Schemaless and Structureless Graph Querying

Shengqi Yang Yinghui Wu Huan Sun Xifeng Yan
University of California Santa Barbara
{sqyang, yinghui, huansun, xyan}@cs.ucsb.edu

ABSTRACT

Querying complex graph databases such as knowledge graphs is a challenging task for non-professional users. Due to their complex schemas and variational information descriptions, it becomes very hard for users to formulate a query that can be properly processed by the existing systems. We argue that for a user-friendly graph query engine, it must support various kinds of transformations such as synonym, abbreviation, and ontology. Furthermore, the derived query results must be ranked in a principled manner.

In this paper, we introduce a novel framework enabling schemaless and structureless graph querying (SLQ), where a user need not describe queries precisely as required by most databases. The query engine is built on a set of transformation functions that automatically map keywords and linkages from a query to their matches in a graph. It automatically *learns* an effective ranking model, *without* assuming manually labeled training examples, and can efficiently return top ranked matches using graph sketch and belief propagation. The architecture of SLQ is elastic for "plug-in" new transformation functions and query logs. Our experimental results show that this new graph querying paradigm is promising: It identifies high-quality matches for both keyword and graph queries over real-life knowledge graphs, and outperforms existing methods significantly in terms of effectiveness and efficiency.

1. INTRODUCTION

Graph querying is widely adopted to retrieve information from emerging graph databases, *e.g.,* knowledge graphs, information and social networks. Searching these real-life graphs is not an easy task especially for non-professional users: either no standard schema is available, or schemas become too complicated for users to completely possess. For example, a single knowledge graph could have more than 10K types of entities, as illustrated in Table 1, not to mention the different presentations of entity attributes.

Proceedings of the VLDB Endowment, Vol. 7, No. 7

Knowledge graphs	node types	relation types	entities
DBpedia [1]	359	800	3.7M
YAGO2 [3]	6,543	349	2.9M
Freebase [2]	10,110	9,101	40.3M

Table 1: Knowledge graphs

This kind of complexity contrasts to the impatience of web users who are only interested in finding query answers in a short period. The existing structured query techniques such as XQuery [6] and SPARQL [22] are barely able to address such challenge. Keyword queries (*e.g.,* [11, 14, 27]) were proposed to shield non-professional users from digesting complex schemas and data definitions. Unfortunately, most of keyword query methods only support a predefined similarity measure, such as approximate string matching [18] and ontology-based matching [30]. A general, systematic approach that automatically supports multiple measures (*e.g.,* synonym, abbreviation, ontology, and several more summarized in Table 2) all together is lacking.

In this paper, we present a principle that could take multiple matchings into account and demonstrate its great potential. Under this principle, given a query Q, query evaluation is conducted by checking if its matches in a graph database G can be "transformed" from Q through a set of transformation functions.

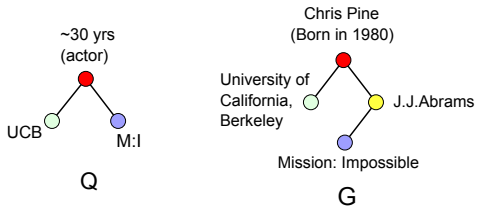


Figure 1: Searching with transformations

Example 1: To find a movie star in a knowledge graph, a graph query Q is issued (Figure 1), which aims to find an actor whose age is around 30 ("`30 yrs`"), graduated from UC Berkely ("`UCB`"), and may relate to movie "Mission: Impossible" ("`M:I`"). One may identify a match for Q as shown in Figure 1. The match indicates that "`30 yrs`" in Q refers to an actor "Chris Pine" who was born in 1980, "`UCB`" is matched to the University of California, Berkeley, and "`M:I`" refers to the movie "Mission:Impossible". Traditional keyword searching based on IR methods or string similarity cannot identify such matches. □

Given a few transformation functions, one might find many matches of Q in a graph database. A transformation-

Transformation	Category	Description	Example
First or Last token	String	take the first/last token	'Anne Hathaway' → 'Anne', 'Justin Bieber' → 'Bieber'
Abbreviation	String	initials of all but last	'Jeffrey Jacob Abrams' → 'J.J. Abrams'
Drop	String	drop the last token	'US Airways Company' → 'US Airways'
Bag of words	String	entity described by the keyword	'Yankees hat' → 'Tom Cruise' ('. . . signs Yankees hat')
Prefix	String	take prefix (*e.g.,* 3 letters)	'Street' → 'Str'
Acronym	String	initials of each token	'International Business Machines' → 'IBM'
Synonym	Semantic	replaced by synonym	'lawyer' → 'attorney'
Ontology	Semantic	replace using ontology	'teacher' → 'educator'
Date Gap	Numeric	year gap of two dates	'2010' → '3 yrs ago' (as of, *e.g.,* 2013)
Date Abbreviation	Numeric	simplify date	'2008.8.8' → '2008.8'
Range	Numeric	find numbers in a range	'∼30 yrs' → '33 yrs'
Unit Conversion	Numeric	convert measurements	'0 Celsius' → '32 Fahrenheit', '3 mi'→ '4.8 km'
Distance	Topology	edge to shortest path	'Pine'-'M:I' → 'Pine'-'J.J.Abrams'-'M:I'

Table 2: Transformations

friendly query engine must address the following two questions: (1) *how to determine which match is better*? (2) *how to efficiently identify the top ranked matches*?

Intuitively, the selectivity, the popularity, and the complexity of transformation functions shall be considered and used as a ranking metric for these matches. How to choose, from many possible transformations, an appropriate ranking metric that leads to good matches? First, a searching algorithm should be deployed to determine the best transformation for different portions of a query. For example, "UCB" should be automatically transformed to entities using it as acronym, rather than string edit distance. This requires a weighting function for various transformations. Second, to identify such a function, manual tuning should be reduced to a minimum level. Instead of asking users to tune the weights, learning to rank [15, 25] is more appropriate. Unfortunately, it usually needs manually labeled training data, again a daunting task for end users. Finally, since there could be too many matches to inspect, it is important to only return top-k results. While desirable, this top-k search problem is much more challenging due to the presence of different transformations, compared to its single transformation counterpart.

Contributions. This work proposes a first-kind of graph querying framework that answers all these questions.

(1) We propose a new, generalized graph searching problem: Given a query Q, a graph G and a library of transformation functions $\mathcal{L}$, where there are multiple matches in G that can be transformed from Q by applying $\mathcal{L}$, it is to find the top-k ranked matches for Q. In contrast to traditional graph searching using single, predefined similarity metric such as string similarity, we use a metric combining transformations of various kinds. The metric itself is automatically learned.

(2) We propose SLQ, a general graph query framework for schemaless and structureless querying. It consists of two phases: *offline learning* and *online query processing*.

(a) *Given multiple matches transformed from Q, how to decide a proper ranking metric?* Certainly a manually picked combination function, *e.g.,* averaging, is not going to work elegantly. We show that this problem can be solved by a parameterized ranking model. In this work, we adopt conditional random fields [24], as it not only gives a good ranking model, but also indicates a fast matching search algorithm. In the offline learning phase, the framework needs to solve the cold-start problem, *i.e.,*, where to find training samples to train the model. Manually labeled matches might be too costly for a few sample queries. A systematic approach is hence introduced to create sample queries and answers by extracting subgraph queries from G, inject transformations to these queries, and form query-answer pairs for training.

(b) *Given a ranking metric, how to efficiently find top ranked matches?* For general graph queries and keyword queries, we prove that the problem is NP-hard. We propose a polynomial time heuristic top-k algorithm for online query processing. The problem is tractable for tree-structured queries, and an exact, polynomial time algorithm is developed. Both algorithms stop once k best matches are identified, without inspecting every match. In practice, they run very fast.

(3) Using several real-life data/knowledge graphs, we experimentally verify the performance of our graph querying engine. It outperforms traditional keyword (Spark [18]) and approximate graph searching (NeMa [12]) algorithms in terms of quality and efficiency. For example, it is able to find matches that cannot be identified by the existing keyword or graph query methods. It is 2-4 times faster than NeMa, and is orders of magnitude faster than a naive top-k algorithm that inspects every match.

To the best of our knowledge, these results are among the first efforts of developing a unified framework for schemaless and structureless querying. SLQ is designed to help non-professional users access complex graph databases in a much easier manner. It is a flexible framework capable of finding good matches when structured query languages do not work. New transformations and ranking metrics can be *plugged in* to this framework easily. The contribution of this study is not only at providing a novel graph querying paradigm, but also at the demonstration of unifying learning and searching for much more intelligent query processing. The proposed techniques can be adapted easily to a wide range of search applications in databases, documents and the Web.

