# An Efficient Parallel Keyword Search Engine on Knowledge Graphs

Yueji Yang [#1], Divykant Agrawal [*], H.V. Jagadish[†], Anthony K. H. Tung [#2], Shuang Wu [#3]

[#] *School of Computing, National University of Singapore*
[#] {[1] yueji, [2] atung, [3] wushuang}@comp.nus.edu.sg

[*] *Department of Computer Science, University of California at Santa Barbara*
[*] agrawal@cs.ucsb.edu

[†] *Department of Electrical Engineering and Computer Science, University of Michigan of Ann Arbor*
[†] jag@umich.edu

***Abstract*—Keyword search has recently become popular as a way to query relational databases, and even graphs, since it allows users to issue queries without learning a complex query language and data schema. Evaluating a keyword query is usually significantly more expensive than evaluating an equivalent selection query, since the query specification is less complete, and many alternative answers have to be considered by the system, requiring considerable effort to generate and compare. Current interest in big data and AI are putting even more demands on the efficiency of keyword search. In particular, searching of knowledge graphs is gaining popularity. As knowledge graphs often comprise tens of millions of nodes and edges, performing real-time search on graphs of this size is an open challenge.**

**In this paper, we attempt to address this need by leveraging advances in hardware technologies, e.g. multi-core CPUs and GPUs. Specifically, we implement a parallel keyword search engine for Knowledge Bases (KB). To be able to do so, and to exploit parallelism, we devise a new approach to keyword search, based on a concept we introduce called Central Graph. Unlike the Group Steiner Tree (GST) model, widely used for keyword search, our approach can naturally work in parallel and still return compact answer graphs with rich information. Our approach can work in either multi-core CPUs or a singe GPU. In particular, our GPU implementation is two to three orders of magnitudes faster than state-of-the-art keyword search method. We conduct extensive experiments to show that our approach is both efficient and effective.**

## I. Introduction

Keyword queries have become popular in recent years, for use against relational databases, graphs, and other structured data stores. This is because keyword queries are easy for a non-technical user to specify, removing the burden of understanding database structure (or schema) in addition to the burden of learning a query language. The query only comprises keywords; the corresponding output is one or more subgraphs, each of which cover the input keywords and are embedded in the original data graph. Much excellent work has been done to support keyword queries, as we discuss in Sec. II. The solution technique is, roughly, to identify matches for each keyword individually and then to combine matches based on an appropriate notion of proximity (such as shortest join path). A Group Steiner Tree (GST) is the data structure commonly used for this purpose. Top-k answers for GST consists of top-ranked trees (according to some scoring function) embedded in the original data graph.

Since there can be many possible join paths between any pair of matches, and since combinations of paths must be considered, keyword query evaluation is typically expensive. If keyword queries can be evaluated in "interactive" time, then users can re-submit keyword queries to retrieve better answers, just as they do in Google web search. Unfortunately, this turns out to be challenging to do, particularly as the size of the database increases. The efficiency issues become particularly challenging for knowledge graphs, which are often very large today. At the same time, the heterogeneity of knowledge graphs makes keyword querying particularly valuable. In this work, we show how to make this primary need. To make matters concrete, we focus on one specific important knowledge graph, Wikidata Knowledge Base [1], [2]. We provide an online query service and name it *WikiSearch*. Interested readers can try it out at http://dbgpucluster-2.d2.comp.nus.edu.sg. Note that our approach applies to other knowledge graphs as well, such as Freebase and Yago. Note that these knowledge graphs can all be represented in an RDF graph.

As the size of knowledge graphs grows rapidly, the efficiency issues naturally come up. Unfortunately, there are few approaches that can respond to keyword queries in real-time on a KB with hundreds of millions of edges. Current approaches that adopt the GST model [6], including BANKS-I [3], BANKS-II [4] and BLINKS [5]. Nodes containing the same keyword form a group and the answer is a tree embedded in the data graph and covers one leaf node from each group. Since GST is known to be NP-hard and there exists no polynomial approximate algorithm with constant approximation ratio, the above methods only conceptually approximate GST without any error bound [6], [7]. In view of the hardness of traditional keyword search models, we are motivated to seek a solution to keyword search problem that can work in real-time.

Having witnessed great advances in computer hardware (e.g. multi-core CPUs and GPUs), we are inspired to think whether we can make use of parallel computational power of modern hardware to address the efficiency issues. Unfortunately, it

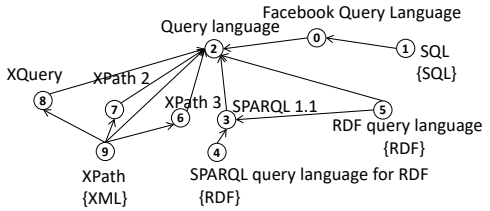



Fig. 1. Example answer graph by our proposed approach for input keywords *XML, RDF, SQL*. Edges' types are omitted. For tree-shaped answers rooted at $v_2$, there are multi-paths from keyword nodes, four paths from $v_9$ and two from $v_4$ and $v_5$. Different combination of these paths give different tree answers, which are repetitive.

turns out to be very difficult for traditional approaches as mentioned [3], [4], [5], since their search procedures are based on shortest paths and have many intrinsic dependencies during traversal. Specifically, a priority queue is used to decide which node to explore next based on current status. Therefore, we are motivated to develop a new model, called Central Graph, which can work in parallel and still return meaningful compact answers.

In addition to the efficiency improvement, our model is particularly suitable searching knowledge bases which typically have richness and heterogeneity of information. Different from the typical GST methods that produce trees as answers, our model returns graphs as answers. Often, tree-shaped answers are too condensed to be able to convey the rich information contained in a KB. In consequence, tree-shaped answers tend to be verbose and repetitive. For example, as shown in Fig. 1, a graph-shaped answer can not only include cycles (from $v_9$ to $v_2$), but also admit more than one node containing the same keyword (eg. $v_4$ and $v_5$ containing "*RDF*"). It needs several tree-shaped answers to convey the same information included in such a graph answer. As can be seen, graph-shaped answers can convey much more information with fewer repetitions.

**Contributions.** To overcome the challenges brought about by both volume and variety of today's huge Knowledge Bases, in this paper we make the following contributions:

First, we introduce the novel concept of Central Graphs to model the answers of a keyword search problem. The final answers are then pruned and ranked by a keyword co-occurrence based novel approach, called level-cover strategy. Central Graphs can naturally work in parallel and still return compact answers. In addition, Central Graphs allow multi-paths from one keyword, leading to much more expressive answers than tree-shaped ones, e.g. Fig. 1.

Second, we develop a two-stage parallel algorithm framework that can work not only on multi-core CPUs, but also GPUs. Our algorithm works in a lock-free way during traversal, which is critical for efficiency. In the first stage, we find a set of potential Central Graphs in a bottom-up manner starting from nodes containing keywords (keyword nodes). In the second stage, we extract, prune and select the top-ranked Central Graphs derived from the first stage in a top-down manner starting from Central Nodes, which are centers of respective Central Graphs.

Third, we conduct extensive experiments to evaluate both efficiency and effectiveness of proposed algorithm.

**Organization.** Sec. II discusses the related work. Sec. III introduces the problem definitions and also introduces as well as the novel concept of a Central Graph. Then we show the definition of minimum activation level and how it affects the search procedure in Sec. IV. Sec. V reveals the details of the proposed two-stage parallel algorithm. Lastly, Sec. VI shows the experiment results. We conclude our work in Sec. VII.

## II. RELATED WORK

**Keyword search.** Early works on keyword search, like DBXplorer [8] and Discover [9], conduct a BFS over the schema graph of tables in targeted relational databases. The schema graphs are connected through foreign-and-primary key relations and tend to be quite small. These works are limited to relational databases. ObjectRank [10] is an authority-based method and the output is top-k relevant nodes. BANKS-I [3], BANKS-II [4] and BLINKS [5] model keyword search problem by approximating Group Steiner Tree (GST) Problem. However, the GST Problem is NP-Hard and difficult to approximate with an ideal bound in polynomial time [11]. As mentioned, to produce top-k results, the search steps of these methods have sequential dependency and thus have difficulties making use of the parallel approaches. Although there are a few works [12], [13], [14] trying to harness parallelism, they mainly focus on RDBMS or XML datasets.

