Fast Top-K Search in Knowledge Graphs

Shengqi Yang*, Fangqiu Han*, Yinghui Wu†, Xifeng Yan*
*University of California Santa Barbara   †Washington State University
Email: *{sqyang, fhan, xyan}@cs.ucsb.edu, †yinghui@eecs.wsu.edu

Abstract—Given a graph query Q posed on a knowledge graph G, top-k graph querying is to find k matches in G with the highest ranking score according to a ranking function. Fast top-k search in knowledge graphs is challenging as both graph traversal and similarity search are expensive. Conventional top-k graph search is typically based on threshold algorithm (TA), which can no long fit the demand in the new setting.

This work proposes STAR, a top-k knowledge graph search framework. It has two components: (a) a fast top-k algorithm for star queries, and (b) an assembling algorithm for general graph queries. The assembling algorithm uses star query as a building block and iteratively sweeps the star match lists with a dynamically adjusted bound. For top-k star graph query where an edge can be matched to a path with bounded length d, we develop a message passing algorithm, achieving time complexity $O(d^2|E| + m^d)$ and space complexity linear to $d|V|$ (assuming the size of Q and k is bounded by a constant), where m is the maximum node degree in G. STAR can further be leveraged to answer general graph queries by decomposing a query to multiple star queries and joining their results later. Learning-based techniques to optimize query decomposition are also developed. We experimentally verify that STAR is 5-10 times faster than the state-of-the-art TA-style graph search algorithm, and 10-100 times faster than a belief propagation approach.

I. INTRODUCTION

Querying knowledge graphs is a challenging task. Due to their complex schemas and varying information descriptions, it is hard for users to formulate structured queries in SQL and SPARQL. Instead, user-friendly query forms such as keyword query, natural language query, exemplary query, and graph query are more preferred. These query forms are related to each other. For example, one can parse a natural language question to a dependency graph, which can later be converted to a graph query [1]. In this work, we target graph query and use it as a vehicle to connect to other query forms.

The query task is as follows [1]–[3]: *Given a knowledge graph G, a scoring function F, and a graph query Q, top-k subgraph search over G returns k answers with the highest matching scores.*

A common practice in top-k graph search [4]–[10] follows conventional top-k aggregation methods over relational databases, *e.g.,* threshold algorithm [11], to find top matches by traversing sorted node/edge lists and checking if there are good matches. Nevertheless, knowledge graph search often requires approximate matches in terms of content and structure. This new requirement, together with the sheer size of knowledge graphs, introduces new challenges and opportunities.

(1) For each query, the matching scores of potential answers are computed *online*. While it is possible to index keywords in nodes and edges, it is too expensive to build indices for complicated aggregation function like the one used in [2] (see Eq. 2 in Section II). Using TA algorithms on sorted node/edge lists is not going to quickly prune the search space as nodes/edges with top scores might not be connected like a subgraph similar to the query.

(2) Queries typically have *inexact* matches: A query edge could have valid matches with paths of bounded length. Finding such inexact matches is costly over big graphs. While indices can be constructed to speedup searching, it often comes with expensive preprocessing, *e.g.,* $O(|V|^3)$ for computing transitive closure [4], [5]. This is no longer practical for big graphs.

(3) Query graphs are usually not big. As observed in [12], most real-world SPARQL queries in RDF stores such as DBpedia are *star-like*. It is more urgent to build a fast query engine for simple structures first.
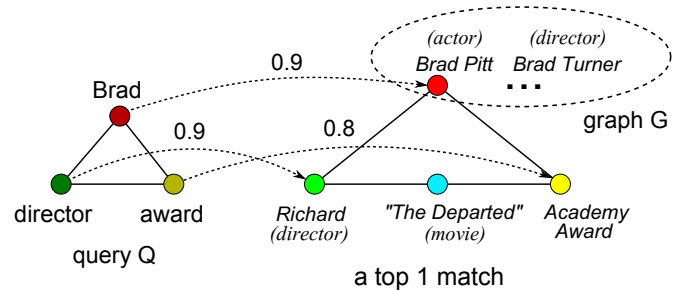

Fig. 1: Top-k subgraph querying

Example 1: Consider graph query Q on a movie knowledge graph, shown in Figure 1. It searches for movie makers who worked with "Brad" and have won awards.

It is nontrivial to find the top answers. Each query node and edge may correspond to an excessive number of possible matches. For example, a node Brad may have matches with any person whose first or last name is Brad. An edge (movie maker, award) may match a path through an intermediate node movie. It is not efficient to enumerate all the possible matches and then ranking them.   □

To the best of our knowledge, we are among the first to recognize the need of performing top-k search using sophisticated ranking functions in large knowledge graphs. This paper proposes STAR, a top-k graph query engine that copes with the new challenges. In a nutshell, it develops search algorithms for star-shaped queries and makes use of them to solve general graph queries. We summarize our contribution as follows.

(1) Given a star-shaped query Q^* and data graph G, STAR finds top-1 matches for the center query node (called *pivot node*) of Q^* in G and expands new matches from there. For each node matching the pivot node, it is able to generate the best matches of Q^* in decreasing order of matching score. We

show that in order to find top k results of Q^* pivoted at $v \in V$, we need not find top k matches for individual leaf nodes in Q^*. The algorithm takes $O(|E|)$ time and space, assuming k and the size of Q^* are bounded by a constant.

(2) We further modify the algorithm to support inexact matching where an edge can be matched to a path with bounded length d. For each node/edge in data graph, it propagates matching scores to their neighbors, retains the maximal matching score and then propagates further. Since it does not compute transitive closure, the time complexity is reduced to $O(d^2|E| + m^d)$ and the space complexity is linear to $d|V|$, where m is the maximum node degree of G.

(3) Given a general (cyclic) graph query Q, STAR decomposes it to a set of star-queries, and assembles top answers from individual star queries. Decomposing a graph query into substructures for fast query processing is not new (see e.g., [13]). Our novelty is the basic substructure we use here, notably, *stars*. STAR generates top answers of star-queries in monotonic decreasing order of matching score, the answer set is equivalent to a pre-sorted list. This nice property makes it possible to apply monotonic ranked joins such as [5] to produce the final top k answers for Q without losing completeness.

(4) Since a query Q can be decomposed to multiple star queries in different manners, we identify new query optimization opportunities in our problem setting, experiment a few designs and also demonstrate their effectiveness.

(5) We evaluate the runtime performance of our algorithms. In comparison with a highly-optimized threshold-based algorithm (TA) and a belief propagation method (BP) employed in [2], [14], it was found that STAR is 5-10 times faster than TA and 10-100 times faster than a recently proposed brief propagation method (BP) for top k graph search [2].

We conclude that optimizing star query processing not only solves the most popular queries in knowledge graphs, but also contributes as a building block of efficient algorithms for answering general graph queries. By effective query decomposition and star query processing, top k answers can be found in a much faster manner.

II. PRELIMINARIES

We start with notions and problem formulation.

Knowledge graphs. We consider *knowledge graph G* as a labeled graph $(V, E, \mathcal{L})$, with node set V and edge set E. Each node $v \in V$ (edge $e \in E$) has a description $\mathcal{L}(v)$ ($\mathcal{L}(e)$) that specifies node (edge) information, and each edge represents a relationship between two nodes. $\mathcal{L}$ could be structured with a schema, *e.g.,* in XML, RDF, and Freebase, not structured, *e.g.,* keywords only,, or with mixed structure, *e.g.,* DBpedia. $\mathcal{L}$ may also include heterogeneous attributes, entities and relations of various types [15].

Queries. We consider query Q as a graph (V_Q, E_Q). Each *query node* in Q provides information/constraints about an entity, and an edge between two nodes specifies the relationship or the connectivity constraint posed on the two nodes. Specifically, we use Q^* to denote star-shaped query.

Example 2: Figure 1 illustrates querying without node schema. The query Q contains nodes as simple keywords, *e.g.,* Brad, to describe the entities it refers to. For each node in the knowledge graph G, a node description $\mathcal{L}$ may specify a type (*e.g.,* actor) and an entity name (*e.g.,* Brad Pitt), or simply a keyword (*e.g.,* Academy Award). Note that $\mathcal{L}$ may also pertain to specified schema, where each node has uniformed attributes, and attribute values in accordance. □

Subgraph Matching. Given a graph query Q and a knowledge graph G, a match $\phi(Q)$ of Q in G is a subgraph of G, specified by a one-to-one matching function ϕ. It maps each node u (resp. edge $e=(u', v)$) in Q to a node match $\phi(u)$ (resp. edge match $\phi(e)=(\phi(u), \phi(u'))$) in $\phi(Q)$. In Section V, we will relax the edge mapping to path mapping to support approximate matching.