2. PRELIMINARY

Property graph model. We adopt a *property graph model* [23]. A graph $G = (V, E)$ is a labeled graph with node set V and edge set E, where each node $v \in V$ has a property list consisting of multiple attribute-value pairs, and each edge $e \in E$ represents a relationship between two entities. The model is widely adopted to present real-life schemaless graphs To simplify our presentation, we will first treat all the information associated with nodes and edges as keywords, and then differentiate type and value in Section 7.

Queries. We formulate a query Q as a property graph (V_Q, E_Q). Each query node in Q describes an entity, and an edge between two nodes, if any, specifies the connectivity constraint posed on two query nodes. Q could be disconnected when a user is not sure about a specific connection. This query definition covers both keyword query [27] (query nodes only) and a graph pattern query [7] (connected query graph). For the ease of discussion, we first focus on the query that is connected. How to handle disconnected queries including keyword queries is given in Section 7.

Traditional graph querying assumes structured queries formulated from well-defined syntax and vocabulary (*e.g.,* XPath and SPARQL). This work considers general queries that might not exactly follow the structure and semantic specifications coded in a graph database.

Transformations and matches. To characterize the matches of Q, we assume a library $\mathcal{L}$ of transformation functions (or simply transformations). A transformation f can be defined on the attributes and values of both nodes and edges of Q. The transformation functions can be specified in various forms, *e.g.,* (string) transformation rules [4]. Table 2 summarizes several common transformations. These transformations consider string transformation, semantic transformation, numeric transformation, and topological transformation (as edge transformations). For example, "Synonym" allows a node with label "`Tumor`" to be mapped to the node "`Neoplasm`". All these transformations are supported in our implementation. New transformations, such as string similarity (*e.g.,* spelling error) [17] and Jaccard distance on word sets [12] can be readily plugged into $\mathcal{L}$. The focus of this work is to show a design combining different transformations, not to optimize a specific transformation.

A node or edge in Q matches its counterparts in a data graph G with a set of transformed attributes/values, specified by a matching (function) ϕ. A match of Q, denoted as $\phi(Q)$, is a connected subgraph of G induced by the node and edge matches. In this work, for each attribute/value, we only consider one-time transformation, as the chance for transforming multiple times is significantly lower.

3. SCHEMALESS AND STRUCTURELESS QUERYING

In this section, we provide an overview of SLQ, and its three key components: matching quality measurement, offline learning, and online query processing.

Matching quality measurement. Given Q and a matching ϕ of Q, we need to measure the quality of $\phi(Q)$ by aggregating the matching quality of corresponding nodes and edges. Intuitively, an identical match should always be ranked highest; otherwise, $\phi(Q)$ shall be determined by the transformations, as well as their *weights* to indicate how "important" they are in contributing to a reasonable match. One possible strategy is to assign equal weight to all transformations. Certainly, it is not the best solution. For example, given a single node query, "Chris Pine", nodes with "C. Pine" (Abbreviation) shall be ranked higher than nodes with "Pine" (Last token). A predefined weighting function is also not good, as it is hard to compare transformations of different kinds. In this work, we introduce a novel learning approach to figure out their weights (Section 4).

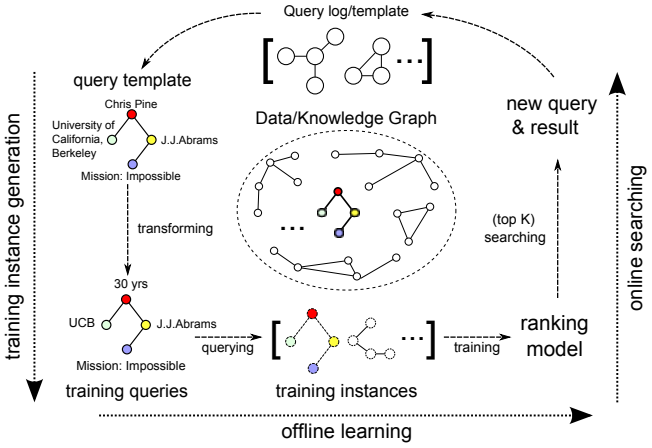


Figure 2: SLQ: graph querying framework

Offline model learning. There might exist multiple matches for Q in a graph G using different transformations. An advanced model should be parameterized and be able to adjust the weights of all possible transformations. If a historical collection of queries and user-preferred answers is available, through a machine learning process, one can automatically estimate weights so that the user-preferred answers could be ranked as high as possible.

As suggested from previous work [15], the best practice for learning a model is to employ a query log generated by real users. However, the log might not be available at the beginning. On the other hand, the system does need a set of good-quality query-answer pairs to have its weights tuned. This becomes the chicken or the egg dilemma. In Section 4.2, we introduce a method to automatically generate training instances from the data graph.

Online top-k searching. Once the parameters of the ranking function are estimated in the offline learning, one can process queries online. Fast query processing techniques are required to identify top ranked matches based on the ranking function. This becomes even more challenging when multiple transformations are applicable to the same query, and the answer pool becomes very large. While the problem is in general intractable, we resort to fast heuristics. The idea is to construct a small *sketch graph* by grouping the matches in terms of Q and the transformations. The algorithm first finds the matches in the sketch graph that are likely to contain the top-k answers. It then "drills down" these matches to extract more accurate matches from the original graph G. This design avoids the need of inspecting all the matches.

Putting the above components together, Figure 2 illustrates the pipeline of SLQ. It automatically generates training instances from data graphs and any available query log. Using the training set, it learns a ranking model by estimating proper weights for the transformations. In the online stage, it applies efficient top-k searching to find best matches for new queries. A user can provide feedback by specifying good answers in the top-k matches, which can be put back to the query log to further improve the ranking model. In the following sections, we discuss each step in detail.

4. OFFLINE LEARNING

Given G and a library $\mathcal{L}$ of transformations, the offline learning module generates a ranking model, without resort-

ing to human labeling efforts. In this section, we present two key components, the parameter estimation and automatic training instance generation.

4.1 Ranking Function

Given Q and $\phi(Q)$, a node matching cost function $F_V(v, \phi(v))$ is introduced to measure the transformation cost from a query node v to its match $\phi(v)$. It aggregates the contribution of all the possible transformations $\{f_i\}$ with corresponding weight $\{\alpha_i\}$,

$$F_V(v, \phi(v)) = \sum_i \alpha_i f_i(v, \phi(v)) \quad (1)$$

where each f_i returns a binary value: it returns 1 if its two inputs can be matched by the transformation, and 0 otherwise. Analogously, an edge matching cost function is defined as

$$F_E(e, \phi(e)) = \sum_i \beta_i f_i(e, \phi(e)) \quad (2)$$

which conveys the transformation(s) from a query edge e to its match $\phi(e)$. $\phi(e)$ can be a path in $\phi(Q)$ with the two endpoints matched with those in e. $\{f_i\}$ can be extended to support real-valued similarity functions. We instantiate our querying framework with a set of commonly used transformations, as in Table 2. Other user-specified transformations can be plugged in too.

We now introduce a ranking function that could combine multiple nodes and edges matches together. There are two important factors to consider. First, using training data, it shall be able to optimize parameters $\{\alpha_i\}$ and $\{\beta_i\}$ for good ranking quality. Second, the ranking function shall have a mechanism to search top-k matchings quickly. Enumerating all possible matches of a query graph and then sorting their scores is not a good mechanism. In this work, we give a *probabilistic formulation* that satisfies both requirements. The superior performance of SLQ can already be demonstrated by this formulation. We leave the search and comparison of various probabilistic models in terms of ranking quality and query response time to future work.

Given Q and a match $\phi(Q)$, we use probability $P(\phi(Q)|Q)$ as a measure to evaluate the matching quality,

$$P(\phi(Q)|Q) = \frac{1}{Z} \exp\Big(\sum_{v \in V_Q} F_V(v, \phi(v)) + \sum_{e \in E_Q} F_E(e, \phi(e)) \Big) \quad (3)$$

where Z is a normalization function so that $P(\cdot) \in [0, 1]$.

The ranking function $P(\phi(Q)|Q)$ can be naturally interpreted with *conditional random fields* (CRFs), a widely applied graphical model (see [24] for more details). In our formulation, the nodes and edges in each query Q are regarded as the observed nodes and structures in CRFs; the nodes and edges in each match $\phi(Q)$ to be predicted are regarded as the output variables. CRFs directly models the distribution of the output variables given the observed variables, which naturally serves as our matching quality measure.