Aditya et al. [3] propose a backward search algorithm, which is applicable to both relational data and graph data. As the graph size increases, the scalability problem of backward search algorithm becomes salient. There are works [5], [15], [16] trying to partition graphs and search only necessary parts of a whole graph in the hope of improving scalability problem. BLINKS [5] needs to pre-compute *keyword-node lists* and *node-keyword map*, which are infeasible on Wikidata KB with 30 million nodes and over 5 million keywords after stopping word filtering and word stemming. These algorithms have to rely on either on Dijkstra's Algorithm or complex index structures to find the nearest node to traverse, since storing all-pair shortest distances is too expansive.

There are dynamic programming algorithms that try to directly solve the Group Steiner Tree problem. [7] is effective when number of keywords is small, but is not very scalable in terms of the number of keywords as pointed by [6]. Specifically, the complexity of [7] is $\mathcal{O}(3^l n + 2^l((l+\log n)n+m))$, where $l$ is the number of keywords and $n, m$ the number of nodes and edges, respectively. However, [6] needs to pre-compute all-pair shortest distances between super nodes, which inevitably needs a huge storage. In addition, it is not clear how to collect top-k answers by [6].

There are several approaches adopting graph-shaped answers to keyword search problem. EASE [17] proposes r-radius Steiner Graph for structure, semi-structure and unstructured datasets. However, EASE is not scalable for large graphs. Moreover, Kargar et al. [18] point out that EASE may miss some highly ranked r-radius Steiner Graphs if they are

TABLE I
SUMMARY OF NOTATIONS

| Notations | Meaning |
|---|---|
| $G(V, E)$ | undirected node-weighted graph $G$ |
| $w_i$ | the weight of node $v_i$ |
| $a_i$ | the minimum activation level of $v_i$ |
| $e_{ij}$ | the undirected edge between $v_i$ and $v_j$ |
| $r$ | a relationship type |
| $R_i$ | the set of relationship types incident to $v_i$ |
| $r_{ij}$ | the relationship between $v_i$ and $v_j$ |
| $Q, t_i$ | a keyword query, a keyword term |
| $T_i$ | the set of nodes containing it |
| $B_i$ | a BFS instance w.r.t. $t_i$ |
| $h_j^b$ | the hitting level of $v_j$ w.r.t. $B_b$ |
| $P_i^b$ | the set of all hitting paths of $v_i$ w.r.t. $B_b$ |
| $C$, $d(C)$ | Central Graph, the depth of Central Graph |
| $\overline{A}$ | average shortest distance of graph $G$ |
| $\alpha$ | a tunable parameter to control $a_i$ |
| $l, l_{max}$ | BFS level and the max expansion depth (level) |
| $M, m_{ij}$ | node-keyword matrix, the value for $v_i$ and $t_j$ |

included in some other Steiner Graphs with larger radius. They propose r-clique problem to model keyword search. However, r-clique is not efficient if keywords correspond to large number of nodes. In addition, the output of r-cliques method is a set of keyword nodes. Although, the author provides an algorithm to extract Steiner Trees from a r-clique, the two procedures may cost too much time to return top-k answers. In addition, since the Steiner Trees are generated from already found r-cliques, they may not be global optimal. In other words, there may exists better Steiner Trees that cross two r-cliques. In addition, like aforementioned methods, for efficiency purpose, instead of maintaining a distance matrix, r-clique method maintains a *neighbor index* that records shortest distances that are smaller than R, where R should be larger than r. These parameters may be difficult to fix in a graph with large variety. Qin et al. [19] propose a graph-shaped answers to keyword queries. However, they target relational datasets and may produce redundant answers, as pointed by [18]. Our answers are also modeled by graphs, which are more expressive than tree-shaped answers.

**BFS on modern hardwares.** Our methods are partly inspired by Breadth First Search (BFS) on multi-core systems. There are many works that use multi-core hardwares to implement efficient graph processing algorithms, e.g. [20], [21] for CPUs and [22], [23], [24], [25] for GPUs. In particular, [23], [24] are typical approaches that focus on using GPUs to accelerate BFS. Merrill et al. [26] studies various scheduling policies and parallel algorithms that facilitate BFS over large graphs on GPUs. To the best of our knowledge, we are not aware of any works that propose an algorithm framework that harnesses multi-core CPUs or GPUs to address keyword search problem on Knowledge Bases.

## III. PROBLEM AND CENTRAL GRAPH DEFINITION

To enhance the connection between nodes, we model Wikidata KB as a bi-directed node-weighted graph with both nodes and edges labeled, denoted as $G = (V, E)$, where V and E are the sets of nodes and edges respectively. We use $w_i$ to denote the weight of a node $v_i \in V$. For each edge $e_{ij} = (v_i, v_j) \in E$, $r_{ij}$ denotes the relationship (label) of $e_{ij}$. In our settings, a BFS instance starts from a set of nodes and proceeds level by level with initial expansion level 0. Every node can only be hit once in terms of one BFS instance. A keyword query consists of a set of keywords $Q = \{t_0, t_2, .., t_{q-1}\}$. For each keyword $t_i$, we denote the set of nodes that contain $t_i$ as $T_i$. In our approach, every keyword $t_i$ corresponds to an independent BFS instance $B_i$ with source node set $T_i$. Every BFS instance expands at the same global expansion level.

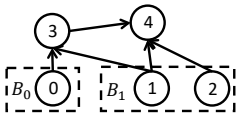



Fig. 2. A simple example to illustrate definitions.

### A. Hitting Level and Hitting Path

**Definition 1.** *(Hitting Level) Given a BFS instance, $B_b$, the hitting level of a node $v_j$ w.r.t $B_b$, denoted by $h_j^b$, is defined as the first BFS expansion level $l$ where $v_j$ becomes a frontier (to expand) in $B_b$.*

**Example 1.** *As shown in Fig. 2, there are two BFS instances, $B_0$ starting from $v_0$ and $B_1$ from $v_1$ and $v_2$. For $B_1$, $v_1$ and $v_2$ have hitting level $h_1^1 = h_2^1 = 0$, since they are source nodes and expand at level 0. $h_3^1 = h_4^1 = 1$, because $v_3$ and $v_4$ are hit at BFS level 0 and become frontiers at level 1. $v_3$ will not expand to $v_4$ in $B_1$, since $v_4$ has already been hit.*

**Definition 2.** *(Hitting Path) Given a BFS instance $B_b$, the hitting path of a node $v_j$ is any expansion path (from source nodes) that hits $v_j$ and makes $v_j$ a frontier in the next expansion level. We denote the set of all hitting paths of $v_j$ w.r.t $B_b$ as $P_j^b$.*

**Example 2.** *In Fig. 2, for $B_1$ and $v_4$, only $v_1 \rightarrow v_4$ and $v_2 \rightarrow v_4$ are hitting paths. $v_1 \rightarrow v_3 \rightarrow v_4$ is not, since it is not an expansion path. There is no expansion from $v_3$ to $v_4$.*

In the next section, we introduce a constraint, called minimum activation level, that lower bounds hitting levels of a node, i.e. a node cannot be hit until the constraint is satisfied. In this way, we can let hitting levels reflect semantic relevance between keywords and nodes, so that nodes with smaller hitting levels can be considered more relevant.

### B. Central Graph and Top-(k,d) Central Graph Problem

We first give the definition and then explain the rationale.

**Definition 3.** *(Central Graph) Given a keyword query $Q = \{t_0, t_2, .., t_{q-1}\}$. For a node $v_j$, if $P_j^i \neq \emptyset$ (w.r.t. $B_i$ and $T_i$) for every $i$, then we define $C = \bigcup_{i=0}^{q-1} P_j^i$ as the Central Graph centered at $v_j$. We denote $v_j$ as Central Node. The size, or equivalently depth, of Central Graph $C$ is defined as the largest hitting level of Central Node $v_j$ by Equation 1.*

$$d(C) = \max_{i \in \{0,1,..,q-1\}} h_j^i \quad (1)$$

We formalize Central Graph model from three perspectives. First, a Central Graph should contain hitting paths from all input keywords and thus connects every keyword. Second, for one keyword, a Central Graph contains all hitting paths to the Central Node, i.e. it allows multi-paths for one keyword. Furthermore, these hitting paths are "shortest" in terms of hitting levels of the Central Node. Third, the depth of Central Graphs is bounded by the maximum hitting level of the respective Central Node. Thus, Central Graphs with smaller depth tend to be more compact.