Yang et al. [2] adopt a probabilistic approach to learn a sophisticated ranking function. When mapping a query node/edge to a data node/edge, it supports various kinds of transformations such as synonym, abbreviation, and ontology. For example, "teacher" can be matched with "educator," and "J.J. Abrams" with "Jeffrey Jacob Abrams." Each match produces a similarity score. This allows ordinary users to post queries without spending hours or even days to digest the vocabulary and complex schema specified in a knowledge graph. A node matching cost function $F_V(v, \phi(v))$ aggregates the contribution of all the possible similarity measures $\{f_i\}$ with weight $\{\alpha_i\}$,

$$F_V(v, \phi(v)) = \sum_i \alpha_i f_i(v, \phi(v)), \tag{1}$$

where f_i refers to the matching score under the i_{th} similarity measure. Analogously, an edge matching cost function, $F_E(e, \phi(e))$ can be defined.

All the node and edge scores are then combined together to produce a final score between query Q and its match $\phi(Q)$. The matching score is computed by a function $F(\phi(Q))$ as

$$F(\phi(Q)) = \sum_{v \in V_Q} F_V(v, \phi(v)) + \sum_{e \in E_Q} F_E(e, \phi(e)) \tag{2}$$

When it is not ambiguous, we write $F_V(v, \phi(v))$ as $F(\phi(v))$, $F_E(e, \phi(e))$ as $F(\phi(e))$ respectively. In practice, we usually have an additional constraint: $F_V(v, \phi(v))$ and $F_E(e, \phi(e))$ have to be greater than a threshold, which assures each node and edge have a good match in an answer.

Top-k subgraph querying. Given Q, G, and $F(\cdot)$, the top-k subgraph querying is to find a set of k matches $Q(G, k)$, such that for any match $\phi(Q) \notin Q(G, k)$, for all $\phi'(Q) \in Q(G, k)$, $F(\phi'(Q)) \geq F(\phi(Q))$.

Example 3: For Q and G in Figure 1, a match $\phi(Q)$ consists of node matches Brad Pitt, Richard and Academy Award, where the function ϕ maps Brad to Brad Pitt with node score 0.9 (as shown in Figure 1). Let the three edge scores be 1.0, 1.0 and 0.8 , then the total score $F(\phi(Q))$ is 5.4, the sum of node and edge scores. Note that an edge (movie maker, award) in Q is matched with a path from Richard (a director) to Academy Award in G. □

Procedure graphTA

Input: a subgraph query Q, a knowledge graph G, integer k.
Output: top-k match set $Q(G, k)$.

1. initialize candidate list L for each node and edge in Q;
2. **for each** list L
3.   sort L following a ranking function;
4. Set a cursor to each list; set an upper bound U;
5. **for each** cursor c in each list L **do**
6.   generate a match that contains c; update $Q(G, k)$;
7.   update a threshold θ with the lowest score in $Q(G, k)$;
8.   move all cursors one step ahead;
9.   update the upper bound U;
10.   **if** k matches are identified and $\theta \geq U$ **then break** ;
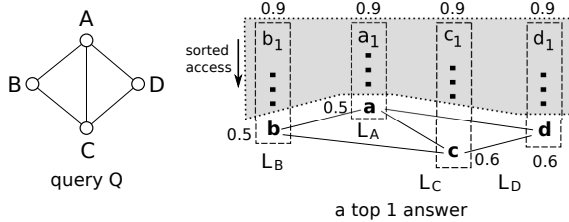11. **return** $Q(G, k)$;

Fig. 2: Procedure graphTA

Fig. 3: An example of wasted enumerations

Framework STAR

Input: a graph query Q, a knowledge graph G, integer k.
Output: top-k match set $Q(G, k)$.

1. decompose Q to a star query set $\mathcal{Q}$;
2. **while** top-k matches are not identified **do**
3.   invoke stark or stard to retrieve new top matches
4.   for queries in $\mathcal{Q}$;
5.   invoke starjoin to assemble new matches;
6.   update $Q(G, k)$;
7. **return** $Q(G, k)$;

Fig. 4: Framework STAR

the next match from each list. If U is smaller than the current lower bound θ, graphTA terminates.

Limitation of graphTA. We use an example to demonstrate the limitations of directly applying TA-style top-k algorithm. Consider a subgraph query Q and its top-1 answer in Figure 3. We observe the following limitations.

(1) Matches for nodes and edges with high matching score alone do not necessarily indicate top answers. For example, the top-1 answer is joined from a set of node and edge matches with quite low matching scores, if ranked independently (Figure 3). Sorted accessing over single node and edge match lists, as in graphTA, leads to an excessive amount of useless visits and enumeration of partial matches.

(2) To explore single node/edge match in a large graph often leads to expensive match expansion, resulting in significant performance degradation. For example, each time a new node match is visited, expanding from single node match requires a subgraph isomorphism search [1].

(3) It is often hard to estimate a tight enough upper bound, by using the node or edge matches alone. For example, if one follows sorted access to L_B to b, while all other cursors are at the top of L_A, L_C and L_D, respectively, the current upper bound, determined by 0.5, 0.9, 0.9 and 0.9, can be far from the "real" upper bound determined by 0.5, 0.5, 0.6 and 0.6. Indeed, the upper bound in conventional TA algorithm is designed for joining attribute values, where no topological linkage is enforced. This typically generates quite loose upper bound that reduces the possibility of early termination.

IV. STAR-BASED TOP-K MATCHING

While a straightforward application of TA has the limitations in subgraph querying, we next outline a framework to mitigate it by utilizing larger structures as building blocks. The idea is to find maximal subqueries for which (a) top-k matches can be quickly retrieved without any TA-based joins, and (b) the matches of subqueries can be effectively assembled for the top-k complete matches. We identified *star shaped queries* as such structures. This framework kills two birds with one stone. First, it is observed that most of real-life subgraph queries on knowledge graphs are "star-like" queries [3], [12]. To deriving a fast solution for star queries is very appealing. Second, as a basic building block, it will lead to efficient top-k search for complex graph queries.

The top-k querying framework, denoted as STAR and illustrated in Figure 4, has the following steps.

III. THRESHOLD ALGORITHM

We first introduce a top-k graph querying procedure based on threshold algorithm (TA) [11]. The procedure is adopted in several state-of-the-art graph pattern matching methods [1], [16]. We analyze the limitation of this approach.

The threshold algorithm [11] finds the top-k best tuples from a relational table by optimizing a monotonic aggregation function. The common practice in existing top-k subgraph matching is to treat each query node and edge as an attribute, with its matching score as an attribute value. If a set of matches can be joined to form a complete match, they are selected to compute a threshold. An upper bound of the matching score is estimated for the rest "unseen" matches. Following [11], top-k matches are identified when the upper bound is smaller than the threshold. The procedure and its variants are invoked in a range of existing subgraph matching methods [1], [6], [8], [9].

A TA-algorithm for subgraph matching. We outline the procedure, denoted as graphTA, in Figure 2. The procedure typically follows three steps. (1) It initializes a candidate list L for each query node and edge. (2) It then sorts each list following a certain ranking function. For each sorted list, a cursor is assigned at the head of the list. (3) graphTA iteratively starts an exploration based subgraph isomorphism search to expand the node match pointed by each cursor, until a complete match is identified. It moves all cursors one step forward. The above step repeats until k matches are identified and it is impossible to generate better matches, or no match can be generated from the lists.

To achieve early termination, graphTA dynamically maintains (1) a lower bound θ as the smallest top-k match score so far, and (2) an upper bound U to estimate the largest possible score of a complete match from unseen matches. For example, an upper bound can be established by aggregating the score of

Procedure stark
_____

Input: a star query Q^*, a knowledge graph G, integer k.
Output: top-k match set R.

1.   initializes set $R=\varnothing$;
2.   initializes priority queue $P=\varnothing$;
3.   identify candidate node matches V' for the pivot node in Q^*;
4.   find top-1 match pivoted at each node v in V';
5.   add best k matches among the top-1 matches to P;
6.   **while** $|R| < k$ **do**
7.   pop the best match M (pivoted at v) from P; $R = R \cup \{M\}$;
8.   generate the next best match M' pivoted at v;
9.   insert M' to P;
10.   **return** R as $Q^*(G,k)$;
_____

Fig. 5: Procedure stark

(1) *Query decomposition.* Given a query Q, STAR invokes a procedure to decompose Q to a set of star queries $\mathcal{Q}$ (Section VI-B). A star query contains a pivot node and a set of leaves as its neighbors in Q. After query decomposition, $\mathcal{Q}$ is sent to the star querying engine.

(2) *Star querying engine.* Using $\mathcal{Q}$ generated in (1) as input, It invokes a procedure, called stark (resp. stard for approximate matching), to efficiently generate a set of top matches for each star query in $\mathcal{Q}$ (Section V). The procedure stark guarantees that the matches are generated progressively in the descending order of the match score for each star query.