Example 2: Recall the Q and a match $\phi(Q)$ (Figure 3). Each node, *e.g.,* `30 yrs`, may have multiple matches via multiple transformations, as remarked earlier. The quality of the match $P(\phi(Q)|Q)$ is computed by aggregating the quality of each node and edge match in $\phi(Q)$, determined by a weighted function of all transformations. □

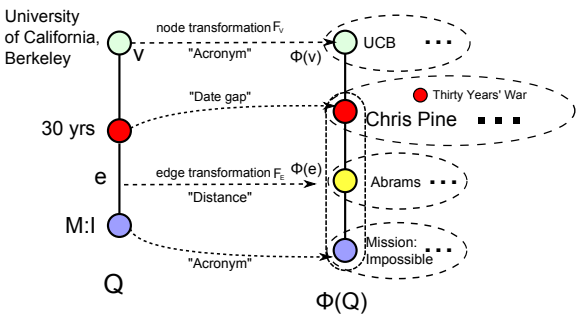


Figure 3: Ranking function

Two key differences between SLQ and the existing graph query algorithms are (1) we support multiple transformations; and (2) the weight of these transformations are learned, rather than user-specified. The probabilistic ranking function is a vehicle to enable these two differences.

4.2 Transformation Weights

To determine the weights of transformations $W = \{\alpha_1, \alpha_2, ...; \beta_1, \beta_2, ...\}$, SLQ automatically learns from a set of training instances. Training instances can be regarded as past query experiences, which can teach the system how to rank the results when new queries arrive. Each training instance is a pair of a query and one of its relevant answers. Intuitively, we want to identify the parameters W that can rank relevant answers as high as possible for a given query in the training set T. We choose parameters such that the log-likelihood of relevant matches is maximized,

$$W = \arg\max_W \sum_T \log P(\phi(Q)|Q) \quad (4)$$

Optimizing objective functions like Eqn. 4 has been studied extensively in machine learning community [24]. We adopt the standard Limited-memory BFGS (L-BFGS) [16] algorithm, as it requires less memory than other approaches.

Complexity. Based on the analysis of [24], the worst time complexity of training CRFs in our problem setting is $O(N|Q||V_m|^2|T|)$, where (1) N is the number of gradient computations performed by the optimization, (2) $|Q|$ is the size of the largest query in the training set, (3) $|V_m|$ is the largest number of the matches a query node or edge may have, and (4) $|T|$ is the number of training instances. Experimental results (Table 4 in Section 8) show that its training time is affordable for large real-life graphs, as only a small sample of the graph is needed. $|V_m|^2$ is also not an issue here as one can avoid using less-selective queries.

4.3 Automatic Training Instance Generation

A key issue in SLQ is how to cold-start the system when no user query log is available. We developed an innovative strategy to generate artificial training instances. It turns out that this strategy works far better than just giving equal weight to all transformations. Our system first randomly extracts a set of subgraphs from the data graph and treat them as query templates. For each template $\hat{Q}$, it injects a few transformations to $\hat{Q}$ and generates a set of training queries Q. Intuitively, each training query Q should have $\hat{Q}$ as its good match, since Q can be transformed back to $\hat{Q}$. The system also identifies exact matches of Q in G. Consequently, the matches identical to Q form training instances

too. The weights of transformation functions are learned by ranking $\hat{Q}$ as high as possible in the matches of Q, but below those identical matches of Q in G.

The identical matches play a key role of determining the weight of transformations. For example, with respect to a query template "Barack Obama", a match "B. Obama" is more preferred than "Obama" as there are less identical matches of "B. Obama" (*i.e.,* with higher selectivity). Therefore, by populating the training instances with random queries and results, the method can gauge the impact of transformations automatically in terms of selectivity. The second reason for this cold-start strategy to work well is that it covers different cases comprehensively, as it randomly and uniformly samples subgraphs from the data graph.

5. ONLINE QUERY PROCESSING

In this section, we introduce the online query processing technique that finds top-k ranked matches for Q in G with the highest scores. To simplify the discussion, we assume that each transformation f_i checks if a query node (resp. edge) matches a node (resp. path) in G in constant time.

The query processing problem is in general NP-hard, as one may verify that subgraph isomorphism [21] is its special case. To precisely compute $P(\phi(Q)|Q)$, one has to inspect every possible match, which is a daunting task. A straightforward algorithm identifies the match candidates for query node/edge via all transformations in $O(|Q||G||\mathcal{L}|)$ time, enumerates all possible result matches, and computes their rank scores to find top-k ones. Its complexity is $O(|Q||G||\mathcal{L}| + |G|^{|Q|})$, which does not scale over large G.

Observing the hardness of the exact searching (e.g., subgraph isomorphism), one shall not expect a fast solution with complete answers (except for tree queries). Instead, we resort to two heuristics. The first one leverages an inference technique in graphical models that has been verified to be efficient and accurate in practice [31] (Section 5.1). The second one further improves it by building a sketch of G so that low-score matches can be pruned quickly (Section 5.2). Our top-k algorithm based on these two techniques (Section 5.3) could reduce the query processing time in orders of magnitude, while only small loss of answer quality is observed (less than 1% in our experiments). Moreover, it can deliver *exact* top-k matches when Q are trees (Section 5.4), which is desirable as many graph queries are indeed trees.

Section 5.1 briefly introduces the first heuristic, LoopyBP, which needs some background knowledge to digest [28]. The readers may skip it without difficulty in understanding the remaining sections.

5.1 Finding Matches

The idea of LoopyBP is to treat Q as a graphical model, where each node is a random variable with a set of matches as possible assignments. It finds top assignments (matches) that maximizes the joint probability for Q (with highest matching quality). To this end, LoopyBP leverages inferencing techniques [28], which iteratively propagates "messages" among the nodes to estimate the matching quality.

Given Q, LoopyBP identifies a match $\phi(Q)$ that maximizes $P(\phi(Q)|Q)$ by seeking $\max_{u_i} b(u_i)$ [31]. For each node $v_i \in V_Q$ and its match u_i, $b(u_i)$ is formulated as:

$$b(u_i) = \max_{u_i} F_V(v_i, u_i) \Pi_{v_j \in N(v_i)} m_{ji}^{(t)}(u_i), \quad (5)$$

for each match u_i of v_i and each v_j in the neighborhood set $N(v_i)$ of v_i in Q. Here $m_{ji}^{(t)}(u_i)$ is a message (as a value) sent to u_i from the matches of $v_j \in N(v_i)$ at the t^{th} iteration:

$$m_{ji}^{(t)}(u_i) = \max_{u_j} F_V(v_j, u_j) F_E((v_j, v_i), (u_j, u_i)) \cdot \prod_{v_k \in N(v_j) \setminus v_i} m_{kj}^{(t-1)}(u_j), \quad (6)$$

for each match u_j of v_j. (u_j, u_i) represents the match of the query edge (v_j, v_i). Intuitively, the score $b(u_i)$ is determined by the quality of u_i as a node match to v_i (F_V), the quality of edge matches, *e.g.,* (u_j, u_i), attached to u_i (F_E), and the match quality of its neighbors u_j as messages ($m_{ji}(u_i)$). Hence the node u with the maximum $b(\cdot)$ and its "surrounded" node and edge matches naturally induce a match with good quality in terms of matching probability.

Algorithm. Based on the formulation, LoopyBP finds top matches in three steps. (1) It first initializes the messages of each node $m^{(0)}(\cdot) = 1$. (2) It iteratively updates $b(\cdot)$ following message propagation until none of $b(\cdot)$ in successive iterations changed by more than a small threshold. (3) LoopyBP identifies best node matches $u = \mathsf{argmax}_u b(u)$, and then extracts top-k matches $\phi(Q)$, following a backtracking strategy [31]. More specifically, LoopyBP first selects a match u with the highest score $b(\cdot)$, and induces a top 1 match $\phi(Q)_1$ following the node matches with top $b(\cdot)$ scores connected to u. It then finds a next best match by performing two message propagations: (a) it identifies a match u' of a node v in Q with the second highest score $b(\cdot)$ among all the matches and is not in $\phi(Q)_1$, and then performs a propagation to find a top match $\phi(Q)_2$, fixing $\phi(v) = u'$; (b) it performs a second round of propagation where $\phi(v) \neq u'$, to "trace back" to an earlier state of the scores in (a), and prepare to extract a next best match. It repeats the above process until k matches are identified (see details in [31]).

Complexity. The propagation only sends messages following edge matches as paths of bounded length d constrained by edge transformation in $\mathcal{L}$. Thus each propagation traverses, for each match u of v, up to the set V_d of d hops of u in G. The algorithm takes $O(I|Q||V||V_d|)$ time for message propagation in total I iterations, where a single iteration completes when each node exchanges a message with each of its neighbors. In addition, it takes $O(|Q|)$ time to construct a best (top-1) matching ϕ for Q. Putting these together, the process of finding one match takes overall $O(I|Q||V||V_d|)$ time. To find k matches, at most $2k$ rounds of propagation are conducted with backtracking, where each round denotes the start to the convergence of a propagation. Hence it identifies top k matches in $O(k * I|Q||V||V_d|)$ time. Note that d is typically small: Edges are usually matched with short paths as observed in keyword and graph searching [12, 14].