**Example 3.** *In Fig. 2, there are two Central Graphs. One centered at $v_3$ with depth 1, covering hitting paths $v_0 \to v_3$ and $v_1 \to v_3$. The other is centered at $v_4$ with depth 2, covering hitting paths $v_0 \to v_3 \to v_4$, $v_1 \to v_4$ and $v_2 \to v_4$.*

**Definition 4.** *(top-(k,d) Central Graph Problem) Given a keyword query $Q$ and $k$, find all Central Graphs with depth no larger than $d$, s.t. $d$ is the smallest possible value to obtain at least $k$ Central Graphs.*

The set of all top-(k,d) Central Graphs is a super set of final top-k answers and reduces search space to only consider Central Graphs with smallest depths. In our two-stage algorithm, the first stage is to solve the top-(k,d) Central Graph Problem in Definition 4, given the value k in top-k. Then, the second stage is to further prune and select the final top-k Central Graphs from the set of top-(k,d) Central Graphs. To avoid repetition, once a node is identified as a Central Node, it becomes unavailable for future expansion.

## IV. Minimum Activation Level

In order to generate meaningful answers, we have to properly weight the graph. Otherwise, in an unweighted graph, our search procedure reduces to multiple independent standard BFSes. The resulting Central Graphs would be arbitrary and meaningless. Therefore, we introduce for every node a constraint, called minimum activation level (denoted as $a_i$ for $v_i$), that lower bounds the hitting level of it. Roughly speaking, minimum activation level acts like a "switch", it gradually turns nodes active for search. Early active nodes have more chances appearing in the final answers. Those nodes are expected to be informative and interesting. In this section, we first introduce minimum activation level in Sec. IV-A. Then, we show the effect of minimum activation level in search. Lastly, we explain the intuition behind a parameter $\alpha$, which is tunable in run time and allows users to control the effect of minimum activation level.

### A. Calculation of Minimum Activation Level

In Wikidata KB, there are many *summary nodes* that easily become a shortcut during search. For example, human node (with over 2M in-edges) connects any two nodes representing people by edge *instance of*, and a conference node would connect any two papers that publish in that conference (usually with around hundreds of in-edges) by edge *published in*. Such nodes are pointed to by a large number of same-labeled edges. These *summary nodes* only summarize some trivial commonality of a lot of nodes and tend to be a shortcut that leads to meaningless connections. We use *degree of summary* to denote the extent to which a node tends to be a summary node. In this setting, human node has a large *degree of summary*. We quantify this *degree of summary* by two observations. First, a node with large number of same-labeled in-edges tend to be a summary node. Second, a node with small number of different labels of in-edges tend to be a summary node. For example, *data mining* node has over 1000 in-edges but only 11 different labels of in-edges. It has a large *degree of summary* and it is indeed a *summary node* representing very general topic. Thus, the connection of two papers via *data mining* may not be informative by edge *main topic*. We use *degree of summary* as weight of nodes. Let $R_i$ denote the set of in-edge labels incidental to $v_i$, and for $r \in R_i$, let $\overline{r}$ denote the number of in-edges of label $r$ pointing to $v_i$.

$$w_i = \frac{\sum_{r \in R_i} \overline{r} \log_2 (1 + \overline{r})}{\sum_{r \in R_i} \overline{r}} \quad (2)$$

The term $\log_2 (1 + \overline{r})$ rescales the number of in-edges with label $r$ and represents the contribution from those edges to *degree of summary* of the node. Then the average over all edges is taken to be the total *degree of summary* of that node. By averaging over all edges, we take into consideration the diversity of in-edge labels. That is, if a node has many different in-edge labels, it may still be meaningful even if it has a relatively large number of in-edges with certain label. For ease of processing and explanation, we further normalize $w_i$ by $w_i' = \frac{w_i - min(w)}{max(w) - min(w)}$ and use $w_i$ to denote $w_i'$.

After obtaining node weight (*degree of summary*), we propose a **Penalty-and-Reward mapping** to obtain minimum activation level $a_i$ from $w_i$, with a tunable parameter $\alpha \in (0, 1)$ that allows users to set preference for *degree of summary* in run time. In fact, other mapping strategies may also work. The intuition behind the mapping strategies is to grant informative nodes with small weight and low minimum activation level so that they have higher search priorities over *summary nodes*.

To apply **Penalty-and-Reward mapping**, specifically, we first compute the average distance (hops) between two nodes in the graph by sampling. Then based on the weight of nodes and $\alpha$, we calculate $a_i$ by either increasing (penalty) or decreasing (reward) the average distance to some extent. In this way, we can make $a_i$ in a reasonable range to control search. Let $\overline{A}$ denote the average shortest distance. The sampling data is shown Table II of Sec. VI. The mapping process is reflected by Equation 3, 4 and 5.

$$Penalty(v_i) = \overline{A} \times \frac{(w_i - \alpha)}{1 - \alpha}, \text{ if } w_i > \alpha \quad (3)$$

$$Reward(v_i) = \overline{A} \times \frac{(\alpha - w_i)}{\alpha}, \text{ if } w_i < \alpha \quad (4)$$

$$a_i = \begin{cases} Rounding(\overline{A} - Reward(v_i)) & w_i < \alpha \\ Rounding(\overline{A}) & w_i = \alpha \\ Rounding(\overline{A} + Penalty(v_i)) & w_i > \alpha \end{cases} \quad (5)$$

Equation 3 and 4 scale $w_i$ according to $\overline{A}$. If $w_i > \alpha$, we use the part of $w_i$ exceeding $\alpha$ as a penalty to be added to $\overline{A}$. If $w_i < \alpha$, then we use the part of $\alpha$ exceeding $w_i$ as a reward to be subtracted from $\overline{A}$. In Equation 5, we round the resulting value to its nearest integer, since minimum activation level controls search by comparing with BFS expansion level.

### B. Effect of Minimum Activation Level

During search, nodes with small minimum activation level become active and available for search in an early stage. Specifically, for non-keyword nodes, their hitting level is lower bounded by their minimum activation level, i.e. they are only available for search when the global BFS expansion level reaches their minimum activation level. For keyword nodes, we make a compromise by allowing keyword nodes to be hit without restriction of minimum activation level but to expand only when the BFS expansion level matches its minimum activation level. This adjustment makes it possible to return keyword nodes with high minimum activation level.

### C. Intuition behind $\alpha$

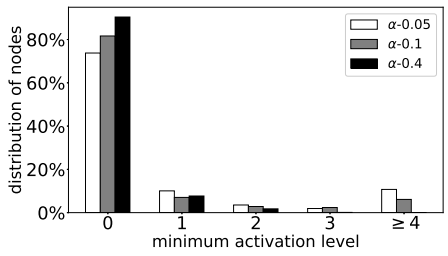



Fig. 3. Nodes' distribution for different $\alpha$'s. Total number of nodes is over 30 millions.

Fig. 3 shows the distribution of nodes for three different $\alpha$ values on Wikidata KB with estimated average distance $\overline{A} = 3.68$. As $\alpha$ becomes larger, nodes with large weight can also map to a relatively small minimum activation level. This shows users can adjust the effect of minimum activation level by changing $\alpha$. Nodes with larger weight tend to express general meanings or topics, such as *conference* nodes and *human* node. These nodes can often lead to meaningless answers but not always. The topic node *data mining* has over 1000 in-edges and only 11 different in-edge labels. It is given a relatively high weight. If the input keywords are {*data, mining, information, retrieval*}, for users who are familiar with these fields may wish to see answers containing specific works related to *data mining* rather than the topic node, but for users who do not know *data mining* or even do not have back ground on Computer Science may wish to see the introduction of data mining topic instead of specific research articles. In practice, we observe that the node *data mining* does not appear in the top answers when $\alpha = 0.1$, but it does when $\alpha = 0.4$. This suggests that larger $\alpha$ maps more nodes to a smaller minimum activation level and thus "decreases" the weight of *data mining* to some extent. Therefore, users can use a larger $\alpha$ to retrieve more nodes with higher *degree of summary*.

## V. Two-stage parallel algorithm

### A. Overview

**Algorithm 1:** Two-stage Parallel Algorithm

```
   /* Bottom-up Search to solve top-(k,d) Central
      Graph Prblem.                                  */
1  BFSLevel ← 0;
2  fork(); Initialize B_i for all t_i in Q; join();
3  while not terminate do
4  |  Enqueue frontiers// Only parallelize on GPU
5  |  fork(); Identify Central Nodes; join();
6  |  fork(); Expansion; join();
7  |  BFSLevel++;
   /* Top-down Processing                            */
8  fork();Extract, prune and rank every Central Graph;join();
```

The search is divided into two stages, bottom-up search and top-down processing as illustrated in Algorithm 1. The two-stage algorithm works in a fork-and-join manner. Threads are synchronized between steps. The details of every step is given in the following sections. In addition, we discuss the complexity and load balancing problem for multi-core CPU and GPU implementations, respectively. We store the graph in Compressed Sparse Row (CSR) format and we do not need any node distance index which may incur huge storage.