(3) *Top-k rank join.* The top matches produced by stark (or stard) for multiple star queries are then collected and joined by a procedure starjoin, to produce top-k complete matches of Q (Section VI). It terminates once the top-k matches are identified, or there is no chance to generate better matches.

V. Top-k Star Query

We first examine star query evaluation. For simplicity's sake, we only use node matches to describe our algorithms.

Top-k tree pattern matching has been extensively studied, *e.g.,* [4], [5] and its newest improvement [17]. While the design of these algorithms can be reused for star query, there are two new problems: (1) Most of these studies assume there is a pre-sorted node (resp. edge) match list with respect to query node (resp. edge), which is not true in our problem setting: $F_V(v, \phi(v))$ and $F_E(e, \phi(e))$ are computed online; (2) For edge-to-path approximate graph matching, the existing studies typically require the construction of transitive closure, which is infeasible over large graphs.

A. Top-k Search

Algorithm. Our star query processing algorithm stark (illustrated in Figure 5) takes the following steps. (1) It first identifies the candidate node match in graph G for the pivot query node (line 3). For each candidate v, it then identifies top-1 match pivoted at v (line 4), by finding the best matches for the leaf nodes of Q^* in v's neighbors and assembling them as the top-1 match pivoted at v (not shown). Among these matches of Q^*, it selects top k matches to form a pseudo top-k set P (lines 5). (2) It pops up the best match M from P, insert it into an answer set R. For the pivot node in M, it generates the next best match M' and inserts it to P (lines 7-9). (3) The procedure stark repeats step (2) until $|R| = k$ (lines 6-10).
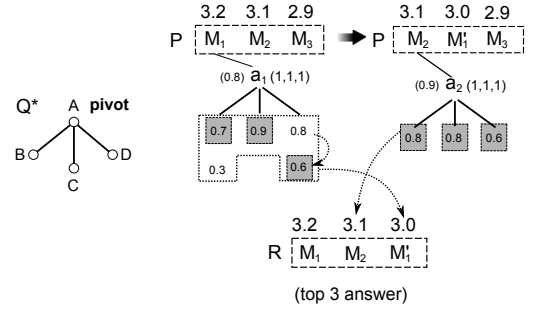

Fig. 6: Top-k star querying

In the first step, stark identifies the pseudo top-k set P that contains all the potential pivot nodes in the real top k answers. We use the following notions. (1) Given a node $v \in V$, a match M in G is *pivoted at v* if v is matched to the pivot node in Q^*. (2) Given G, Q^*, F, and k, let $\mathcal{M}$ be the set of top-1 matches of Q^* pivoted at each node in G. The *pivot node set*, V_p, refers to the set of nodes v, where a match M is among the top k matches in $\mathcal{M}$ and is pivoted at v.

Lemma 1: *Top-k matches of Q^* can only come from the matches pivoted at v, $v \in V_p$.* □

For the complexity of step (1), we shall use a well-known result for finding top k numbers in an unsorted list.

Lemma 2: *[18] Given a set of n numbers and an integer k, it takes $O(n)$ time to find top k number, and $O(n + k \log k)$ time to find sorted top k numbers.* □

The procedure stark takes $O(|V|)$ time to find the potential node candidate match of the pivot node of Q^* (line 3). This can be further optimized with various indices, as discussed in [2]. stark next finds a top-1 match pivoted at v. It scans all v's neighbor nodes, thus taking $O(|E||V^*|)$ time (line 4). It next finds the k best matches among these matches. This can be done in $O(|V|)$ time (Lemma 2). Therefore, the first step of stark takes $O(|E||V^*|)$ time.

Example 4: We demonstrate how stark computes top-3 matches for the star query Q^* in Figure 6. It first finds the top-1 matches for each match of A, and then selects the best 3 of the matches M_1, M_2 and M_3, pivoted at *e.g.,* a_1, a_2 and a_3 (a_3 is omitted from the figure), respectively. The procedure stark adds all the three matches to a priority queue P. The minimum value in the queue is 2.9, which is the score of the top-1 match for pivot a_3. It next pops from P the best match M_1, which is the top-1 match pivoted at a_1, with cursor index $(1,1,1)$. M_1 is then inserted into R. Based on M_1, stark fetches the next best match from the union of the three lists L_B, L_C, and L_D calculated from the neighbors of a_1. This gives one new match M_1' pivoted at a_1, with match score 3.0. It pushes M' to P. In this case, it is not going to search node a_3 any more. The searching continues over each match in P, until $|R|$=3. When stark terminates, R is returned as $\{M_1, M_2, M_1'\}$, with scores 3.2, 3.1, and 3.0, respectively. □

In the second step, stark retrieves the top-1 match M from P and its pivot node v (line 7). It then fetches the next best match pivoted at v (line 8). As v's neighbors are *not* sorted with respect to their similarity to the leaf nodes in Q^*, one needs to scan the entire neighbor set to find the second largest
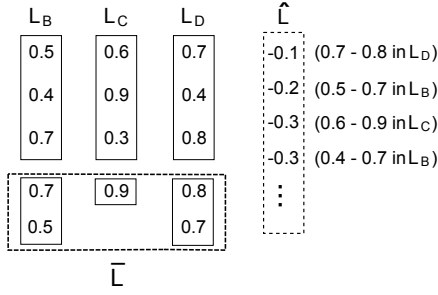
Fig. 7: Optimal match selection

value w.r.t each query leaf node. This takes $O(m|V^*|)$ time, where m refers to the largest node degree in G. The issue becomes more severe if all of the remaining top-k matches actually come from v. The cost will grow to $O(k * m|V^*|)$, as step (2) will run k-1 times on v. Instead, stark finds and sorts top-k node matches w.r.t each query leaf node, which is in $O((m + k \log k)|V^*|)$ time. Better still, the following result shows that the cost can be reduced to $O(m|V^*| + k \log k)$, which is optimal in the worst case.

Proposition 3: *Given s lists of unsorted numbers $L_1, L_2, \ldots, L_s$, each of size m, and an aggregation function,*

$$F = \sum_{i=1}^{s} x_i, x_i \in L_i,$$

(1) there exists a set $\bar{L} \subseteq L = \bigcup_i L_i$, $|\bar{L}| \leq k + s - 1$, s.t., any number in $L \setminus \bar{L}$ does not contribute to the top-k values of F; (2) It takes $O(sm)$ time to find $\bar{L}$. $\square$

Proof sketch: We first construct $\bar{L}$. Denote the largest number in L_i as x_i^{max}, $\hat{L}_i = \{x - x_i^{max}|x \in L_i\}$, $L^{max} = \{x_i^{max}\}$ and $\hat{L} = \bigcup_i \hat{L}_i$. Let $\bar{L} = \{x \in L \setminus L^{max}|x - x_i^{max}$ ranks top-k+s-1 in $\hat{L}\}$. We then prove the theorem by contradiction: Suppose $x' \in L \setminus (\bar{L} \cup L^{max})$ contributes to one of the top-k sums, denoted as F'. It is easy to see $F' \leq x' + \sum_{j \neq 1} x_j^{max} \leq x' - x_1^{max} + \sum_j x_j^{max}$, where w.l.o.g. $x' \in L_1$. However, besides the s largest numbers $\{x_i^{max}\}$, there are at least $k - 1$ numbers $x_i \in \bar{L}$, such that $x' - x_1^{max} \leq x_i - x_i^{max}$. Thus F' is not among the top-k sums since there are at least k sums no less than F'. $\square$

Prop. 3 shows that in order to find top-k w.r.t the score function F, we need not find top-k numbers for each list L_i. Instead, with a minor modification, we only need to find $k + s - 1$ numbers in the union of the lists.

Example 5: Consider three lists L_B, L_C and L_D in Figure 7, with largest number x_B^{max}=0.9, x_C^{max}=0.7 and x_D^{max}=0.8, respectively. To find top-3 values of the function F, one only needs to find the largest numbers of each list and additional 2 numbers from the combined list. To this end, a set $\hat{L}$ is constructed by subtracting the largest number in each list, *e.g.,* 0.8 in L_D, from each other numbers in the list, *e.g.,* 0.7. The set $\bar{L}$ is then constructed by including the three largest numbers from each list, and two numbers 0.7 and 0.5, corresponding to top ranked numbers -0.1 and -0.2 in $\hat{L}$, respectively. $\square$

Following Prop. 3, we only need to retain at most $s+k-1$ numbers from $\bigcup_i L_i$ to fetch the next best match.