5.2 Sketch Graph

With LoopyBP, one still needs to inspect a large number of node and edge matches. Observe that these matches can be naturally grouped in terms of transformations: Each match contributes the *same* matching score when it conducts the same type of transformation. Following this, we construct a *sketch graph* G_h from G induced by Q and $\mathcal{L}$. The idea is to efficiently extract matches from a much smaller G_h, and then drill down to find more accurate "lower level" ones.

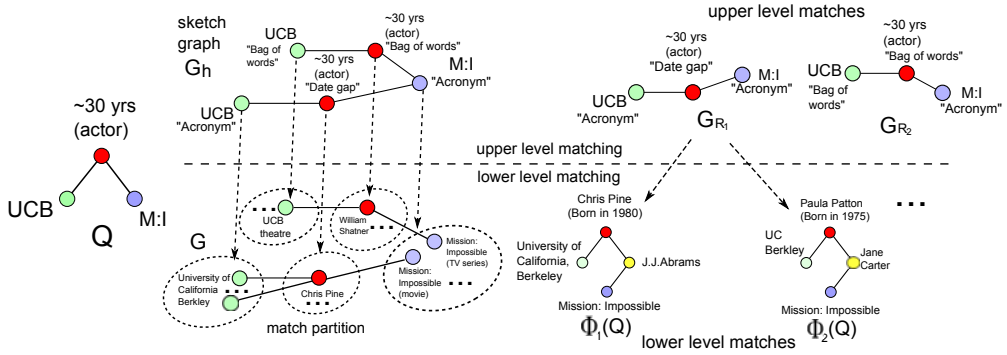


Figure 4: Graph sketch and top-k searching

We denote as σ_i the set of all matches for a node v_i in Q. (1) A *match partition* of σ_i is a set of partitions $\{\sigma_{i1}, \ldots, \sigma_{in}\}$ of σ_i, such that for any two nodes in σ_{ij}, they can be mapped to v_i via the same transformation $f_j \in \mathcal{L}$. (2) The sketch graph G_h of G contains a set of *hyper nodes*, where each hyper node $u_{(v_i,f_j)}$ denotes a match set σ_{ij} of v_i in Q induced by f_j. There is an edge connecting two hyper nodes $u_{(v_i,f_m)}$ and $u_{(v_j,f_n)}$ if and only if (v_i, v_j) is an edge of Q. Thus the match score of an edge in G_h establishes an *upper bound* of its underlying edge matches in G, since any edge in G_h is an exact match of an edge in Q. Intuitively, G_h sketches G by grouping the matches of each query node as a single node, as long as they can match to the query node by the same transformation. Note that a sketch graph G_h can also be queried by LoopyBP. We denoted as G_R an "upper level match" from G_h, and distinguish it from a "lower level match" G_r as a subgraph of G. G_r is contained in G_R if each node of G_r is in a hyper node of G_R.

One may verify that the rank score of each upper level match G_R indicates an *upper bound* of the rank scores of all the lower level matches it contains:

Lemma 1: *For any upper level match G_R (specified by matching ϕ_R) and any lower level match G_r contained in G_R (specified by ϕ_r),* $\max_{\phi_r(v_i)} b(\phi_r(v_i)) \leq \max_{\phi_R(v_i)} b(\phi_R(v_i))$, *where v_i ranges over the query nodes in V_Q.* □

Proof sketch: We prove by induction on the iterations that for any $v_i \in V_Q$ at any iteration t, $m_{ji}^{(t)}(\phi_r(v_i)) \leq m_{ji}^{(t)}(\phi_R(v_i))$. (1) Let $t = 1$. Since $F_V(v_i, \phi_r(v_i)) = F_V(v_i, \phi_R(v_i))$ and $F_E(e, \phi_r(e)) \leq F_E(e, \phi_R(e))$, we have $m_{ji}^{(1)}(\phi_r(v_i)) \leq m_{ji}^{(1)}(\phi_R(v_i))$, by the definition of G_R and Eqn. 6. (2) Assume $m_{ji}^{(t)}(\phi_r(v_i)) \leq m_{ji}^{(t)}(\phi_R(v_i))$ for $t < n$. When $t = n$, one can verify that $m_{ji}^{n}(\phi_r(v_i))$ is no larger than $m_{ji}^{n}(\phi_R(v_i))$, again with Eqn. 6. Hence, by Eqn. 5, $b(\phi_r(v_i)) \leq b(\phi_R(v_i))$. The Lemma hence follows. □

Note that the size of G_h is independent of $|G|$: it is bounded by $O(|Q|^2|\mathcal{L}|^2)$ where $|Q|$ and $|\mathcal{L}|$ are typically small. Moreover, G_h can be efficiently constructed using indexing techniques (Section 6).

Example 3: A sketch graph G_h is illustrated for the query Q in Figure 4. A node UCB with label "Acronym" in G_h points to a group of matches via transformation "Acronym". Given Q and G_h, LoopyBP provides an upper level match G_{R_1}, which contains two lower level matches $\phi_1(Q)$ and $\phi_2(Q)$, with rank scores bounded by that of G_{R_1}. □

Input: Graph G, query Q, transformations $\mathcal{L}$, integer k;
Output: a list L of top k ranked matches.

1. top k list $L := \emptyset$; terminate:=false;
2. **for each** node v of Q **do**
3. initialize valid match candidates *w.r.t.* $\mathcal{L}$;
4. construct sketch graph G_h;
5. $G_R :=$ LoopyBP(G_h);
6. **while** terminate = false **do**
7. update L with top k matches from LoopyBP(G_R);
8. $G_R :=$ LoopyBP(G_h);
9. update terminate;
10. **return** L;

Figure 5: Algorithm topK

5.3 Top-k Search

Using LoopyBP and sketch graph as building blocks, we next present our top-k searching algorithm. The algorithm, denoted as topK, is illustrated in Figure 5.

Given Q, G, $\mathcal{L}$ and integer k, topK initializes a top k match list L, and a Boolean flag terminate to indicate if the termination condition (as will be discussed) is satisfied (line 1). It next constructs a sketch graph G_h (lines 2-4). Given G, Q and G_h, it dynamically updates L with newly extracted matches, by applying LoopyBP over the sketch graph G_h and G iteratively (lines 5-9). More specifically, topK repeats the following two steps, until the termination condition is satisfied (terminate = true).

(1) The algorithm topK first performs LoopyBP over G_h, and produces a best upper level match of Q, *e.g.,* G_R, as a subgraph of G_h (line 5). Note that G_R corresponds to a subgraph of G, induced by all the nodes in G that are contained in the hyper nodes of G_R following edge matches.

(2) topK then "drills down" G_R to obtain the subgraph it corresponds to, and conducts LoopyBP over the subgraph to update L with more accurate lower level matches (line 7). Matches in L are replaced with new matches with higher scores. In addition, topK also performs necessary propagation over the subgraphs from earlier upper level matches, if they contain nodes with updated scores due to messages from the updated matches in L. It updates L with new lower level matches from these subgraphs, if any, until no more new matches can be identified to update L. It next extracts a next upper level match G_R from G_h (line 8).

The above steps (lines 7-8) complete a round of processing. At the end of each round, topK checks if the termination condition below is satisfied (line 9): (a) L already contains

k matches, and (b) the match ranked at k in L already has a score higher than the next upper level match G_R (if any) from G_h. If the condition is satisfied (or all possible matches in G are visited), topK terminates and returns L (line 10). Otherwise, it extracts a new high level match from G_h, and repeats steps (1) and (2).

Analysis. topK always terminates, as the message (value) propagation stops when the change of the value is below a threshold. Moreover, the top k matches returned by topK will be the same as those returned by LoopyBP if sketch graph is not involved, due to Lemma 1.