### B. Bottom-up Search

**Initialization.** There are three data structures we maintain in order to realize a lock-free procedure. First, *FIdentifier* records 1 for nodes becoming frontiers in the next iteration, otherwise 0. After enqueuing frontiers at every iteration, *FIdentifier* are set to all 0 in parallel. Second, *CIdentifier* records 1 for node already identified as Central Node, otherwise 0. Note that the sum of all elements of *CIdentifier* gives the number of already identified Central Nodes. Third, a node-keyword matrix, $M$ for short, is initialized. Its element, $m_{ij}$ (i-th row and j-th column), records the hitting level of $v_i$ for keyword $t_j$. $m_{ij} = 0$ if $v_i$ contains keywords otherwise $\infty$. The size of *FIdentifier* and *CIdentifier* is $\Theta(|V|)$. The size of $M$ is $\Theta(|V|q)$, where $q$ is the number of keywords. For GPU implementation,$M$ is directly initialized on GPU and transferred back to CPU after search is done. The size of $M$ is not a bottleneck for our problem. Here, we give a concrete example. Given a graph of 30M nodes and a query of 10 keywords, the total size of $M$ is only 300MB, since one byte is all we need to record a hitting level ($m_{ij}$). Given a bandwidth of around 12GB/sec from GPU to CPU of today's hardware, the transfer time of $M$ takes only around 25ms, which is small enough to produce real-time responses.

**Enqueuing frontiers.** At the beginning of each iteration (a new expansion level), we extract and enqueue nodes into frontier queue by examining the flags in $FIdentifier$ which was modified in last iteration or in initialization phase. This enqueuing process is sequentially writing nodes with flag 1

in *FIdentifier* to frontier queue. On GPU, we parallelize the process of checking *FIdentifier* and write nodes to frontier queue with lock. However, we find that on CPU locked writing is so expensive and the fastest way is to enqueue frontiers in a sequential manner. This difference is due to the extremely high bandwidth of GPU with **DDR5X**. Note that we only maintain one frontier queue to record nodes to traverse at each BFS expansion level for all BFSes. This is called joint frontier array [27]. A node becomes a frontier as long as it is one in any BFS instance. Therefore, different BFS instances may share a frontier. In expansion procedure, we describe how a frontier knows which BFS it belongs to.

**Identifying Central Nodes.** From $M$, it is easy to identify whether $v_i$ is Central Node by checking $m_{ij}$ for each keyword $t_j$. Note that we only need to check frontiers, since frontiers are nodes that were just modified at last BFS level. In addition, the depth of the Central Graph can be correctly obtained according to Lemma V.1.

**Lemma V.1.** *The depth $d$ of a Central Graph centered at $v_i$ equals the BFS level where $v_i$ is identified as a Central Node.*

*Proof.* Suppose $v_i$ is identified at level $l$, then $v_i$ must be modified by some BFS $B_j$ at level $l-1$, which means $m_{ij}$ is set to $l$, the maximum BFS level currently. Based on Equation 1, the depth $d$ of a Central Graph centered at $v_i$ is $l$. □

**Expansion (Algorithm 2).** After the previous steps, only frontiers not identified as Central Nodes can be expanded in this procedure. We explain the expansion procedure in a sequential way and then introduce the parallelization of Algorithm 2. There are three loops in the expansion procedure. To put it simply, for every frontier $v_f$ (line 1), we iterate every $B_i$ to see if $v_f$ is a frontier of $B_i$ (line 8). If $v_f$ satisfies expansion conditions, then we iterate through all its neighbor w.r.t. $B_i$ (line 12). For a frontier $v_f$ to expand in $B_i$, $a_f$ and $m_{fi}$ should both be no larger than BFS expansion level $l$ (line 5 and 9). For a neighbor $v_n$ to accept expansion, it should not be visited in $B_i$ and $a_n$ should be at least $l$ (line 14 - 20).

The parallelization is slightly different for CPUs and GPUs. On GPU, we have far more threads than CPU. We let one warp handle one $B_i$ of a frontier $v_f$ (a warp is a group of threads active simultaneously in SIMD model). The threads within a warp handle different neighbors of $v_f$. In comparison, on CPU, we have fewer but more powerful threads. However, the communication cost among threads on CPU is also more expensive. If we parallelize the innermost loop as GPU, we need to synchronize threads many times and schedule them dynamically, which incurs a high time cost. Therefore, we use a coarse-grained parallelism. We simply let threads on CPU handle different frontiers with a dynamic scheduling, which means once a thread finishes a frontier, it looks for another. Note that the number of total frontiers is far smaller than that of total neighbors. The communication cost for dynamic scheduling is thus tolerable.

**Algorithm 2:** Expansion Procedure

```
   Input : data graph G, M, frontiers, node weights, BFS expansion
           level l, α
   Output: modified M and FIdentifier
   /* CPU threads parallel level                          */
1  foreach frontier f do
2      if CIdentifier[v_f] = 1 then
3          continue;
4      calculate a_f from w_f and α;
5      if a_f > l then
6          FIdentifier[v_f] ← 1;
7          continue;
       /* GPU warps parallel level                        */
8      foreach BFS instance B_i do
9          h^i_f ← m_fi;
10         if h^i_f > l then
11             continue;
           /* GPU threads parallel level                  */
12         foreach neighbor v_n of v_f do
13             h^i_n ← m_ni;
14             if h^i_n ≠ ∞ then
15                 continue;
16             if v_n is not a keyword node then
17                 calculate a_n from w_n and α;
18                 if a_n > l + 1 then
19                     FIdentifier[v_f] ← 1;
20                     continue;
21             m^i_n ← l + 1;// Set hitting level in M
22             FIdentifier[v_n] ← 1;
23 return;
```

Theorem V.2 gurantees the lock-free property of writes and reads. Theorem V.3 guarantees we identify all Central Nodes for top-(k,d) Central Graphs.

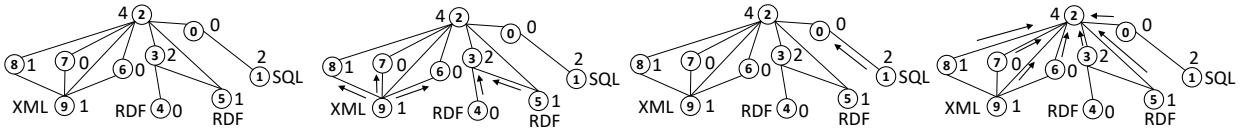
**Theorem V.2.** *Algorithm 2 is a lock-free procedure and all reads and writes are guaranteed correct.*

*Proof.* We only need to examine all reads and writes for *FIdentifier* and $M$, since only they may be modified. First, all values written to *FIdentifier* and $M$ is 1 and $l+1$ respectively, where $l$ is the BFS level. Therefore, we do not need to add lock when two writes are for a same location. Second, all values read from $M$ (there is no read from *FIdentifier*) may change but do not affect the truth value in if-condition (line 10 and 14). At line 10, $m_{fi}$ may be changed from $\infty$ to $l+1$ in the situation where $v_f$ is a neighbor of another frontier in $B_i$. In either case ($m_{fi} = l+1$ or $m_{fi} = \infty$), we should not expand $v_f$. And it is similar for if-condition at line 14. □

**Theorem V.3.** *Given top-k value, the bottom-up search procedure solves top-(k,d) Central Graphs Problem correctly.*

*Proof.* The search procedure is in line with the definition of Central Graphs and stops at the smallest level $d$ where we collect at least k Central Graphs. □

**Load balancing.** We implement on CPU by OpenMP. Load balancing is automatically handled by using the dynamic scheduling feature of OpenMP. On GPU, each warp handles one frontier and one BFS instance. The load balancing problem only becomes severe when a frontier has too many edges. Such nodes tend to have large *degree of summary*, resulting in



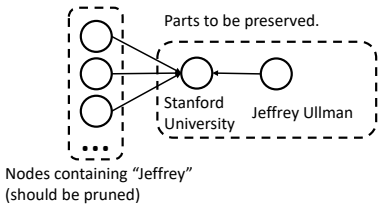

(a) BFS expansion level 0 (b) BFS expansion level 1 (c) BFS expansion level 2 (d) BFS expansion level 3

Fig. 4. A running example with values of $a_i$ attached to each node from Fig. 1. Arrows denote the expansion.