The remaining step of stark follows the lattice search [4]. It maintains a priority queue (with size at most k) to bookkeep the top k matches. For each match, stark records its pivot node and a cursor to remember the index of sorted lists. It takes $O(k \log k)$ to put the matches in the candidate pool P (line 7) into the queue. When it pops up the current best match from the the queue (line 8), it retrieves the cursor. Let $s = |V^*| - 1$. Assume the cursor index is $(l_1, l_2, \ldots l_s)$, it calculates the F value for $(l_1+1, l_2, \ldots l_s)$, $(l_1, l_2+1, \ldots l_s)$, $\ldots$, $(l_1, l_2, \ldots l_s+1)$. In total, there are total s matches, which shall be pushed into the queue if they are greater than the minimum value in the queue. The time cost is $s \log k$. There is an evidence to further improve it using the lattice structure shown in [4].

Putting the above analysis together, Step 2 takes $O(m|V^*| + k \log k + |V^*| \log k)$ time in total $k - 1$ iterations. As k is a small constant, the time complexity is dominated by $O(m|V^*|)$. Here m is the largest node degree.

Analysis. As stark always selects the top k matches from the matches pivoted at the nodes in pivot node set V_p, it correctly identifies top k matches for Q^* (Lemma 1).

For the time complexity, Step 1 takes $O(|V| + |E||V^*|)$ time to find best k top-1 matches. Step 2 takes $O(mk|V^*| + k^2 \log k + |V^*|k \log k)$, in total. Assuming Q^* and k bounded by a small constant, stark is linear in terms of $O(|E|)$. In practice, not every node in G will be matched with the query pivot node. A cutoff threshold will be applied to retain a few candidate nodes. Let n' be the size of candidate nodes. In this case, Step 1 takes $O(n'm|V^*|)$, dominating the cost. When n' is very large, the aggregation overlay graph and a "push" strategy from [19] could be applied to sharing computation.

B. d-bounded Star Query

d-**bounded subgraph matching**. As illustrated earlier, an edge may be matched with a path of bounded length in querying knowledge graphs. Given G, Q and an integer d, a d-bounded subgraph querying extends subgraph querying with a matching function ϕ_d, such that each edge $e = (u, u')$ can be mapped to a path $\phi_d(e)$, connecting two node matches $\phi(u)$ and $\phi(u')$ with length bounded by d. The conventional subgraph isomorphism is a special case of d-bounded subgraph querying.

Edge-Path Similarity Function. When an edge e in a star query is matched to a path $\phi_d(e)$ in G, we need to define a similarity function $F(e, \phi_d(e))$. The algorithm proposed in this section is valid as long as $F(e, \phi_h(e))$ is monotonically decreasing in terms of d. A typical example is $F(e, \phi_d(e)) = \lambda^{(h-1)}$, $\lambda \in (0, 1]$, where h is the length of path $\phi_d(e)$.

The major challenge of answering d-bounded star queries is twofold. (1) Traversing d-hop neighbors of pivot nodes incurs excessive cost. Moreover, pre-computing d-hop neighborhood indexes for each node is no longer practical for large graphs. (2) The bottleneck is to find top-1 matches from a potentially large number of matches to identify the pivot node set V_p. Following Lemma 1, the next step is to find top-k matches pivoted at nodes in V_p, where traversing is typically more affordable for typically small k.

We present the main result for d-bounded star queries.

Theorem 4: *Given G, Q^*, k, and d, when $|Q^*|$ and k are bounded by a small constant, there exists an algorithm that*
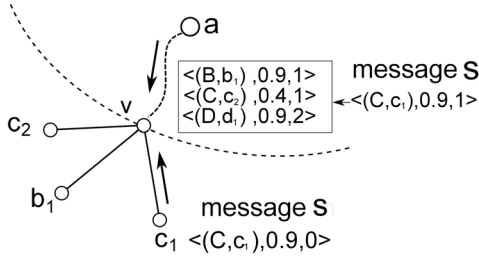
Fig. 8: d-bounded star querying

computes top k answers in $O(d^2|E| + m^d)$ time, where m refers to the largest node degree in G. $\square$

We next introduce a message propagation algorithm, denoted as stard (star-d), that achieves the above time complexity. In a nutshell, it leverages message passing to exchange the maximal node and edge matching scores.

Message propagation. The algorithm stard identifies all the node matches v for each query leaf node u^* in Q^*. Instead of "pulling" the neighbors' score for each potential pivot node match in G, it collects, aggregates and propagates messages encoding the matching score of each node in G to its 1-hop neighbors and repeats d times.

Message. The algorithm stard encodes a message s as a set of triples $<(u^*, v), F, h>$. If a node v' in G receives such a message, it means in h hop of v' in G, there is a node v matched to u^* with score F.

Example 6: Consider query Q^* in Figure 8. A message s is initialized at c_1 as $<(C, c_1), 0.9, 0>$, indicating that c_1 is matched to query node C, with node score 0.9, and the message resides at node c_1 (with hop number 0). $\square$

Message passing. stard initializes a message s that contains a single triple $<(u^*, v), F(u^*, v), h = 0>$ at each match v of u^*. It then propagates s (by forking it to multiple copies and distributing all the copies to the nodes at the same time) from v towards its 1-hop neighbors. For each node v that receives a message $s_1 = <(u^*, v_1), F_1, h_1>$, it increases h_1 by 1 and then performs the following aggregation task:

1) If v has no local copy of any message containing u^*, it keeps a copy of s_1.
2) If v has a local copy of a message, $s_2 = <(u^*, v_2), F_2, h_2>$. If $F_1 \leq F_2$ and $h_1 \geq h_2$, discard s_1; Otherwise keep both s_1 and s_2.

Intuitively, stard always keeps track of the node match with a greater "potential" to be the top-1 match, measured by the sum of its node score and "up to the moment" edge score $F(e)$ at the hth hop of propagation.

Example 7: Given query Q^* in Figure 8 for 3-bounded search, F_E is defined as 0.8^{h-1} for a path match of length h. stard iteratively propagates s from node c_1 to its neighbors. When s is propagated to node v, it finds a local copy of message s' with an entry $<(C, c_2), 0.4, 1>$, indicating that a match c_2 is 1 hop away from v. stard replaces the entry of c_2 with c_1, and continue propagation with s. $\square$

Algorithm. We outline the complete algorithm stard below. Given a d-bounded star query Q^*, it performs d rounds message propagation, from all the leaf node matches in G.

It then selects V_p along the same line as stark. For each node match in V_p, it performs a traversal to collect the distance and score information to compute top-k matches, similar to stark.

Ping-Pong effect. It is possible that a node v could have a similarity score with both the pivot node and a leaf node u^* in Q. When a message initiated at v for matching u^* is passed around, it may arrive at v again. When v is matched to the pivot node, it might lose the trace of any other node that could be matched to u^*. In this case, we can not derive the top-1 match pivoted at v correctly. One way to solve this issue is to record two best matches for u^* and pass them around. This will guarantee at least one match can be used later.

Analysis. When the message propagation terminates, the algorithm stard correctly computes top-k matches, following stark. Hence it suffices to show that all the top-1 matches are correctly gathered and computed. Indeed, stard keeps the invariant below: (1) at any time, the message s which carries the information of top-1 node and edge matches are not replaced by any other message; and (2) when the propagation terminates, all the s in (1) are guaranteed to be fetched. The correctness of stard hence follows.

The main time cost of stard is dominated by the message passing. There are at most d rounds of message propagation for every node. For each node in G, we need to maintain at most $d|V^*|$ messages (ie., $h = 1, 2, \ldots, d$). Hence the total time is in $O(d^2|E||V^*|)$ for finding the pivot node set V_p. Once it is found, the time to find top-k matches is in $O(m^d|V^*| + k^2 \log k + |V^*|k \log k)$. When $|Q^*|$ and k are bounded by a small constant, the total time complexity is $O(d^2|E| + m^d)$.

The above analysis completes the proof of Theorem 4.

Remark. The implementation of stard allows multi-level of parallelism. In an extreme case of vertex-centric programming [20], each node can exchange messages between their neighbors in parallel, which can complete all message propagation in at most d rounds of communication.

C. Alternatives

The algorithms graphTA, stark, and stard adapt different search strategies to find top-k answers for star-shaped queries. It is possible to factorize them to a few components and recombine them to generate new alternatives. In the following discussion, we present one of them. Empirical examination of these alternatives will be left to future study.

We next outline an approach that combines the search strategies of graphTA, stark, and stard. The algorithm searches pivot and leaf node lists alternatively. In Stage 1, (1) Sort $L_A, L_B, L_C, \ldots$ in decreasing order of node score. (2) Fetch nodes from $L_A, L_B, L_C, \ldots$ alternatively in a top-down manner. For a node fetched from L_A, the algorithm searches its neighbors for its top-1 match. Update the pseudo top-k set. For a node fetched from $L_B, L_C, \ldots$, the algorithm propagates its score to its neighbors. Update the upper bound of the best answer that can be found at each neighbor. Whenever finding a new top-1 match, update the pseudo top-k set. (3) When the upper bound of the remaining unseen answers is smaller than the pseudo top-k scores found so far, stop searching. In Stage 2, follow the remaining steps in stark by searching other good matches for each pivot node in the pseudo top-k set, and generate the final top-k matches.