For the complexity, one may verify the following. (1) It takes $O(|V_Q||V||\mathcal{L}|)$ time to identify all the partition sets, and construct G_h (lines 2-4). Note that we assume every transformation checking is done in constant time, as remarked earlier. (2) The total runtime consists of two parts: the upper level LoopyBP (over G_h) and the lower level LoopyBP (over G) . The upper level LoopyBP (line 5,8) takes in total $O(I_1|\mathcal{L}|^2|Q|^3)$ time, since G_h has in total $|V_Q||\mathcal{L}|$ nodes, and it takes $O(I_1|Q|(|\mathcal{L}||V_Q|)^2)$ time for upper level LoopyBP, where I_1 denotes the total number of upper level iteration. Note that the performance of upper level LoopyBP is independent of the size $|G|$. (3) The lower level LoopyBP (line 7) takes in total $O(I_2|Q||V_t|^2)$ time, where I_2 is the iteration number for lower level LoopyBP, and $|V_t|$ denotes the total number of nodes in G visited by lower level LoopyBP when topK terminates. Putting these together, algorithm topK takes in total $O(|V_Q||V||\mathcal{L}| + I(|Q|^3|\mathcal{L}|^2 + |Q||V_t|^2))$ time, for in total I (*i.e.,* I_1+I_2) iterations.

In practice, $|Q|$ and $|\mathcal{L}|$ are typically small. Moreover, indexing techniques to efficiently identify node matches (lines 2-4) can be readily applied, reducing its time complexity from $O(|V_Q||V||\mathcal{L}|)$ to $O(|V_Q|)$ (see Section 6). Our experiments show that topK achieves near-linear runtime w.r.t. graph size (see Figure 11 in Section 8). A possible reason is that most of possible node matches are not connected with each other in terms of edge matches. The number of message passing among them is much smaller than the worst case $|V_t|^2$. They cannot form a high quality subgraph that matches the entire query graph. In the first few iterations, they are quickly pruned by LoopyBP.

Example 4: Consider the query Q in Figure 4. The algorithm topK finds top 2 matches for Q in G as follows. (1) topK first computes a sketch graph G_h of G as show in Fig. 4. (2) It then computes a top ranked result G_{R_1} from G_h, where the node `UCB` in Q is matched (via transformation "Acronym") with a hyper node that contains the node `University of California, Berkeley` . topK then computes a top K list by drilling down G_{R_1} (Figure 4), and identifies two lower level matches $\phi_1(Q)$ and $\phi_2(Q)$ from G_{R_1} (Figure 4), indicating actors in the movie "mission:impossible". (3) It next identifies a second high level match G_{R_2}, specified by "Bag of words" and "Acronym". Without drilling-down to lower level matches, topK identifies that the ranking score of G_{R_2} is already lower than $\phi_2(Q)$. This indicates that no lower level matches better than $\phi_2(Q)$ can be found. topK thus returns $\phi_1(Q)$ and $\phi_2(Q)$ as the top 2 matches. □

5.4 Exact Matching for Trees

When Q is a tree, which is quite common in practice, topK can be readily revised, leading to efficient *exact* top-k search.

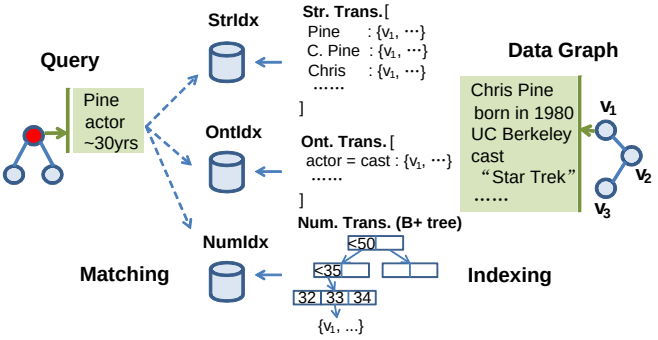


Figure 6: Indexing

Algorithm. The algorithm topK for tree queries iteratively performs LoopyBP over G_h and G, similarly as for general graph queries. The difference is that it uses a simplified propagation: it only performs two passes of propagation to extract an optimal match [28]. More specifically, given a tree query Q, it designates a *root* in Q, and denotes all nodes as *leaves*. topK then computes a top ranked match by conducting two passes of propagation: one from the matches for all leaves to those of the root, and the other from the matches of the root to all the matches of the leaves. It repeats the process to fetch top k best results.

Correctness and Complexity. Following [28], the two passes of propagation in topK for a tree query Q is guaranteed to converge in at most m steps, where m is the diameter of Q, *i.e.,* the length of the longest shortest path between two nodes in Q. Moreover, the propagation computes the exact rank value $P(\phi(Q)|Q)$ (Section 4). The correctness of topK hence follows. One may verify that topK is in total $O(|V_Q||V||\mathcal{L}| + |Q|^3|\mathcal{L}|^2 + |Q||V_t|^2)$ time over tree queries, with (a) 2 passes of propagations, and (b) each propagation directs messages up to a few steps in both G_h and G. Here V_t is similarly defined as its counterpart for general queries.

6. INDEXING

The remaining issue is to find transformed matches of query nodes quickly. For example, a node in Q with label "Chris Pine" shall be matched to a node in G has a label "Chris," "Pine," "C. Pine," etc. A straightforward method rewrites each label l from Q to a label set using all possible transformations, and inspects every node label in G to find matches. Obviously, scanning the entire graph is expensive. For each (or each category of) transformation, an appropriate index is needed to support fast search.

Several indices are adopted in SLQ, in accordance with the category of the transformations it supports in Table 2. Nevertheless, experimenting various kinds of indexing techniques is not the focus of this work.

(1) String index, StrIdx, is built for all the string labels in G. The index contains a list of key-value pairs $<l, S_l>$, where (a) each key is a distinct label l, and (b) S_l is a node set, such that each node v in S_l has a label l_v, such that $f_i(l, l_v) = 1$ for string transformation f_i. In other words, S_l corresponds to the matches of nodes who have labels that can be transformed from label l via string transformations. The nodes in S_l are further grouped in terms of their associated transformations to form a partition of S_l.

Let D be the set of all the labels in G. To construct StrIdx, each transformation is applied on each label of all

the nodes in G. The transformed label set is denoted as Λ, which hence forms the keys in StrIdx. For each key l, nodes with labels associated to l via a transformation are grouped as a set S_l. The pair $<l, S_l>$ is then inserted to StrIdx as an entry. One may verify that (1) the construction of StrIdx takes $O(|\mathcal{L}||D|)$ time, and (2) the space cost of StrIdx is in $O(|\mathcal{L}||\Lambda||V|)$ for at most $|\mathcal{L}|$ string transformations. SLQ does not necessarily build a specific index for each transformation. (1) Transformations can be grouped according to their category (*e.g.,* "String"), supported by a single index (*e.g.,* StrIdx). (2) Searching for some transformations, *e.g.,* Unit Conversion, can be trivially performed as direct mapping. As demonstrated in Section 8, the worst case space cost is seldom demonstrated. The index size can be further reduced by index optimization *e.g.,* [29].

(2) Semantic index, OntIdx, leverages the indexing techniques in [30] and [13], to help identify the matches based on semantic transformations, *e.g.,* Ontology and Synonym.

(3) Numeric index, NumIdx, is constructed for searching involving labels with numeric values, *e.g.,* $\leq$ 35 yrs (Range). SLQ builds NumIdx as B+ tree over numeric values.

Figure 6 illustrates the above indexing techniques. As an example, for a node v_1 with label "`Chris Pine`" in G, StrIdx performs string transformations, *e.g.,* Last token, and identifies the label "`Pine`" as a key. It then insert node v_1 into the value entry corresponding to key "`Pine`". Analogously, the nodes, *e.g.,* "`Robert Pine`" and "`Peter Pine`", in G will be mapped to the same entry, associated with key "`Pine`" and transformation Last token.

For each label l in every query node, SLQ searches for the node match candidates by looking up the label (key) in the indices StrIdx, NumIdx and SemIdx. The candidates for label l refer to all the entry values in the indices corresponding to the key l. Thus, it takes only $O(|V_Q|)$ to find the transformed match candidates. Indeed, as verified in Section 8, with the indices, the time for finding transformed match candidates accounts for less than 2% of the total search time.

7. EXTENSIONS

The architecture of SLQ can also support *typed queries*, and *partially connected queries*.

Typed queries. Users may pose explicit type constraints on queries. For example, the query node "`30 yrs`" (Figure 1) can be specified with a type "`actor`". To cope with typed queries, SLQ defines a *type feature function* for a query node v and its type s_v as

$$F_S(v, \phi(v)) = \sum_i \gamma_i f_i(s_v, s_{\phi(v)}) \quad (7)$$

with the transformations $\{f_i\}$ applied to the node types.