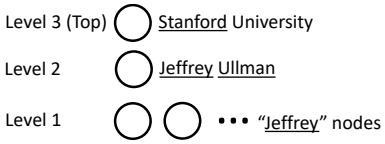
a high minimum level during search. They have little chance to expand. On the other hand, it takes much more cost to evenly divide the neighbors of all frontiers and BFSes to threads, since the total number is only known after scanning all neighbors.

**Time and space complexity.** Since the computation is not overlapped during parallel execution, the time complexity is calculated by dividing sequential time complexity by the number of threads, denoted by $T$. We discuss the time complexity for initialization, enqueuing frontiers, identifying Central Nodes and expansion separately. First, for initialization, we just set the keyword nodes for each $B_i$. Thus, the time complexity is $\mathcal{O}(|V|q)$ in sequential on CPU and $\mathcal{O}(\frac{|V|q}{T})$ in parallel on GPU, where $q$ is the number of keywords. Second, the time complexity of enqueuing frontiers is $\mathcal{O}(\frac{|V|l_{max}}{T})$ where $l_{max}$ is the maximum number of iteration or depth allowed, since we scan through the *FIdentifier* array and a node may be a frontier many times. Third, The time complexity of identifying Central Nodes is $\mathcal{O}(\frac{|V|ql_{max}}{T})$, since for each frontier, we need to scan all hitting levels in $M$ and the number of frontiers is bounded by $|V|$. Last, For expansion procedure, we run $q$ BFS-like instances concurrently. In standard BFS, a visited node will not be visited again, but in our BFS-like search, a node continued to be a frontier if it is inactive or it has inactive neighbors. This causes one BFS-like instance in our algorithm to have time complexity $\mathcal{O}(|V||E|l_{max})$ instead of $\mathcal{O}(|V| + |E|)$ for standard BFS. Since there are $q$ BFSes and $T$ threads, we have $\mathcal{O}(\frac{|V||E|ql_{max}}{T})$. For space complexity, the memory occupation includes graph data in CSR format $\Theta(|V| + |E|)$, the node weight array $\Theta(|V|)$, the *FIdentifier* $\Theta(|V|)$, the *CIdentifier* $\Theta(|V|)$ and the node-keyword matrix $\Theta(|V|q)$ where q is the number of keywords. In total, the space complexity is $\mathcal{O}(q|V| + |E|)$.

**Example 4.** *Fig. 4 illustrates a running example for nodes in Fig. 1. In Fig. 4a, three BFS instances are initiated at* $\{v_9\}$, $\{v_4, v_5\}$ *and* $\{v_1\}$. *At every BFS level, the expansion is in parallel. Since* $a_4 = 0$, *only* $v_4$ *is active to expand, but* $v_3$ *is not active because* $a_3 = 2$. *As a result, there is no expansion. At expansion level 1 shown in Fig. 4b,* $v_9$, $v_4$ *and* $v_5$ *start expansion. Also,* $v_3$ *is able to accept expansion and it becomes a frontier in level 2 which reaches its minimum activation level. The hitting levels of new frontiers are,* $h_6^0 = h_7^0 = h_8^0 = h_3^1 = 2$. *Finally, at level 3 (Fig. 4d), every node near* $v_2$ *can expand to it.* $v_2$ *is identified as a Central Node in the next iteration (not shown) and its depth is 4.*




(a) Central Graph before pruning




(b) Level classification

Fig. 5. Level cover strategy example.

### C. Top-down Processing

There are three major steps in top-down processing. First, given the Central Nodes from the first stage, we need to extract respective Central Graphs. Second, to obtain even more compact answers, we apply a **level-cover strategy** to all Central Graphs to prune redundant nodes based on keyword co-occurrence. Third, after pruning, we select the final top-k answers by proper scoring function. We let one thread to recover one or more Central Graphs. The load balancing problem is handled by OpenMP dynamic scheduling feature. We implement the top-down process on CPU rather than GPU, because it not only needs dynamic memory allocation for recording recovered nodes and paths, but also diverges a lot in terms of program executions.

**Level-cover strategy.** To make the final top-k answers even "thinner", we propose a keyword co-occurrence based level-cover pruning strategy to prune nodes that are redundant within a Central Graph. We classify only keyword nodes within a Central Graph into different levels based on the number of keywords they contribute. The Central Node is always at the top level. We proceed in a greedy manner starting downwards from top level where nodes contain most keywords. If nodes in one level already covers all keyword, we then prune all nodes in the rest levels along with the hitting paths from pruned nodes to Central Nodes. In this pruning strategy, we preserve as many keyword nodes as possible, since nodes will not lead to pruning of nodes within the same level. An example below is given with respect to Fig. 5.

**Example 5.** *As shown in Fig. 5, the input keywords are*

*Stanford, Jeffrey and Ullman. After pruning nodes with only one keyword "Jeffrey", we have an answer with only Stanford University and Jeffrey Ullman nodes.*

**Scoring function.** To select the final top-k answers from the **pruned** top-(k,d) Central Graphs, we propose a ranking function that restricts the "width" of Central Graphs, as in Equation 6.

$$S(C) = d(C)^{\lambda} \sum_{v_i \in C} w_i \tag{6}$$

where $C$ represents a Central Graph and $\lambda \geq 0$ is a parameter that controls the effect of depth of the respective Central Graph. We set $\lambda = 0.2$ by default.

**Algorithm.** By Theorem V.4, we can correctly recover nodes contained in a Central Graph as long as Central Node and node-keyword matrix are known from the first stage. As illustrated in Algorithm 3, we start a standard BFS search from each node. For every frontier $v_f$, we apply Theorem V.4 between $v_f$ and its neighbors to see whether a neighbor can be recovered w.r.t. a keyword $t_i$ (line 8 to 10). We apply **level-cover strategy** after extraction (line 13). At last, we insert the pruned graph to the top-k answer heap (line 14).

**Theorem V.4.** *Suppose $v_i$ expands to $v_j$ during bottom-up search and the extraction is now at $v_j$ to extract $v_i$, we have the following heuristics between $h_i^l$ and $h_j^l$ for a certain keyword $t_l$.*

1) *If $v_j$ contains keywords, $h_j^l = 1 + \max\{a_i, h_i^l\}$,*
2) *If $v_j$ contains no keywords, $h_j^l = 1 + \max\{a_i, h_i^l, a_j - 1\}$.*

*Proof.* The quantitative relationship is due to the fact that $a_j$ lower bounds $h_j^l$ if $v_j$ does not contain keywords. Otherwise, $h_j^l$ is lower bounded by 0. □

**Time and space complexity.** First, for the extraction step, it is a standard BFS traversal from the respective Central Node, and for each node we check through the hitting levels of $q$ keywords. It can also be thought of as $q$ independent standard BFSes, one for each keyword. Therefore the time complexity is $\mathcal{O}(q(|V| + |E|))$. Second, for level-cover strategy, to classify all keyword nodes, we need to scan all hitting levels of these nodes, which is bounded by $\mathcal{O}(q|V|)$. To do pruning, we need to scan from top level to the lowest level, in the worst case no keyword nodes are pruned. In this case, we scan everything again and the time complexity is also $\mathcal{O}(q|V|)$. Therefore, the total time complexity of level-cover strategy is $\mathcal{O}(q|V|)$. Third, to insert result to $T_k$ which is a heap. Thus, the complexity of insertion is $\mathcal{O}(\log_2 k)$ for maintain top-k answers. All together, suppose we have $|C|$ top-(k,d) Central Graphs, the time complexity is then $\mathcal{O}(|C|(q(|V| + |E|) + \log_2 k))$ in sequential execution and $\mathcal{O}(\frac{|C|(q(|V|+|E|)+\log_2 k)}{T})$ in parallel, where $T$ is the number of threads. For space complexity, the major cost arises from storing Central Graphs while extraction, besides node-keyword matrix and graph storage cost. Note that the number of nodes of a Central Graph is bounded by $|V|$, then in worst cases the space complexity is $\mathcal{O}(|C||V| + (q|V| + |E|) + k)$, where k denotes the number of elements in the answer heap and $|C|$ is the number of all top-(k,d) Central Graphs. The part, $\mathcal{O}(|C||V|+(q|V|+|E|)+k)$, comes from node-keyword matrix and graph storage. In practice, the top-(k,d) Central Graphs tend to be compact with small number of nodes.