VI. TOP-K STAR JOIN

The star query framework stark can not only process star queries quickly, but also serve as a foundation to answer general graph queries. A graph query Q can be decomposed to a set of star-shaped queries $\{Q^*\}$. Top-k answers to Q can be assembled by collecting the top matches of each Q^*, followed by a multi-way join process.

There is a great advantage of leveraging star queries. First, stark is able to quickly generate matches in a *monotonic decreasing order* of the matching score. As manifested in Section VI-A, this property is critical when joining multiple subqueries: It produces an upper bound for those matches that have not been seen yet. Second, although a similar method [5] exists for other basic structures like edges, a "bigger" structure like star-shaped subqueries can reduce the number of joins, thus improving query processing time.

We address two challenges in this section:

1) Query decomposition. Consider different query decomposition strategies and determine an efficient way to execute a query.
2) Top-k ranked joins. Efficiently construct a complete match from star matches and derive an upper-bound for the remaining possible matches.

We first study the top-k ranked join problem (Section VI-A). We then develop the intuition from the join problem for query optimization in Section VI-B.

A. Top-k Star Rank Join

Given a query Q decomposed to a set of star queries $\mathcal{Q} = \{Q_1^*, Q_2^*, \dots Q_m^*\}$, the *rank join* is to find the top-k matches for Q by assembling the matches retrieved by stark on each Q_i^*. This is outlined as starjoin in Figure 9.

starjoin performs in a similar way as the hash rank join strategy (*HRJN* [21]). It iteratively fetches k matches for each star and joins them with the existing matches for the other stars (line 5 and 6). In order to compute the joins, a hash table for each L_i maintains the mapping of the joint nodes to the matches seen so far. starjoin keeps track of lower bound θ as the k-th match in the priority queue R (line 7). It can be seen that the efficiency of the algorithm relies on the upper bound θ_i for each star (line 8 and 9).

Procedure starjoin

Input: $\mathcal{Q} = \{Q_1^*, Q_2^*, \dots Q_m^*\}$.
Output: top-k join matches.

1. initialize a priority queue $R=\varnothing$; set $\theta = -\infty$;
2. initialize the match list L_i for each $Q_i^* \in \mathcal{Q}$;
3. **while** $\mathcal{Q} \neq \varnothing$ **do**
4.   **for each** $Q_i^* \in \mathcal{Q}$ **do**
5.   invoke stark on Q_i^* to find the next match M;
6.   join M with L_j $(j \neq i)$ and add the join results to R;
7.   update θ as the k-th score in R if $|R| \geq k$;
8.   compute upper bound θ_i based on M;
9.   add M to L_i; remove Q_i^* from $\mathcal{Q}$ if $\theta_i < \theta$;
10. **return** the first k results in R;

Fig. 9: Procedure starjoin

Upper bound [21]. Consider m match lists $\{L_1, \dots, L_m\}$. For a list L_i of size n_i, denote ϕ_{ij} as the jth ranked match in L_i. The upper bound θ_i is defined as

$$\theta_i = F(\phi_{in_i}) + \sum_{j=1, j\neq i}^{m} F(\phi_{j1}). \qquad (3)$$

Intuitively, an upper bound is estimated as the sum of the scores from the last match in one list and the top-1 matches all the others.

The *HRJN* strategy was widely adopted in RDBMS and demonstrated the superior performance over the traditional join-then-sort approach [21]. However, there is a difference between *HRJN* and starjoin. Directly applying θ_i as Eq. 2 results in an invalid upper bound, as the scores for the joint nodes shared by several stars are counted multiple times. This can be seen as the example shown in Figure 10(a). Given a query Q and the score function F, let $(A = a_n, U = u_n)$ and $(B = b_1, U = u_1)$ be the n-th match and the first match in L_1 and L_2, respectively. According to Eq. 2, $\theta_1 = F(a_n)+F(u_n)+F(u_1)+F(b_1)$, which cannot be considered as the upper bound and directly compared with the lower bound θ for the top-k join results. To overcome this problem, we introduce the starjoin with the α-*scheme*.

Rank Join with α-scheme. Let U be the set of the joint nodes for two stars Q_1^* and Q_2^*, and A (resp. B) is the set of nodes that appear only in query Q_1^* (resp. Q_2^*). Then based on a parameter α, we introduce a new ranking function scheme, denoted as $F'(\phi(Q_1^*)) = F(\phi(A)) + \alpha \cdot F(\phi(U))$ for Q_1^* and $F'(\phi(Q_2^*)) = F(\phi(B)) + (1 - \alpha) \cdot F(\phi(U))$ for Q_2^*. Accordingly, given the two match lists, L_1 for Q_1^* and L_2 for Q_2^*, the upper bound can be refined as

$$\theta_1' = F'(\phi_{1n_1}) + F'(\phi_{21}), \theta_2' = F'(\phi_{11}) + F'(\phi_{2n_2}), \quad (4)$$

where ϕ_{1n_1} and ϕ_{21} are the last match and top match in L_1 and L_2, ϕ_{11} and ϕ_{2n_2} are the top match and last match in L_1 and L_2, respectively. When $\alpha \in [0, 1]$, one may verify that θ_1' and θ_2' are valid upper bound for the search on Q_1^* and Q_2^*, respectively. It is worth mentioning that the selection of α affects the number of matches to be fetched for assembling.

Example 8: Given query Q in Figure 10(a) that is decomposed to two stars Q_1^* and Q_2^*. Denote $a_i(j)$ in the figure as the i-th largest entry in the match list for A with the match score j. For example, $a_2(0.9)$ (L_1, Figure 10(c)) refers to the match a_2 for A with score 0.9 and $u_1(0.5)$ refers to the match u_1 for U with score $1.0 * \alpha = 0.5$. To identify the top-4 join matches as in Figure 10(b), it only needs to reach the top-3 matches in L_1 and L_2 with $\alpha = 0.5$. While for $\alpha = 0.9$, at least top-3 and top-11 matches in L_1 and L_2, respectively, are required. $\square$

The effectiveness of starjoin can be evaluated by the *total search depth*, $D = \sum_i |L_i|$, when the algorithm terminates. Example 8 implicates that when using a proper α, starjoin will likely require a smaller D to identify the top-k join matches, *e.g.,* $D = 6$ (resp. $D = 14$) when $\alpha = 0.5$ (resp. $\alpha = 0.9$) in the example. To determine an optimal α value for minimizing D, nevertheless, is not trivial. We introduce a principled way to determine α in Section VI-C.

The α scheme works for assembling two star matches, *i.e.,* two-way join. For multiple stars, we perform a sequence of two-way join (as a left-deep pipeline [21]) and apply the α scheme for each two-way join.
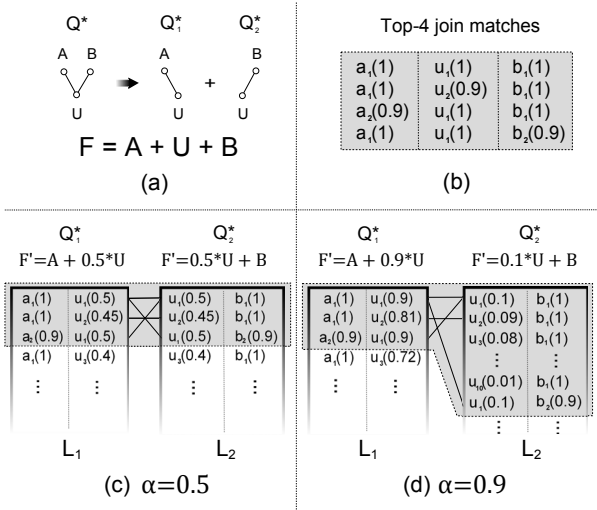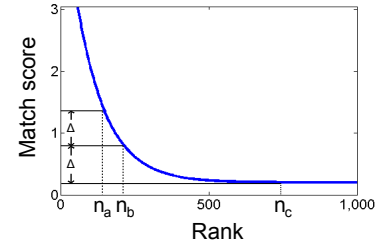
Fig. 10: Selection of α

Fig. 11: The distribution of the match score

where $\delta(Q_i^*)$ is the score decrement of the top matches in L_i, $f(Q_i^*)$ is the feature score of Q_i^* while $\bar{f}$ is their average, *i.e.*, $\frac{1}{m}\sum_{i=1}^{m} f(Q_i^*)$. Intuitively, it aims to maximize the score decrement and minimize the feature difference of the subqueries, where λ is a parameter to make a trade-off.

Since it is costly to accurately compute the score decrement δ and exhaust all the feature measurements, we consider several simple but effective features below:

SimSize: $f(Q_i^*) = |E_i^*|$: Star size.