Partially connected queries. A partially connected query Q contains several connected components. Note that a keyword query is a case of partially connected queries. A user submits partially connected queries when he is not clear about the connection among these nodes. To cope with such queries, a new query Q' is constructed by inserting a set $\tilde{E}$ of implicit edges, where each edge $\tilde{e}$ bridges a pair of nodes from different components. An *implicit edge feature function* can be readily introduced as

$$F_{\tilde{E}}(\tilde{e}, \phi(\tilde{e})) = \sum_i \delta_i f_i(\tilde{e}, \phi(\tilde{e})) \quad (8)$$

Both $F_S(v, \phi(v))$ and $F_{\tilde{E}}(\tilde{e}, \phi(\tilde{e}))$ can be plugged into the ranking function Eq. 3, where $\{\gamma_i\}$ and $\{\delta_i\}$ can be learned using the same training strategy.

8. EXPERIMENTAL EVALUATION

In this section, we perform a set of experiments using real-life large graphs, to demonstrate SLQ framework in terms of effectiveness, efficiency and scalability.

8.1 Experimental Settings

Datasets. We use the following three datasets. (1) *DBpedia* [1] is a knowledge base. Each node represents an entity associated with a set of properties, (*e.g.,* `name`='california', `type`='place', `area`='163,696 sq mi'). The labeled edges indicate various relationships. (2) *YAGO2* [3] is a knowledge base gathered from several open sources. Similarly as *DBpedia*, its nodes and edges preserve rich information. (3) *Freebase* [2] is a collaboratively created graph base that has over 40M topics (nodes) and 1.2B facts. As public repositories, these graphs are maintained by multiple communities, containing highly diverse and heterogeneous entities, attributes and values. The following table gives a summary.

Graph	Nodes	Edges	Node types	Relation	Size
DBpedia	3.7M	20M	359	800	40G
YAGO2	2.9M	11M	6,543	349	18.5G
Freebase	40.3M	180M	10,110	9,101	88G

Transformations. Our system integrated all of the transformations in Table 2 including ontology [30]. More transformations can be seamlessly adopted.

Queries. In the experiments, two sets of query benchmarks are employed. (1) The *DBPSB* benchmark [19] is derived from *DBpedia*. The benchmark is a set of 25 query templates that are originally expressed in *SPARQL* format. The templates resemble real query workload and cover queries with different complexity. The queries can be converted to graph queries. (2) The templates in *DBPSB* have limited types (e.g., "Person") and simple topology (*e.g.,* tree). We hence designed a second set of 20 templates that explore more diverse topics and complex (*e.g.,* cyclic) graph structures.

In offline learning, query templates are generated by instantiating the query benchmarks with the labels from the data graphs. Here a label can be any property of the corresponding entity. These instances also serve as ground truth for the queries. We then perform transformations on randomly selected labels in each query instance, which yield training queries. We show three such queries and their matches in Figure 7. Query 1 is to find an athlete in football team "San Francisco 49ers" who is about 30 years old. Query 2 is to find a person who served in the Union army and attended a battle, and these information *maybe* related with "Missouri". Query 3 identifies a current US senator at his 60 who lives in "NJ" and knows "F. Lautenberge".

Algorithms. We chose the CRFs model as defined in Eq. 3 and developed SLQ in Java. For comparison, the following algorithms are also developed with the best effort.

Baselines. To compare the match quality, we consider the following state-of-the-art techniques. (1) Spark [18], a keyword-based search engine. It supports IR-style ranking heuristics. Since Spark only supports exact string matching, we modified it to accept transformed matches. Spark does

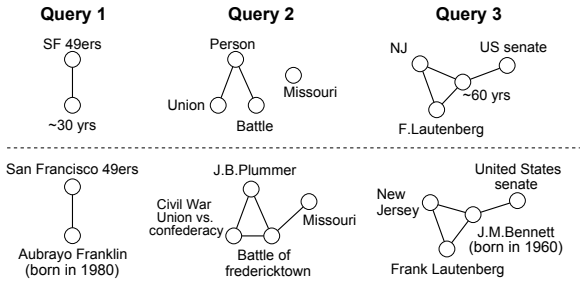


Figure 7: Case study: querying knowledge graphs

not consider edge information in a query as it is keyword oriented. (2) Unit is a variant of SLQ. The only difference is it uses a revised ranking model with equal weight for all the transformations; (3) Card also implements SLQ, while revises its ranking model with weights equal to the selectivity of the transformations as $\frac{1}{card(f)}$. Here $card(f)$ refers to the average size of the matches for a randomly sampled node (or edge) in the graph using transformation f.

For efficiency comparison, we compare SLQ with (1) Exact, which enumerates all possible matches based on the subgraph search algorithm [14, 21], and then rank them with the learned ranking model. This strategy ensures that all the matches including the ground truth can be obtained and ranked. (2) Approximate searching in NeMa [12]. The method directly applies a propagation strategy similar to LoopyBP over data graphs. Note that NeMa only extracts the most probable result, *i.e.,* top 1 match. We enhanced it by applying the techniques in [31] to identify top-k results. For fair comparison, the above baselines are also equipped with our predefined transformations and the indices.

Metrics. Given a query workload $\mathcal{Q}$ as a set of queries Q, we adopt several metrics for the rank evaluation: (1) Precision at k ($P@k$), the number of the top-k answers that contain the ground truth; (2) Mean Average Precision (MAP @k), which means $\text{MAP@}k = \frac{1}{|\mathcal{Q}|}\sum_{Q\in\mathcal{Q}}\frac{1}{k}\sum_{i=1}^{k}\frac{AveP(i)}{R}$, where $AveP(i) = \frac{1}{i}$ when the ith result is a true answer and $AveP(i) = 0$, otherwise. R is the number of the answers; (3) Normalized Discounted Cumulative Gain (NDCG@k), as $NDCG@k = \frac{1}{|\mathcal{Q}|}\sum_{Q\in\mathcal{Q}} Z_k \sum_{i=1}^{k}\frac{2^{r_i}-1}{\log_2(i+1)}$, where r_i is the score of the result at rank i. Following convention, we set r_i as 3 for the *good* match, 1 for the *relevant* match and 0 for the *bad* match. Z_k is a normalization term to let the perfect ranking have score 1. We also tested other metrics, such as SoftNDCG [25]. They share similar intuition and thus are not elaborated. In the experiments, unless otherwise specified, each query workload refers to 1,000 randomly generated queries using different query templates. Note that the set of training queries is different from that for testing.

Setup. We compressed each data graph, *e.g.,* same predicates in the RDFs, and built index based on the transformations. The indexing time and the size is: 61.8min/1.02GB (DBpedia), 37.4min/0.78GB (YAGO2), 263min/12.91GB (Freebase). All the experiments were performed on a machine with Intel Core i7 2.8GHz CPU and 32GB RAM. For each test, we report the average value over 5 runs.

8.2 Case Study: SLQ vs. IR-based search

We provide a case study using *DBpedia*. Consider the three queries in Figure 7. For each query, SLQ identifies meaningful matches of high quality. For example, for Query 2, a historical figure, Colonel J.B. Plummer, is identified to match `Person` who fought in the Battle of Fredericktown during the Civil War in Missouri. Our framework is able to tell the importance of different transformations: for `Person`, *Ontology* is a proper transformation; while for `Union`, *Bag of words* is promoted in the ranking. `Missouri` is selected as an exact match. In addition, the match suggests a direct connection between Missouri and Battle in Query 2, indicating a *refinement* of Query 2 in future. In all cases, Spark gives low IR score and cannot identify matches for Query 2.

8.3 Experimental Results

Exp-1: Manual evaluation. We first conduct manual evaluation on 75 queries that are randomly constructed from the three datasets. 10 students help evaluate the results returned by our search algorithm. For each result, a label, *i.e., Good*, *Relevant* or *Bad*, was assigned by the students regarding the query. The labels are thus considered as the ground truth. The students were not trained beforehand and thus the labels were assigned merely based on their intuition. The metric, $NDCG@k$, can be calculated based on the rank order of the results and the corresponding labels. Table 3 presents the quality of top-5 returned answers. The result confirms that SLQ shows a substantial improvement over the baselines. In terms of answer quality, it is very close to the exhaustive search algorithm, Exact. On the other hand, SLQ is up to 300 times faster than Exact (see Exp-3 for query processing time comparison).

