2}[a] = \mathsf{dist}(v_2, a) = 2$ $\mathsf{mat}_{v_2}[b] = \mathsf{dist}(v_2, b) = 2$
- mat_{y_1}: a match rooted at y_1
$\mathsf{mat}_{y_1}[a] = \mathsf{dist}(y_1, a) = 5$ $\mathsf{mat}_{y_1}[b] = \mathsf{dist}(y_1, b) = 2$

Fig. 7. Key steps: Backward search (Section III-B), refinement and pruning (Example III.3 and III.4), and forward search (Section III-C)

Correctness: By using match refinement, bkws is monotonic. Since $\mathsf{mat}_u[q]$ is refined when a shorter path between u and q in a complete match mat_u or a new path between u and a missing keyword q in a partial match mat_u is identified, the refinement follows the partial order $\preceq$ on mat_u. We recall that $\mathsf{mat}_u[q]$ is from a finite domain $\{\sum_{e_i \in E'} w(e_i)\} \cup \{+\infty\}$, where E' is any subset of E. By Definition III.2, bkws can be parallelized and terminated correctly.

Complexity: bkws takes $O(|Q|(|E| + |V| \log |V|))$, where $|Q|$ is the number of query keywords. For simplicity, we provide the analysis in Appendix B.5 of [17], available online. The size of a match, mat_u, is bounded by $O(|Q|)$. Hence, the space complexity is bounded by $O(|Q||V|)$.

C. Monotonic Forward Search (fkws)

The main purpose of fkws is to retrieve the missing keywords of the partial matches via forward expansion. fkws is also widely used in existing keyword search algorithms, e.g., [5], [16], [21], [34], [47]. Due to a potentially large number of partial matches, forward expansion for the vertices $\bar{\mathcal{V}}$ could be costly. Existing studies can be space-consuming [16] or do not guarantee exact matches [21]. We describe the forward expansion and propose *new bounds* for pruning in fkws.

Forward expansion: Consider a partial match mat_u. Suppose a query keyword $q \in Q$ is missing in mat_u. fkws forward expands from u by using Dijkstra's algorithm to retrieve the nearest node that contains q.

Pruning in fkws: Some forward expansions do not lead to complete matches and can be pruned as shown in Proposition III.2.

Proposition III.2: Consider the forward expansion for vertex u. Suppose the next vertex to be expanded by Dijkstra's algorithm is v, the forward expansion is terminated when any of the following conditions holds.

1) $q \in L(v)$, i.e., the keyword q is found;
2) $\mathsf{dist}(u, v) > \tau$, the vertex containing keyword q is farther than τ from u or does not exist (i.e., $\mathsf{dist}(u, q) > \tau$); or
3) $\mathsf{scr}(u) + \mathsf{dist}(u, v) > S \Rightarrow \mathsf{scr}(u) + \mathsf{dist}(u, q) > S$.

As indicated by Condition 2, if $\mathsf{dist}(u, q)$ has been indexed, early termination can be determined if $\mathsf{dist}(u, q) > \tau$. Furthermore, Condition 3 posits that the current top k-th match score, denoted as S, serves as an upper bound. If $\mathsf{dist}(u, q)$ is indexed, we can employ a tightly estimated upper bound of S to facilitate decisions on early termination. Thus, we engage state-of-the-art indexing techniques – PageRank-based All-distances Sketches (PADS) and PageRank-based Keyword Distance Sketches (KPADS) [19]. Specifically, $\mathsf{PADS}(u)$ is a sketch for u, which indexes the shortest distance between u and the sketch's centers (some vertices in the graph). Given that $\mathsf{PADS}(v_i)$ and $\mathsf{PADS}(v_j)$ may share common centers where $q \in L(v_i) \cap L(v_j)$, these shared centers can be merged. In the process of merging, only the smallest distance is retained. $\mathsf{KPADS}(q)$ sketch is constructed through such merges and is used to index the shortest distance between the keyword q and the centers. These sketches assist in estimating both the upper and lower bounds of the shortest distance between u and q, where u belongs to $\bar{\mathcal{V}}$ and q is a missing keyword in mat_u.

Indexing: PADS and KPADS have been shown to be both space- and time-efficient in practice with theoretical guarantees on the accuracy of the shortest distance which can be readily distributed. However, we remark that [19] considered undirected graphs. To support *directed graphs*, we make a modification to PADS as follows. The sketch of a node u is two sets of vertices and their corresponding shortest distances from (resp. to) u, denoted by $\mathsf{PADS}^{\mathsf{out}}(u) = \{(w, d)\}$ (resp. $\mathsf{PADS}^{\mathsf{in}}(u) = \{(w, d)\}$), where $w \in V$ and $d = \mathsf{dist}(u, w)$ (resp. $d = \mathsf{dist}(w, u)$). Similarly, the sketch of a keyword q is denoted by $\mathsf{KPADS}^{\mathsf{out}}(q) = \{(w, d)\}$ (resp. $\mathsf{KPADS}^{\mathsf{in}}(q) = \{(w, d)\}$), where $w \in V$ and $d = \mathsf{dist}(q, w)$ (resp. $d = \mathsf{dist}(w, q)$). For brevity, we leave the construction pseudo-code of PADS and KPADS in [17].

Since PADS yields estimated bounds, fkws needs to handle both approximate and exact matches. Specifically, fkws computes the *upper* bound of the score for any $u \in \bar{\mathcal{V}}$, $\mathsf{scr}(u)$, by estimating the shortest distance between u and the missing keywords, i.e., $\Sigma\mathsf{dist}(u, q_i) + \Sigma\mathsf{mat}_u[q_j]$, where $q_i, q_j \in Q$, q_i is missing from mat_u whereas q_j has been found in mat_u. If the upper bound is smaller than S, mat_u is inserted into $\mathcal{A}$ and S is refined accordingly. To avoid ambiguity, we denote the $\mathcal{A}$ that may consist of exact matches and approximate matches by $\hat{\mathcal{A}}$. The approximate matches in $\hat{\mathcal{A}}$ are further refined during forward expansion. $\hat{\mathcal{A}}$ is eventually refined to yield $\mathcal{A}$.

Next, we present the upper and lower bounds of $\mathsf{dist}(u, q)$ for the termination of forward expansion from u. These bounds can be applied to other keyword search semantics as they involve numerous distance computations, such as [16], [22], [23].

(1) Upper bound of the shortest distance: Given a shortest distance query (u, q), the upper bound is computed by $\mathsf{PADS}^{\mathsf{out}}(u)$ and $\mathsf{KPADS}^{\mathsf{in}}(q)$ as follows:

$$\mathsf{dist}(u, q) \leq \mathsf{dist}(u, w) + \mathsf{dist}(w, q), \tag{1}$$

where $(w, \mathsf{dist}(u, w)) \in \mathsf{PADS}^{\mathsf{out}}(u), (w, \mathsf{dist}(w, q)) \in \mathsf{KPADS}^{\mathsf{in}}(q)$, and w is a common center in $\mathsf{PADS}^{\mathsf{out}}(u)$ and $\mathsf{KPADS}^{\mathsf{in}}(q)$.

Example III.3: Consider the graph in Fig. 7(b1). $\mathsf{PADS}^{\mathsf{out}}(y_1) = \{(w_1, 1)\}$ and $\mathsf{KPADS}^{\mathsf{in}}(a) = \{(w_1, 4)\}$. The common center of $\mathsf{PADS}^{\mathsf{out}}(y_1)$ and $\mathsf{KPADS}^{\mathsf{in}}(a)$ is w_1. Hence, the upper bound of the shortest distance between y_1 and keyword a is derived by $\mathsf{dist}(y_1, w_1) + \mathsf{dist}(w_1, a) = 5$. Then, the upper bound of the score of the match rooted at y_1 is 7. Since the upper bound is smaller than S, the approximate match mat_{y_1} is inserted into $\mathcal{A}$ to yield $\hat{\mathcal{A}}$. S is refined accordingly, $S = \mathsf{scr}(y_1)$.

(2) Lower bound of the shortest distance: We also derive a lower bound of the shortest distance between u and q by exploiting $\mathsf{PADS}^{\mathsf{out}}(u)$ and $\mathsf{KPADS}^{\mathsf{out}}(q)$ to prune unnecessary traversals in an early stage of forward expansion. We have the following inequality.

$$\mathsf{dist}(u, q) \geq \mathsf{dist}(u, w) - \mathsf{dist}(q, w), \tag{2}$$

where $(w, \mathsf{dist}(u, w)) \in \mathsf{PADS}^{\mathsf{out}}(u), (w, \mathsf{dist}(q, w)) \in \mathsf{KPADS}^{\mathsf{out}}(q)$, and w is a common center in $\mathsf{PADS}^{\mathsf{out}}(u)$ and $\mathsf{KPADS}^{\mathsf{out}}(q)$. Therefore, the minimum of $\mathsf{dist}(u, w) - \mathsf{dist}(q, w)$ is the lower bound of $\mathsf{dist}(u, q)$.

If the lower bound is larger than τ, the forward expansion from u is simply skipped, since $\mathsf{dist}(u, q) > \tau$, and Proposition III.2-Condition 2 is already satisfied. Similarly, if the lower bound of the score of the match rooted at u is larger than S, Proposition III.2-Condition 3 is met.