SimTop: $f(Q_i^*) = F(\phi_{i1})$, where ϕ_{i1} is the top-1 match for Q_i^*. Unfortunately, ϕ_{i1} is difficult to observe without executing Q_i^*. Hence we use the top-1 pivot node match score to represent $F(\phi_{i1})$. In practice, we sample nodes in knowledge graphs and calculate the match score.

SimDec: $\delta(Q_i^*) = f(Q_i^*) = \frac{F(\phi_{i1})-F(\phi_{in_i})}{n_i}$, where n_i is the number of top matches checked for Q_i^*. SimDec measures the average match score decrement for Q_i^*. In practice, we approximate n_i by $p^{|V_i^*|-1}\prod_{v \in V_i^*} n_v$, where n_v is the number of node matches and p is the probability that two node matches are connected. p is a parameter estimated off-line by conducting a set of edge queries. n_v is estimated by sampling nodes in knowledge graphs calculating their match score with the pivot node of Q_i^*, and selecting relevant ones.

Query decomposition based on SimSize only considers query structures. Nevertheless, such problems (balanced edge partition) are in general hard (NP-hard) [22]. We employ the efficient greedy algorithm designed in [22] for SimSize. In practice, since most queries would not have many star sub-queries, we use dynamic programming to enumerate possible star decompositions starting with $m = 2$. For each m, the decomposition with the best score of Eq. 5 will be picked and returned immediately.

B. Query Decomposition

We next discuss the query decomposition problem, which has been studied for solving complex queries, *e.g.,* twig queries on XML data [5], [6] and SPARQL on RDFs [3]. However, the traditional techniques are not applicable in our problem setting since the match score has to be calculated online.

Given a query graph, we expect a decomposition to generate a set of star subqueries that minimize the total depth D. Since all match scores are generated on the fly, it is very challenging to analyze the search depth accurately. We investigate several heuristics and evaluate their performance on real-world graphs, based on the following observations.

(1) A reasonable decomposition derives as small number of stars as possible, which intuitively reduces the number of joins.

(2) To make the upper bound estimation tighter in Eq. 3 (Section VI-A), we shall make $F(\phi_{in_i})$ as small as possible. Therefore, a large score decrement for the matches in L_i will likely lead to small search depth.

(3) We observe that many real-world star queries share the similar distribution of the match scores with a long-tail effect, as illustrated in Figure 11. Given a query decomposed to several stars, the search for each star that stops at similar positions, say n_b, is likely to yield smaller D, in comparison with the case that one star search stops at n_a while the others stop at n_c with a much larger position gap. Based on these observations, the third intuition is to decompose a query to a few stars that have similar distribution of matching scores. While it is hard to derive the actual distribution, we approximately characterize it with similar size, similar top-1 match score or similar match score decrement.

Based on the above intuitions, given Q, the objective of the query decomposition is to derive a minimum number of stars with similar features, such that the score decrement of the matches for each star Q_i^* can be maximized. This can be described as an optimization problem,

$$\underset{\{Q_1^*,\ldots,Q_m^*\}}{\text{maximize}} \quad \sum_{i=1}^{m} \delta(Q_i^*) - \lambda \sum_{i=1}^{m} |f(Q_i^*) - \bar{f}| \quad (5)$$

$$\text{subject to} \quad \text{minimum m,} \quad (6)$$

C. Optimization: Determine the Parameters

The above top-k rank join technique has two parameters, α and λ, which can be learned off-line by a testing and validation method. Suppose we have a sample query workload W. Our top-k join algorithm is assumed as a black-box A with three input α, λ and W. The output of A is the aggregated total depth D for the queries in W. Let $\alpha \in [0, 1.0]$ and $\lambda \in [0, 2.0]$. By iteratively running A and setting a small constant *e.g.,* 0.1 as the adjustment step for α and λ, we can derive an optimal setting of α and λ that minimizes D. As verified in Section VII, with proper α and λ, our query optimization technique improves the runtime of baseline algorithms by 45%.

VII. EXPERIMENTAL EVALUATION

We conduct a set of experiments, using real-world knowledge graphs to examine the efficiency of STAR and its

components including the star query engine, stark/stard, and the top-k rank join, starjoin. The effectiveness of top-k search is referred to the previous work, e.g., [2]. We applied 46 similarity functions, covering acronym, synonym, abbreviation, ontology, unit conversion, frequency, TF/IDF, NLP parse tree distance, type, edit distance, path distance etc. The weights of these functions are learned through training [2].

A. Experimental Setup

Datasets. We use three real-world graphs. *DBpedia* [23] is a complex knowledge graph with each node representing a real entity and an edge indicating the relationship between two entities. *YAGO2* [24] is a knowledge base derived from multiple public data sources. *Freebase* [25] is designed as a collaboratively created large knowledge base. Note that these graphs are quite heterogeneous with many different types of nodes (e.g., 'place', 'people') associated with various kinds of links. Additionally, we retain the rich content information attached to each node and edge in the graphs. Thus given a labeled query, a large amount of match candidates might be acquired, with varying matching scores. The following table summarizes these graphs.

Graph	Nodes	Edges	Node types	Relations	Size
DBpedia	4.2M	133.4M	359	800	40G
YAGO2	2.9M	11M	6,543	349	18.5G
Freebase	40.3M	180M	10,110	9,101	88G

Query workload. Two sets of query workload are designed for the evaluation. (1) We adopt the *DBPSB* benchmark [26] and derive 50 star query templates. Each template contains a set of nodes and edges which are augmented by either the real labels, e.g., '*Person*', or a variable label '*?*'. The percent of the variable label is $\leq 50\%$ in each template. The variable node (resp. edge) in a template query can be matched to any node (resp. edge) in the graph. To generate a query, we search the template in the graphs and select the most common labels from the data entity that are matched to the variables. The selected labels are then used to instantiate the variable nodes/edges in the template. (2) Since the templates are only stars, we extend the templates by adding nodes and edges to generate queries with complex structure, *e.g.,* cycles and multiple stars. Figure 1 shows a sample query.

Algorithms. We implement the STAR framework, including algorithms stark, stard and starjoin. In order to run stark for d-bounded star queries, d-hop traversal is performed for each node match of the pivot node. For comparison, we also developed two top-k search algorithms, graphTA and BP.

(1) The algorithm graphTA (Section III) is a direct application of the threshold (TA) algorithm over top-k subgraph querying [1]. For a fair comparison, we implement graphTA with two optimizations. (a) The neighbors and their matching scores are cached in each node when the node is visited during the traversal. The cache serves as an index to reduce the unnecessary graph traversal when the node is visited again; (b) Instead of using the widely adopted DFS traversal, it adopts BFS traversal so that the neighbor nodes are sorted based on their scores before carrying out the next round of exploration. These two strategies reduce the runtime of graphTA by 90%.
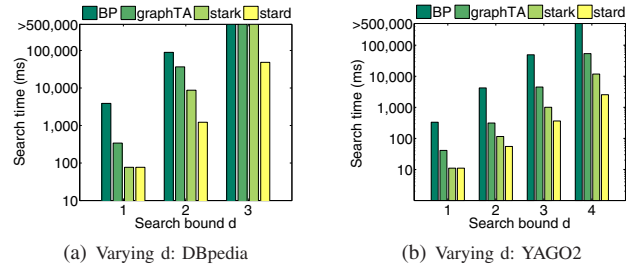

(a) Varying d: DBpedia   (b) Varying d: YAGO2

Fig. 12: The effect of search bound d

(2) The label propagation based algorithm [2], [14], such as Belief Propagation (BP), was also employed recently for approximate top-k pattern matching. BP [2] considers the nodes/edges in a query as a set of random variables and converts the top-k matching problem to the probabilistic inference on the label (match) for each random variable. It finds approximate matches for cyclic patterns, by exchanging probability scores as messages among node matches. For acyclic queries, BP outputs the exact top-k matches. But for cyclic queries, different from the STAR framework, it does not guarantee the completeness. We did not employ the graph sketch technique developed in [2] as it can benefit all the search algorithms.

Metrics. Given the query workload, the search runtime corresponds to the end-to-end query processing time, *i.e.,* the total CPU time spent from receiving the query to the output of the top-k results. The time includes not only the cost of the top-k search time but also the cost of other tasks, such as node matching and query decomposition, which account for a small amount of runtime ($\leq 1\%$).

Setup and Memory Usage. All the algorithms are implemented in Java. We conduct the experiments on a server with Intel Core i7 2.8GHz CPU and 32GB RAM, running 64-bit Linux. To serve online queries, the graph is stored in main memory while the attribute information attached to the nodes/edges is stored in a MongoDB server on a 512GB SSD. Each result reported in the following is averaged over 5 cold runs. The memory consumed by our algorithms is negligible, in comparison with the memory used to store the graph data. The time spent on fetching entities and relations from MongoDB is around $5 - 10\%$ of total query processing time.