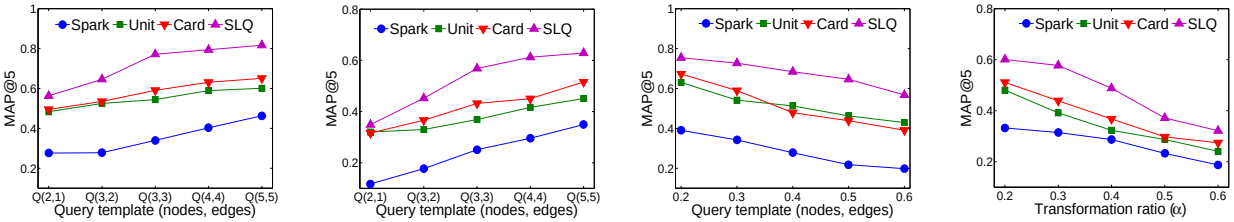
Graph	Spark	Unit	Card	SLQ	Exact
DBpedia	0.707	0.790	0.858	0.935	0.935
YAGO2	0.682	0.849	0.852	0.926	0.928
Freebase	0.636	0.751	0.768	0.859	0.865

Table 3: Manual evaluation ($NDCG@5$)

In the following experiments, we verify the performance of our algorithms by varying query size and transformation ratio. Since finding good matches manually is very costly, we focus on two kinds of intuitively good matches: the original subgraph from which a query is transformed from, and all the identical matches of the query. A good algorithm shall at least rank these good matches as high as possible.

Exp-2: Effectiveness of ranking. This experiment examines the answer quality of SLQ. There are several factors, such as query size, query topology, transformation ratio and data graph, that may affect the ranking. We first study the impact of the query size and topology while fixing the others. The following test is based on the evaluation of the query workload that are randomly sampled from the data graph *w.r.t.* the query templates. Each query is modified by applying random transformations with the ratio $\alpha = 0.3$. The ranking model was trained beforehand for each graph (see Exp-4 for the report on offline training).

Given the queries and the corresponding results, we employ $MAP@k$ as the metric to evaluate the rank and plot the scores in Figure 8(a-b) for *DBpedia* and *YAGO2*, respectively. The results tell us that the methods Unit, Card and SLQ significantly outperform the IR based technique, Spark. The ranking model in Spark only considers a linear



(a) Varying query size: DBpedia (b) Varying query size: YAGO2 (c) Varying ratio: DBpedia (d) Varying ratio: YAGO2

Figure 8: Effectiveness of ranking ($MAP@5$)

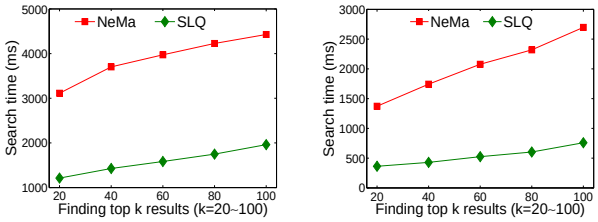
combination of keywords' IR scores. It does not take the selectivity of different transformable conventions and the relations (connections) of the keywords into account. SLQ also achieves better ranking result than its two variants, Unit and Card, indicating that automatic learning of transformation weights could improve answer quality. Figure 8(a-b) also show that when the query size increases, the score increases for all the methods. This is due to the fact that a query with larger size provides more evidences, which help identify good matches easily. This phenomenon implies great potential of the schemaless and structureless querying model: As long as a user provides enough evidences, she can find the answer even her query does not fully comply with the schema and the structure of the underlying graph database.

We next validate the ranking quality with respect to the transformation ratio α. In this test, we derive a set of query workload by varying α of the queries from 0.2 to 0.6. Intuitively, to raise the transformation ratio will increase the "ambiguous" level of the query, making it more difficult to find the true match in the top-k matches. The result is depicted in Figure 8(c-d). As expected, the performance of all the algorithms degrades along with the increase of α. However, our algorithm is still the best. We also examined the performance of Exact, which is slightly better (by $\leq 1\%$) than SLQ and thus is not shown in Figure 8 for simplicity.

Exp-3: Efficiency of top-k search. In this experiment, we demonstrate the runtime improvement of SLQ over Exact and NeMa. SLQ employs graph sketch to quickly skip the low-quality matches. We choose $k = 20$, and use the same query workload as in the previous experiment. The runtime examined here also contains the index search time, which accounts for less than 2% of the total time and thus is not analyzed separately. The runtime of Unit and Card is not reported as it is close to that of SLQ.

Figure 9(a-b) shows the runtime with varying query size, and fixed transformation ratio (0.3). For both graphs, SLQ and NeMa are 5-50 times faster than Exact. This advantage is achieved by top-k search and the merit of approximate search (LoopyBP). Meanwhile, SLQ is 2-4 times faster than NeMa. It implies the graph sketch method can indeed avoid some unnecessary verification. We also evaluate the runtime of SLQ by varying the transformation ratio from 0.2 to 0.6. Figure 9(c-d) show a clear advantage of SLQ over other approaches. For most queries, SLQ can finish the execution within 1 second. Its runtime can further be reduced by employing a multi-thread implementation.

Figure 10(a-b) plots the search time with different k values for *DBpedia* and *YAGO2*, respectively. The test queries are randomly generated with transformation ratio $0.2 \sim 0.6$. Our algorithm again demonstrates outstanding performance



(a) Top-k search: DBpedia (b) Top-k search: YAGO2

Figure 10: Search time: effect of k ($20 \sim 100$)

on runtime, which is up to 1/3 of the time by NeMa.

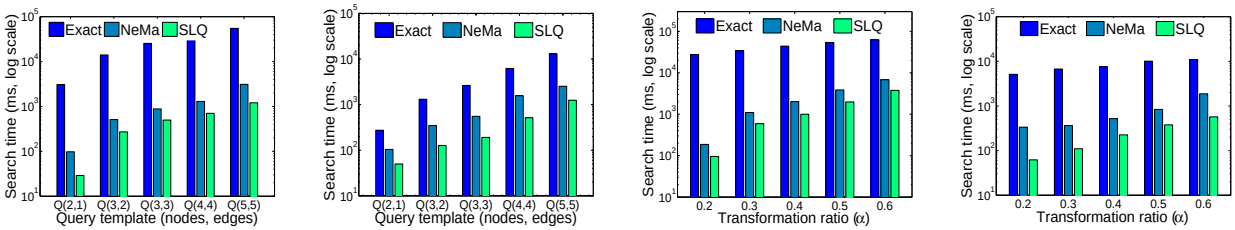
Exp-4: Offline learning. We study the impact of sample size and offline training on the quality of ranking. Recall that the training queries along with the ground truth are randomly sampled from the graph, the coverage of the queries plays a pivotal role in the training. The coverage of a query workload $\mathcal{Q}$ is defined as $C(\mathcal{Q}) = \frac{|\bigcup_{Q \in \mathcal{Q}} Q|}{|G|}$. Since the graphs are highly heterogeneous, we speculate with larger coverage, the learned model would have a better ranking result. To inspect the effect, we conduct two tests with different workload coverage: $0.5\% \sim 2.0\%$ (*DBpedia*) and $0.05\% \sim 0.2\%$ (*Freebase*). The queries in each training workload are generated from randomly selected query templates.

DBpedia			Freebase		
Sample	Time	$P@5$	Sample	Time	$P@5$
0.5%	795s	0.650	0.05%	1,695s	0.685
1.0%	1,588s	0.715	0.1%	3,125s	0.712
2.0%	3,028s	0.722	0.2%	5,828s	0.725

Table 4: Training: sample coverage

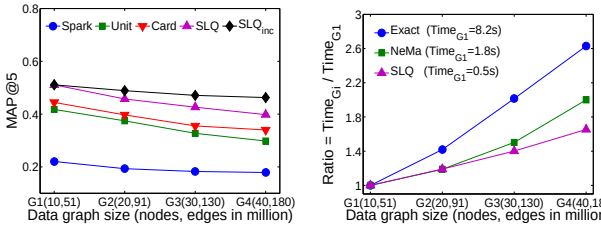
The training time and the quality of ranking ($P@k$) are shown in Table 4. The transformation ratio for each training set is controlled by a 5-fold *cross validation*. Note the test queries are different from those for training. For both of the two datasets, the training time is nearly linear *w.r.t.* $C(\mathcal{Q})$. It can be seen that with higher coverage, we can achieve a clear better ranking performance, with the cost of extra training time. For *DBpedia*, when the coverage increases from 1.0% to 2.0%, the improvement is marginal, *i.e.,* $\leq 1.0\%$. The same effect can be observed for *Freebase*, when we increase the coverage from 0.1% to 0.2%. The experiment validates that only a small sample of the raw data for offline training is enough for good performance.