**Algorithm 3:** Top-down Processing

```
   Input  : G(V, E), M, identified Central Nodes, node weights, α, k
   Output: Final top-k answers
1  Initialize top-k answer heap T_k;
2  foreach v_c in Identified Central Nodes do
3      insert v_c to frontier queue f;
4      while f ≠ ∅ do
5          v_f ← f.next(), f' ← ∅;
           /* Scan neighbors of v_f                          */
6          foreach v_n ∈ N(v_f) do
7              foreach B_i do
8                  if v_f has keywords and h_f^i = 1 + max{a_n, h_n^l}
                     then
9                      Extract v_n, insert v_n into f', if not in f';
10                 if v_f has no keywords and
                     h_f^i = 1 + max{a_n, h_n^l, a_f − 1} then
11                     Extract v_n, insert v_n into f', if not in f';
12         f ← f';
       /* let C_n denote the Central Graph at v_c   */
13     Apply level-cover strategy to C_n;
14     Insert into T_k, if possible;
15 return;
```

## VI. Experiment studies

The first goal of our work is to obtain performance at scale. We evaluate our success in this direction, and report results in the first subsection below. The second goal of our work is to return effective answers in face of the large variety in Wikidata KB. We also evaluate our success in this direction, and report results in the second subsection below.

**Competitors.** We use following implementations.

1) *GPU-Par*. The proposed parallel algorithm on GPU as described in Sec. V. Our online system is based on GPU implementation.
2) *CPU-Par*. The proposed parallel algorithm on CPU as described in Sec. V.
3) *CPU-Par-d*. We implement a parallel algorithm with dynamic memory allocation, which does not require node-keyword matrix but needs locks on writes and reads. In addition, there is no extraction phase needed, since all Central Graphs are recorded during search. By comparing with it, we validate the efficiency of our designs.
4) *BANKS-II* [4]. We compare with the established and widely used method *BANKS-II* [4] for both efficiency and effectiveness. The reasons are as follows. There are few established parallel keyword search methods that can work on graphs. [12] can only apply to relational database. As discussed in [7], in order to find top-k answers, the proposed parameterized DP (dynamic programming) algorithm finds the top-1 result, followed by top-2, top-3, and so on. This process is rather slow, as pointed by [6]. However, [6] finds answers in a progressive manner. In other words, the answers are

TABLE II
WIKIDATA DUMPS

| dataset(year) | # nodes | # edges | $\overline{A}$ | Deviation |
|---|---|---|---|---|
| wiki2017 | 15.1M | 124M | 3.87 | 0.81 |
| wiki2018 | 30.6M | 271M | 3.68 | 0.98 |

TABLE III
PARAMETERS IN EXPERIMENTS.

| Parameter | Meaning | Default |
|---|---|---|
| Topk | Top-k answers to be returned | 20 |
| Knum | The number of keywords in a query | 6 |
| $\alpha$ | The tunable parameter introduced in Sec. IV-A | 0.1 |
| Tnum | The number of threads for parallel | 30 |

generated better and better until the true Steiner Tree is found. It is not clear how to collect top-k results by [6]. BLINKS [4] needs to pre-compute all-pair shortest path between nodes and keywords to build two indexes, which are *keyword-nodes lists* and *node-keyword map*. Similarly, EASE [17] has to use node matrix to pre-compute steiner graphs as well as all-pair keywords distance in addition. Furthermore, [18] needs to build a *neighbor index* as mentioned in Sec. II and requires domain experts to define value $r$, which is difficult on Wikidata KB with so large variety. [16] is a partition method for backward algorithm with a disk solution for small RAM. Taking the above into account, we finally choose *BANKS-II* as our competitor.

**Dataset.** As we focus on Wikidata Knowledge Base, we obtain two dumps as shown in Table II. The last two columns show the sampled average distance and the deviation of sampling. We sample ten thousand pairs of nodes to estimate the average shortest distances. The statistics are collected after we filter out non-English contents.

**Platform.** All algorithms are implemented using C++ 4.8.5 with openmp 3.1 and cuda 8.0. We turn on -O3 flag for compilation. All tests were run on Centos 7.0. We use a single machine with 52-core Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10GHz and a single GPU, GTX 1080 Ti. It is worth noting that our CPU has 1 TB DDR4 as its main memory with data width 64 bits, and our GPU has 11 GB memory with DDR5X 352-bit memory bus width. It can be seen that GPU has a much faster transfer speed (480GB/s) between processors and main memory than CPU (around 56 GB/s).

In all experiments, we set the time limit as 500 seconds. If the running time exceeds this limit, we note it as 500 to compute averages. All running times are in millisecond (ms).

### A. Efficiency Studies

In this section, we evaluate the efficiency of our proposed approaches. Table III summarizes the parameters we study. We vary one parameter at a time while others are set to default values. For each Knum, we randomly select 50 keyword queries from keyword lists of all accepted (over 300) papers in AAAI'14 from UCL repository [28], as these keywords naturally serve as reasonable queries. The running time is calculated as the average of all 50 queries.

**Exp-1 (Vary Knum)** As shown in Fig. 6 and 7, we provide a detailed profiling comparison for each phase in our algorithm. The results of *BANKS-II* are shown only in the last figure of total time. For initialization, *GPU-Par* and *CPU-Par* are both faster than *CPU-Par-d*, since they only need to set the node-keyword matrix in a lock-free way. However, *CPU-Par-d* has to add a lock to each node to record which keyword it has, since the memory is allocated dynamically. The advantage of DDR5X of our GPU is reflected by *Enqueuing Frontier* and *Identifying Central Nodes*, as *GPU-Par* consistently beats other methods. It is also the case for *expansion* procedure. In addition, the proposed lock-free expansion approach is 2 to 3 magnitudes faster than *CPU-Par-d* which needs a lock for modifying any nodes during search. The benefits of *CPU-Par-d* is that there is no need to extract and recover Central Graphs from Central Nodes, since all path information is kept in run time. Therefore, level-cover strategy is the only thing for *CPU-Par-d*. As shown in *Top-down processing*, *CPU-Par-d* is always the fastest. However, this advantage is easily overwhelmed by the slow processing of other phases of *CPU-Par-d*. This also validates the efficiency of heuristics we use for recovering Central Graphs. As shown in *Total time*, *GPU-Par* and *CPU-Par* are 2 to 3 magnitudes faster than *BANKS-II*. This efficiency is essential for keyword search service.

As Knum becomes larger, the number of frontiers and top-(k,d) Central Graphs also becomes larger. However, this change only leads to a small increase of proposed *GPU-Par* and *CPU-Par*, which suggests our method works efficiently and stably for longer keyword queries.

We observe that there are three main reasons for *BANKS-II* to be slow. Firstly, compared to *BANKS-I* which is purely based on backward search, *BANKS-II* adds forward testing to avoid traversing too many neighbors from a node in backward direction. However, when the graph becomes large, the situation now is that there is also a large number of nodes in the forward direction, which causes the program trapped in nodes with many forward edges. Secondly, the top-k termination checking turns out to be very inefficient. To guarantee the correctness of top-k results, *BANKS-II* needs to make sure that there is no better results in the undiscovered answer trees, but the best possible score of undiscovered trees changes slightly. As a result, *BANKS-II* needs to search many nodes to guarantee the correctness of top-k answers. Thirdly, *BANKS-II* expands based on activation of nodes as priorities instead of shortest distances, on which the final scoring is based. This may cause a node to be reached in a shorter distance by the same keyword. Then it needs to broadcast this shorter path to all its parents, which is a recursive update and costs much time when the graph size is large. In contrast, we model the problem in a different way which can execute in parallel naturally. The benefit is that we can explore the potential parts that generate answers and non-potential parts of graph at the same time, which saves time on searching and pruning non-potential nodes

**Exp-2 (Vary Topk)** As shown in the first row in Fig. 8, *GPU-Par* and *CPU-Par* have a stable running time for

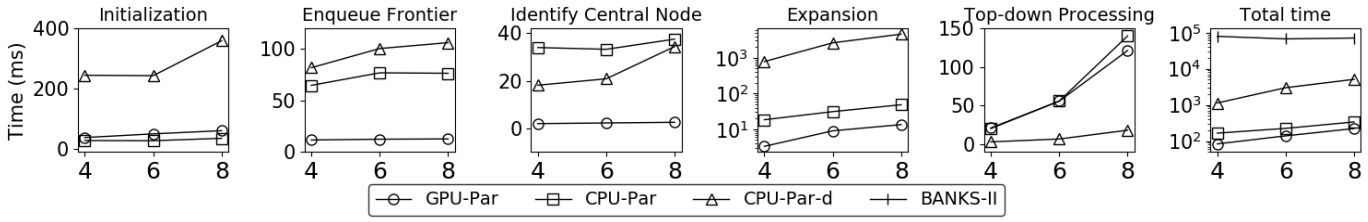



Fig. 6. Vary Knum on wiki2017. The y-axis may be log-scaled, which can be seen from the attached value.