B. Evaluation results

Exp-1: Runtime over star queries. In this experiment, we examine the impact of the search bound d. We employ a query workload consisting of $1,000$ star queries which are randomly generated based on the query templates with different size. By fixing k=20 and varying d, we compare the performance of stark, stard, graphTA and BP.

The results over *DBpedia* and *YAGO2* are reported in Figure 12(a) and (b) respectively, in log scale. The result shows that stark and stard outperform BP and graphTA by almost one order of magnitude. Note that when $d = 1$, stard degrades to stark, thus having the same runtime. When $d \geq 2$, stark is slower than stard as it has to search d-hops for each node. Indeed, for large d, BP, graphTA and stark may incur a humongous amount of message passing and neighborhood
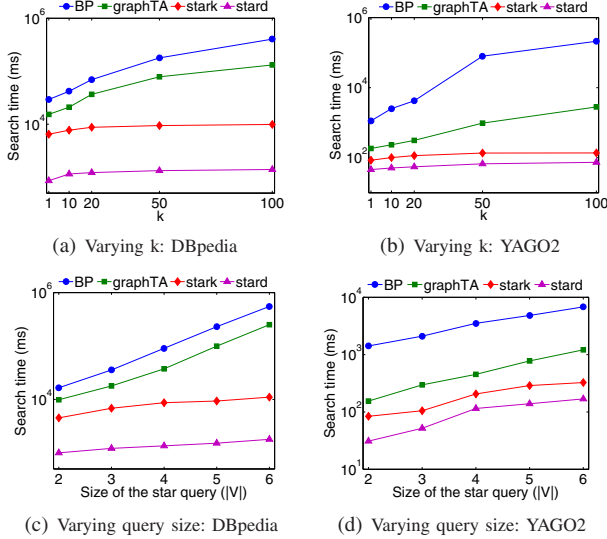
(a) Varying k: DBpedia

(b) Varying k: YAGO2

(c) Varying query size: DBpedia

(d) Varying query size: YAGO2

Fig. 13: Efficiency of star querying

(a) Effect of α ($k = 100$)

(b) Varying k: runtime

(c) Query size ($k = 100$)

(d) Search depth ($k = 100$)

Fig. 14: Evaluation on the top-k join

exploration, which can be reduced by stard.

Exp-2: Impact of k and query size. In this set of experiments, we evaluate the impact of k and query size to the runtime. Fixing $d = 2$ and use the same query workload as in the previous experiment, we vary k from 1 to 100. The results are plotted in Figure 13(a-b), which shows that the runtime of BP and graphTA grows dramatically when k increases. Indeed, both BP and graphTA use top scored node matches to find complete matches, which incurs considerable useless enumeration and traversal, especially for larger k. The top scored node matches might not lead to the best matches of the query. In contrast, stark and stard outperform all other methods in orders of magnitude, and their performance is much less sensitive to the growth of k. We observe that the main bottleneck for stark is the expensive graph traversal, especially for larger d and denser graphs (*DBpedia*). stard copes with this quite well: Almost all results are acquired in 1 second.

To evaluate the impact of query size, we use star query templates with different numbers of nodes varying from 2 to 6. We generate 5 query workloads accordingly, each contains 1,000 instantiated queries. We fix $d=2$ and $k=20$. Figures 13(c-d) show the exponential runtime growth of BP and graphTA, while stark and stard are less sensitive. stark (resp. stard) improves BP and graphTA better over larger queries, and is twice (resp. 8 times) faster than graphTA for even single edge query with 2 nodes.

We conduct the above experiments on more complicated graph queries and had very similar observations. The reason is obvious. Since stark and stard optimize the search based on bigger structures (star vs. single node/edge), their search will have a lower chance to be stuck in local optimum.

Exp-3: Efficiency of top-k join. This experiment running on DBpedia, examines the proposed top-k rank join technique. The three query decomposition methods, *i.e.,* SimSize, SimTop and SimDec, are inspected, respectively. The node score variance in SimTop and SimDec is estimated
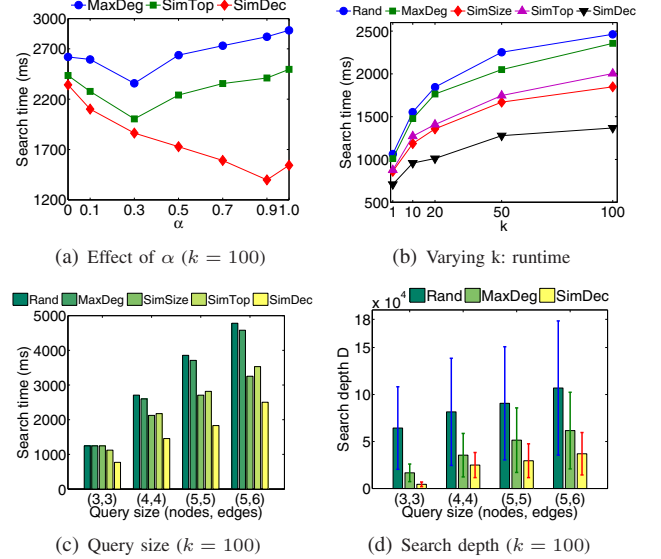
online by randomly sampling 200 matches for each query node. The sampling time only accounts for $\leq 1\%$ of the total search time and hence is not reported separately. In SimDec, $p = 4.5 \times 10^{-4}$, estimated by checking a set of edge queries. Additionally, two baselines are compared: (1) Rand refers to a method that randomly selects the pivot nodes to generate star subqueries; (2) MaxDeg greedily selects the pivots with the highest degree in the query graph.

We first test the effect of the α-schema. A query workload is generated using randomly selected query templates. We choose $k=100$ and $d=1$ in the experiment. Figure 14(a) depicts the average search time by varying α. It shows that a well selected α value indeed leads to less runtime. Considering each method, the best performance can be achieved when $\alpha = 0.3$ for MaxDeg, $\alpha = 0.3$ for SimTop and $\alpha = 0.9$ for SimDec ($\lambda = 1.0$), respectively. These α values are used in the following tests. We choose $\alpha = 0.5$ for Rand and SimSize due to their random and symmetrical nature (verified by real test). We also evaluate each method by varying k and plot the time efficiency in Figure 14(b). The result tells when k increases, the search time increases accordingly. Moreover, SimSize, SimTop and SimDec demonstrate constantly better runtime performance than Rand and MaxDeg for each k setting. Among all the methods, SimDec performs best, saving up to 45% *w.r.t.* Rand in terms of search time.

The experiment in Figure 14(c) examines top-k join by the query workloads with different query size, ranging from $Q(3,3)$ to $Q(5,6)$. We observe when the query size increases, the runtime increases for all the methods. This is because a larger query is usually decomposed into more stars, leading to more expensive multi-way joins. In the figure, SimDec shows the best time efficiency compared with the others. Moreover, the top-k join incurs large search depth for each star subquery. This effect can be seen in Figure 14(d), which reports the average search depth. Among all the methods, SimDec results in the smallest search depth for each query workload. Figure 14(d) also shows the average standard deviation as the error bar for each workload. When serving a query, small depth deviation indicates similar search depth for each star subquery,

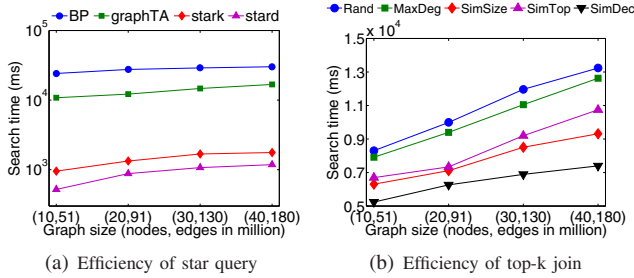(a) Efficiency of star query   (b) Efficiency of top-k join

Fig. 15: Scalability evaluation: Freebase

leading to a balanced search effort. As shown in the figure, the heuristic employed in SimDec is quite effective, showing the smallest deviation. Note that this balance merit might have significant impact on distributed graph query processing and thus is worth investigating in the future.

Exp-5: Scalability. This experiment studies the scalability of the algorithms over *Freebase*. Specifically, we extract a graph $G_1(10M, 51M)$, *i.e.,* 10M nodes and 51M edges, from Freebase and expand it in a BFS manner (each time randomly pick up a node and add the new edge from Freebase) to three larger graphs $G_2(20M, 91M)$, $G_3(30M, 130M)$ and $G_4(40M, 180M)$. We use a query workload with $1,000$ randomly generated queries and fixed $k = 20$ and $d = 2$. Since k is fixed, the runtime might not increase linearly w.r.t the graph size. Figure 15(a) reports the result of top-k star querying in log scale. When the graph size increases, the runtime of all the algorithms increases, as expected. stark and stard outperform their competitors by at least one order of magnitude. Moreover, stard further improve stark by $35\% - 45\%$.