Exp-5: Scalability w.r.t. graph size. We next evaluate the scalability of SLQ by varying the size of the Freebase



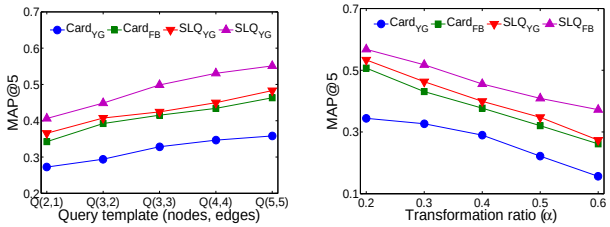
(a) Varying query size: DBpedia (b) Varying query size: YAGO2 (c) Varying ratio: DBpedia (d) Varying ratio: YAGO2

Figure 9: Efficiency of top-k search ($k = 20$)



(a) Effectiveness: MAP@5 (b) Efficiency: search time

Figure 11: Scalability evaluation: Freebase



(a) Varying query size (b) Varying ratio

Figure 12: Cross query evaluation on Freebase

graph. Specifically, we initialize a subgraph G_1 from Freebase with size $(10M, 51M)$ (*i.e.,* 10M nodes and 51M edges) and gradually grow it to $G_4(40M, 180M)$. This setting will test the performance of SLQ in a streaming mode. Figure 11(a) depicts the result. Specifically in the figure, SLQ shows the performance of the ranking model trained only based on the initial graph (G_1), while SLQ_{inc} shows the performance of the ranking model with incremental update based on the growing graph. The test queries are generated separately for each graph. With the growing of the graph, the rank performance generally decreases since there are more data to confuse top-k ranking. Among the four algorithms, SLQ_{inr} is the best. Moreover, although it degrades *w.r.t.* SLQ_{inr}, SLQ still outperforms the other methods dramatically, indicating a comparatively stable result.

In terms of search time, to illustrate the significant time difference, we plot the runtime increasing ratio, $\frac{Time_{G_i}}{Time_{G_1}}$, in Figure 11(b), for top-k search ($k = 20$). All the algorithms take more time for searching larger graphs. Moreover, despite the significant difference on the search time on G_1, *i.e.,* $Time_{G_1}$ as shown in the legend of Figure 11(b), SLQ achieves near-linear runtime increase regarding the size of the graph. It takes up to 25% of the time by NeMa and is at least one order of magnitude faster than Exact. We also inspect the runtime of SLQ_{inr}. Recall that the model in SLQ_{inr} is continuously updated, the training time is negligible. With the setting of 0.1% training sample coverage and 1000 test queries, the amortized runtime of SLQ_{inr} is at least as twice as that of SLQ and thus is not shown in Figure 11(b) for simplicity.

Exp-6: Training on YAGO2 and querying Freebase. Finally, we do a bold experiment: Can we apply the model trained on one graph and query another graph? To answer this question, we test the model trained on YAGO2 by ranking the results of queries on Freebase (SLQ_{YG}), and compare it with the models trained on Freebase (SLQ_{FB}). Figure 12(a) reports the result by varying the query size. The transformation ratio is 0.3. The model in SLQ_{YG} is the same as that trained for YAGO2 in Exp-2. It shows SLQ_{YG} still works, and is even slightly better than Card_{FB}. This is a strong evidence showing that the knowledge learned (the weights of different transformations) can be transferred between different graph databases. A similar result is also observed when we vary the transformation ratio, as shown in Figure 12(b).

9. RELATED WORK

Graph searching is studied for structured queries (*e.g.,* XQuery, SPARQL), keyword queries [11, 14, 27] and graph pattern queries (*e.g.,* [5]). These methods focus on fixed schemas and ranking functions. To relax the constraints of schema and structure, approximate matching is studied, for *e.g.,* graph pattern matching [7,12], and for keyword queries over knowledge graphs [11]. The searching semantics are relaxed to identify more meaningful matches with similar structures or similar attributes to a given query.

Closer to our work is NeMa [12] and NAGA [11]. (1) NeMa defines node similarity by comparing the neighborhood similarity of two nodes, and iteratively infer the matching quality using similarity propagation as in a graphical model. (2) NAGA supports keyword querying over the YAGO knowledge base. It defines match quality with confidence, informativeness and compactness, and ranks the answers based on probabilistic models, where the parameters in the ranking model are tuned by users. Nevertheless, all of these studies use predefined ranking metrics. This significantly limits the power of these methods as it is hard to justify them. Our work shows that a ranking model shall be learned automatically through the existing queries and their associated answers, not given beforehand.

Top-k search is also extensively studied in the database community. Fagin's algorithms [9] read attributes from sorted lists, construct tuples, and perform random access to find missing scores. They stop when k tuples are constructed from the top-ranked attributes that have been seen, thus allowing early termination with approximate top-k matches. Top-k graph searching is studied for *e.g.,* twig queries [10]

and graph patterns [7]. These algorithms are developed for fixed schemas and vocabularies. In contrast to these studies, we do not assume sorted list of matches and monotonic ranking functions. Moreover, how to select a proper ranking metric is not discussed in these studies.

Machine learning techniques are leveraged to find matched entity pairs by combining multiple similarity metrics. For example, weights of various transformation rules are learned for object identification [26]. These methods differ from ours in the following. (1) Time-consuming manual labeling and training data. In contrast, our system requires no manual effort for generating training examples. (2) Homogeneous data. Thus, they can not be easily extended to deal with heterogeneous graphs as studied in this work.

There are several other topics complementary to this work. Query disambiguation [20] is an effort to identify the search intent from the query context. Query interpretation [8] provides a user with multiple plausible interpretations of a query. These techniques can play as add-ons for our framework to further improve the result quality.

10. CONCLUSION

We identified a key problem that frustrates nonprofessional users for accessing emerging graph databases. We argued that a user-friendly query engine must support various kinds of transformations directly, such as synonym, abbreviation, and ontology. We developed a novel searching framework, SLQ, to (a) learn a ranking model that combines multiple transformations, which does not require manually labeled training instances; and (b) efficiently find top-k matches for graph and keyword queries. As verified by our experiments, SLQ achieves much better query results in comparison with the existing approaches, and is able to process queries quickly. Better still, SLQ can be readily extended to integrate new transformations, indices and query logs. Surrounding this new query paradigm, there are a few emerging topics worth studying in future, *e.g.,* comparison of different probabilistic ranking models, compact transformation-friendly indices, and distributed implementation.

11. ACKNOWLEDGMENTS

This research was sponsored in part by the Army Research Laboratory under cooperative agreements W911NF-09-2-0053 and NSF IIS-0954125. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

12. REFERENCES

[1] Dbpedia. *dbpedia.org*.
[2] Freebase. *freebase.com*.
[3] Yago2. *mpi-inf.mpg.de/yago*.
[4] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.
[5] P. Barceló, L. Libkin, and J. L. Reutter. Querying graph patterns. In *PODS*, pages 199–210, 2011.
[6] D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001.
[7] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.
[8] E. Demidova, P. Fankhauser, X. Zhou, and W. Nejdl. Divq: diversification for keyword search over structured databases. In *SIGIR*, pages 331–338, 2010.
[9] R. Fagin. Combining fuzzy information from multiple systems. *JCSS*, 58(1):83–99, 1999.
[10] G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, pages 581–594, 2008.
[11] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.
[12] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: fast graph search with label similarity. In *VLDB*, pages 181–192, 2013.
[13] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
[14] G. Li, B. 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*, pages 903–914, 2008.
[15] P. Li, Q. Wu, and C. J. Burges. Mcrank: Learning to rank using multiple classification and gradient boosting. In *NIPS*, pages 897–904, 2007.
[16] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.
[17] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *SIGMOD*, pages 373–384, 2013.
[18] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, pages 115–126, 2007.
[19] M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo. Dbpedia sparql benchmark – performance assessment with real queries on real data. In *ISWC*, pages 454–469, 2011.
[20] R. Navigli. Word sense disambiguation: A survey. *CSUR*, 41(2):10, 2009.
[21] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *PAMI*, 26(10):1367–1372, 2004.
[22] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *TODS*, 34(3):16:1–16:45, 2009.
[23] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, 2013.
[24] C. Sutton and A. McCallum. An introduction to conditional random fields for relational learning. *Introduction to statistical relational learning*, 93:142–146, 2007.
[25] M. Taylor, J. Guiver, S. Robertson, and T. Minka. Softrank: optimizing non-smooth rank metrics. In *WSDM*, pages 77–86, 2008.
[26] S. Tejada, C. A. Knoblock, and S. Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *KDD*, pages 350–359, 2002.
[27] H. Wang and C. Aggarwal. A survey of algorithms for keyword search on graph data. *Managing and Mining Graph Data*, pages 249–273, 2010.
[28] Y. Weiss and W. T. Freeman. On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs. *Information Theory*, 47(2):736–744, 2001.
[29] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, pages 219–232, 2009.
[30] Y. Wu, S. Yang, and X. Yan. Ontology-based subgraph querying. In *ICDE*, pages 697–708, 2013.
[31] C. Yanover and Y. Weiss. Finding the m most probable configurations using loopy belief propagation. In *NIPS*, 2004.