DKWS: A Distributed System for Keyword Search on Massive Graphs

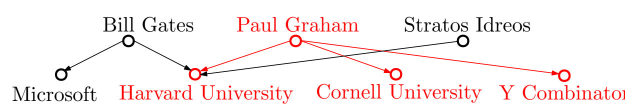
Jiaxin Jiang, Byron Choi, Xin Huang, Jianliang Xu and Sourav S Bhowmick

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.

1 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]

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]).



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

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 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 dis-

- *Jiaxin Jiang is with the School of Computing, National University of Singapore, Singapore. Some work was done while Dr. Jiang was with Hong Kong Baptist University.*
  E-mail: jxjiang@nus.edu.sg
- *Byron Choi, Xin Huang and Jianliang Xu are with the Department of Computer Science, Hong Kong Baptist University, Hong Kong. Corresponding author: Byron Choi.*
  E-mail: {bchoi, xinhuang, xujl}@comp.hkbu.edu.hk
- *Sourav.S. Bhowmick is with School of Computer Engineering, Nanyang Technological University, Singapore.*
  E-mail: assourav@ntu.edu.sg

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

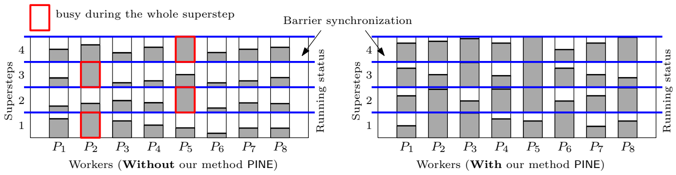
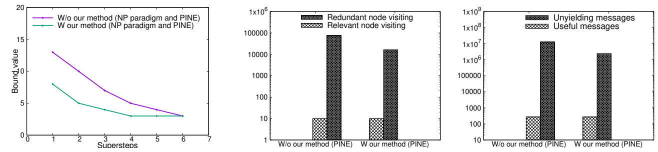


Fig. 2: Illustration of stragglers of distributed keyword search (grey denotes the worker P_i is busy, whereas white color denotes the worker P_i is idle.)



(a) Pruning bounds (b) Visited nodes (c) Messages

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

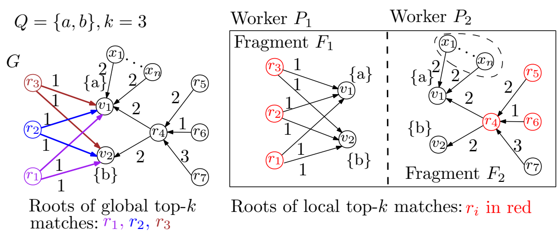


Fig. 3: An example of distributed keyword search

tributed 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. 4a. 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. 4b. 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.

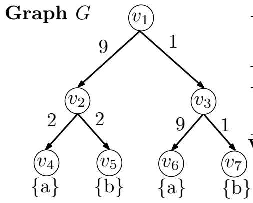
Challenge 3: Message passing. Since keyword search on graphs often involves numerous traversals, keyword search on distributed graphs might cause massive message passing. For instance, in previous studies [47] and [28], the local matches rooted at r_i ($i \in [1, 2, 3]$) (resp. r_i ($i \in [4, 5, 6]$)) 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. 4c, 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 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



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: An example of frequently used graph notations

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. Sec. 2 provides some background and the problem statement. In Sec. 3, we illustrate an efficient monotonic sequential keyword search algorithm. In Sec. 4, we propose DKWS and its two novel ideas, namely the notify-push paradigm and the PINE model. Sec. 5 reports experimental results. Sec. 6, presents the related work. We conclude the paper and present the future works in Sec. 7.

2 Preliminaries and problem statement

This section presents some background and the problem statement. Some frequently used notations are summarized in Table 1.

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) $L{:}V \rightarrow \Sigma$ 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 2.1. 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.*, $w(v_1, v_2) = 9$.

Partition strategy. Given a number m, a strategy Par partitions a graph G into *fragments* $\mathcal{F} = \{F_1, \ldots, F_m\}$ 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.

TABLE 1: Frequently used notations

Notation	Meaning
Q	a set of query keywords $Q = \{q_1, q_2, \ldots q_l\}$
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
$\mathsf{mat}_u/\mathsf{mat}^b_u/\mathsf{mat}^f_u$	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) $\{F_1, \ldots, F_m\}$
M_i	messages designated to worker P_i

TABLE 2: 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]

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 Sec. 4) 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 Tab. 2. 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 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

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

(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 2.1 (Score function $\mathsf{scr}(u)$***).*** Given a match, $T = \{u, \langle v_1, \dots v_l\rangle\}$, 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.

3 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 Sec. 3.2 and 3.3, respectively). The monotonic property of a few popular keyword search algorithms, such as [9], [16], [31], can be analyzed similarly, which is omitted.

3.1 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 3.1 (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 3.2 (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 $\mathsf{mat}_u[q]$. $\mathsf{mat}_u[q] \in \{\sum_{e_i\in E'} w(e_i) | E' \subseteq E\} \cup \{+\infty\}$.

(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 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.

3.2 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 Algo. 2, to be discussed with PINE in Sec. 4.2. 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.*, $u_i = \arg\min_u \mathsf{dist}(u, v)$, 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 3.1. 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,

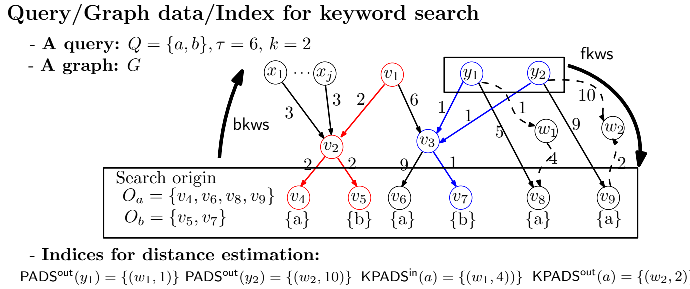


Fig. 6: A query, a data graph (top) and indexes (bottom) for the illustration of the key steps of bkws and fkws

$O_a = \{v_4, v_6, v_8, v_9\}$ 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 3.1. 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 Appx. A.1 of [17]. □

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. Prop. 3.1 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 (Sec. 3.3).

Example 3.2. 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.

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 $\{\sum_{e_i \in E'}$

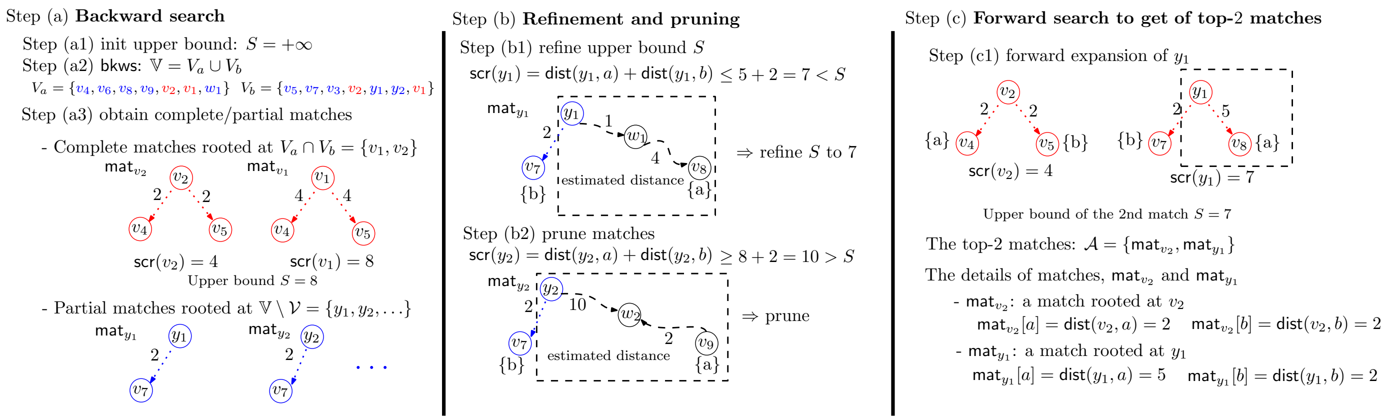


Fig. 7: Key steps: backward search (Sec. 3.2), refinement and pruning (Example 3.3 and 3.4), and forward search (Sec. 3.3)

$w(e_i)\} \cup \{+\infty\}$, where E' is any subset of E. By Def. 3.2, 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 Appx. B.5 of [17]. The size of a match, mat_u, is bounded by $O(|Q|)$. Hence, the space complexity is bounded by $O(|Q||V|)$.

3.3 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 Prop. 3.2.

***Proposition* 3.2.** 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.
(i) $q \in L(v)$, *i.e.*, the keyword q is found;
(ii) $\mathsf{dist}(u, v) > \tau$, the vertex containing keyword q is farther than τ from u or does not exist (*i.e.*, $\mathsf{dist}(u, q) > \tau$); or
(iii) $\mathsf{scr}(u) + \mathsf{dist}(u, v) > S \Rightarrow \mathsf{scr}(u) + \mathsf{dist}(u, q) > S$.

As indicated by Condition (ii), if $\mathsf{dist}(u, q)$ has been indexed, early termination can be determined if $\mathsf{dist}(u, q) > \tau$. Furthermore, Condition (iii) 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:

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

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 3.3. 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.

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

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 Prop. 3.2-Condition (ii) is already satisfied. Similarly, if the lower bound of the score of the match rooted at u is larger than S, Prop. 3.2-Condition (iii) is met.

Example 3.4. 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].

4 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.

Algorithm 1: API of DKWS

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

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 (Sec. 4.1) at runtime; and (2) PINE consists of PEval, IncEval (n subtasks), and one Assemble functions (Sec. 4.2) 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 Sec. 3) between them are possible.

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

Definition 4.1 (Notify API). Notify(i, S_i) is an API that a worker refines the global upper bound S with the local

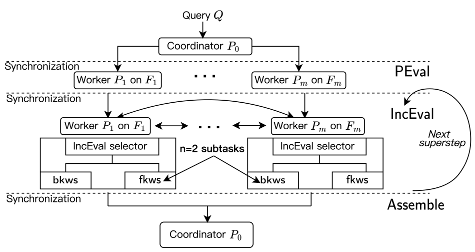


Fig. 8: Workflow of DKWS

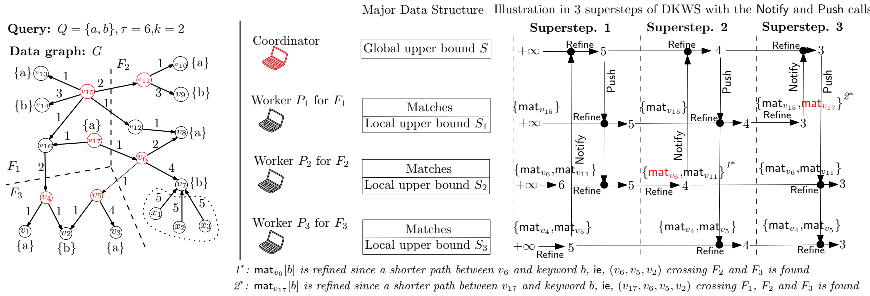


Fig. 9: Illustration of the change of bounds during query processing of DKWS

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 4.2 **(**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 4.1. $\forall i \in [1, m]$, $S_i \leq S$.

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 4.1. 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 Sec. 3.2. 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.

4.2 PINE **programming model**

4.2.1 Overview of PINE

The overview of PINE is illustrated with Fig. 8. PINE consists of PEval and IncEval of 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 Sec. 1).

We next illustrate the PINE programming model with an efficient implementation of bfkws. There are two subtasks, bkws (Sec. 4.2.2) and fkws (Sec. 4.2.3), 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_u^b (resp. mat_u^f) to denote the partial match found by bkws (resp. fkws) rooted

Algorithm 2: PEval for bkws

Input: $F_i(V, E, L)$, $Q = \{q_1, \ldots, q_l\}$, τ
Output: $Q(F_i)$ consisting of current 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 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** 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: $F_i(V, E, L)$, $Q = \{q_1, \ldots, q_l\}$, τ, $Q(F_i)$, message M_i
Output: $Q(F_i \oplus M_i)$ consisting of current $\mathsf{mat}_u^b \in \mathcal{A}_i$, where $u \in V$

init V_q, $\mathcal{P}_q$ for each query keyword $q \in Q$
foreach $\mathsf{mat}_u^{b,\mathsf{in}} \in M_i$ **do**
  foreach $q \in Q$ and $\mathsf{mat}_u^b[q] > \mathsf{mat}_u^{b,\mathsf{in}}[q]$ **do**
    $\mathsf{mat}_u^b[q] = \mathsf{mat}_u^{b,\mathsf{in}}[q]$
    $\mathcal{P}_q$.push($\langle u, \mathsf{mat}_u^b[q]\rangle$)
BackwardExpand($\mathcal{P}$) // $\mathcal{P} = \{\mathcal{P}_q | q \in Q\}$
Message segment: $M_i = \{\mathsf{mat}_u^b | u \in F_i.I\}$

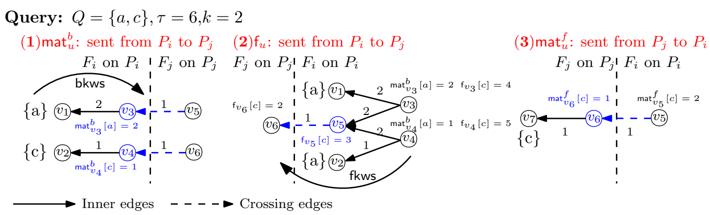


Fig. 10: Message exchange during query processing (mat_u^b (resp. mat_u^f): keep track the shortest distance between u and a query keyword by bkws (resp. fkws); and f_u: the longest distance needed to be forward expanded starting from u)

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.

4.2.2 PI for bkws

Message declaration. DKWS declares a variable mat_u^b for each vertex u, where mat_u^b is a map such that $\mathsf{mat}_u^b[q] = \langle \mathsf{v}, \mathsf{d}\rangle$ 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.

(1) Partial evaluation (PEval) for bkws (Algo. 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 11-22 present the pseudo-code of bkws (described in Sec. 3.2). 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 (Algo. 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 $V^s_{q_j,i}$. We denote the union set of all the visited vertices by $\mathbb{V}$. Hence, $\mathbb{V} = \bigcup_{i\in[1,m], j\in[1,l], s\in[1,R]} V^s_{q_j,i}$, where m is the number of workers and $l = |Q|$. We have the following proposition.

Proposition 4.1. 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 Appx. A.2 of [17]. □

Example 4.2. 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.

4.2.3 PI for fkws

Message declaration. DKWS declares a variable mat_u^f, where mat_u^f is a map, $\mathsf{mat}_u^f[q] = \langle \mathsf{v}, \mathsf{d}\rangle$, 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_u^f 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 (Algo. 4). fkws mainly conducts the forward expansion to complete the partial

Algorithm 4: PEval for fkws

Input: $F_i(V, E, L)$, $Q = \{q_1, \ldots, q_l\}$, τ
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 mat_u^f and a forward distance 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                 $\mathsf{mat}_u^f[q] = \mathsf{dist}(u, v)$
13   marks that $q \in \mathsf{f}_u$ is found
14   **else if** $\mathsf{dist}(u, v) > \mathsf{mat}_u^f[q]$ **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` mat_u^f `by a found match` mat_v
17                 $\mathsf{mat}_u^f[q] = \min\{\mathsf{mat}_u^f[q], \mathsf{dist}(u, v) + \mathsf{mat}_v[q]\}$
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** 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         $S_i = \mathsf{scr}(\mathcal{A}_i.\mathsf{top}().\mathsf{u})$
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:* $M_i^1 = \{\mathsf{mat}_u^f | u \in F_i.I\}$ and $M_i^2 = \{\mathsf{f}_u | u \in F_i.O\}$

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 Prop. 3.2 is met, the expansion is terminated (Lines 11-15). Suppose u is expanded to vertex v and the missing keyword q is found in $\mathsf{mat}_v = \mathsf{mat}_v^b \cup \mathsf{mat}_v^f$, $\mathsf{mat}_u^f[q]$ is refined (Lines 16-17). 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_u^f (resp. f_u) are grouped into M_i^1 (resp. M_i^2) in worker P_i. $M_{i,j} \in M_i$ is sent to worker P_j. Formally, $M_{i,j}^1 = \{\mathsf{mat}_u | u \in F_i.I \cap F_j.O\}$ and $M_{i,j}^2 = \{\mathsf{f}_u | u \in F_i.O \cap F_j.I\}$ 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. Firstly, P_i receives the partial matches, mat_u^f 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 $\mathsf{mat}_u^f[q]$ is refined. IncEval propagates the distance refinement to the ancestor vertices.

(2.2) Incremental forward expansion. Secondly, 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 $\mathsf{f}_u^{\mathsf{in}} \in M_j^2$ 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 $\mathsf{f}_u^{\mathsf{in}}[q]$ is smaller than $\mathsf{f}_u[q]$, $\mathsf{f}_u^{\mathsf{in}}[q]$ is skipped.

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

Example 4.3. 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 $\mathsf{mat}_{v_6}^f[c] = 1$ (shown in Fig. 10.(3)), $\mathsf{mat}_{v_6}^f[c] = 1$ is sent to F_i via the portal node $v_6 \in F_j.I$.

4.2.4 Preemptive execution of IncEvals in PINE

Even if the complexity of bkws (analyzed in Sec. 3.2) is smaller than that of fkws (analyzed in Sec. 3.3), 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 $\mathbb{B}_i^b$ and $\mathbb{B}_i^f$ to keep track of backward and forward messages from other workers. The more messages are accumulated in $\mathbb{B}_i^b$ (resp. $\mathbb{B}_i^f$), the earlier P_i should start bkws (resp. fkws) computation, and vice versa.

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*.

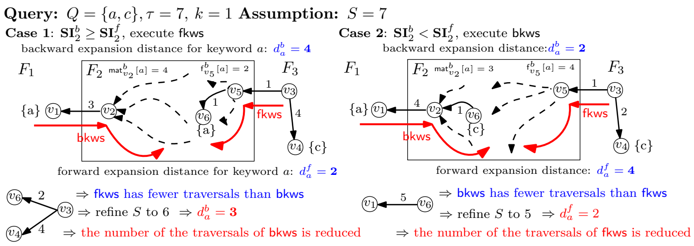


Fig. 11: Illustration of the preemptive execution

<u>*Staleness indicators.*</u> 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:

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

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

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 4.4. 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.

4.2.5 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 4.5. 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.

4.3 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 4.2. 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 Prop. 4.2, DKWS inherits all properties of GRAPE (Theorem 1 of [13]), including convergence and correctness theorems.

Theorem 4.2. 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

(a) I_i satisfies the monotonic condition,[4] for all $i \in [1, n]$; and
(b) 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 (Sec. 3.2 and 3.3) and Theorem 4.2.

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

5 EXPERIMENTAL STUDY

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

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.

TABLE 3: Statistics of real-world datasets

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

5.1 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 Sec. 4).

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 Sec. 3).

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.

5.2 Experimental results

Exp-1: Efficiency. We firstly 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 Fig. 12(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.

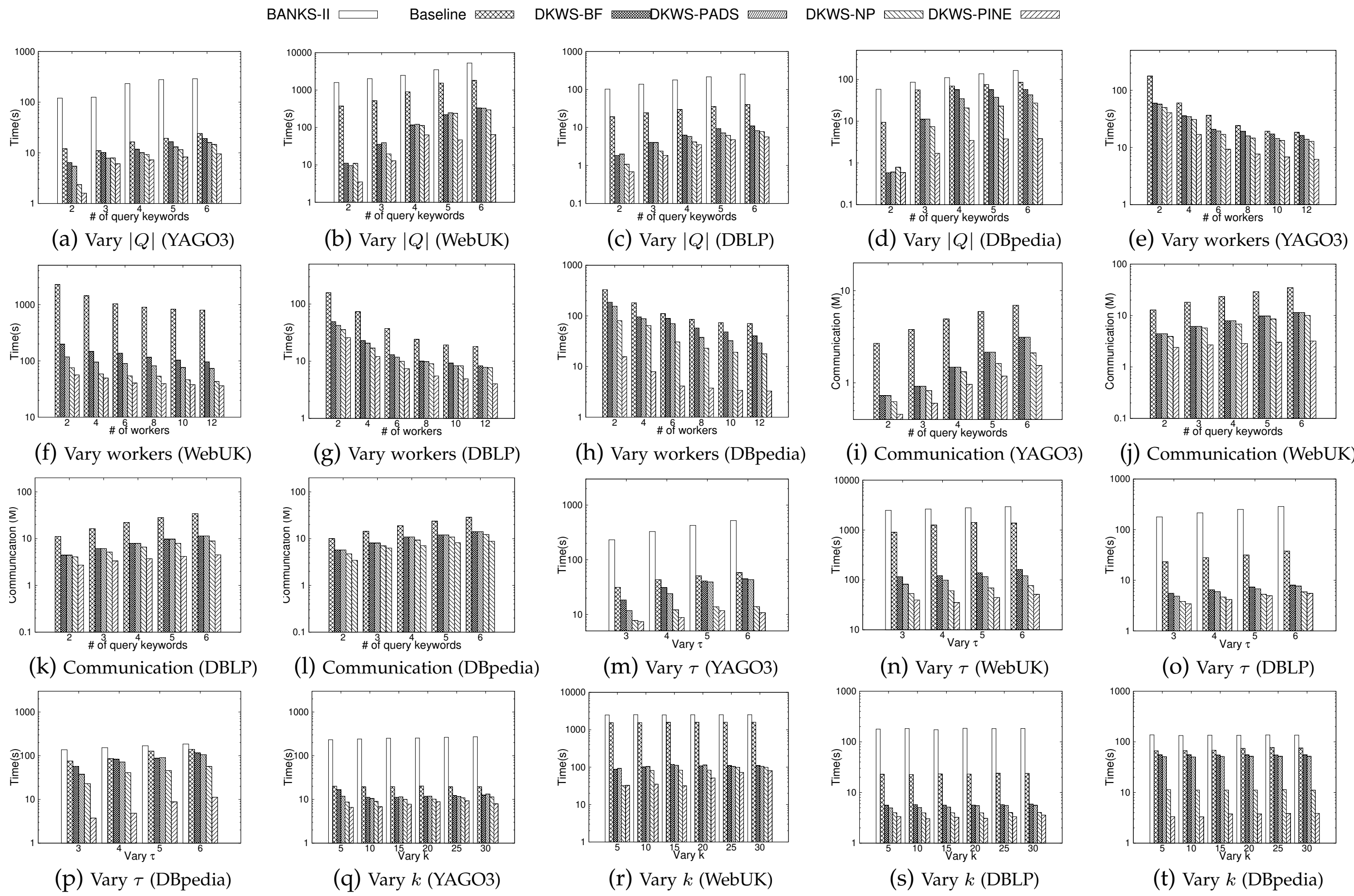
(b) On WebUK, DKWS-BF is on average 14.6 times faster than Baseline. The reason for such a significant speedup is that a tight local upper bound on WebUK is derived early, since WebUK is denser than the other three datasets. Hence, the vertices, which require forward expansion, are few. DKWS-PADS (resp. 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 (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 Fig. 12(h). DKWS-BF and DKWS-PADS take less time when the number of workers increases.



(a) Vary $|Q|$ (YAGO3) (b) Vary $|Q|$ (WebUK) (c) Vary $|Q|$ (DBLP) (d) Vary $|Q|$ (DBpedia) (e) Vary workers (YAGO3)
(f) Vary workers (WebUK) (g) Vary workers (DBLP) (h) Vary workers (DBpedia) (i) Communication (YAGO3) (j) Communication (WebUK)
(k) Communication (DBLP) (l) Communication (DBpedia) (m) Vary τ (YAGO3) (n) Vary τ (WebUK) (o) Vary τ (DBLP)
(p) Vary τ (DBpedia) (q) Vary k (YAGO3) (r) Vary k (WebUK) (s) Vary k (DBLP) (t) Vary k (DBpedia)

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

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.

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 $(40\text{M}, 1\text{B})$ to $(200\text{M}, 5\text{B})$. 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.

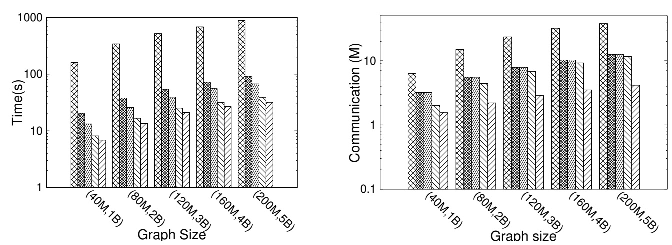


Fig. 13: Scalability on synthetic graphs

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 Fig. 12(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 Fig. 12(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 Fig. 12(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.

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].

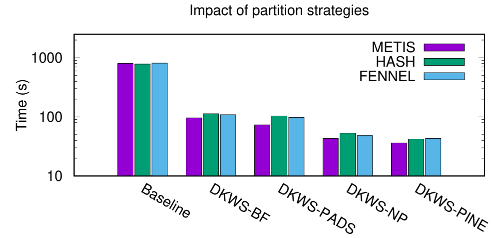


Fig. 14: Impact of partition strategies (WebUK)

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 Fig. 12(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.

6 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.

7 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.

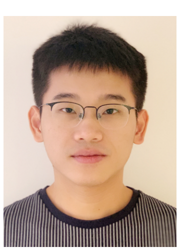
In the future, we plan to implement PINE into the latest codebase of GRAPE. Moreover, we will extend DKWS to support *approximate* analysis for some keyword search semantics, such as [22], [23], [34].

Acknowledgements. This work is supported by HKRGC GRF 12203123, 12201119, 12200022, and 12202221, and C2004-21GF.

References

[1] DBLP. *https://dblp.org/*.
[2] DBpedia. *http://wiki.dbpedia.org/Datasets*.
[3] WebUK. *http://law.di.unimi.it/webdata/uk-union- 2006-06-2007-05*.
[4] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.
[5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440. IEEE, 2002.
[6] J. Coffman and A. C. Weaver. An empirical performance evaluation of relational keyword search techniques. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):30–42, 2012.
[7] E. Cohen. All-distances sketches, revisited: Hip estimators for massive graphs analysis. *IEEE Transactions on Knowledge and Data Engineering*, 27(9):2320–2334, 2015.
[8] W. Fan, C. Hu, M. Liu, P. Lu, Q. Yin, and J. Zhou. Dynamic scaling for parallel graph computations. *Proceedings of the VLDB Endowment*, 12(8):877–890, 2019.
[9] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, pages 155–169. ACM, 2017.
[10] W. Fan, R. Jin, M. Liu, P. Lu, X. Luo, R. Xu, Q. Yin, W. Yu, and J. Zhou. Application driven graph partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1765–1779, 2020.
[11] W. Fan, P. Lu, W. Yu, J. Xu, Q. Yin, X. Luo, J. Zhou, and R. Jin. Adaptive asynchronous parallelization of graph algorithms. *ACM Transactions on Database Systems (TODS)*, 45(2):1–45, 2020.
[12] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang. Grape: Parallelizing sequential graph computations. *Proceedings of the VLDB Endowment*, 10(12):1889–1892, 2017.
[13] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *SIGMOD*, pages 495–510. ACM, 2017.
[14] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *PVLDB*, 9(12):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 *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.
[16] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.
[17] J. Jiang, B. Choi, X. Huang, J. Xu, and S. S. Bhowmick. Dkws: An efficient distributed system for keyword search on massive graphs. *https://www.comp.hkbu.edu.hk/%7Ebchoi/DKWS.pdf*, 2023.
[18] J. Jiang, B. Choi, J. Xu, and S. S. Bhowmick. A generic ontology framework for indexing keyword search on massive graphs. In *TKDE*, 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 *ICDE*, pages 457–468, 2020.
[20] M. Jiang, A. W.-C. Fu, and R. C.-W. Wong. Exact top-k nearest keyword search in large networks. In *SIGMOD*, pages 393–404. ACM, 2015.
[21] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *PVLDB*, pages 505–516, 2005.
[22] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.
[23] M. Kargar, L. Golab, D. Srivastava, J. Szlichta, and M. Zihayat. Effective keyword search over weighted graphs. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.
[24] G. Karypis and V. Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.
[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. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.
[26] F. Mahdisoltani, J. Biega, and F. Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *Seventh Biennial Conference on Innovative Data Systems Research*, 2014.
[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010.
[28] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *Proceedings of the 31st*

International Conference on Very Large Data Bases, pages 637–648, 2005.
[29] A. Pacaci and M. T. Özsu. Experimental analysis of streaming algorithms for graph partitioning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1375–1392, 2019.
[30] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):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 *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 827–838. ACM, 2014.
[32] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.
[33] J. Shi, D. Wu, and N. Mamoulis. Top-k relevant semantic place retrieval on spatial rdf data. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1977–1990, 2016.
[34] Y. Shi, G. Cheng, and E. Kharlamov. Keyword search over knowledge graphs via static and dynamic hub labelings. In Y. Huang, I. King, T. Liu, and M. van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 235–245. ACM / IW3C2, 2020.
[35] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.
[36] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, pages 567–580, 2008.
[37] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014.
[38] L. G. Valiant. General purpose parallel architectures. In *Algorithms and Complexity*, pages 943–971. Elsevier, 1990.
[39] H. Wang and C. C. Aggarwal. A survey of algorithms for keyword search on graph data. In *Managing and mining graph data*, pages 249–273. Springer, 2010.
[40] Y. Wu, S. Yang, M. Srivatsa, A. Iyengar, and X. Yan. Summarizing answer graphs induced by keyword queries. *PVLDB*, 6(14):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 *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.
[42] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.
[43] J. Yang, W. Yao, and W. Zhang. Keyword search on large graphs: A survey. *Data Science and Engineering*, 6(2):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 *2019 IEEE 35th international conference on data engineering (ICDE)*, pages 338–349. IEEE, 2019.
[45] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu. Autog: A visual query autocompletion framework for graph databases. *PVLDB*, 9(13):1505–1508, 2016.
[46] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 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 Transactions on Knowledge and Data Engineering*, 29(6):1212–1225, 2017.
[48] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao. Semantic sparql similarity search over rdf knowledge graphs. *PVLDB*, 9(11):840–851, 2016.



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

Byron Choi is a Professor in the Department of Computer Science at the Hong Kong Baptist University. He received the bachelor 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. His research interests include graph data management and time series data analyses.

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

Jianliang Xu is a Professor in the Department of Computer Science, Hong Kong Baptist University (HKBU). He held visiting positions at 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, TODS, TKDE, and VLDBJ.

Sourav S. Bhowmick is an Associate Professor in 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. Sourav was inducted into Distinguished Members of the ACM in 2020.

Appendix A
Appendix

A.1 Proof of Prop. 3.1

Proposition A.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: 1) Suppose $u \notin \mathbb{V}$. We assume that the last vertex which is backward expanded for keyword q_i as v_i. The distance between v_i and the query keyword $q_i \in Q$ is $\mathsf{dist}(v_i, q_i)$. It is worth noting that in Step 2 of the backward search, the vertex with the shortest distance to O_i is chosen for expansion.

Since u has not been visited,

$$\mathsf{dist}(u, q_i) \geq \mathsf{dist}(v_i, q_i) \tag{5}$$

Hence,

$$\mathsf{scr}(u) = \Sigma\mathsf{dist}(u, q_i) \geq \Sigma\mathsf{dist}(v_i, q_i) > S \tag{6}$$

The match rooted at u is not among the top-k matches. Therefore, $\mathsf{mat}_u \notin \mathcal{A}$.

2) We prove the second part of the proposition by contradiction. Suppose $\mathsf{mat}_u \in \mathcal{A}$. If $u \notin \mathbb{V}$, we have $\mathsf{scr}(u) > S$ based on the proof above. This contradicts with that the mat_u is among the top-k matches. □

A.2 Proof of Prop. 4.1

Proposition A.2. (**Completeness**) Suppose the top-k matches to a keyword query is $\mathcal{A}$ and all the visited vertices $\mathbb{V}$, we have:
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: 1) Suppose $u \notin \mathbb{V}$, and $u \in F_i.V$. The match rooted at u is mat_u. We consider the nearest query keyword $q_x \in Q$ of u, *i.e.*, $\mathsf{dist}(u, q_y) \geq \mathsf{dist}(u, q_x)$, where $q_y \in Q$. We denote the shortest path between u and q_x by $\mathsf{Path}(u, q_x)$.

We consider the following cases:

Case 1: $\mathsf{Path}(u, q_x)$ is localized on F_i completely, *i.e.*, $\mathsf{Path}(u, q_x)$ is a subgraph of fragment F_i. The proof is the same with that of Proposition 3.1.

We assume that the last vertex which to be backward expanded for keyword q_y on F_i as v_y. Since u has not been visited, we have

$$\mathsf{dist}(u, q_x) \geq \mathsf{dist}(v_y, q_y) \tag{7}$$

Hence,

$$\mathsf{scr}(u) = \sum_{y\in[1,l]} \mathsf{dist}(u, q_y) \geq l\mathsf{dist}(u, q_x) \geq \sum_{y\in[1,l]} \mathsf{dist}(v_y, q_y) \tag{8}$$

Based on the termination condition (Algo. 3), we have

$$\sum_{y\in[1,l]} \mathsf{dist}(v_y, q_y) > S_i \tag{9}$$

Combining Equ 8 and Equ 9, we derive

$$\mathsf{scr}(u) > S_i \tag{10}$$

Hence, mat_u is not among the local top-k matches on F_i. It is not among the global top-k matches either in nature, *i.e.*, $\mathsf{mat}_u \notin \mathcal{A}$.

Case 2: $\mathsf{Path}(u, q_x)$ is not localized on F_i completely, *i.e.*, the path spans through multiple fragments. We denote all the vertices which have been visited by the backward expansion of the query keyword q_x on fragment F_i by $\mathbb{V}_{q_x,i} = \bigcup_{s\in[1,R]} V^s_{q_x,i}$. We consider the vertex u and all the portal nodes on $\mathsf{Path}(u, q_x)$. Since u is never visited, we denote the sequence of u and the portal nodes by $\mathbb{P} = \{p_n, \ldots, p_{z+1}, p_z, \ldots, p_1\}$, where $p_n = u$, such that $p_z \in \mathbb{V}_{q_x,j}$, $p_{z+1} \notin \mathbb{V}_{q_x,j}$, $p_z \in F_j.O$ and $p_{z+1} \in F_j.I$ or $p_{z+1} = u$, where $j \in [1, m]$. Intuitively, p_{z+1} is the first portal node which is not expanded by bkws in the portal node sequence.

The proof is similar to the **Case 1.** We assume that the last vertex to be backward expanded for keyword q_y on F_j as v_y. Since p_{z+1} has not been visited by bkws of the query keyword q_x on F_j, we have

$$\mathsf{dist}(p_{z+1}, q_x) \geq \mathsf{dist}(v_y, q_y) \tag{11}$$

Hence,

$$\mathsf{scr}(u) = \sum_{y\in[1,l]} \mathsf{dist}(u, q_y) \geq l\mathsf{dist}(u, q_x) \geq l\mathsf{dist}(p_{z+1}, q_x) \geq \sum_{y\in[1,l]} \mathsf{dist}(v_y, q_y) \tag{12}$$

Based on the termination condition (Algo. 3), we have

$$\sum_{y\in[1,l]} \mathsf{dist}(v_y, q_y) > S_j \tag{13}$$

Combining Equ 12 and Equ 13, we derive

$$\mathsf{scr}(u) > S_j \tag{14}$$

Hence, mat_u is not better than the local top-k matches on F_j. It is not among the global top-k matches, *i.e.*, $\mathsf{mat}_u \notin \mathcal{A}$. □

Theorem A.1. Consider a PINE algorithm which 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), and any partition strategy Par. If (a) P_i and I_i satisfy the monotonic condition, and (b) P_i, I_i and E are correct *w.r.t.* Par, then DKWS with P_i, I_i and E guarantee to terminate correctly.

Proof: In each superstep, PINE performs I_i ($i \in [1, n]$) selected by the switch statement. Hence the monotonic condition of PINE is identical to that of I_i. □

A.3 Pseudocode of IncEval of fkws

(2) Incremental computation (IncEval) for fkws (Algo. 5). IncEval is derived from fkws of bfkws with the following two modifications.

(2.1) Refinement propagation. Firstly, P_i receives the partial matches, mat^f_u, to the forward expansion *request* f_u in previous supersteps from other fragments in M^1_i via the portal nodes, where $u \in F_i.O$. If a shorter path between u and q is found crossing multiple fragments (Line 3), the forward match $\mathsf{mat}^f_u[q]$ is refined (Line 5). Then, IncEval propagates the distance refinement to the ancestor vertices.

Algorithm 5: IncEval for fkws

Input: $F_i(V, E, L)$, $Q = \{q_1, \ldots, q_l\}$, τ, message M_i
Output: $Q(F_i \oplus M_i)$ consisting of current $\mathsf{mat}_u \in \mathcal{A}_i$

1 **foreach** $q \in Q$ **do**
2 init a priority queue $\mathcal{P}_q = \emptyset$ for q to store the refinement
3 **foreach** $\mathsf{mat}_u^{f,\mathsf{in}} \in M_i^1$ **do**
4 **if** $\mathsf{mat}_u^f[q] > \mathsf{mat}_u^{f,\mathsf{in}}[q]$ **then**
5 $\mathsf{mat}_u^f[q] = \mathsf{mat}_u^{f,\mathsf{in}}[q]$
6 $\mathcal{P}_q$.insert$(\langle u, \mathsf{mat}_u^f[q] \rangle)$
7 PropagrateUpdate$(\mathcal{P}_q, q)$
8 **foreach** $\mathsf{f}_u^{\mathsf{in}} \in M_i^2$ **do**
9 **if** $u \notin \bar{\mathcal{V}}$ **then**
10 $\bar{\mathcal{V}}$.add(u)
11 **else**
12 **foreach** $q \in \mathsf{f}_u^{\mathsf{in}}$ **do**
13 $\mathsf{f}_u[q] = \max\{\mathsf{f}_u[q], \mathsf{f}_u^{\mathsf{in}}[q]\}$
14 **foreach** $u \in \bar{\mathcal{V}}$ **do**
15 forwardExpand$(u, Q, S_i, \mathcal{A}_i)$
16 **Function** PropagrateUpdate$(\mathcal{P}_q, q)$
17 init a visited vertices set Vis $= \emptyset$
18 **while** $\mathcal{P}_q$ is not empty **do**
19 $\langle u, d \rangle = \mathcal{P}_q$.top()
20 Vis.add(u)
21 $\mathcal{P}_q$.pop()
22 **foreach** $e = (u', u) \in E$ **and** $u' \notin$ Vis **do**
23 $d' = w(e) + d$
24 **if** $d' \leq \mathsf{mat}_{u'}^f[q]$ **then**
25 $\mathsf{mat}_{u'}^f[q] = d'$
26 $\mathcal{P}_q$.insert$(\langle u', d' \rangle)$
27 *Message segment:* $M_i^1 = \{\mathsf{mat}_u^f | u \in F_i.I\}$ and $M_i^2 = \{\mathsf{f}_u | u \in F_i.O\}$

It also notifies the coordinator once the local upper bound is refined in forwardExpand (Line 15).

(2.2) Incremental forward expansion. Secondly, upon receiving some forward expansion requests from other fragments, worker P_i further forward expands to retrieve missing keywords on F_i via the incoming portal nodes, $F_i.I$. Specifically, if $\mathsf{f}_u^{\mathsf{in}} \in M_i^2$ is received and $u \notin \bar{\mathcal{V}}$, u is added into $\bar{\mathcal{V}}$ (Line 9). Since the search requests come from different fragments, f_u keeps the largest one for each keyword (Line 12). If u is forward expanded in previous iterations for query keyword q or $\mathsf{f}_u^{\mathsf{in}}[q]$ is smaller than $\mathsf{f}_u[q]$, $\mathsf{f}_u^{\mathsf{in}}[q]$ is skipped.

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

Appendix B
Optimizations for DKWS

B.1 Backtrack graph for refinement propagation

Refinement propagation is potentially costly when messages are large. If we conduct the refinement propagation from the portal nodes in $F.O$, one by one, the time complexity is bounded by $O(|Q||F.O|(|E|+|V|\log|V|))$. To address this issue, we propose backtrack pointers for fkws and extend them to the outgoing portal nodes to build a backtrack graph.

Backtrack pointer. To reduce duplicate forward traversals, fkws maintains a backtrack pointer $\mathcal{I}$ for each visited vertex v in the forward expansion of u. Specifically, let's assume the sequence of the shortest path that starts from u and ends with v is $[v_1, \ldots, v_n]$ such that $v_1 = u$, $v_n = v$, $\mathcal{I}_{v_{j+1}} = v_j$ and $q \in L(v_n)$. Once v_n is expanded, bfkws refines the matches as follows: $\mathsf{mat}_{v_j}[q] = \mathsf{mat}_{v_{j+1}}[q] + w(v_j, v_{j+1})$ recursively, for all $j \in [1, n-1]$.

Backtrack graph. Given a query keyword $q \in Q$, a backtrack graph $G_{\mathcal{I}}^q = (V_{\mathcal{I}}^q, E_{\mathcal{I}}^q)$ stores all the paths between the vertices in $\bar{\mathcal{V}}$ and the outgoing portal nodes $F.O$ when conducting the forward expansion to retrieve the missing keyword q. DKWS propagates the refinement in *one batch* of the outgoing portal nodes for one query keyword rather than *vertex by vertex* from all the outgoing portal nodes. *The time complexity on fragment F is reduced to* $O(|Q|(|E_{\mathcal{I}}^q|+|V_{\mathcal{I}}^q|\log|V_{\mathcal{I}}^q|))$.

Construction of backtrack graph. We consider a vertex $u \in \bar{\mathcal{V}}$. For any vertex $v \in F.O$ that is forward expanded during the forward expansion starting from u for retrieving query keyword q, the sequence of the shortest path between u and v, $[v_1, \ldots, v_n]$ (such that $v_1 = u$, $v_n = v$), is added into $G_{\mathcal{I}}^q$. That is, DKWS inserts the edge (v_j, v_{j+1}) ($j \in [1, n-1]$) to $E_{\mathcal{I}}^q$, implemented by backtrack pointers.

Proposition B.1. The size of $G_{\mathcal{I}}^q$ is bounded by $O(|V|+|E|)$.

B.2 Indexing for pruning false matches

As proposed in Sec. 3, PADS and KPADS prune some partial matches locally. Here, we extend PADS and KPADS to propose BPADS that skips some forward expansions that span across multiple fragments.

Boundary-PADS (BPADS). For each fragment F, we build a sketch for all the out-portal nodes $F.O$, denoted by BPADS(F). BPADS(F) is built by merging PADS$^{\mathsf{out}}$ of out-portal nodes, *i.e.,*PADS$^{\mathsf{out}}(v)$, where $v \in F.O$. Formally, given a center $(w_j, d_j) \in$ PADS$^{\mathsf{out}}(v)$, $(w_j, d_j) \in$ BPADS(F) *iff* $\forall (w_j, d_j') \in$ PADS$^{\mathsf{out}}(v')$ and $v' \in F.O$, $d_i' \geq d_i$.

Pruning by indexes. Consider a vertex $u \in \bar{\mathcal{V}}$ that needs to be forward expanded. The forward expansion starting from u can be skipped if there is either no *local path* or no path *across multiple fragments* between u and q whose length is smaller than $\mathsf{f}_u[q]$: a) pruning forward expansion locally. We denote the lower bound of the shortest distance between u and each missing keyword $q \in Q$ by $\check{\mathsf{dist}}(u, q)$. If $\check{\mathsf{dist}}(u, q) > \mathsf{f}_u[q]$, then there is no local path between u and q whose length is smaller than $\mathsf{f}_u[q]$ in F. $\check{\mathsf{dist}}(u, q)$ is derived by Eq 2 with PADS$^{\mathsf{out}}(u)$ and KPADS$^{\mathsf{out}}(q)$ (Line 27 of Algo. 4); and b) pruning forward expansion across multiple fragments. We denote the lower bound of the shortest distance between u and the nearest out-portal node by $\check{\mathsf{dist}}(u, F.O)$. If $\check{\mathsf{dist}}(u, F.O)$ is larger than $\mathsf{f}_u[q]$, then no path across multiple fragments between u and q which is smaller than $\mathsf{f}_u[q]$ could be found. $\check{\mathsf{dist}}(u, F.O)$ is derived by PADS$^{\mathsf{out}}(u)$ and BPADS (Line 25 of Algo. 4). If $\check{\mathsf{dist}}(u, q)$ and $\check{\mathsf{dist}}(u, F.O)$ are both larger than $\mathsf{f}_u[q]$, the forward expansion starting from u can simply be skipped, since such expansion cannot yield any matches.

Complexities. This optimization does not change the time complexity of fkws. A vertex $u \in \bar{\mathcal{V}}$ that needs to be forward expanded may be pruned in $O(\ln|V|)$ time. Given a vertex u, the size of PADS(u) is $O(\ln|V|)$. Given a keyword q,

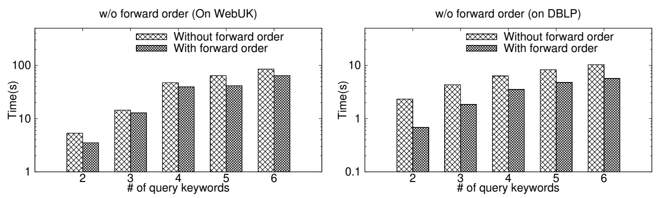


Fig. 15: Performance of the forward expansion order

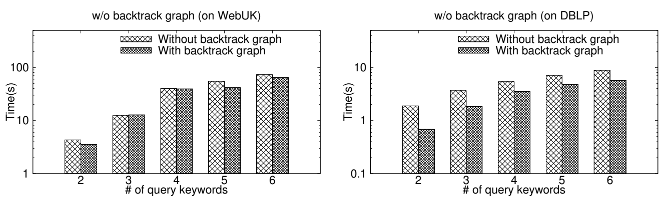


Fig. 16: Performance of the backtrack graph

the size of KPADS(q) is $O(|V|)$. The space complexity of BPADS is $O(|F.O|\ln|V|)$.

B.3 Forward expansion order for fkws

A vertex may be expanded multiple times in different forward expansions. To reduce this, we propose forward expansion orders, *i.e.,* Lines 4-5 of Algo. 4 and Line 14 of Algo. 5. DKWS stores $\bar{\mathcal{V}}$ in a priority queue of *the match score* $\mathsf{scr}(u)$ *in descending order*. Intuitively, the larger $\mathsf{scr}(u)$ is, the easier $\mathsf{scr}(u)$ exceeds the upper bound S. As a consequence, the smaller region is expanded for u. Consider a forward expansion that starts from another vertex $u' \in \bar{\mathcal{V}}$, where $\mathsf{scr}(u') < \mathsf{scr}(u)$. If the expansion meets u, the following expansion starting from u could be skipped, since $\mathsf{dist}(u, q)$ has already been computed and stored at $\mathsf{mat}_u[q]$ (Line 16 of Algo. 4). And the distance between u' and q via u is computed by $\mathsf{dist}(u', u) + \mathsf{mat}_u[q]$.

Exp-5: Effectiveness of the optimizations. We performed a set of experiments to investigate the effectiveness of the proposed optimizations on WebUK and DBLP. The results on other datasets exhibit similar trends and are hence not shown.

Effectiveness of expansion order. We turned the expansion order of fkws on and off. The results are reported in Fig. 15. With the optimization of expansion order, DKWS-PINE is 1.13 (resp. 1.86) times faster on WebUK (resp. DBLP). The improvement with expansion order is more significant on DBLP than that on WebUK since shorter forward expansions are computed early and some longer forward expansions can be computed from the shorter ones.

Effectiveness of backtrack graph. We also investigated the effectiveness of the backtrack graph by turning the optimization on and off. The results are reported in Fig. 16. With the optimization of the backtrack graph, DKWS-PINE is 1.34 (resp. 2.22) times faster on WebUK (resp. DBLP) on average. The improvement of this optimization is more significant on DBLP than that on WebUK since more refinement propagation on DBLP is shared.

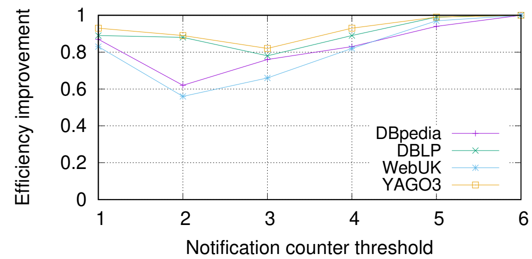


Fig. 17: Impact of notification counter threshold

B.4 Supplementary experiment

Impact of Notification Counter. As depicted in Fig. 17, we normalized the execution time by the time taken when the notify-push paradigm was not activated. We observed that on the four real-life datasets, setting the notification counter to 2 or 3 resulted in a comparatively good performance. However, when the notification counter exceeded 4, there was no substantial difference in the performance acceleration compared to when the notify-push paradigm was not used. In the context of keyword search semantics, this observed behavior can be traced back to the fact that, within these datasets, the discrepancy in the number of times each worker refines the bounds rarely surpasses four times; thus, the push function was seldom invoked. We also noted that, on larger graphs, setting the threshold to 2 performed better than setting it to 3. This is because the rapid synchronization of global bounds on larger graphs provides more significant performance acceleration.

B.5 Time complexity of bkws

bkws takes $O(|Q|(|E|+|V|\log|V|))$, where $|Q|$ is the number of query keywords. To derive the complexity, we add a dummy vertex v_i^d for each query keyword $q_i \in Q$. Also, we add a set of dummy edges to connect v_i^d and the vertices in the search origin V_{q_i}. We denote the graph with the dummy vertex v_i^d and edges by G_i^d. The complexity of bkws for each query keyword on G is identical to that of Dijkstra's algorithm starting at v_i^d on G_i^d, *i.e.,* $O(|E|+|V|\log|V|)$. Hence, the complexity of bkws is bounded by $O(|Q|(|E|+|V|\log|V|))$.

B.6 PageRank-based All Distance Sketches (PADS)

In this subsection, we review ADS and then propose our index. It is known that ADS is small in size, accurate, and efficient in answering shortest distance queries. Our main idea is to use PageRank to determine the chance of a node to be included in the sketch (*i.e.,* the index).

All-Distances sketches (ADS). Recall that in [7], given a graph $G = (V, E)$, each vertex v is associated with a sketch, which is a set of vertices and their corresponding shortest distances from v. To select the vertices in V and put them as the centers in the sketch of v, each vertex is initially assigned a *random* value in $[0, 1]$. If a vertex $u \in V$ has the k-th largest value among the vertices which have been traversed from v in the Dijkstra order, then u is added to the sketch of v. k is a user-defined parameter set by user. A larger k results

Algorithm 6: PADS construction

```
Input: Graph G = (V, E)
Output: PADS
compute the PageRank pr of the vertices in G
initialize PADS(v) = {(v, 0)} for each vertex v ∈ V
sorted the vertices V by the descending order of pr(v)
for v ∈ V do
    for u in the Dijkstra's traversal do
        if |{(w, d) ∈ PADS(u) | d ≤ d(v, u)}| < k then
            add (v, d(v, u)) into PADS(u)
        else
            continue the traversal on the next vertex
return PADS
```

in larger and more accurate sketches. The shortest distance between u and v can be estimated by the intersection set of ADS(u) and ADS(v) (a.k.a. the common centers).

A drawback of ADS is that it does not consider the relative importance of the vertices when generating the sketch. The vertices with high PageRanks, which roughly estimates the importance of the vertices in a graph, should be added to the sketch to cover the shortest paths. On the contrary, the vertices with low PageRanks are unlikely to be on many shortest paths and should not be added to the sketch.

PageRank. We employ any efficient algorithms to obtain the PageRank of the vertices of a graph G. We use a function pr: $V \rightarrow$ [0,1] to denote the PageRank of a vertex v by $pr(v)$.

Dijkstra rank. We recall that we can efficiently obtain the Dijkstra rank of a vertex v w.r.t a source vertex s as follows. We run the Dijkstra's algorithm starting at s and obtain the order of the visited nodes $[v_1, v_2, \dots, v_l]$. The Dijkstra rank of v_i w.r.t s is i, denoted as $\pi(s, v_i) = i$.

PageRank based all-distances sketches (PADS**)**. Given a Dijkstra rank π, the PageRank, a vertex v, and a threshold k, the PADS of v is defined as follows:

$$\text{PADS}(v) = \{(u, d(v,u)) \mid pr(u) \geq k(v,u)\}, \qquad (15)$$

where $k(v,u)$ is the k-th largest PageRank among the nodes from v to u according to π.

Example B.1. (PADS construction) Consider the graph G in Fig. 18. Assume $k = 1$. We compute the PageRank values for all the vertices in the graph, as shown below the vertices' labels. v_{13} covers 41 out of 156 shortest paths in the graph G in total, which is the largest among all the vertices. This shows that the node having a large PageRank value, $pr(v_{13}) = 0.130$, can be an effective center. To determine the PADS of v_1, we run the Dijkstra's algorithm by taking v_1 as the source vertex to obtain the Dijkstra ranked list $[v_1, p_1, p_2, v_{13}, v_4, \dots, p_7]$. Since the PageRank value of v_{13} is the highest among the first four vertices in the ranked list, v_{13} is added to PADS(v_1) with its distance to v_1. Similarly, v_1 is added to PADS(v_1).

Shortest distance estimation. Given a shortest distance query (u, v) and the PADS, $\hat{d}(u, v)$ is computed by the intersection of PADS(u) and PADS(v) as follows:

$$\hat{d}(u, v) = \min\{(d_1 + d_2)\}, \qquad (16)$$

where $(w, d_1) \in \text{PADS}(u)$, $(w, d_2) \in \text{PADS}(v)$.

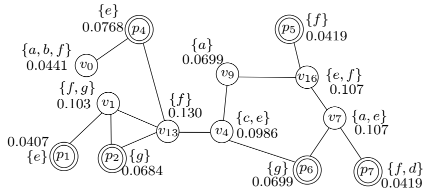


Fig. 18: A graph (fragment) and the PageRank

TABLE 4: An ADS label for the graph in Fig. 18

Vertex ID	ADS
v_0	$\{(v_0, 0), (p_4, 1), (v_1, 3), (p_1, 4), (p_7, 6)\}$
p_4	$\{(p_4, 0), (v_1, 2), (p_1, 3), (p_7, 5)\}$
v_{13}	$\{(v_{13}, 0), (p_4, 1), (v_1, 1), (p_1, 2), (p_7, 4)\}$
v_1	$\{(v_1, 0), (p_1, 1), (p_7, 5)\}$
p_1	$\{(p_1, 0), (p_7, 6)\}$
p_2	$\{(p_2, 0), (v_1, 1), (p_1, 2), (p_7, 5)\}$
v_4	$\{(v_4, 0), (v_{13}, 1), (v_9, 1), (p_4, 2), (v_1, 2), (p_1, 3), (p_7, 3)\}$
v_9	$\{(v_9, 0), (p_4, 3), (v_1, 3), (p_7, 3)\}$
p_6	$\{(p_6, 0), (v_4, 1), (v_{13}, 2), (v_9, 2), (p_7, 2)\}$
v_{16}	$\{(v_{16}, 0), (v_9, 1), (p_7, 2)\}$
v_7	$\{(v_7, 0), (v_{16}, 1), (p_7, 1)\}$
p_5	$\{(p_5, 0), (v_9, 2), (p_7, 3)\}$
p_7	$\{(p_7, 0)\}$

Spae complexity. The expected size of PADS(v) is $O(k \ln n)$, where n is the number of nodes reachable from v, which is bounded by $O(k \ln|V|)$. (The analysis of [7] can be applied to PADS.)

Time complexity. The time complexity is bounded by $O(k|E|\ln|V|)$.

Consider the graph G in Fig. 18. We set $k = 1$ and compute its ADS shown in Tab. 4 and its PADS shown in Tab. 5. We can see that there are two advantages of PADS. First, the size of PADS is significantly smaller than that of ADS. Second, the PADS's estimation is much more accurate than that of ADS.

Example B.2. (Shortest distance estimation.) Consider the graph G in Fig. 18 and its PADSs in Tab. 5. Given two vertices v_9 and v_7, there are two common centers v_{16} and v_{13} in PADS(v_9) and PADS(v_7). The shortest distance is estimated by Eq 16, *i.e.*, $\hat{d}(v_9, v_7) = 2$ (*i.e.*, 0% error). By ADS, $\hat{d}(v_9, v_7) = 4$ is returned (*i.e.*, 100% error). More specifically, we compare the estimation accuracy of ADS and PADS between all pairs of the vertices in Fig. 18. The average error of PADS (resp. ADS) is around 3% (resp. 38%).

It is worth noting that PADS exhibits the theoretical guarantee of the shortest path estimation stated below.

Lemma B.1. The distance between two vertices u and v is estimated using Eq 16 with an approximation factor $(2c - 1)$, where $c = \lceil \frac{\ln|V|}{\ln k} \rceil$ with a constant probability, *i.e.*, $\hat{d}(u, v) \leq (2c - 1)d(u, v)$.

Proof: Let $d = d(u, v)$. Let $N_i(u)$ denote the neighbors of vertices u within id hops. For simple exposition, we denote the intersection and union of $N_i(u)$ and $N_i(v)$ as $I_i = N_i(u) \cap N_i(v)$ and $U_i = N_i(u) \cup N_i(v)$, respectively. It is worth noting that $I_i \subseteq U_i \subseteq I_{i+1}$. Consider the ratio of $\frac{|I_i|}{|U_i|}$ and a ratio threshold $\frac{m}{k}$. Given the vertices with k largest pr values in U_i, if one of them (say w) hits I_i, w belongs to both PADS(v) and PADS(u). The real distance d can be estimated within $2id$. The probability of *at least one of the vertices, which has the k largest PageRank values in U_i, hits the I_i* is $1 - (1 - \frac{\alpha}{k})^k \approx 1 - e^{-\alpha}$. Since there are n vertices in

TABLE 5: The PADS label for the graph in Fig. 18

Vertex ID	PADS
v_0	$\{(v_0, 0), (p_4, 1), (v_{13}, 2)\}$
p_4	$\{(p_4, 0), (v_{13}, 1)\}$
v_{13}	$\{(v_{13}, 0)\}$
v_1	$\{(v_1, 0), (v_{13}, 1)\}$
p_1	$\{(p_1, 0), (v_1, 1), (v_{13}, 2)\}$
p_2	$\{(p_2, 0), (v_1, 1), (v_{13}, 1)\}$
v_4	$\{(v_4, 0), (v_{13}, 1)\}$
v_9	$\{(v_9, 0), (v_4, 1), (v_{16}, 1), (v_{13}, 2)\}$
p_6	$\{(p_6, 0), (v_4, 1), (v_7, 1), (v_{13}, 2)\}$
v_{16}	$\{(v_{16}, 0), (v_7, 1), (v_{13}, 3)\}$
v_7	$\{(v_7, 0), (v_{16}, 1), (v_{13}, 3)\}$
p_5	$\{(p_5, 0), (v_{16}, 1), (v_7, 2), (v_{13}, 4)\}$
p_7	$\{(p_7, 0), (v_7, 1), (v_{16}, 2), (v_{13}, 4)\}$

TABLE 6: The KPADS label for the graph in Fig. 18

Terms	KPADS
a	$\{(v_9, 0), (v_4, 1), (p_4, 1), (v_7, 0), (v_{13}, 2), (v_{16}, 1), (v_0, 0)\}$
b	$\{(v_0, 0), (v_{13}, 2), (p_4, 1)\}$
c	$\{(v_{13}, 1), (v_4, 0)\}$
d	$\{(v_{13}, 4), (v_7, 1), (p_7, 0), (v_{16}, 2)\}$
e	$\{(v_{13}, 1), (v_4, 0), (v_1, 1), (v_7, 0), (p_4, 0), (v_{16}, 0), (p_1, 0)\}$
f	$\{(p_5, 0), (v_1, 0), (v_{13}, 0), (p_4, 1), (v_7, 1), (v_{16}, 0), (v_0, 0), (p_7, 0)\}$
g	$\{(p_6, 0), (v_1, 0), (v_4, 1), (v_{13}, 1), (v_7, 1), (p_2, 0)\}$

graph G at most, $|U_i| \leq n$. Hence, there exists $i \leq \log_{k/\alpha} n$. □

B.7 PageRank-based Keyword Distance Sketches (KPADS)

We denote the *shortest distance between a vertex* v and *a keyword* t by $d(v, t)$, where $d(v, t) = \min\{d(v, u) | t \in L(u), u \in V\}$. To estimate the distance between a given vertex and keyword, we propose KPADS, which is constructed by PADS-merging: Given any two vertices u and u' where $t \in L(u)$ and $t \in L(u')$, there may exist common centers in PADS(u) and PADS(u'). Hence, we only keep the smallest one among $\hat{d}(v, u')$ and $\hat{d}(v, u)$, since both of them are the upper bound of $d(v, t)$.

Keyword-PADS (KPADS). For each keyword $t \in \Sigma$, we build a sketch KPADS(t). KPADS(t) can be built by merging PADS of those vertices that contain t, *i.e.*,PADS(v) where $t \in L(v)$. More formally, given a center $(w_i, d_i) \in$ PADS(v), $(w_i, d_i) \in$ KPADS(t) *iff* $\forall (w_i, d'_i) \in$ PADS(v') and $t \in L(v')$, $d'_i \geq d_i$.

Shortest keyword-vertex distance estimation. Given a vertex v and a keyword t, the shortest distance $\hat{d}(v, t)$ can be computed as follows:

$$\hat{d}(v, t) = \min\{(d_1 + d_2) | (w, d_1) \in \text{PADS}(v) \text{ and } (w, d_2) \in \text{KPADS}(t)\} \quad (17)$$

Example B.3. Consider the graph G in Fig. 18 and its PADS in Tab. 5. The KPADS is shown in Tab. 6. Consider the shortest distance between a and p_4. The distance can be estimated by the intersection of KPADS(a) and PADS(p_4). There are two common centers, p_4 and v_{13}. $\hat{d}(a, p_4) = 1$ is returned by the common center p_4.