We also verify the scalability of starjoin and report the result in Figure 15(b). Using the α schema, the proposed decomposition techniques, SimSize, SimTop and SimDec, are $20\% - 44\%$ faster than the baselines Rand and MaxDeg. This again demonstrates the effectiveness of the α-scheme and the decomposition heuristics.

VIII. RELATED WORK

The top-k search has been studied extensively in various contexts, including relational data, XML and graph.

Top-k Relational Queries. Top-k search over relational data is to find top-k tuples for a scoring function [27]. Given a monotonic aggregation function, and a sorted list for each attribute, Fagin's algorithm [28] reads the attribute values from the lists and constructs complete tuples. It stops when k complete tuples are found from the top-ranked attributes that have been seen. It then performs random access to find missing scores. The algorithm is optimal with high probability for some monotonic scoring functions. The threshold algorithm (TA) [11] improves Fagin's algorithm in that it is optimal for all monotonic scoring functions, and allows early termination. In a nutshell, it reads the scores of a tuple from the lists and performs sorted access by predicting maximum possible score in the unseen ones, until top-k tuples are identified.

Ranked join queries are studied over relational data [21], [29]–[31]. Assuming that random access is not available, J^* search [29] tries to identify a combination of attributes at the top of priority queues by selecting the stream to be joined,

and pulls the next tuple from the selected stream. Ranked join queries are also studied in NoSQL databases [32], which leverage indices and MapReduce optimization, as well as statistical structures (histogram and bloom filter) to reduce the cost and identify promising values. Distance join index is proposed in [31] to find matches with static scores for graph patterns, where edges can be matched to paths. A recent work [33] introduces the hybrid indexing on weighted attribute graphs. The indexing considers the weights of the attributes on the nodes as well as the structure of the graph.

In this work we study top-k queries on knowledge graphs. (1) We do not assume static node/edge weights; instead, the matching scores are computed online. (2) We study a general graph matching problem, where the matching quality is determined by scores from nodes and edges, and edges can be matched to paths of bounded length.

Top-k graph search. Top-k graph search have been studied for keyword queries [34], [35], twig queries [4], [5] and subgraph isomorphism [1], [6], [8]–[10], [36].

Keyword search. Top-k efficient keyword search on relation data, XML, RDFs, was extensively studied [34], [35], [37]–[40]. There are works on query relaxation and approximate matching on SPARQL queries, e.g., [41]. Most of the existing keyword search algorithms use relatively simple ranking functions such as TF/IDF and do not consider the links among keywords. In this study, the scoring function is far more sophisticated, combining 46 similarity measures (structure and content) together, which makes query processing a challenging issue. The effectiveness of using a learning-based ranking function over traditional keyword methods has already been demonstrated in [2]. It is like Google vs. traditional IR.

Twig query. In a more general setting, top-k graph pattern matching for twig queries are studied [4], [5], [17]. A bottom-up strategy is studied [4] where sorted access is used to generate matches for the leaf nodes in the twig query, and top matches for subqueries are obtained by merging top matches from their leaf nodes. [5] studies top-k graph pattern matching when strict monotonicity may not hold for some twig queries. These studies typically require sorted node/edge matches and the construction of transitive closure for the data graph, which are expensive over large graphs. In contrast, our method does not require transitive closure and is able to perform top-k join using partial matching lists generated online.

Graph query. Top-k search for general graph queries was studied [1], [6], [8]–[10], [36]. The common practice in these studies for early termination is, in general, conservative TA-style test. [9] uses multiple match lists of spanning trees from a pattern to answer top-k graph pattern matching. Instead of accessing node/edge matches in the list, we resort to big structure – star subquery, which can be solved in an efficient manner. This is not addressed in [9]. Furthermore, our decomposition technique identifies promising stars as the subqueries when serving the general graph query.

IX. CONCLUSIONS

We developed STAR, a top-k subgraph searching framework over knowledge graphs. We have shown that STAR can efficiently solve popular star queries posed on knowl-

edge graphs. It can also be conveniently exploited for more complicated graph queries, by incorporating a join algorithm and an upper bound scheme. Experimental results show that STAR is 5-10 times faster than the state-of-the-art TA-style subgraph matching algorithm, and 10-100 times faster than belief propagation based graph matching algorithms.

X. Acknowledgments

References

[1] L. Zou, R. Huang, H. Wang, J. X. Yu, W. He, and D. Zhao, "Natural language question answering over RDF: a graph data driven approach," in *SIGMOD*, 2014.

[2] S. Yang, Y. Wu, H. Sun, and X. Yan, "Schemaless and structureless graph querying," *VLDB*, 2014.

[3] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *VLDB*, 2011.

[4] G. Gou and R. Chirkova, "Efficient algorithms for exact ranked twig-pattern matching over graphs," in *SIGMOD*, 2008.

[5] Y. Qi, K. S. Candan, and M. L. Sapino, "Sum-max monotonic ranked joins for evaluating top-k twig queries on weighted data graphs," in *VLDB*, 2007, pp. 507–518.

[6] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum, "Topx: efficient and versatile top-k query processing for semistructured data," *VLDB*, 2008.

[7] L. Zou, L. Chen, and Y. Lu, "Top-k subgraph matching query in a large graph," in *Ph.D. workshop in CIKM*, 2007.

[8] X. Ding, J. Jia, J. Li, J. Liu, and H. Jin, "Top-k similarity matching in large graphs with attributes," in *Database Systems for Advanced Applications*, 2014, pp. 156–170.

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

[10] X. Zeng, J. Cheng, J. Yu, and S. Feng, "Top-k graph pattern matching: A twig query approach," *WAIM*, 2012.

[11] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 614–656, 2003.

[12] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, "An empirical study of real-world SPARQL queries," in *USE-WOD workshop*, 2011.

[13] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 788–799, 2012.

[14] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan, "Nema: fast graph search with label similarity," in *VLDB*, 2013.

[15] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang, "String similarity measures and joins with synonyms," in *SIGMOD*, 2013.

[16] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han, "Top-k interesting subgraph discovery in information networks," in *ICDE*, 2014.

[17] L. Chang, X. Lin, W. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "Optimal enumeration: Efficient top-k tree matching," in *Proc. VLDB Endow.*, vol. 8, no. 5, 2015, pp. 533–544.

[18] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461.

[19] J. Mondal and A. Deshpande, "EAGr: supporting continuous ego-centric aggregate queries over large dynamic graphs," in *SIGMOD*, 2014, pp. 1335–1346.

[20] 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," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, 2012.

[21] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting top-k join queries in relational databases," *The VLDB Journal*, vol. 13, no. 3, pp. 207–221, 2004.

[22] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," *SIGKDD*, 2011.

[23] "Dbpedia," *dbpedia.org*.

[24] "Yago2," *mpi-inf.mpg.de/yago*.

[25] "Freebase," *freebase.com*.

[26] M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo, "DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data," in *ISWC*, 2011.

[27] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys (CSUR)*, vol. 40, no. 4, p. 11, 2008.

[28] R. Fagin, "Combining fuzzy information from multiple systems," in *PODS*, 1996.

[29] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter, "Supporting incremental join queries on ranked inputs," in *VLDB*, 2001.

[30] M. Xie, L. V. Lakshmanan, and P. T. Wood, "Efficient rank join with aggregation constraints," *VLDB*, 2011.

[31] L. Zou, L. Chen, and M. T. Özsu, "Distance-join: Pattern match query in a large graph database," *VLDB*, 2009.

[32] N. Ntarmos, I. Patlakas, and P. Triantafillou, "Rank join queries in nosql databases," *VLDB*, 2014.

[33] S. B. Roy, T. Eliassi-Rad, and S. Papadimitriou, "Fast best-effort search on graphs with multiple attributes," *TKDE*, vol. 99, p. 1, 2014.

[34] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: ranked keyword search over XML documents," in *SIGMOD*, 2003.

[35] H. He, H. Wang, J. Yang, and P. S. Yu, "Blinks: ranked keyword searches on graphs," in *SIGMOD*, 2007.

[36] A. Wagner, V. Bicer, and T. Tran, "Pay-as-you-go approximate join top-k processing for the web of data," in *The Semantic Web: Trends and Challenges*, 2014, pp. 130–145.

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

[38] 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 *SIGMOD*, 2008, pp. 903–914.

[39] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data," in *ICDE*, 2009, pp. 405–416.

[40] S. Elbassuoni, M. Ramanath, R. Schenkel, and G. Weikum, "Searching rdf graphs with sparql and keywords," *IEEE Data Eng. Bull.*, vol. 33, no. 1, p. 1624, 2010.

[41] C. Hurtado, A. Poulovassilis, and P. Wood, "A relaxed approach to RDF querying," in *ISWC*, 2006, pp. 314–328.