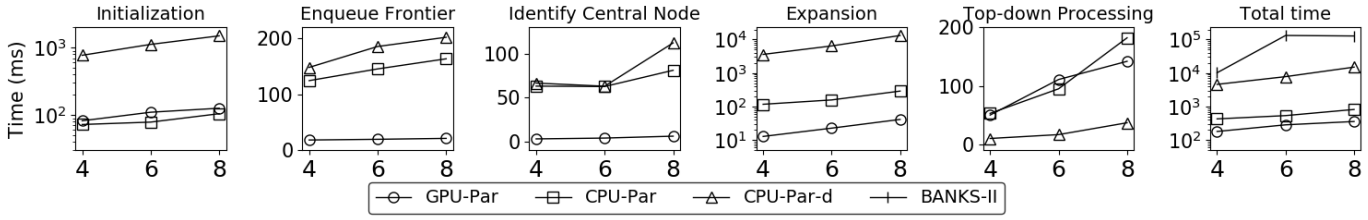



Fig. 7. Vary Knum on wiki2018. The y-axis may be log-scaled, which can be seen from the attached value.

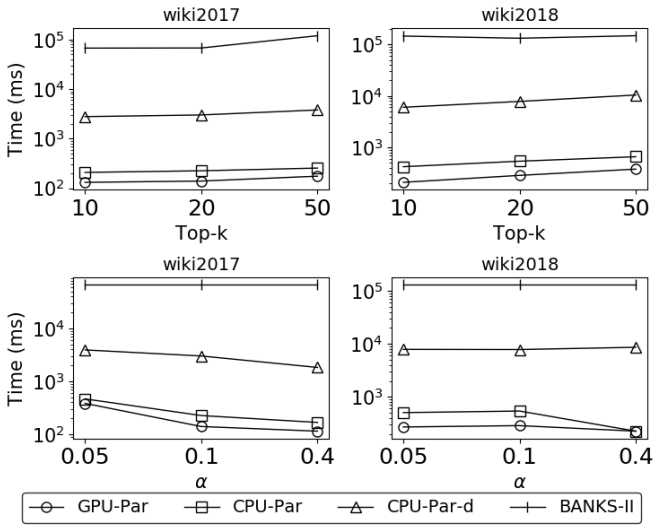



Fig. 8. Vary topk and $\alpha$ on both datasets.

different Topk settings. The reason is that the top-k answers are selected from the set of all top-(k,d) Central Graphs and the running time increases saliently only when more levels (larger d) need to be searched for obtaining k answers.

**Exp-3 (Vary $\alpha$)** As shown in the second row in Fig. 8, the running time goes down as $\alpha$ becomes smaller. This is because larger $\alpha$ grants more nodes with a smaller minimum activation level, which facilitates search and thus answers can be discovered faster. These answers tend to include some nodes with high *degree of summary* (Sec. IV-A).

**Exp-4 (Vary Tnum)** As shown in Fig. 9 and 10, we vary the number of threads from 1 to 50. $Tnum = 1$ means we are running everything sequentially on CPU. Note that GPU implementation (*GPU-Par*) keeps parallelism and is only affected in top-down processing step by Tnum on CPU. For CPU implementation, with larger Tnum, the acceleration is salient especially for *Identifying Central Nodes , Expansion* and *Top-down Processing* steps. For *CPU-Par-d*, it does not benefit so much from large number of threads, since the locked writes and reads slow down the whole processing and overwhelm the benefits from parallelism. This validates the success of our proposed lock-free algorithm.

TABLE IV
RUNNING STORAGE COST ON GPU (KNUM=8, TOPK=50).

| dataset | pre-storage | max. running storage |
|---|---|---|
| wiki2017 | 1.19GB | 1.46GB |
| wiki2018 | 2.41GB | 2.92GB |

**Run time storage** Table IV shows the pre-storage and the maximum running storage cost (including pre-storage) on GPU for *GPU-Par* and it is the same but not a primary concern on CPU since CPU has sufficient memory. Note that the storage does not includes texture and content information of the datasets, which can be stored in external memory. The pre-storage includes the weight of all nodes and adjacency matrix in CSR format. The dynamic memory cost includes *FIdentifier*, *CIdentifier* and node-keyword matrix. We set Knum to be 8 and topk 50, so the size recorded in Table IV is the largest in all experiments. The total size of global memory on GTX 1080 Ti is around 11 GB, which suggests that we can handle much larger graphs.

As can be seen from all experiments, *GPU-Par* always performs the best thanks to its high bandwidth and large number of threads. We thus implement our online search engine using GPU. It is worth noting that the price of GTX 1080 Ti is much lower than that of multi-core CPU.

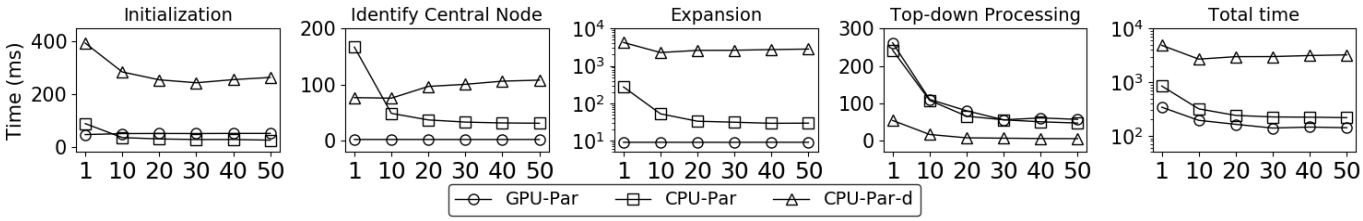



Fig. 9. Vary Tnum (the number of Threads) on wiki2017.

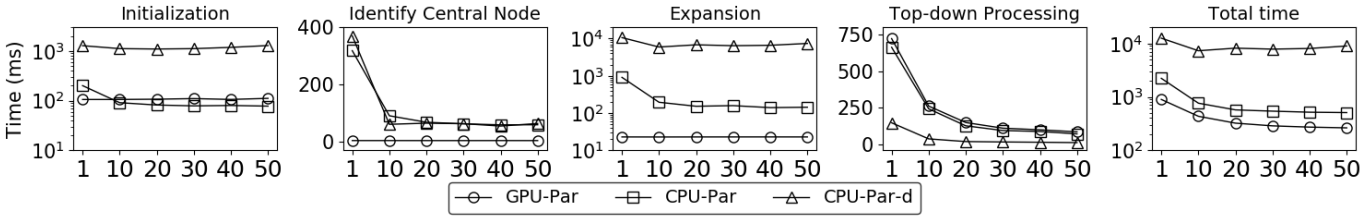
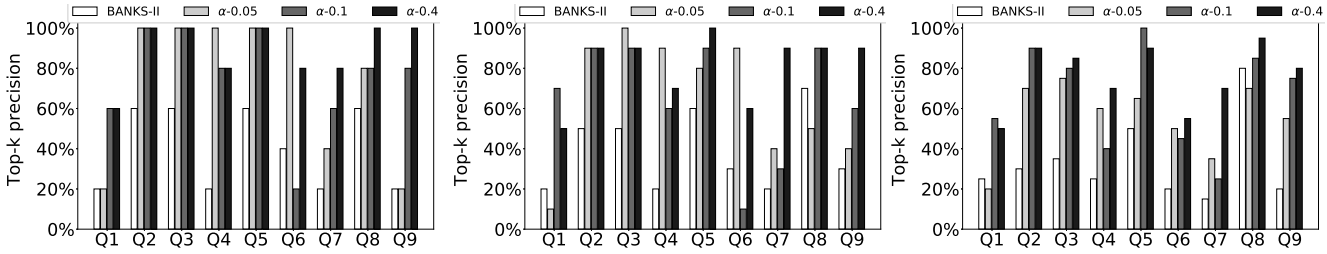
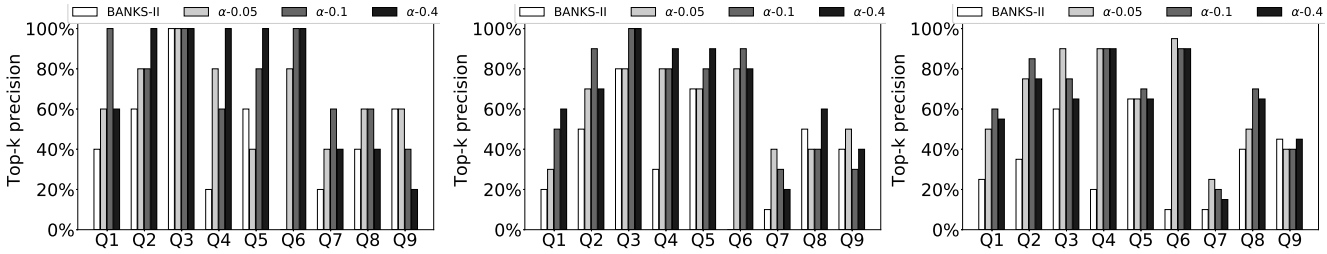



Fig. 10. Vary Tnum (the number of Threads) on wiki2018.