Example III.4: Consider the graph in Fig. 7(b2). Suppose $\mathsf{PADS}^{\mathsf{out}}(y_2) = \{(w_2, 10)\}$ and $\mathsf{KPADS}^{\mathsf{out}}(a) = \{(w_2, 2)\}$. The common center of $\mathsf{PADS}^{\mathsf{out}}(y_2)$ and $\mathsf{KPADS}^{\mathsf{out}}(a)$ is w_2. The lower bound of the shortest distance between y_2 and keyword a, $\mathsf{dist}(y_2, a)$, is derived by $\mathsf{dist}(y_2, w_2) - \mathsf{dist}(a, w_2) = 8$. The lower bound of the score of the match rooted at y_2 is $10 > S$. The forward expansion of y_2 is pruned.

Correctness: The analyses of partial order and the finite domain of fkws are similar to those of bkws. Hence, fkws can be parallelized correctly since it has the monotonic property.

Complexity: In the worst case, fkws performs a single source shortest path computation for each vertex $u \in \bar{\mathcal{V}}$. Therefore, the time complexity of fkws is bounded by $O(|\bar{\mathcal{V}}|\ (|E| + |V| \log |V|))$. The space complexity fkws is identical to that of bkws, which is bounded by $O(|Q||V|)$, whereas the space for PADS and KPADS is $O(|V| \log |V|)$ [19].

IV. Distributed Keyword Search (DKWS)

We illustrate PIE [12] with a keyword search algorithm, denoted as kws. (a) PEval is *partial evaluation* of kws. Partial results are passed to the next function. (b) IncEval is *incremental evaluation* of kws that takes partial results and computes the changes. IncEval is repeated until no more changes are computed. (c) Assemble collects local matches from workers. These functions are evaluated in a non-preemptive manner and defined formally as follows.

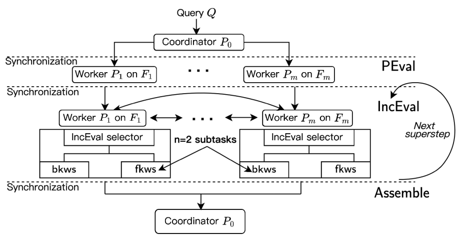


Fig. 8. Workflow of DKWS

The partial evaluation (PEval) utilizes a query Q and a fragment F_i of the graph G as inputs. PEval then concurrently computes partial answers, represented as $Q(F_i)$, consisting of current mat_u for all $u \in V$ at each worker P_i.

The incremental evaluation (IncEval) takes four inputs: a query Q, a fragment F_i of graph G, partial results derived from the application of the query to the fragment $Q(F_i)$, and a message M_i. The function then incrementally computes $Q(F_i \oplus M_i)$, optimizing the computation of $Q(F_i)$ from the previous superstep to maximize efficiency. After every execution of IncEval, DKWS updates its state by considering $F_i \oplus M_i$ and $Q(F_i \oplus M_i)$ as the new F_i and $Q(F_i)$, respectively, forming the input for the incremental computation in the next superstep.

Assemble starts its computation when M_i is empty for any worker P_i. Assemble accepts $Q(F_i \oplus M_i)$ as inputs. It consolidates for all $i \in [1, m]$, $Q(F_i \oplus M_i)$, to compute the final answer $Q(G)$.

Architecture of DKWS *(Fig. 8):* The coordinator P_0 is responsible for receiving and transmitting the query to all workers. Workers P_1 to P_n are in charge of computing the query on their fragments F_1 to F_n. When receiving the query, the workers perform PEval. During each superstep (IncEval) of query computation, a selector of each P_i decides to perform either bkws or fkws on F_i. When all workers meet the termination condition, the coordinators assemble the (local) top-k matches and select the (global) top-k matches from the local ones.

Programming model of DKWS*:* DKWS differs from previous studies in two major ways: (1) DKWS is the first to introduce a *notify-push* paradigm into a distributed programming model. The notify-push paradigm allows the coordinator and workers to asynchronously exchange refined bounds (Section IV-A) at runtime; and (2) PINE consists of PEval, IncEval ($\underline{n}$ subtasks), and one Assemble functions (Section IV-B) that the users can use to solve their problems by composing several PI algorithms and assembling the matches at the end, rather than only one PIE algorithm. DKWS runs the PI algorithms in a *preemptive* manner and therefore interactions, such as exchange of tighter bounds (presented in Section III) between them are possible.

A. Notify-Push (NP) Paradigm

With the Notify-Push paradigm, the bounds can be exchanged at run-time and provide a global scope for each worker. The

Algorithm 1: API of DKWS.

```
1 Function Notify (Worker id $i$, Local upper bound $S_i$):
2     Worker $P_i$ sends $S_i$ to notify the coordinator $P_0$
3     Coordinator refines the global upper bound $S$ by
        $\min\{S, S_i\}$
4 Function Push (Worker id $i$, Global upper bound $S$):
5     Coordinator $P_0$ pushes $S$ to worker $P_i$
6     Worker $P_i$ refines the local upper bound $S_i$ by $\min\{S, S_i\}$
```

pruning techniques are more efficiently on the workers with the tighter bounds.

*Definition IV.1 (*Notify *API):* Notify(i, S_i) is an API that a worker refines the global upper bound S with the local upper bound S_i. Notify(i, S_i) takes a worker's id i and a local upper bound S_i as input. Notify must be invoked by a worker P_i to notify the coordinator with its worker id i and the local upper bound S_i.

Local upper bound S_i: For fragment F_i, DKWS maintains a local upper bound S_i to prune false matches locally. DKWS maintains a priority queue $\mathcal{A}_i$ with a fixed size k to store the local top-k matches, which are ordered in descending order of the score of the matches for each fragment F_i. Once a better match is inserted into $\mathcal{A}_i$, S_i is refined locally. The worker P_i sends the refined local upper bound S_i to the coordinator P_0 and *notifies* the coordinator to refine the global upper bound by calling function Notify(i, S_i).

*Definition IV.2 (*Push *API):* Push(i, S) is an API that the coordinator P_0 broadcasts the global upper bound S to all the workers and refines the local upper bounds. Push(i, S) takes a worker's id i and the global upper bound S in the coordinator P_0 as input. Push is invoked by the coordinator P_0 and pushes the global upper bound S to worker P_i.

Global upper bound S: When the coordinator receives a local upper bound from a worker, it refines its local upper bound table which records the local upper bounds from all the workers. The global upper bound S is the smallest among the local upper bounds. To avoid excessive refinements, the coordinator maintains a notification counter N_i for each fragment F_i. Consider any N_i. If $\max\{N_j | j \in [1, m]\} - N_i$ is larger than a threshold, and $S_i > S$, this implies P_i may be doing unnecessary computation on the fragment F_i for a long time. The coordinator *push*es the global upper bound to F_i by calling Push(i, S). Once P_i receives the global upper bound S, it refines the local upper bound S_i with S.

Note that the notify-push paradigm is established on the fact that the local upper bound S_i is the upper bound of the global upper bound S. We formalize this as follows.

Lemma IV.1: $\forall i \in [1, m]$, $S_i \leq S$.

Proof: We can prove this assertion by contradiction. Let's suppose that $S_i > S$. By definition, S_i (resp. S) denotes the score of the local (resp. global) k-th match, represented by mat_{u_k} (resp. $\mathsf{mat}_{u'_k}$). Considering any local top-k match mat_{u_j} with $j \in [1, k]$, $\mathsf{scr}(\mathsf{mat}_{u_j}) \leq \mathsf{scr}(\mathsf{mat}_{u_k}) < \mathsf{scr}(\mathsf{mat}_{u'_k})$. This implies that $\mathsf{mat}_{u'_k}$ is not included among the global top-k matches, as there are k matches with lower scores on F_i. Hence, we deduced that $S_i \leq S$.

Example IV.1: Given a distributed graph G, which has been partitioned into three fragments (F_1, F_2, and F_3), shown in Fig. 9 , assume that the query keywords are $Q = \{a, b\}$ and $k = 2$. In the first superstep of DKWS, P_3 finishes the computation earlier since the size of F_3 is smaller. The local upper bound S_2 on F_2 is 6. Without the NP paradigm, P_2 does not terminate until all vertices of F_2 are traversed. The vertices x_1, x_2, and x_3, are not pruned by S_2 since $\mathsf{dist}(x_1, b) = 5 < S_2$ and the termination condition of backward expansion is not met as presented in Section III-B. With the paradigm, P_3 sends $S_3 = 5$ to the coordinator by Notify$(3, S_3)$. Then, the coordinator refines the global upper bound S with S_3 and pushes the global upper bound to all the workers, e.g., P_2, by Push$(2, S)$. Once P_2 receives the global upper $S = 5$, it refines the local upper bound S_2 (denoted by $6 \xrightarrow{R} 5$) accordingly. Since the paradigm allows exchanging the bounds during a superstep, if x_1, x_2 and x_3 have not been visited, they are pruned, since $\mathsf{dist}(x_i, b) = 5 \geq S_2$. Then, the termination condition of backward search is met. Similarly, in Superstep 2, the backward expansion at v_{16} is skipped.

Remarks: DKWS is efficient for several reasons: (a) Notify API provides a way for each worker to send refined bounds which help to prune more false matches on stragglers; and (b) The communication cost is small since DKWS only exchanges the local upper bounds rather than intermediate matches during distributed query evaluation.

B. PINE *Programming Model*

1) Overview of PINE*:* The overview of PINE is illustrated with Fig. 8. PINE consists of PEval and IncEval of $\underline{n}$ subtasks, along with one Assemble. In the first superstep, PEval of all subtasks are executed in each worker P_i. In subsequent supersteps, each worker P_i features an IncEval selector that decides which subtask's IncEval to execute. This granular level of execution is designed to address the straggler problem (refer to the Challenge 1 in Section I).

We next illustrate the PINE programming model with an efficient implementation of bfkws. There are two subtasks, bkws (Section IV-B.2) and fkws (Section IV-B.3), respectively. For each subtask, we only need to declare its messages, PEval and IncEval. We propose preemptive execution of IncEvals in DKWS (shown in Fig. 8). We use mat^b_u (resp. mat^f_u) to denote the partial match found by bkws (resp. fkws) rooted at u. Finally, we implement Assemble by collecting the local top-k matches from all fragments to yield the global top-k matches after both the IncEvals terminate.

2) PI *for* bkws*: Message declaration:* DKWS declares a variable mat^b_u for each vertex u, where mat^b_u is a map such that $\mathsf{mat}^b_u[q] = \langle \mathsf{v}, \mathsf{d} \rangle$ is used to denote the shortest distance between u and a query keyword $q \in L(\mathsf{v}) \cap Q$, i.e., $\mathsf{d} = \mathsf{dist}(u, q)$. Intuitively, u is considered as the root of a match, while v is a leaf vertex of the match, the labels of which contain a query keyword, q.

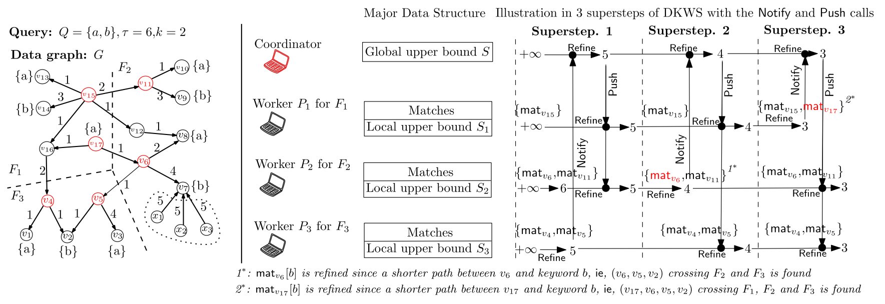

Query: $Q = \{a, b\}, \tau = 6, k = 2$
Data graph: G
Major Data Structure
Illustration in 3 supersteps of DKWS with the Notify and Push calls
Coordinator
Global upper bound S
Worker P_1 for F_1
Worker P_2 for F_2
Worker P_3 for F_3
Matches
Local upper bound S_1
Local upper bound S_2
Local upper bound S_3
Superstep. 1
Superstep. 2
Superstep. 3
Refine
Push
Notify
1*: $\mathsf{mat}_{v_6}[b]$ is refined since a shorter path between v_6 and keyword b, ie, (v_6, v_5, v_2) crossing F_2 and F_3 is found
2*: $\mathsf{mat}_{v_{17}}[b]$ is refined since a shorter path between v_{17} and keyword b, ie, (v_{17}, v_6, v_5, v_2) crossing F_1, F_2 and F_3 is found

Fig. 9. Illustration of the change of bounds during query processing of DKWS

Algorithm 2: PEval for bkws.

```
    Input: $F_i(V, E, L)$, $Q = \{q_1, \ldots, q_l\}$, $\tau$
    Output: $Q(F_i)$ consisting of current $\mathsf{mat}_u^b$ for all $u \in V$
1   init a local upper bound $S_i$ with a large value
2   For each node $u \in V$, init a match variable $\mathsf{mat}_u^b$ to null
3   foreach $q \in Q$ do // init the searching origin
4       init search priority queue $\mathcal{P}_q = \emptyset$
5       init visited vertices set $V_q = \emptyset$
6       foreach $u \in O_q$ do
7           $\mathsf{mat}_u^b[q] = 0$
8           $\mathcal{P}_q$.push($\langle u, 0 \rangle$)
9   BackwardExpand($\mathcal{P}$)// $\mathcal{P} = \{\mathcal{P}_q | q \in Q\}$
10  Function BackwardExpand($\mathcal{P}$)
11      while $\exists \mathcal{P}_q$ is not empty and $S_i > \Sigma\mathcal{P}_q$.top().d do
12          pick $\mathcal{P}_q$ from all the search queues with minimal
              $\mathcal{P}_q$.top().d
13          $\langle u, d \rangle = \mathcal{P}_q$.top()
14          $V_q$.add($u$)
15          foreach $e = (u', u) \in E$ and $u' \notin V_q$ do
16              $d' = w(e) + d$
17              if $d' < \tau$ and $d' < \mathsf{mat}_{u'}^b[q]$ then
18                  $\mathsf{mat}_{u'}^b[q] = d'$
19                  $\mathcal{P}_q$.push($\langle u', d' \rangle$)
20                  if $\mathsf{mat}_u^b$ is a complete match and scr($u$) $< S_i$
                      then
21                      $\mathcal{A}_i$.push($\langle u, \mathsf{mat}_u^b \rangle$)
22                      $S_i$ = scr($\mathcal{A}_i$.top().u)
23                      Notify($i$, $S_i$)
24  Message segment: $M_i = \{\mathsf{mat}_u^b | u \in F_i.I\}$
```

Algorithm 3: IncEval for bkws.

```
    Input: $F_i(V, E, L)$, $Q = \{q_1, \ldots, q_l\}$, $\tau$, $Q(F_i)$, message $M_i$
    Output: $Q(F_i \oplus M_i)$ consisting of current $\mathsf{mat}_u^b \in \mathcal{A}_i$, where
              $u \in V$
1   init $V_q$, $\mathcal{P}_q$ for each query keyword $q \in Q$
2   foreach $\mathsf{mat}_u^{b,\mathsf{in}} \in M_i$ do
3       foreach $q \in Q$ and $\mathsf{mat}_u^b[q] > \mathsf{mat}_u^{b,\mathsf{in}}[q]$ do
4           $\mathsf{mat}_u^b[q] = \mathsf{mat}_u^{b,\mathsf{in}}[q]$
5           $\mathcal{P}_q$.push($\langle u, \mathsf{mat}_u^b[q] \rangle$)
6   BackwardExpand($\mathcal{P}$) // $\mathcal{P} = \{\mathcal{P}_q | q \in Q\}$
7   Message segment: $M_i = \{\mathsf{mat}_u^b | u \in F_i.I\}$
```

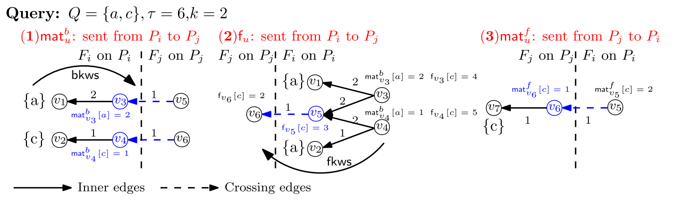

Query: $Q = \{a, c\}, \tau = 6, k = 2$
(1)mat_u^b: sent from P_i to P_j
(2)f_u: sent from P_i to P_j
(3)mat_u^f: sent from P_j to P_i
F_i on P_i
F_j on P_j
bkws
fkws
$\mathsf{mat}_{v_3}^b[a] = 2$
$\mathsf{mat}_{v_4}^b[c] = 1$
$\mathsf{f}_{v_3}[c] = 4$
$\mathsf{f}_{v_6}[c] = 2$
$\mathsf{f}_{v_5}[c] = 3$
$\mathsf{mat}_{v_4}^b[a] = 1$
$\mathsf{f}_{v_4}[c] = 5$
$\mathsf{mat}_{v_6}^f[c] = 1$
$\mathsf{mat}_{v_5}^f[c] = 2$
Inner edges
Crossing edges

Fig. 10. Message exchange during query processing (mat_u^b (resp. mat_u^f): keep track the shortest distance between u and a query keyword by bkws (resp. fkws); and f_u: the longest distance needed to be forward expanded starting from u)

(1) Partial evaluation (PEval) for bkws (Algorithm 2). Upon receiving a query Q, PEval computes the partial matches of bkws, mat_u^b on F_i locally, for all $i \in [1, m]$ in parallel. P_i initializes its local upper bound S_i with a large constant value and initializes a match variable mat_u^b for each vertex (Lines 1–2). Lines 3–8 initialize the search origins and the priority queue for the search. Lines 11-12 present the pseudo-code of bkws (described in Section III-B). In addition, in the NP paradigm, at runtime, PEval sends the local upper bound S_i to the coordinator and notifies it to refine the global upper bound when S_i is refined (Line 23). In Line 24, the messages are grouped into M_i at the incoming portal nodes on fragment F_i. Partial matches that are relevant to F_j ($M_{i,j} = \{\mathsf{mat}_u^b | u \in F_i.I \cap F_j.O\} \in M_i$) are transmitted to worker P_j.

(2) Incremental computation (IncEval) for bkws (Algorithm 3). Upon receiving messages M_i, IncEval iteratively computes the partial matches, mat_u^b, on F_i with the updates (messages) M_i. Specifically, if the distance between u and $q \in Q$, i.e., $\mathsf{d} = \mathsf{mat}_u^b[q]$, is refined by using message M_i, u is pushed into the priority queue $\mathcal{P}_q$ with the refined distance. Then, IncEval propagates the distance refinement to the affected area by bkws. Worker P_i notifies the coordinator P_0 once the local upper bound S_i is refined by invoking Notify(i, S_i). At the end of IncEval, the messages are grouped into M_i at the incoming portal nodes and sent to the relevant workers, similar to PEval.

Completeness: We assume that DKWS takes R supersteps to finish the evaluation of a keyword query. We denote the vertices that have been visited on F_i at the s-th ($s \leq R$) superstep for a query keyword $q_j \in Q$ by $V_{q_j,i}^s$. We denote the union set of all the visited vertices by $\mathbb{V}$. Hence, $\mathbb{V} = \bigcup_{i \in [1,m], j \in [1,l], s \in [1,R]} V_{q_j,i}^s$, where m is the number of workers and $l = |Q|$. We have the following proposition.

Proposition IV.1: Suppose the top-k matches of a keyword query is $\mathcal{A}$ and all the visited vertices $\mathbb{V}$, the following hold:

(1) $\forall u \notin \mathbb{V}$, $\mathsf{mat}_u \notin \mathcal{A}$; and (2) $\forall \mathsf{mat}_u \in \mathcal{A}$, $u \in \mathbb{V}$.

Proof: The proof is presented in Appendix A.2 of [17], available online.

Example IV.2: As shown in Fig. 10 .(1), when bkws expands from v_1 to v_3, $\mathsf{mat}_{v_3}^b[a] = 2$ is sent from F_i to F_j since $v_3 \in F_i.I$. Similarly, $\mathsf{mat}_{v_4}^b[c] = 1$ is sent from F_i to F_j. IncEval of bkws is invoked in F_j to search for matches.

Algorithm 4: PEval for fkws.

```
Input: $F_i(V, E, L)$, $Q = \{q_1, \ldots, q_l\}$, $\tau$
Output: $Q(F_i)$ consisting of current $\mathsf{mat}_u \in \mathcal{A}_i$ for all $u \in V$
1  load the indexes PADS and KPADS
2  maintain the vertices to be forward expanded in $\bar{\mathcal{V}}$, i.e., roots
     of partial matches
3  for $u \in \bar{\mathcal{V}}$, init a forward match $\mathsf{mat}^f_u$ and a forward distance
     $\mathsf{f}_u$
4  foreach $u \in \bar{\mathcal{V}}$ do
5      forwardExpand$(u, Q, S_i, \mathcal{A}_i)$
6  Function forwardExpand$(u, Q, S_i, \mathcal{A}_i)$
7      foreach $v$ in the Dijkstra's traversal of $u$ do
8          if not isCandidate$(u)$ or all the $q \in \mathsf{f}_u$ are found then
9              break
10         foreach $q \in \mathsf{f}_u$ do
11             if $q \in L(v)$ then // missing keyword is
                 found
12                 $\mathsf{mat}^f_u[q] = \mathsf{dist}(u, v)$
13                 marks that $q \in \mathsf{f}_u$ is found
14             else if $\mathsf{dist}(u, v) > \mathsf{mat}^f_u[q]$ or
                 $\mathsf{dist}(u, v) + \mathsf{scr}(u) > S_i$ then
15                 marks that $q \in \mathsf{f}_u$ is found // Prop.3.2
                     Condition (iii) is met
16             else if $\mathsf{dist}(u, v) + \mathsf{mat}_v[q] < \tau$ then
                   // Refine $\mathsf{mat}^f_u$ by a found match
                   $\mathsf{mat}_v$
17                 $\mathsf{mat}^f_u[q] = \min\{\mathsf{mat}^f_u[q], \mathsf{dist}(u, v) + \mathsf{mat}_v[q]\}$
18             else if $v \in F_i.O$ then
                   // foward expansion on other
                   fragments
19                 $\mathsf{f}_v[q] = \max\{\mathsf{f}_v[q], \mathsf{f}_u[q] - \mathsf{dist}(u, v)\}$
20     if $\mathsf{mat}_u$ is a complete match and $\mathsf{scr}(u) < S_i$ then
21         $\mathcal{A}_i$.push$(\langle u, \mathsf{mat}_u\rangle)$
22         $S_i = \mathsf{scr}(\mathcal{A}_i.\mathrm{top}().\mathrm{u})$
23         Notify$(i, S_i)$
24 Function isCandidate$(u)$
25     $\check{\mathsf{dist}}(u, F_i.O) \leftarrow$ estimate the lower bound between $u$ and
         $F_i.O$ by Eq 2
26     foreach $q \in \mathsf{f}_u$ do
27         $\check{\mathsf{dist}}(u, q) \leftarrow$ estimate the lower bound between $u$ and
             $q$ by Eq 2
28         if $\check{\mathsf{dist}}(u, q) > \mathsf{f}_u[q]$ and $\check{\mathsf{dist}}(u, F_i.O) > \mathsf{f}_u[q]$ then
29             return False
30     return True
31 Message segment: $M^1_i = \{\mathsf{mat}^f_u | u \in F_i.I\}$ and
     $M^2_i = \{\mathsf{f}_u | u \in F_i.O\}$
```

3) PI for fkws: Message declaration: DKWS declares a variable mat^f_u, where mat^f_u is a map, $\mathsf{mat}^f_u[q] = \langle \mathsf{v}, \mathsf{d}\rangle$, where d is the shortest distance between vertex u and a query keyword $q \in L(\mathsf{v}) \cap Q$, i.e., $\mathsf{d} = \mathsf{dist}(u, q)$. mat^f_u is to keep track of the updates to u during the forward expansion. DKWS also declares a variable f_u for each vertex u to indicate the distances of the longest forward expansion of retrieving missing keywords starting from u. Formally, f_u is a map (q, d), where $q \in Q$ is a query keyword and d is the longest distance needed to be forward expanded starting from u to retrieve the query keyword q.

(1) Partial evaluation (PEval) for fkws (Algorithm 4). fkws mainly conducts the forward expansion to complete the partial matches. Lines 2-3 are the initialization of the vertices for the expansion and match variables. In the forward expansion starting from u, if any condition(s) in Proposition III.2 is met, the expansion is terminated (Lines 16-17). Suppose u is expanded to vertex v and the missing keyword q is found in $\mathsf{mat}_v = \mathsf{mat}^b_v \cup \mathsf{mat}^f_v$, $\mathsf{mat}^f_u[q]$ is refined (Lines 11-15). If $v \in F_i.O$, the remaining distance of the forward expansion to retrieve the query keyword q on other fragments is stored in $\mathsf{f}_v[q]$ (Lines 23).

At the end of PEval (Line 31), messages mat^f_u (resp. f_u) are grouped into M^1_i (resp. M^2_i) in worker P_i. $M_{i,j} \in M_i$ is sent to worker P_j. Formally, $M^1_{i,j} = \{\mathsf{mat}_u | u \in F_i.I \cap F_j.O\}$ and $M^2_{i,j} = \{\mathsf{f}_u | u \in F_i.O \cap F_j.I\}$ are sent from worker P_i to worker P_j. Moreover, PEval sends the refined S_i to the coordinator and notifies it to refine the global upper bound S once the local upper bound is refined.

(2) Incremental computation (IncEval) for fkws is derived with the following two modifications.

(2.1) Refinement propagation: First, P_i receives the partial matches, mat^f_u in previous supersteps from other fragments through the portal nodes. If a shorter path between u and q is found crossing multiple fragments, the forward match $\mathsf{mat}^f_u[q]$ is refined. IncEval propagates the distance refinement to the ancestor vertices.

(2.2) Incremental forward expansion: Second, upon receiving some forward expansion requests from other fragments, worker P_j further forward expands to retrieve missing keywords on F_j through the incoming portal nodes, $F_j.I$. Specifically, if $\mathsf{f}^{\mathsf{in}}_u \in M^2_j$ is received and $u \notin \bar{\mathcal{V}}$, u is added to $\bar{\mathcal{V}}$. Since search requests come from different fragments, f_u keeps the largest one for each keyword. If u is forward expanded in previous iterations for query keyword q or $\mathsf{f}^{\mathsf{in}}_u[q]$ is smaller than $\mathsf{f}_u[q]$, $\mathsf{f}^{\mathsf{in}}_u[q]$ is skipped.

At the end of IncEval, the partial matches found by forward expansions are grouped into M^1_i and the remaining forward expansion requests are grouped into M^2_i, respectively, for fragment F_i and sent to the corresponding fragments, which is the same as that of PEval.

Example IV.3: As shown in Fig. 10.(2), when fkws expands from v_4 to v_5 to search for the missing keyword c, $\mathsf{f}_{v_5}[c] = 3$ is sent from F_i to F_j since $v_5 \in F_i.O$. IncEval of fkws is invoked in F_j to search on F_j forwardly. Once the keyword c is retrieved in F_j as recorded in $\mathsf{mat}^f_{v_6}[c] = 1$ (shown in Fig. 10.(3)), $\mathsf{mat}^f_{v_6}[c] = 1$ is sent to F_i via the portal node $v_6 \in F_j.I$.

4) Preemptive Execution of IncEvals *in* PINE*:* Even if the complexity of bkws (analyzed in Section III-B) is smaller than that of fkws (analyzed in Section III-C), running bkws first and then fkws may not exhibit the best query performance in practice. In particular, we provide three insights: (a) bkws increases the size of $\bar{\mathcal{V}}$ but k of top-k is fixed. Relatively more vertices of $\bar{\mathcal{V}}$ may not be backward expanded to final matches; (b) some early messages from bkws may not effectively refine into tight upper bounds for fkws; and (c) some workers running bkws can be stragglers, as fkws is blocked by them, i.e., it cannot yet start. Hence, PINE provides a lightweight selector (as shown in Fig. 8) and allows the computation of bkws and fkws in a preemptive manner. At runtime, each worker P_i determines to *execute either* bkws *or* fkws, *independently*. Each of them maintains a set of *status parameters* to estimate the performance improvement of executing either bkws or fkws.

Message buffers: Each worker P_i maintains two message buffers $\mathbb{B}^b_i$ and $\mathbb{B}^f_i$ to keep track of backward and forward messages from other workers. The more messages are accumulated in $\mathbb{B}^b_i$ (resp. $\mathbb{B}^f_i$), the earlier P_i should start bkws (resp. fkws) computation, and vice versa.

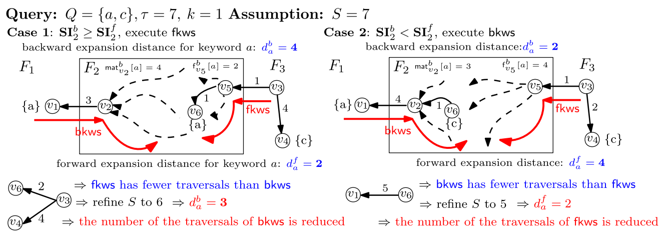

Query: $Q = \{a, c\}, \tau = 7,\ k = 1$ Assumption: $S = 7$
Case 1: $\mathbf{SI}_2^b \geq \mathbf{SI}_2^f$, execute fkws
backward expansion distance for keyword a: $d_a^b = 4$
F_1
F_2
F_3
$\mathsf{mat}_{v_2}^b[a] = 4$
$\mathsf{f}_{v_5}^b[a] = 2$
{a}
{c}
bkws
fkws
forward expansion distance for keyword a: $d_a^f = 2$
⇒ fkws has fewer traversals than bkws
⇒ refine S to 6 ⇒ $d_a^b = 3$
⇒ the number of the traversals of bkws is reduced
Case 2: $\mathbf{SI}_2^b < \mathbf{SI}_2^f$, execute bkws
backward expansion distance: $d_a^b = 2$
$\mathsf{mat}_{v_2}^b[a] = 3$
$\mathsf{f}_{v_5}^b[a] = 4$
forward expansion distance: $d_a^f = 4$
⇒ bkws has fewer traversals than fkws
⇒ refine S to 5 ⇒ $d_a^f = 2$
⇒ the number of the traversals of fkws is reduced

Fig. 11. Illustration of the preemptive execution

Expansion distance: Denote $\mathsf{mat}_u^{b,\mathsf{in}}$ as a message generated by bkws and maintained in $\mathbb{B}_i^b$. If $\mathsf{mat}_u^b[q]$ is larger than $\mathsf{mat}_u^{b,\mathsf{in}}[q]$, DKWS needs to backward expand starting from u. We call $d_q^b = \min\{S - \mathsf{mat}_u^{b,\mathsf{in}}[q], \tau\}$ *backward expansion distance starting from u for query keyword q*. Without prior statistics, if d_q^b is larger, the backward expansion is more costly and the messages are less likely to finally yield one of the top-k matches. Worker P_i may stop expanding $\mathsf{mat}_u^b[q]$ by postponing the execution of bkws, but start fkws to prune some unyielding messages. Similarly, we define $d_q^f = \min\{\mathsf{f}_u^{\mathsf{in}}[q], \tau\}$, the *forward expansion distance*.

Staleness indicators: Inspired by the complexities of bkws and fkws, we propose the staleness indicators of the accumulated backward messages and forward messages for each worker P_i, denoted by SI_i^b and SI_i^f. SI_i^b and SI_i^f are formally defined below:

$$\mathsf{SI}_i^b = \begin{cases} +\infty, & \text{if } \mathbb{B}_i^b \text{is empty} \\ \frac{\sum_{u\in F_i.O}\sum_{q\in Q} d_q^b}{|\mathbb{B}_i^b|}, & \text{otherwise} \end{cases} \quad (3)$$

$$\mathsf{SI}_i^f = \begin{cases} +\infty, & \text{if } \mathbb{B}_i^f \text{ is empty} \\ \frac{\sum_{u\in F_i.I}\sum_{q\in Q}^{|Q|} d_q^f}{|\mathbb{B}_i^f|}, & \text{otherwise} \end{cases} \quad (4)$$

where d_j^b (resp. d_j^f) is the average backward (resp. forward) searching distance for query keywords and $|\mathbb{B}_i^b|$ (resp. $|\mathbb{B}_i^f|$) is the size of backward (resp. forward) messages buffer.

If $\mathsf{SI}_i^b < \mathsf{SI}_i^f$, worker P_i conducts bkws. Otherwise, worker P_i conducts fkws. PINE is able to simulate PIE by enforcing SI_i^b to $+\infty$ at the even supersteps and SI_i^b to $+\infty$ at the odd supersteps.

Example IV.4: Consider the two cases in Fig. 11 . In Case 1, when the backward expansion from v_1 is performed via v_2, the backward expansion distance starting from v_2 is 4 and $\mathsf{SI}_2^b = 4$. When the forward expansion from v_3 is performed via v_5, the forward expansion distance from v_5 is 2 and $\mathsf{SI}_2^f = 2$. Since $\mathsf{SI}_2^f < \mathsf{SI}_2^b$, fkws has a higher priority to be executed. We can observe that an answer rooted at v_3 is returned, and the upper bound S is refined to 6. Consequently, d_a^b is refined to 3, and fewer traversals are required in the next iteration. Similarly, in Case 2, bkws has a higher priority and produces the matches earlier, which reduces the number of traversals of fkws after the upper bound S is refined.

5) Assemble for bfkws: DKWS only collects the local top-k matches to yield the global top-k matches $\mathcal{A}$ by selecting the top-k matches from $\bigcup_{i\in[1,m]} \mathcal{A}_i$ after the executions of IncEvals of bkws and fkws have terminated. Hence, the cost of collecting local matches from all the workers is bounded by $O(km)$.

Example IV.5: Consider the graph and query in Fig. 9. Local matches of F_1 are rooted at v_{15} and v_{17} and $\mathsf{scr}(v_{15}) = 4$ and $\mathsf{scr}(v_{17}) = 3$. Similarly, we have two local matches on F_2 with $\mathsf{scr}(v_{11}) = 4$ and $\mathsf{scr}(v_6) = 4$ and two local matches on F_3 with $\mathsf{scr}(v_4) = 2$ and $\mathsf{scr}(v_5) = 5$. Hence, the coordinator collects all the 6 local matches. The matches rooted at v_4 and v_{17} are returned since they are the top-2 among the 6 matches.

C. Analysis of bfkws on DKWS

In this section, we present an analysis of the correctness of PINE. Following [38] and [11], a parallel model model_1 can be optimally simulated by another one model_2 if there exists a compilation algorithm that transforms any program on model_1 with a constant cost C to a program on model_2 with a cost $O(C)$.

Proposition IV.2: A PINE algorithm can be compiled into a PIE algorithm with a cost $O(C)$.

Proof: Any PINE algorithms developed on DKWS can be compiled into a PIE algorithm. Given a PINE algorithm algo that consists of n PEvals (denoted by P_i, where $i \in [1, n]$), n IncEvals (denoted by I_i, where $i \in [1, n]$), and one Assemble (denoted by E). algo is compiled into GRAPE by a PIE algorithm as follows. (a) PEval of GRAPE runs P_is sequentially over the workers. The messages are exchanged by PEval after P_n is executed. (b) IncEval of GRAPE introduces a selection control mechanism by a switch statement. GRAPE plugs I_i into the i-th branch of the switch statement. The control flow of IncEval execution is determined by staleness indicators provided by users. The messages are exchanged at the end of each round of IncEval. (c) Assemble of GRAPE is identical to E.

Due to Proposition IV.2, DKWS inherits all properties of GRAPE (Theorem 1 of [13]), including convergence and correctness theorems.

Theorem IV.2: The general form of a PINE algorithm consists of the following:

1) n PEvals (denoted by P_i, where $i \in [1, n]$),
2) n IncEvals (denoted by I_i, where $i \in [1, n]$), and
3) one Assemble (denoted by E), and any partition strategy Par.

The PINE algorithm on DKWS terminates correctly if

1) I_i satisfies the monotonic condition,[4] for all $i \in [1, n]$; and
2) P_i, I_i and E are correct w.r.t. Par.[5]

Proof: The proof is presented in A.1 of [17].

The correctness of bfkws implemented using PINE is assured by the correctness of bkws and fkws (Sections III-B and III-C) and Theorem IV.2.

Complexities: The time complexity of bkws (resp. fkws) is $O(|Q|(|E| + |V| \log |V|))$ (resp. $O(|\bar{\mathcal{V}}|(|E| + |V| \log |V|))$). The space complexity of bkws and fkws is bounded by

[4]There exists a partial order on the variables attached on the vertices such that IncEval updates the variables in the partial order [13].

[5]P_i is correct if it returns correct answer on an input graph G for any queries. I_i is correct if it returns correct answer on an input graph G and a set of messages for any queries. E is correct if it yields the answer on the input graph G by assembling all the local matches.

$O(|Q||V|)$. The size of PADS(u) is bounded by $O(\log |V|)$. Hence, the overall index size of PADS is bounded by $O(V \log |V|)$ (cf. [19]).

V. Experimental Study

We experimentally evaluate (1) efficiency, (2) performance under different settings, and (3) communication costs on massive graphs with competitors [31] and [9].

A. Experimental Setup

Software and hardware: Our experiments were run on a cluster with eight machines. Each machine had one Xeon X5650 CPU, 128GB memory and was running CentOS 7.4. The implementation was made memory-resident. We used METIS [24] as the graph partition strategy.

Algorithms: We implemented all algorithms in C++. The settings followed [31] and [9] whenever appropriate. Our implementation of PINE was done by modifying the PIE model running on the platform of GRAPE [13]. We used the following implementations for algorithms.

1) DKWS-BF*:* We implemented bfkws using the PIE programming model (detailed in Section IV).
2) DKWS-PADS*:* We applied PADS and KPADS to DKWS-BF for deriving a lower bound between a vertex and a query keyword for pruning the forward expansion as proposed (detailed in Section III).
3) DKWS-NP*:* We applied NP paradigm to DKWS-PADS.
4) DKWS-PINE*:* We applied PINE model to DKWS-NP.
5) Baseline*:* We implemented the distributed algorithms proposed in [31] and [9], both of which share the same keyword semantics as ours. These were established on GRAPE[13], serving as our baseline algorithms. We did not compare DKWS with [47] since their algorithm (a) returns a set of approximate matches, and (b) proposes a different subtree semantic.
6) BANKS-II*:* BANKS-II [21] is the only sequential algorithm we could run on a single machine. In particular, BANKS-II does not require massive indexes. BANKS-II is widely used in the experimental comparison of existing works, such as [16], [44].

Datasets: We used four popular real-world graphs: (a) YAGO3 [26], a large knowledge base with 2.6 million entities and 5.26 million factors; (b) WebUK [3], a large Web graph with 106 million nodes and 3.7 billion edges; (c) DBLP [1] is a social network with 2.2 million authors and 5.4 million collaboration relationships; and (d) DBpedia [2] is a knowledge base with 5.8 million entities and 15.7 million factors. These datasets are widely used in previous keyword search works such as [16], [22], [23], [34] or used to test the scalability of distributed graph evaluation systems, such as [13], [42].

Queries: We followed [47] to generate the queries by varying the number of query keywords $|Q|$. The number ranged from 2 to 6. The average query time is stable when the number of queries is 50. Hence, we generated 50 random synthetic keyword queries for each query size in our experiments and reported the average evaluation time.

Default settings: We fixed $k = 10$, the number of query keywords $|Q|$ to 4, the number of workers to 8, and the τ to 3. Each worker was assigned one fragment. Unless specified otherwise, we conducted experiments with default values of the parameters and varied values of a specific parameter.

B. Experimental Results

Exp-1. Efficiency: We first evaluated the efficiency of DKWS by varying the number of query keywords $|Q|$ from 2 to 6. All algorithms take longer when $|Q|$ gets larger since the size of search space increases. The results are shown in Fig. 12(a) to (d). (a) On YAGO3, DKWS-PADS is on average 1.24 times faster than DKWS-BF. The main reason is that most forward expansions are pruned by PADS. DKWS-NP is 2.32 times faster than Baseline as DKWS-NP avoids the straggler problem. The slower workers are terminated early by using the global upper bound. DKWS-PINE is 3.3 times faster than Baseline since the computing tasks of DKWS-PINE are finer-grained, avoiding the straggler problem and tighter bounds are retrieved by taking advantage of both bkws and fkws.

(b) On WebUK, DKWS-BF is on average 14.6 times faster than Baseline. The reason for such a significant speedup is that a tight local upper bound on WebUK is derived early, since WebUK is denser than the other three datasets. Hence, the vertices, which require forward expansion, are few. DKWS-PADS (resp. DKWS-NP and DKWS-PINE) is 17.05 (resp. 21.45 and 46.8) times faster than Baseline.

(c) On DBLP, the query time of DKWS-BF is 5.73 times faster than Baseline. Since the diameter of DBLP is small, the forward expansion distance is not far. DKWS-PADS is 6.12 times faster than Baseline. Pruning by PADS on DBLP is not as obvious as that on the other three datasets due to the small graph diameter. DKWS-NP (resp. DKWS-PINE) is on average 9.22 (resp. 12.86) times faster than Baseline.

(d) The query performance improvement on DBpedia is similar to that on YAGO3. DKWS-BF (resp. DKWS-PADS, DKWS-NP, and DKWS-PINE) is 5.06 (resp. 5.31, 5.83, and 22.32) times faster than Baseline.

In a nutshell, the performance improvement is due to the following reasons: (a) DKWS-BF avoids the exhaustive explorations by the backward search and forward search; (b) DKWS-PADS prunes the redundant forward search by computing the tight lower bound of the shortest distance between a vertex and a query keyword, which prunes some unnecessary forward search at an early stage; (c) DKWS-NP improves the query performance by exchanging the local upper bounds to yield a global upper bound, which reduces the stragglers' computation; and (d) DKWS-PINE further improves the query performance since it is finer-grained.

Exp-2. Scalability: We next investigated the scalability of DKWS over real-life graphs by varying the number of workers (m) from 2 to 12. a) All algorithms take a shorter time when the number of workers becomes larger, as expected. b) All algorithms scale reasonably well with the increase of m. When m increases from 2 to 12, the running time of Baseline

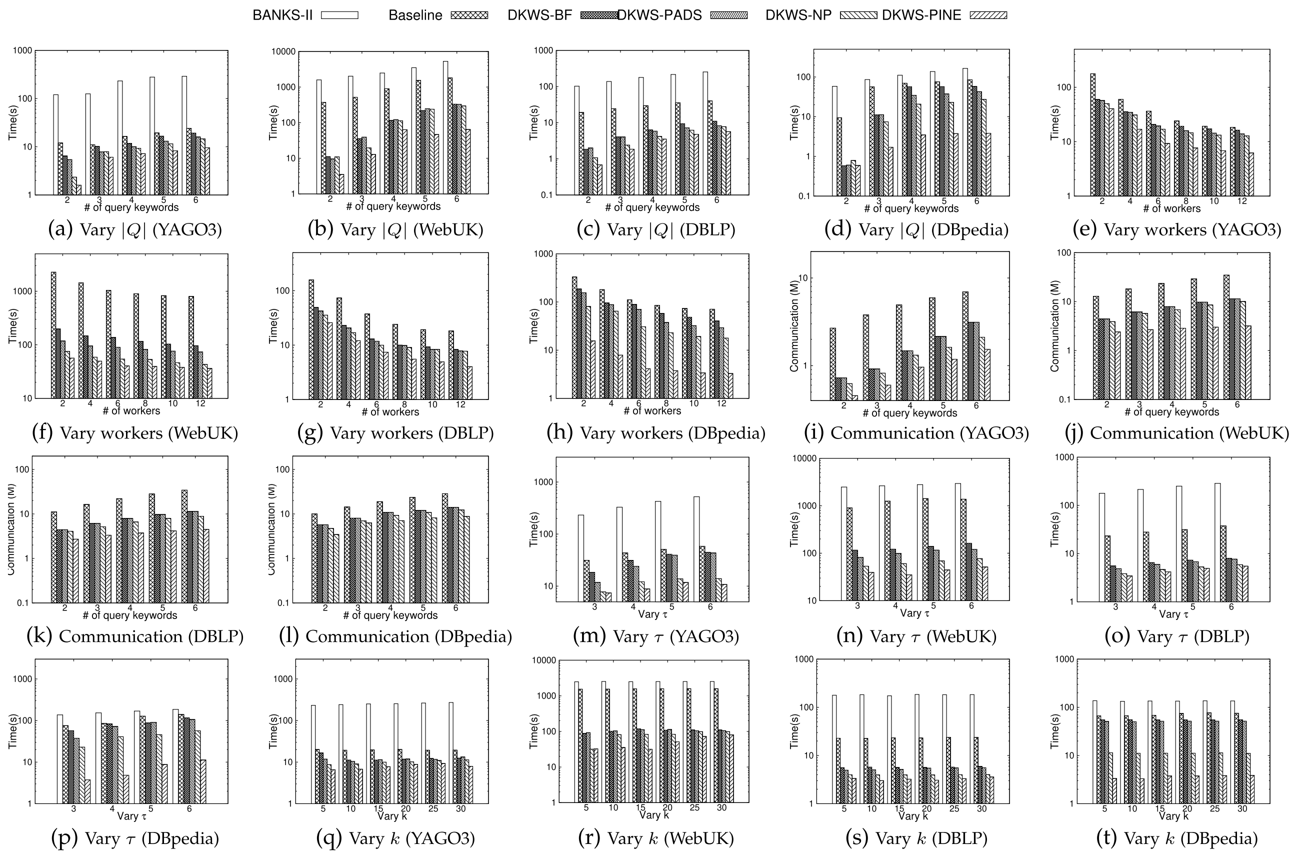


Fig. 12. Query performance on the four real-life datasets

(resp. DKWS-BF, DKWS-PADS, DKWS-PINE and DKWS-NP) decreases to 21.63% (resp. 23.18%, 28.99%, 31.62%, and 25.75%) on average. c) DKWS-NP consistently outperforms Baseline, DKWS-BF, DKWS-PADS and DKWS-PINE for all queries. Specifically, the results are shown in Fig. 12(f) to (h). DKWS-BF and DKWS-PADS take less time when the number of workers increases. More specifically, DKWS-BF is on average 1.65 (resp. 8.81, 2.65, and 1.60) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia), when the number of workers varies from 2 to 12. DKWS-PADS is on average 1.81 (resp. 13.11, 2.92, and 2.11) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). The reason is that DKWS-BF, and DKWS-PADS avoid exhaustive search and prune some redundant stale computations. By exchanging the local upper bounds, DKWS-NP exploits parallelism, since it reduces the straggler problem. DKWS-NP is on average 2.03 (resp. 21.19, 3.31, and 3.67) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). DKWS-PINE is the most efficient since the computing tasks are finer-grained. On average, DKWS-PINE is 3.47 (resp. 26.94, 5.00, and 22.82) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). It is also worth noting that in a single-machine environment, there is no difference in the performance of DKWS-PINE, DKWS-NP, and DKWS-PADS. This is because the fine-grained execution of PINE and notify-push paradigm are not activated in a single-machine setting.

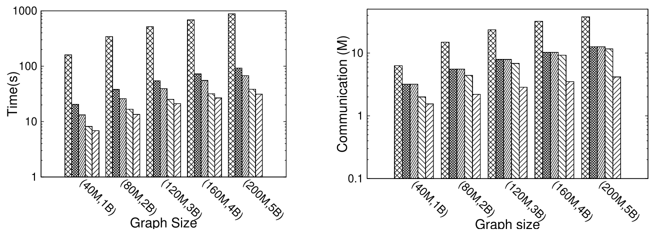


Fig. 13. Scalability on synthetic graphs

Impact of the graph size $|G|$*:* We also evaluated the scalability of DKWS over larger synthetic graphs. We use the graph generator of [13] to produce graphs $G = (V, E, L)$ with L drawn from an alphabet $\mathcal{L}$ of 50 labels. It is controlled by the numbers of nodes $|V|$ and edges $|E|$, up to 200 million and 5 billion, respectively. Fixing $n = 12$, we varied $|G|$ from (40M,1B) to (200M,5B). As reported in Fig. 13, the results are consistent with Fig. 12 over real-life graphs. (a) All algorithms take a longer time when the G gets larger, as expected. (b) DKWS scales reasonably well with the increase of $|G|$. When G increased by 5 times, the running time of Baseline (resp. DKWS-BF, DKWS-PADS, DKWS-PINE and DKWS-NP) increases by 6.7 (resp. 6.3, 6.8, 6.3 and 6.1) times. DKWS-PINE consistently outperforms Baseline, DKWS-BF, DKWS-PADS and DKWS-NP.

Exp-3. Impact of parameters: The elapsed time of keyword search is relevant to the threshold, τ and the number of matches, k. We next present the impact of these parameters.

Impact of threshold τ: τ has been a crucial parameter of keyword search. According to the findings of [6], [47], $\tau = 5$ is large enough to obtain satisfactory matches in real applications. Hence, we next evaluated the scalability of DKWS by varying τ from 3 to 6. a) All algorithms take longer when τ becomes larger, as expected, since there are more candidate answers generated during the backward expansion and forward expansion. b) All algorithms scale reasonably well with the increase of τ. When τ increases from 3 to 6, the running time of Baseline (resp. DKWS-BF, DKWS-PADS, DKWS-PINE, and DKWS-NP) increases by 71.59% (resp. 82.61%, 138.97%, 81.38%, and 83.56%). c) DKWS-PINE consistently outperforms Baseline, DKWS-BF, DKWS-PADS and DKWS-NP for all queries. Specifically, the results are presented in Fig. 12(m) to (p). In particular, DKWS-BF is 1.08 times (resp. 9.18, 4.37, and 1.25) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). DKWS-PADS is 1.33 (resp. 11.80, 4.76, and 1.48) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). DKWS-PADS is more efficient on τ since it can prune longer forward searches when τ increases. DKWS-NP is 3.02 (resp. 19.0, 6.10, and 2.66) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). DKWS-NP is more efficient since it pushed and notified tighter bounds early which was more efficient when τ was large. DKWS-PINE is 3.71 (resp. 29.32, 6.65, and 16.2) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia).

Impact of k: We evaluated the scalability of DKWS by varying k. a) All algorithms take longer when k gets larger since more matches are retrieved. b) All algorithms perform well with the increase of k. When k increases from 5 to 30, the running time of Baseline (resp. DKWS-BF, DKWS-PADS, DKWS-PINE, and DKWS-NP) increases by 4.63% (resp. 1.65%, 9.99%, 60.20%, and 47.22%). c) DKWS-NP outperforms Baseline, DKWS-BF, DKWS-PADS and DKWS-PINE for all queries. Specifically, the experiments are shown in Fig. 12(r) to (s). On average, DKWS-BF is 1.58 (resp. 14.96, 4.07, and 1.30) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). DKWS-PADS is 1.67 (resp. 15.97, 4.39, and 1.39) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). DKWS-NP is 1.98 (resp. 22.92, 5.83, and 6.37) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia). DKWS-PINE is 2.54 (resp. 35.63, 7.14, and 19.66) times faster than Baseline on YAGO3 (resp. WebUK, DBLP, and DBpedia).

Exp-4. Communication costs: We further investigated the communication cost in terms of the total message size. The communication costs on WebUK and DBLP are reported in Fig. 12(j) and (k). The results on other datasets exhibit similar trends. We obtained the following findings. (a) The communication cost of DKWS-PADS is the same as that of DKWS-BF since DKWS-PADS only prunes the local traversals. DKWS-BF and DKWS-PADS ship 33.6% (resp. 36%) of data transmitted by Baseline on WebUK (resp. DBLP). (b) DKWS-NP ships 29.8% (resp. 30.4%) compared to that of Baseline on WebUK (resp. DBLP). This is because DKWS-NP yields tighter bounds and reduces unnecessary message exchange early. (c) DKWS-PINE ships 13.0% (resp. 18%) compared to that of Baseline on WebUK (resp. DBLP). DKWS-PINE takes the advantage of preemptive execution of both bkws and fkws, which reduces long or useless traversals. Consequently, the communication cost is reduced since the messages caused by such traversals have been avoided.

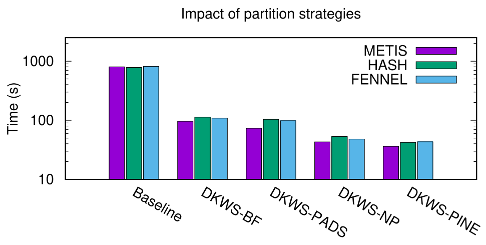


Fig. 14. Impact of partition strategies (WebUK)

Exp-5. Impact of notification counter threshold: We observed that on the four real-life datasets, setting the notification counter threshold to 2 or 3 resulted in a comparatively good performance. However, when the threshold exceeded 4, there was no substantial difference in the performance improvement compared to when the notify-push paradigm was not used. This can be attributed to the fact that on these datasets, the number of times the local bounds were refined rarely exceeded 4; thus, the push function was seldom invoked. The threshold of the notification counter in the coordinator can be determined by a simple experiment offline on the dataset. The details are presented in [17].

Exp-6. Impact of graph partition: We evaluated the impact of different partition strategies, including METIS [24], HASH [12], and FENNEL [37] in Fig. 14. Among these strategies, all algorithms, except for the Baseline, demonstrated faster performance under METIS partitioning. Considering the significance of efficiency, we selected METIS as the default partition strategy for our experiments, as mentioned earlier. Furthermore, we observed that METIS improved performance in DKWS-NP and DKWS-PINE. This can be attributed to the notify-push paradigm employed that helps alleviate the impact of load imbalances.

Exp-7. Comparison with a sequential algorithm: We further compared our works with a sequential algorithm, BANKS-II [21]. The results are shown in Fig. 12(a) to (d). On average, DKWS-PINE is 82.58 times faster than BANKS-II. This verifies that DKWS-PINE has exploited the efficiency of a distributed environment.

VI. Related Work

Keyword search semantics: Recently, keyword search has attracted a lot of interest from both industry and research communities. Bhalotia et al. [5] proposed keyword search on relational databases. He et al. [16] proposed an index, called Blinks to reduce the search time. Kargar et al. [22] proposed distance restrictions on the keyword nodes, i.e., the distance between each pair of keyword nodes is smaller than τ. Shi et al. [34] proposed hub labelings to solve Group Steiner Trees (GST). Kargar et al. [23] proposed an approximate algorithm to retrieve the GST on weighted graphs. These studies optimize a specific keyword search semantic. Jiang et al. [18] proposed a generic index for keyword search semantics running on a standalone machine.

Distributed systems: Several distributed systems have been proposed for graphs. Popular graph systems include Pregel [27], Giraph [4], GraphX [41], GraphLab [25], PowerGrapah [15], Giraph++ [35], Blogel [42], GPS [32], GRAPE [13], and AAP [11]. Pregel [27] and Giraph [4] are implemented with the vertex-centric programming model. A superstep executes a user-defined function at each vertex in parallel. GraphX [41] is a component built on top of Spark for graphs which exposes a set of operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel [27]. Blogel [42], Giraph++ [35] and GRAPE [13] are implemented with the block-centric programming model. AAP [11] proposes an adaptive asynchronous parallel model for graph computations on [13]. These systems are general-purpose. Keyword search algorithms have not been exploited. For instance, DKWS can also be beneficial to existing systems. By integrating PINE, the systems could make the query evaluation more fine-grained. By integrating the notify-push paradigm, DKWS allows each worker to broadcast local information to their peer workers which can alleviate the straggler problem.

Distributed kws *algorithms:* Lu et al. [31] proposed a scalable algorithm for keyword search in MapReduce. However, the false matches were pruned at the last superstep, which may cause large messages. Yuan et al. [47] proposed a search strategy based on a compressed signature to avoid the exhaustive flooding search. [47] sent all the local candidate matches to the coordinator at runtime which may require large messages and extra synchronization cost. DKWS differs from the above in the following aspects: (a) each worker computes the top-k matches locally. DKWS sends the local matches to the coordinator when all of the workers terminate rather than sends massive local candidates matches; and (b) DKWS exchanges the local upper bounds which prune some traversals early.

VII. Conclusions and Future Works

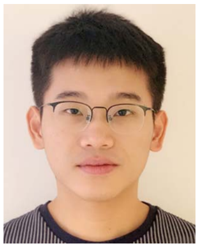
In this paper, we propose a distributed keyword search system called DKWS. We derive new bounds for pruning some keyword searches that tackle the performance challenges of a general distributed system. We show that bfkws, which can be used to express query algorithms for popular keyword semantics, has a monotonic property that ensures the correct parallelization. We propose a notify-push paradigm allows asynchronously exchanging the upper bounds across the workers and the coordinators. We also propose a programming model PINE for DKWS which fits keyword search algorithms as they have distinguished n phases, to allow preemptive searches to mitigate staleness in a distributed system. We verify that DKWS significantly reduces the runtimes of distributed top-k keyword searches.

In the future, we plan to implement PINE into the latest codebase of GRAPE. Moreover, we will extend DKWS to support *approximate* analysis for some keyword search semantics, such as [22], [23], [34].

References

[1] DBLP, 2021. [Online]. Available: https://dblp.org/
[2] DBpedia, 2022. [Online]. Available: http://wiki.dbpedia.org/Datasets
[3] WebUK, 2022. [Online]. Available: http://law.di.unimi.it/webdata/uk-union-2006--06-2007-05
[4] C. Avery, “Giraph: Large-scale graph processing infrastructure on hadoop,” in *Proc. Hadoop Summit. Santa Clara*, vol. 11, no. 3, pp. 5–9, 2011.
[5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, “Keyword searching and browsing in databases using banks,” in *Proc. IEEE Int. Conf. Data Eng.*, 2002, pp. 431–440.
[6] J. Coffman and A. C. Weaver, “An empirical performance evaluation of relational keyword search techniques,” *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 1, pp. 30–42, Jan. 2014.
[7] E. Cohen, “All-distances sketches, revisited: Hip estimators for massive graphs analysis,” *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 9, pp. 2320–2334, Sep. 2015.
[8] W. Fan, C. Hu, M. Liu, P. Lu, Q. Yin, and J. Zhou, “Dynamic scaling for parallel graph computations,” in *Proc. VLDB Endowment*, vol. 12, no. 8, pp. 877–890, 2019.
[9] W. Fan, C. Hu, and C. Tian, “Incremental graph computations: Doable and undoable,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 155–169.
[10] W. Fan et al., “Application driven graph partitioning,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1765–1779.
[11] W. Fan et al., “Adaptive asynchronous parallelization of graph algorithms,” *ACM Trans. Database Syst.*, vol. 45, no. 2, pp. 1–45, 2020.
[12] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang, “GRAPE: Parallelizing sequential graph computations,” in *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1889–1892, 2017.
[13] W. Fan et al., “Parallelizing sequential graph computations,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 495–510.
[14] Y. Fang, R. Cheng, S. Luo, and J. Hu, “Effective community search for large attributed graphs,” in *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1233–1244, 2016.
[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: Distributed graph-parallel computation on natural graphs,” in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, pp. 17–30, 2012.
[16] H. He, H. Wang, J. Yang, and P. S. Yu, “Blinks: Ranked keyword searches on graphs,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*2007, pp. 305–316.
[17] J. Jiang, B. Choi, X. Huang, J. Xu, and S. S. Bhowmick, “DKWS: An efficient distributed system for keyword search on massive graphs,” 2023. [Online]. Available: https://www.comp.hkbu.edu.hk/%7Ebchoi/DKWS.pdf
[18] J. Jiang, B. Choi, J. Xu, and S. S. Bhowmick, “A generic ontology framework for indexing keyword search on massive graphs,” *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 6, pp. 2322–2336, Jun. 2020.
[19] J. Jiang, X. Huang, B. Choi, J. Xu, S. S. Bhowmick, and L. Xu, “PPKWS: An efficient framework for keyword search on public-private networks,” in *Proc. IEEE Int. Conf. Data Eng.*, 2020, pp. 457–468.
[20] M. Jiang, A.-C. Fu, and R. C.-W. Wong, “Exact top-k nearest keyword search in large networks,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 393–404.
[21] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, “Bidirectional expansion for keyword search on graph databases,” in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 505–516.
[22] M. Kargar and A. An, “Keyword search in graphs: Finding r-cliques,” in *Proc. VLDB Endowment*, vol. 4, no. 10, pp. 681–692, 2011.
[23] M. Kargar, L. Golab, D. Srivastava, J. Szlichta, and M. Zihayat, “Effective keyword search over weighted graphs,” *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 2, pp. 601–616, Feb. 2022.
[24] G. Karypis, “METIS: Unstructured graph partitioning and sparse matrix ordering system,” Dept. Comput. Sci., Univ. Minnesota, Minneapolis, MN, USA, Tech. Rep., 1997.
[25] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed graphlab: A framework for machine learning and data mining in the cloud,” in *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[26] F. Mahdisoltani, J. Biega, and F. Suchanek, "YAGO3: A knowledge base from multilingual Wikipedias," in *Proc. 7th Biennial Conf. Innov. Data Syst. Res.*, 2014.
[27] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[28] S. Michel, P. Triantafillou, and G. Weikum, "Klee: A framework for distributed top-k query algorithms," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 637–648.
[29] A. Pacaci and M. T. Özsu, "Experimental analysis of streaming algorithms for graph partitioning," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 1375–1392.
[30] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian, "Top-k nearest keyword search on large graphs," in *Proc. VLDB Endowment*, vol. 6, no. 10, pp. 901–912, 2013.
[31] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in mapreduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 827–838.
[32] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. 25th Int. Conf. Sci. Stat. Database Manage.*, 2013, pp. 22:1–22:12.
[33] J. Shi, D. Wu, and N. Mamoulis, "Top-k relevant semantic place retrieval on spatial RDF data," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1977–1990.
[34] S. G. Cheng and E. Kharlamov, "Keyword search over knowledge graphs via static and dynamic hub labelings," in*Proc. Web Conf.*, Taipei, Taiwan, 2020, pp. 235–245.
[35] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," in *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.
[36] Y. Tian, R. A. Hankins, and J. M. Patel, "Efficient aggregation for graph summarization," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 567–580.
[37] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Proc. 7th ACM Int. Conf. Web Search Data Mining*, 2014, pp. 333–342.
[38] L. G. Valiant, "General purpose parallel architectures," in *Algorithms and Complexity*. Amsterdam, The Netherlands: Elsevier, 1990, pp. 943–971.
[39] H. Wang and C. C. Aggarwal, "A survey of algorithms for keyword search on graph data," in *Managing and Mining Graph Data*. Berlin, Germany: Springer, 2010, pp. 249–273.
[40] Y. Wu, S. Yang, M. Srivatsa, A. Iyengar, and X. Yan, "Summarizing answer graphs induced by keyword queries," in *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1774–1785, 2013.
[41] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on spark," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, pp. 1–6.
[42] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," in *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.
[43] J. Yang, W. Yao, and W. Zhang, "Keyword search on large graphs: A. survey," *Data Sci. Eng.*, vol. 6, no. 2, pp. 142–162, 2021.
[44] Y. Yang, D. Agrawal, H. Jagadish, A. K. Tung, and S. Wu, "An efficient parallel keyword search engine on knowledge graphs," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 338–349.
[45] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu, "AutoG: A visual query autocompletion framework for graph databases," in *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1505–1508, 2016.
[46] J. X. Yu, L. Qin, and L. Chang, "Keyword search in relational databases: A survey," *IEEE Data Eng. Bull.*, vol. 33, no. 1, pp. 67–78, 2010.
[47] Y. Yuan, X. Lian, L. Chen, J. X. Yu, G. Wang, and Y. Sun, "Keyword search over distributed graphs with compressed signature," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 6, pp. 1212–1225, Jun. 2017.
[48] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao, "Semantic sparql similarity search over RDF knowledge graphs," in *Proc. VLDB Endowment*, vol. 9, no. 11, pp. 840–851, 2016.



Jiaxin Jiang received the BEng degree in computer science and engineering from Shandong University, in 2015 and the PhD degree in computer science from Hong Kong Baptist University (HKBU), in 2020. He is a research fellow with the School of Computing, National University of Singapore. His research interests include graph-structured databases, distributed graph computation and fraud detection.

Byron Choi received the bachelor's of engineering degree in computer engineering from the Hong Kong University of Science and Technology (HKUST), in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania, in 2002 and 2006, respectively. He is a professor with the Department of Computer Science, Hong Kong Baptist University. His research interests include graph data management and time series analysis.

Xin Huang received the PhD degree from the Chinese University of Hong Kong (CUHK), in 2014. He is currently an associate professor with Hong Kong Baptist University. His research interests mainly focus on graph data management and mining.

Jianliang Xu (Senior Member, IEEE) is a professor with the Department of Computer Science, Hong Kong Baptist University (HKBU). He held visiting positions with Pennsylvania State University and Fudan University. He has published more than 150 technical papers in these areas, most of which appeared in leading journals and conferences including SIGMOD, VLDB, ICDE, *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, and *VLDB Journal*.

Sourav S. Bhowmick is an associate professor with the School of Computer Science and Engineering (SCSE), Nanyang Technological University, Singapore. His core research expertise is in data management, human-data interaction, and data analytics. His research has appeared in premium venues such as ACM SIGMOD, VLDB, and *VLDB Journal*. He is co-recipient of Best Paper Awards in ACM CIKM 2004, ACM BCB 2011, and VLDB 2021. He is also co-recipient of the 2021 ACM SIGMOD Research Highlights Award. He was inducted into distinguished members of ACM in 2020.