(a) top-5 (b) top-10 (c) top-20

Fig. 11. wiki2017




(a) top-5 (b) top-10 (c) top-20

Fig. 12. wiki2018

### B. *Effectiveness Studies*

We follow the tradition of effectiveness experiments for keyword search problem [3], [4], [18], [10], [5], [17]. The effectiveness of keyword search results is measured by **top-k precision** and the relevances of answers are judged manually. We compared the results from our approach with *BANKS-II* on both datasets and three settings of $\alpha$'s, denoted by $\alpha$-0.05,$\alpha$-0.1,$\alpha$-0.4. Top-k precision measures the percentage of relevant answers that appear in top-k results and is used by many previous works. The queries we used are listed in Table V.

The results are shown in Fig. 11 and 12. We do not show the results for Q10 and Q11, because all settings can return all relevant results and this arises from two perspectives of reasons worth mentioning. For Q10, these keywords have lots of co-occurrences and it is easy to find all relevant small answers. For Q11, the input keywords have little ambiguity and can be mapped to a very small number of entity nodes. As a result, any connected answers tend to be very relevant for Q11.

For other queries except Q10 and Q11, we find that *WikiSearch* can always find a setting of $\alpha$ that can match or outperform the effectiveness of *BANKS-II*. We identify two designs that make *WikiSearch* more effective than *BANKS-II*.

TABLE V
QUERIES FOR EFFECTIVENESS EXPERIMENT. **KWF1** AND **KWF2** DENOTE THE AVERAGE KEYWORD FREQUENCY ON WIKI2017 AND WIKI2018.

| Query | keywords | kwf1 | kwf2 |
|---|---|---|---|
| Q1 | XML relational search | 7555 | 54744 |
| Q2 | database indexing ranking search | 2470 | 17452 |
| Q3 | Bayesian inference Markov network | 2969 | 20700 |
| Q4 | statistical relational learning inference | 6999 | 56815 |
| Q5 | SQL RDF knowledge base | 4674 | 36498 |
| Q6 | supervised learning gradient descent machine translation | 4193 | 18732 |
| Q7 | transfer learning auxiliary data retrieval text classification | 4203 | 44127 |
| Q8 | XML RDF knowledge base sharing | 4143 | 31833 |
| Q9 | network mining medicine retrieval technique | 6353 | 46981 |
| Q10 | natural language processing machine learning | 10333 | 54940 |
| Q11 | Wikidata Freebase Yahoo Neo4j SPARQL | 369 | 448 |

Firstly, *BANKS-II* approximates Steiner Tree and its scoring function takes the sum of length of paths from root to every leaf node. This scoring metric fails taking into consideration of co-occurrences of keywords. As a result, *BANKS-II* fails Q4, Q6 and Q7. Phrases fail to appear together, which results in irrelevant answers, e.g. "Statistical relational learning" or "statistical inference" of Q4, "gradient descent" or "machine translation" of Q6 and "transfer learning" of Q7. In contrast, *WikiSearch* allows multi-paths from one keyword node sets and then prunes the final top-k results by level-cover strategy, which is based on co-occurrences of keywords. This helps maintain the nodes containing phrases and remove nodes that only contribute isolated keywords. The answers are thus more relevant. Secondly, *BANKS-II* searches for small connected trees, which incur many repetitions of answers which overlap a large portion of nodes. The repetitions can cause problem, if the part that repeats is irrelevant. This directly causes all repetitive answers containing that part are irrelevant. In Q11 on wiki2018 dataset, the node representing an irrelevant article *Genotyping on a thermal gradient DNA chip.* appears in 16 different answers of top-20, contributing the keyword "gradient" for 16 times. In contrast, Central Graphs cover more parts of the underlying graphs and we remove the Central Graph that completely contains smaller ones. As a result, a Central Graph covers what it can cover at most, leading to fewer repetitions.

## VII. CONCLUSION

We propose the Central Graph model, which can naturally work in parallel and return meaningful answers on Knowledge Bases. We carefully design a two-stage parallel algorithm to work in a lock-free way which is critical to efficiency. Finally, we implement an online service, *WikiSearch*, for Wikidata Knolwdeg Base.

## REFERENCES


[1] T. Pellissier Tanon, D. Vrandečić, S. Schaffert, T. Steiner, and L. Pintscher, "From freebase to wikidata: The great migration," in *Proc. WWW'16*, pp. 1419–1428.

[2] F. Erxleben, M. Günther, M. Krötzsch, J. Mendez, and D. Vrandeăić, "Introducing wikidata to the linked data web," in *Proceedings of the 13th International Semantic Web Conference - Part I*, ser. ISWC '14, pp. 50–65.

[3] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, P. Parag, and S. Sudarshan, "Banks: Browsing and keyword searching in relational databases," in *Proc.VLDB'02*, pp. 1083–1086.

[4] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *Proc. VLDB'05*, 2005, pp. 505–516.

[5] H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: Ranked keyword searches on graphs," in *Proc. SIGMOD'07*, pp. 305–316.

[6] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Efficient and progressive group steiner tree search," in *Proc. SIGMOD'16*, pp. 91–106.

[7] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding top-k min-cost connected trees in databases," in *ICDE'07*, pp. 836–845.

[8] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: Enabling keyword search over relational databases," in *Proc. SIGMOD'02*, pp. 627–627.

[9] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *Proc.VLDB'02*, pp. 670–681.

[10] A. Balmin, V. Hristidis, and Y. Papakonstantinou, "Objectrank: Authority-based keyword search in databases," pp. 564–575.

[11] E. Ihler, "The complexity of approximating the class steiner tree problem," in *Graph-Theoretic Concepts in Computer Science*, G. Schmidt and R. Berghammer, Eds., 1992, pp. 85–96.

[12] L. Qin, J. X. Yu, and L. Chang, "Ten thousand sqls: Parallel keyword queries computing," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 58–69, Sep. 2010.

[13] D. Yue, G. Yu, J. Liu, T. Zhang, T. Nie, and F. Li, "Efficient keyword search for slca in parallel xml databases," in *2011 8th Web Information Systems and Applications Conference*, pp. 29–34.

[14] B. Ning, X. Zhou, and Y. Shi, "Parallel processing the keyword search in uncertain environment," in *2012 International Conference on System Science and Engineering (ICSSE)*, pp. 409–414.

[15] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan, "Keyword search on external memory data graphs," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 1189–1204, Aug. 2008.

[16] W. Le, F. Li, A. Kementsietsidis, and S. Duan, "Scalable keyword search on large rdf data," *TKDE*, vol. 26, no. 11, pp. 2774–2788, 2014.

[17] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: An effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *Proc. SIGMOD'08*, pp. 903–914.

[18] M. Kargar and A. An, "Keyword search in graphs: Finding r-cliques," *Proc. VLDB'11*, vol. 4, no. 10, pp. 681–692.

[19] L. Qin, J. X. Yu, L. Chang, and Y. Tao, "Querying communities in relational databases," in *Proc. ICDE'09*, pp. 724–735.

[20] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, 2011, pp. 78–88.

[21] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, 2011, pp. 65:1–65:12.

[22] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proc. HiPC'07*, pp. 197–208.

[23] Z. Fu, H. K. Dasari, B. Bebee, M. Berzins, and B. Thompson, "Parallel breadth first search on gpu clusters," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 110–118.

[24] L. Luo, M. Wong, and W. m. Hwu, "An effective gpu implementation of breadth-first search," in *Design Automation Conference*, June 2010, pp. 52–55.

[25] J. Zhong and B. He, "Parallel graph processing on graphics processors made easy," *Proc. VLDB Endow.*, vol. 6, no. 12, pp. 1270–1273, Aug. 2013.

[26] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *SIGPLAN Not.*, vol. 47, no. 8, pp. 117–128, Feb. 2012.

[27] H. Liu, H. H. Huang, and Y. Hu, "ibfs: Concurrent breadth-first search on gpus," in *Proc. SIGMOD'16*, pp. 403–416.

[28] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml