GQBE: Querying Entity-Relationship Graphs by Example Tuples

ABSTRACT

We see an unprecedented proliferation of entity data graphs that capture entity properties and relationships in many applications. Such data is difficult to use. The challenges lie in the gap between large and complex graphs and non-expert users. If writing structured queries over "simple" tables is difficult, complex graphs are only harder to query. As an initial step toward building usable query systems for entity-relationship graphs, we propose to query such data by example entity tuples, without requiring users to write complex graph queries. To the best of our knowledge, there has not been such proposal in the past.

Our system GQBE tackles the challenges in supporting this query approach. It derives a hidden maximal query graph based on input query tuples, to capture a user's query intent. Edges in the maximal query graph are weighted by several heuristic ideas. Given the derived maximal query graph, there can be a large space of approximate answer tuples, thus it is imperative for our system to efficiently find the top ranked answers based on how well they match the query tuple. Our top-*k* query algorithm guarantees to evaluate only the query graphs that are necessary for obtaining top-*k* answers. We conduct experiments on the large Freebase dataset to evaluate the accuracy and efficiency of our system.

1. INTRODUCTION

As our society enters the era of big data, we see an unprecedented proliferation of *entity-relationship graphs* in many computing domains. In an entity-relationship graph, nodes represent entities (e.g., persons, products, organizations) and edges represent their relationships. As a concrete example, Figure 1 is an excerpt of an entity-relationship graph. For instance, the node Stanford represents the Stanford University and Jane Stanford a person who *founded* that university. The edges labeled *headquartered_in* are between several pairs of entities such as (Apple Inc., Cupertino), representing that Apple Inc. has a headquarter in Cupertino. Real-world instances of such graphs include knowledge bases (e.g., DBPedia [1], YAGO [25], Freebase [4]), social networks, citation graphs (e.g., DBLP, PubMed, gene and protein databases (e.g., UniProt, Protein Data Bank, Genbank, U.S. government data, mobile networks, and program analysis graphs, to name just a few.

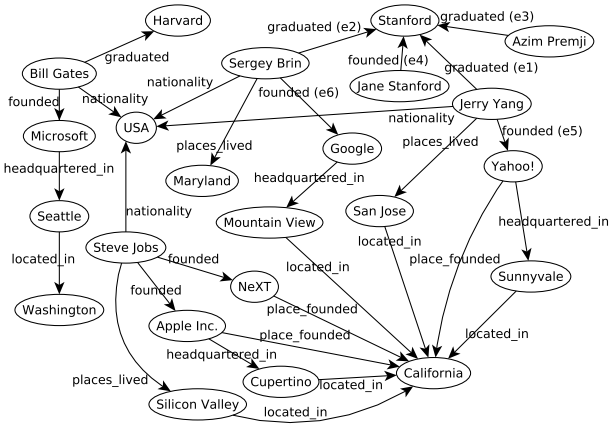


Figure 1: An Excerpt of An Entity-Relationship Graph.

Users and developers are trying hard to tap into the large amount of entity-relationship graph data for numerous applications. Google's Knowledge Graph, for instance, enhances users' search experience using data from Freebase as well as its proprietary search logs and Web index. It echoes the shift of Web users' interest towards entity-related information that is evidenced by estimation that 71% of Microsoft Bing search queries contain named entities [33]; Hunch.com's "taste graph" enables personalized recommendations of things based on people's social relationships and shared tastes; Citizens and journalists use data on government spending, campaign finance, congress voting record, and census to answer questions such as "Who graduated from the same university as a US president and worked at a company that received government funding during the president's term?" [3]; Scientific fields such as biology become data-driven and scientists face sheer challenges in analyzing big biology entity graphs. For example, given a DNA sequence and a protein, a biologist can find out if the protein is involved in an interaction with another protein that is encoded by the DNA sequence [2].

Entity-relationship graphs are difficult to use, for both users and application developers. This largely has to do with the sheer size and complexity of such data. As of March 2012, the Linking Open Data community had interlinked over 52 billion RDF triples spanning over several hundred datasets. Freebase alone has over 22 million entities and 350 million relationships in about 100 domains. Before users and developers can do anything meaningful with the data, they often are overwhelmed by the daunting task of at-

tempting to digest and understand it.

More specifically, the challenges lie in the gap between complex data and non-expert users. While a number of graph database systems, triplestores, and RDF stores have emerged in recent years (e.g., Neo4j, Pregel [21] , GBase [16] , AllegroGraph, and many others), *usability* has not been the focus of innovation. In retrieving data from these databases, the norm is often to use structured query languages such as SQL, SPARQL, and those alike. In the literature on graph querying (cf. a comprehensive tutorial in [20]), the starting point is virtually always a query graph, which is a graphical representation of structured query. However, writing structured queries is hard. It requires extensive experiences in query language and data model and good understanding of particular datasets. For this very reason, database usability has received considerable attention lately. (See an excellent overview in [13].) Graph data is not easier than relational data in either query language or data model. The fact that it is schema-less makes it even more intangible to understand a data graph and express queries on it. *If querying "simple" tables is difficult, is not that complex graphs are harder to query?*

We aim to tackle the challenges in building *usable* query systems for entity-relationship graphs. As an initial step toward this objective, in this paper we propose to query graphs by example entity tuples, without requiring users to write complex graph queries. The paradigm of *query by example* (QBE), though not seen before in querying graph data, has a long history in relational databases [34]. The idea is to let users express queries by filling example tables with variables and constants in various fields, where constants indicate selection conditions, and shared variables in multiple tables are for join conditions. Its simplicity and improved user productivity make it an influential query language in database systems. By proposing to query entity-relationship graphs by example tuples, our conjecture is that it will enjoy similar advantages and success. The technical challenges and approaches are vastly different, due to the fundamentally different data models.

Specifically, given a data graph and a query tuple consisting of entities, our goal is to find similar answer tuples. To understand the concept, consider the data graph in Figure 1 again. Suppose a user provides a 2-entity query tuple ⟨Jerry Yang, Yahoo!⟩. The system can return answer tuples such as ⟨Sergey Brin, Google⟩, ⟨Steve Jobs, Apple Inc.⟩ and ⟨Bill Gates, Microsoft⟩, since they are all pairs of ⟨PERSON, COMPANY⟩ and have the same relationship–the person founded the company, captured by edges labeled *founded*. If the query tuple consists of 3 or more entities (e.g., ⟨Jerry Yang, Yahoo!, Stanford⟩), the answers will be similar tuples of the same number of entities.

There are several challenges in building a query system with such capability. To start with, the input is an entity tuple instead of a query graph. The system needs to figure out what are important "features" of the query tuple that need to be "matched" in answer tuples. More concretely, the entities in the query tuple are vertices in a data graph. These entities and other entities in their neighborhood are related to each other in certain ways. For an answer tuple, the answer entities and their neighboring entities are expected to be also related in such ways. Hence, with regard to *query semantics*, our system needs to derive a hidden query graph, based on the query tuple, to represent a user's query intent. Edges in the hidden query graph need to be weighted, in order to capture the importance of different relationships in the neighborhood of the query tuple entities.

The answers to such a hidden query graph are approximate by nature, because no answer tuple can really match the query tuple exactly (otherwise they become identical). This means that there could be many answer tuples and they match the query tuple to different extents. This also implies that the answer tuples must be ranked, by how well they match the query tuple. In the aforementioned example, answer tuple ⟨Sergey Brin, Google⟩ intuitively is a more accurate answer than others, because both Larry Page and Jerry Yang are graduates of Stanford, and both Google and Yahoo! are headquartered in California. For similar reasons, ⟨Steve Jobs, Apple Inc.⟩ perhaps should be considered a better answer than ⟨Bill Gates, Microsoft⟩ because Apple Inc. is also headquartered in California, although they are both weaker answers than ⟨Sergey Brin, Google⟩. Even with regard to the edges labeled *founded*, it is not necessary that the two entities in an answer tuple must bear such an edge between them. There can be multiple direct and indirect paths between two entities. The entities in query and answer tuples only need to have some paths in common. In summary, with regard to *query processing*, there can be a large space of approximate answer tuples, thus it is imperative for our system to find the top ranked answers efficiently.

In this paper, we describe GQBE, a system that queries entity-relationship graphs by example tuples. GQBE has two major components, as follows. (1) *Query graph discovery*: The system derives a hidden query graph, termed *maximal query graph*, from a query tuple. The maximal query graph is a subgraph of the *neighborhood graph* which consists of vertices and edges within the vicinity of the query tuple entities. Edges in the maximal query graph are weighted by several heuristic ideas, including that edges rare in both globally the data graph and locally the neighborhood of an entity are more important with regard to that entity, and that edges closer to query tuple entities are more important. Intuitively, the maximal query graph captures important features of the query tuple that require to be matched by answer tuples. Any of its subgraphs containing all query tuple entities is a query graph. A query graph and its corresponding answer graphs are isomorphic subgraphs of the data graph, barring that their entities (vertices) may not match. Given an answer graph, those entities in it that correspond to query tuple entities form an answer tuple.

(2) *Query processing*: The query graphs, which are all different subgraphs of the maximal query graph, form a *query graph lattice* based on their subsumption relationship. This lattice can be large. To obtain top-*k* ranked answer tuples, the brute-force approach of evaluating all query graphs in the lattice can thus be prohibitively expensive. To tackle this challenge, we propose an algorithm that guarantees to only evaluate the query graphs that are *necessary* for obtaining the top-*k* answers. Without delving into details, the gist of the idea is to explore the lattice in the order of the query graphs' upper-bound scores. The lower-bound score of a lattice node (i.e., a query graph) is the total weight of edges in it, representing the score of its corresponding answer graphs. The upper-bound score of a node is the highest lower-bound score among its unpruned ancestors. The algorithm starts from the bottom of the lattice, called *minimal query trees*, which are the smallest query graphs. In a bottom-up fashion, the algorithm always chooses to evaluate the smallest query graph with the highest upper-bound score. If evaluating a node produces empty result, which means exact match of the corresponding query graph does not exist, the node and all its ancestors (i.e., super-query graphs) are pruned. The upper-bound scores of the pruned nodes' descendants are recalculated. The algorithm terminates when it has obtained at least *k* answer tuples with scores higher than the highest possible upper-bound score among all unevaluated nodes.

Contributions. In summary, our work makes the following contributions:

- For better usability of graph querying systems, we propose the novel approach of querying entity-relationship graphs by exam-

ple entity tuples, which saves users the burden of forming explicit query graphs. To the best of our knowledge, there has not been such proposal in the past. Our system GQBE tackles the challenges in supporting this query approach.
- The query graph discovery component of GQBE derives a hidden maximal query graph based on input query tuples (Section 4). Previous works on finding subgraphs to capture the relationships between input nodes fall short in our problem scenario (cf. Section 2).
- The query graph lattice based on the derived maximal query graph may contain a large number of query graphs. The brute-force approach of evaluating all query graphs in the lattice can be prohibitively expensive. Our top-*k* query algorithm guarantees to only evaluate query graphs that are *necessary* for obtaining the top-*k* answers.
- We conduct experiments on the large Freebase dataset to evaluate the accuracy and efficiency of our system.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the data model and query model, and it provides the problem statement. We describe the query graph discovery component of GQBE in Section 4 and the query processing component in Section 5. Section 6 presents experimental evaluation results. In Section 7, we conclude the paper.

2. RELATED WORK

Our work is the first to query an entity-relationship graph based on example entity tuples. It differs from prior related works in several aspects, as described below.

Due to the very different data and query model, the work is a departure from prior study on database usability that mainly focuses on relational databases. With regard to graph databases, substantial progress has been made on various intuitive query mechanisms that do not require explicit query graphs or help users construct query graphs. Such mechanisms include keyword search in graphs [11, 29, 17] that finds substructures of a graph containing query keywords, keyword-based query formulation [23, 32] that formulates graph queries from keywords, natural language questions [31], interactive and form-based query formulation [7, 14], and visual interface for query graph construction [6, 15].

The proposed query mechanism is related to *set expansion*, where the goal is to expand a set of objects by seed objects. Examples of set expansion systems include SEAL [30], SEISA [12], and the now discontinued Google Sets and Squared services [1]. Chang et al. [5] find the top-*k* correlated keyword terms from an information network given a set of keyword terms, where each term can be an entity. GQBE is different from this line of work on two main aspects– (1) While these systems, except [5], find existing answers within structures in web pages such as HTML tables and lists, GQBE works on data graphs. (2) Our work is more general in that each input query tuple in GQBE contains multiple heterogeneous entities, whereas these systems, except Google Squared, take multiple input objects in which each input is a single entity.

There are several works [27, 18, 24, 8] that identify the best subgraphs/paths in a data graph to describe how several input nodes are related. The query graph discovery component of GQBE is different from these works in several important ways– (1) Some of these works [27, 24] focus on homogeneous graphs where all nodes are of the same type and all edges represent instances of the same relationship, e.g., social networks representing friendships between people and co-authorship graphs representing co-authoring relationships between authors. What are similar to homogeneous graphs are those with only a few different types of entities and relationships. GQBE and others [18, 8] focus on heterogeneous graphs with many different types of entities and relationships. (2) The graphs identified by these works contain only those paths that connect the input nodes, but the hidden query graphs discovered in GQBE include also relationships pertinent to individual query tuple entities. The REX system [8] further has the limitation of allowing only two input entities. (3) GQBE uses the discovered query graph to find answer graphs and answer tuples, which is not within the focus of the aforementioned works.

The query processing component of GQBE is related to the problem of inexact subgraph matching, which identifies all approximate occurrences of a query graph in a data graph. There have been a large number of studies on inexact subgraph matching in large graphs, including G-Ray [28], TALE [26], NESS [19] and many other works. (See [10, 20] for surveys.) We note that GQBE is different from these works, on several aspects. First, GQBE requires to match only edge labels, but node labels are not required to be always matched. This is equivalent to matching a query graph with all unlabeled nodes, and thereby significantly increases the problem complexity. Only a few previous graph querying methods, e.g., NESS [19] allow unlabeled query nodes. Second, in GQBE, query tuple entities have fundamentally higher significance than other entities in the maximal query graph, therefore the top-*k* query algorithm is centered around query tuple entities. More specifically, edges in the maximal query graph are weighted by their distances to query tuple entities, and an answer graph must have entities corresponding to all query tuple entities. On the contrary, previous methods do not consider such priority treatment of query tuple entities. They give equal importance to all nodes in a query graph. Our empirical results show that this difference makes NESS produce less accurate answers than GQBE. Finally, the works on subgraph matching assume explicit query graphs as input while GQBE derives hidden query graphs from input query tuples.

3. DATA MODEL, QUERY MODEL, AND PROBLEM STATEMENT

In this section, we formally define the data model, query model, and problem statement of GQBE and we provide examples to explain various concepts.

Definition 1 (Entity-Relationship Graph) An *entity-relationship graph*, or simply a *data graph*, is a directed multi-graph $G(V, E)$, with node set V and edge set E. We also use $V(G)$ and $E(G)$ to denote the node set and the edge set, respectively. Each node $v \in V$ represents an entity. Each node has a unique identifier $id(v)$. [2] Each edge $e=(v_i, v_j) \in E$ denotes a directed relationship from entity v_i to entity v_j. Each edge has a label, denoted as $label(e)$. Multiple edges can have the same label. ■

Definition 2 (n-Tuple) Given an entity-relationship graph G, an *n-tuple* $t=\langle v_1, v_2, \ldots, v_n \rangle$ is an ordered list of n entities, where each entity is a node in G. ■

An n-tuple provided as user input is referred to as a *query tuple*. The entities in a query tuple are called the *query tuple entities* or simply *query entities*. Our goal is to find the top-*k* most "similar" n-tuple answers to the query tuple. Given a data graph G, the set of possible answer tuples is the n-ary Cartesian product over the entire node set V, i.e., V^n. The answer set can be prohibitively massive

[1] http://en.wikipedia.org/wiki/List_of_Google_products

[2] Without loss of generality, we use an entity's name as its identifier in presenting examples, assuming entity names are unique.

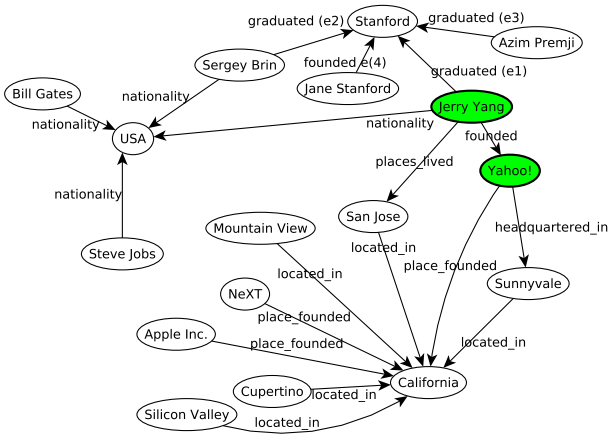


Figure 2: Neighborhood Graph for ⟨Jerry Yang, Yahoo!⟩.

under a large G. Hence, the key to our approach is to design an appropriate tuple-wise similarity measure. We define similarity between a query tuple and a candidate answer tuple by matching the relationships between entities in the two tuples. Observe that entities not only have direct relationships between them, but also have indirect relationships through other entities. Therefore, the essence of the problem is the matching between two graphs, constructed from the query tuple and the answer tuple, respectively. In this context, we define the *neighborhood graph* for an n-tuple, which is based on the concept of undirected path.

An *undirected path* in a directed graph is a path whose edges are not oriented in the same direction. Unless otherwise stated, we will refer to undirected path simply as "path". We consider undirected path instead of directed path because both incoming edges and outgoing edges of a node can represent important relationships with other nodes. More formally, we define a path p as a sequence of edges $e_1, e_2, \ldots, e_n$ and we say each edge $e_i \in p$. The path connects two nodes v_0 and v_n through intermediate nodes $v_1, v_2, \ldots, v_{n-1}$, where either e_i=(v_{i-1}, v_i) or e_i=(v_i, v_{i-1}), for all $1 \leq i \leq n$.[3] The length of the path, $len(p)$, is n and the endpoints of the path, $ends(p)$, are $\{v_0, v_n\}$. Note that there is no undirected cycle in a path, i.e., the entities $v_0, v_1, \ldots, v_n$ are all distinct.

Definition 3 (Neighborhood Graph) Given an entity relationship graph G and a query tuple t=$\langle v_1, \ldots, v_n \rangle$, the corresponding *neighborhood graph* H_t is the *weakly connected subgraph*[4] of G that consists of all undirected paths in G of length d or smaller, with at least one endpoint of each such path being a query entity in t. The *path length threshold*, d, is an input parameter. More formally, the nodes and edges in H_t are defined as follows:

$V(H_t) = \{v | v \in t$, or $v \in V(G)$ and $\exists p$ s.t. $ends(p) = \{v_i, v\}$ where $v_i \in t, len(p) \leq d\}$;

$E(H_t) = \{e | e \in E(G)$ and $\exists p$ s.t. $ends(p)$=$\{v_i, v\}$ where $v_i \in t, len(p) \leq d$, and $e \in p\}$. ■

Example 1 (Neighborhood Graph) Given the data graph in Figure 1, the neighborhood graph for query tuple ⟨Jerry Yang, Yahoo!⟩ with path length threshold d=2 is shown in Figure 2. The nodes in dark color are the query entities. Other nodes are those to which there exists an undirected path of length at most 2 from either Jerry Yang, or Yahoo!, or both in the data graph. ■

[3] Since a path is undirected, we will not make distinction between starting entity and ending entity. [4] A directed graph is *weakly connected*, if there exists an undirected path between every pair of vertices.

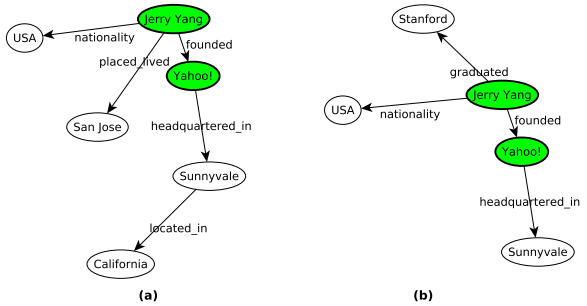


Figure 3: Query Graphs in the Neighborhood Graph in Figure 2.

Intuitively the neighborhood graph, by capturing how query entities and other entities in their neighborhood are related to each other, represents "features" of the query tuple that are to be matched in query answers. It can thus be viewed as a hidden query graph derived for capturing user's query intent. The query answers are approximate by nature, because many answers may match the neighborhood graph to different extents. Another way of looking at this is that there are many query graphs, which are all subgraphs of the neighborhood graph. By retrieving exact matches of these query graphs, we find different approximate answers to the neighborhood graph. We formalize the concept of query graph as follows.

Definition 4 (Query Graph) Given the neighborhood graph H_t of a query tuple t, a *query graph* Q is a weakly connected subgraph of H_t that contains all the query entities. We use $\mathcal{Q}_t$ to denote the set of all query graphs for t, i.e., $\mathcal{Q}_t$=$\{Q | Q$ is a weakly connected subgraph of H_t s.t. $\forall v \in t, v \in V(Q)\}$. ■

Continuing the running example, Figure 3 shows two query graphs for the neighborhood graph in Figure 2.

Echoing the intuition behind neighborhood graph, the definitions of answer graph and answer tuple are based on the idea that an answer tuple is similar to the query tuple if both their entities participate in similar relationships in their neighborhoods. The definitions are as follows.

Definition 5 (Answer Graph) Given a query graph $Q \in \mathcal{Q}_t$ for query tuple t, an *answer graph* A is a weakly connected subgraph of G that is also isomorphic to Q, except that node identifiers may not necessarily match. Formally, there exists a bijection $f : V(Q) \rightarrow V(A)$ such that:

- For every edge $e = (v_1, v_2) \in E(Q)$, there exists an edge $e' = (f(v_1), f(v_2)) \in E(A)$ such that $label(e) = label(e')$;
- For every edge $e' = (u_1, u_2) \in E(A)$, there exists an edge $e = (f^{-1}(u_1), f^{-1}(u_2)) \in E(Q)$ such that $label(e) = label(e')$.

We use $\mathcal{A}_Q$ to denote the set of all answer graphs for a query graph Q. ■

Definition 6 (Answer Tuple) Given a query graph $Q \in \mathcal{Q}_t$ for a query tuple t=$\langle v_1, v_2, \ldots, v_n \rangle$ and an answer graph $A \in \mathcal{A}_Q$, the *answer tuple* in A is t'=$\langle f(v_1), f(v_2), \ldots, f(v_n) \rangle$, where $f : V(Q) \rightarrow V(A)$ is the bijection between A and Q. We also call t' the *projection* of A, denoted as $t' \leftarrow A$. ■

Example 2 (Answer Graph and Answer Tuple) Figures 4 and 5 each show two answer graphs for query graphs (a) and (b) in Figure 3, respectively. The answer tuples in Figure 4 are ⟨Sergey Brin, Google⟩

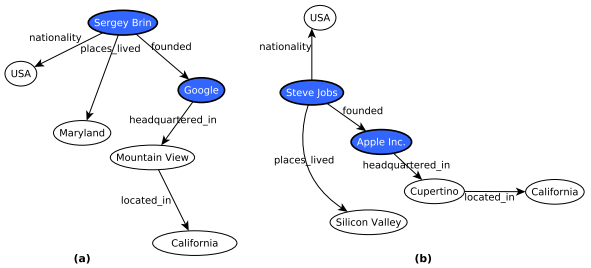


Figure 4: Answer Graphs for Query Graph in Figure 3(a).

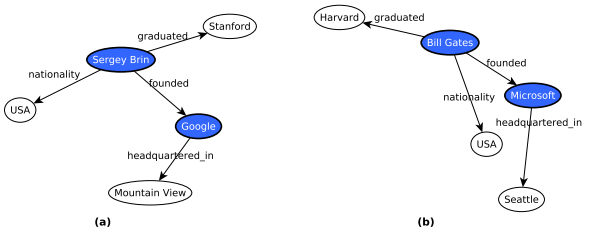


Figure 5: Answer Graphs for Query Graph in Figure 3(b).

and ⟨Steve Jobs, Apple Inc.⟩, and the answer tuples in Figure 5 are ⟨Sergey Brin, Google⟩ and ⟨Bill Gates, Microsoft⟩. Note that all the edges in a query graph are also present in its answer graphs, while matching entity nodes are not necessarily identical. ■

We note that a query tuple trivially matches itself, therefore is not considered an answer tuple.

The same answer tuple may be projected from multiple answer graphs, which in turn can match different query graphs. For instance, Figures 4(a) and 5(a) have the same projection– ⟨Sergey Brin, Google⟩. In this case, the two answer graphs are for different query graphs. A trivial similar case is when an answer graph is a subgraph of another answer graph. (Thus their corresponding query graphs also form a subgraph-supergraph relationship.) They will project on to the same answer tuple. Furthermore, two answer graphs of the same query graph may project on to the same answer tuple too. Consider the simple query graph in Figure 6(a), for query tuple ⟨Jerry Yang, California⟩. The two answer graphs shown in Figure 6(b) and (c) correspond to the same answer tuple ⟨Steve Jobs, California⟩.

Our objective is to return only distinct answer tuples. Given a query tuple t, among all answer graphs that project on to the same answer tuple t', those attaining the highest score are the *best answer graphs* of t'. That highest score is assigned as the score of t', capturing how well t' matches t.

Definition 7 (Answer Tuple Score) Given a query tuple t, the set of answer tuples are $\{t'|t' \leftarrow A, A \in \mathcal{A}_Q, Q \in \mathcal{Q}_t\}$. The score of an answer tuple t', i.e., the similarity between t' and t, is defined by

$$\mathsf{sim}(t', t) = \max_{A \in \mathcal{A}(Q), Q \in \mathcal{Q}(t)} \{\mathsf{score}_{Q,H_t}(A) | t' \leftarrow A\}. \quad (1)$$

The score of an answer graph A, $\mathsf{score}_{Q,H_t}(A)$, captures the similarity between A and the query graph Q (thus the neighborhood graph H_t). By Definition 5, an answer graph is isomorphic to the corresponding query graph, i.e., it is an exact match, barring that the identifiers of corresponding nodes in these two graphs do

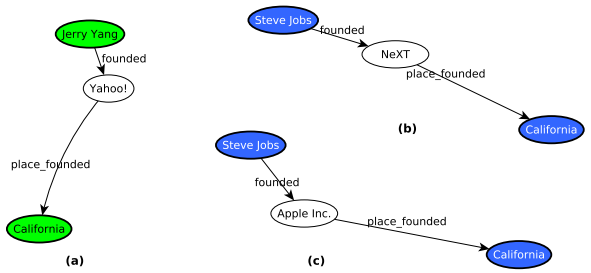


Figure 6: Simpler Query Graph

not necessarily match. Since the query graph is a subgraph of the neighborhood graph, the answer graph is an approximate match of the neighborhood graph. We thus use their similarity to capture how similar t' is to t.

The exact equation of $\mathsf{score}_{Q,H_t}(A)$ is given in Section 5. Without delving into details, we note that $\mathsf{score}_{Q,H_t}(A)$ sums up two values–the total weight of edges in Q (the query graph that A matches) and the extra credits given to matching nodes in A and Q. How edge weights are derived is discussed in Section 4. The rationale for the extra credits is that although node matching is not required, the more nodes are matched, the more similar A and Q (thus H_t) are.

We now formally define our problem statement below.

Problem Definition: Given an entity-relationship graph G and a query tuple t, find the top-k answer tuples t' with highest scores $\mathsf{sim}(t, t')$.

We note that the neighborhood graph may be quite large even for a small path length threshold d, especially in a well-connected data graph. For example, using Freebase as the data graph, the query tuple ⟨Jerry Yang, Yahoo!⟩ produces a neighborhood graph with 800K nodes and 900K edges, for $d = 2$. Such a large neighborhood graph not only makes query semantics obscure but also increases the complexity of query evaluation. GQBE alleviates this problem by several measures, including a heuristic to prune unimportant edges, a weighting scheme for capturing the relative importance of remaining edges, and an efficient algorithm for exploring the large space of possible query graphs to find the top-k answer tuples. In Section 4 we discuss the query graph discovery component of GQBE, for obtaining the *maximal query graph*, which aims to capture the most important edges in the neighborhood graph with much smaller size. In Section 5 we present the query processing component of GQBE, which aims to efficiently find top-k answer tuples by only exploring a small portion of the possible space of answers.

4. QUERY GRAPH DISCOVERY

In this section, we describe the query graph discovery component of GQBE, which constructs a weighted *maximal query graph* from the neighborhood graph H_t of a query tuple t. Given a large data graph, H_t itself can be quite large, even under a small path length threshold d. Such a large H_t will obscure query semantics because there might be only very few nodes and edges in it that capture important relationships in the neighborhood of t. Moreover, in finding approximate answers to a large H_t as query graph, the query evaluation cost can be high. Hence, we aim to drastically reduce the size of H_t and obtain a small subgraph that captures only important features of the query tuple, which are to be matched in answer tuples. This graph is termed the maximal query graph

because it is treated as the largest query graph allowed for t and any other query graph needs to be its subgraph.

The maximal query graph is obtained by several ideas. First, clearly unimportant edges are removed from the neighborhood graph. Then, remaining edges in the reduced graph are weighted by considering both their global frequencies in the data graph and their local frequencies in the reduced graph. We use a greedy method to form a connected subgraph of the reduced graph with the largest total edge weight, under a size constraint on number of edges. The resulting graph is the maximal query graph. Finally, weights of its edges are revised, by considering the aforementioned global and local edge frequencies, as well as distances to query entities. The rest of this section explains these ideas.

Unimportant Edges. Consider the neighborhood graph H_t in Figure 2. Edge e_1=(Jerry Yang, Stanford) and $label(e_1)$=*graduated*. Two other edges labeled *graduated*, e_2 and e_3, are also incident on node Stanford. Figure 2 is based on the data graph excerpt in Figure 1. The neighborhood graph on a complete real-world data graph may contain many more such edges because a lot of people have graduated from Stanford University. Among these edges, e_1 is the most important edge because it is closer to the query entity Jerry Yang. The intuition is that edge e_1 represents an important relationship between query tuple Jerry Yang and Stanford, while the other edges represent the relationships between Stanford and other entities instead of the query entities, which are deemed unimportant with respect to the query tuple.

We formalize the definition of *unimportant edges* as follows. Given an edge e=$(u, v) \in E(H_t)$, e is unimportant if it is unimportant from the perspective of its either end, u or v, i.e., if $e \in UE(u)$ or $e \in UE(v)$. Given a node $v \in V(H_t)$, we use $E(v)$ to denote the set of edges in H_t that are incident on v. $E(v)$ is partitioned into three mutually exclusive subsets– $IE(v)$, $UE(v)$, and the remaining edges in $E(v)$. The important and unimportant edges with regard to v, $IE(v)$ and $UE(v)$, respectively, are defined as follows:

$IE(v) = \{e \in E(v) \mid \exists v_i \in t, p$ s.t. $e \in p, ends(p)$=$\{v, v_i\}$, $len(p) \leq d\}$;

$UE(v) = \{e \in E(v) \mid e \notin IE(v), \exists e' \in IE(v)$ s.t. $label(e) = label(e'), (e$=$(u, v) \wedge e'$=$(u', v)) \vee (e$=$(v, u) \wedge e'$=$(v, u'))\}$.

With regard to v, an edge e incident on v belongs to $IE(v)$ if there exists a path between v and any query entity in the query tuple t, through e, with path length no more than the threshold d. To understand, consider v=Stanford in Figure 2. Edge e_1 belongs to $IE(v)$ by this definition. An edge e belongs to $UE(v)$ if (1) it does not belong to $IE(v)$ (i.e., there exists no such aforementioned path) and (2) there exists $e' \in IE(v)$ such that e and e' have the same label and they are both either incoming into or outgoing from v. In Figure 2, e_2 and e_3 have the same label and direction as e_1. Furthermore, there is no path between Stanford and Jerry Yang (or Yahoo) that contains e_2 or e_3. Hence e_2 and e_3 belong to $UE(v)$. In the same neighborhood graph, e_4 does no belong to either $IE(v)$ or $UE(v)$.

All edges deemed unimportant by the above definition are removed from H_t, which may become unconnected. However, it can be proved that there remains a weakly connected subgraph of H_t containing all query entities in t. This subgraph is called the *reduced neighborhood graph*.

Theorem 1 Given the neighborhood graph H_t for a query tuple t, the reduced neighborhood graph H'_t is the weakly connected subgraph of H_t containing all query entities in t and no unimportant edges in H_t. H'_t always exists. ■

PROOF. Proof Omitted.

The reduced neighborhood graph H'_t may still be large. It is thus critical to distinguish between the importance of the remaining edges. We weight these edges by the following measures and design a greedy method to form a connected subgraph of H'_t with the largest total edge weight, under a constraint on number of edges.

Inverse Edge Label Frequency. Edge labels that appear more frequently than others in the entire entity-relationship graph G are usually less important. For example, edges labeled *founded* (for capturing that someone is a founder of a company) can be rare and more important than edges labeled *nationality* (for capturing the nationality of a person). We capture this notion by the *inverse edge label frequency*, which is defined below.

$$\mathsf{ief}(e) = \log \frac{|E(G)|}{\#label(e)} \quad (2)$$

where $|E(G)|$ is the number of edges in G, and $\#label(e)$ is the number of edges in G with the same label as e.

Participation Degree. We define the *participation degree* $p(e)$ of an edge $e = (u, v)$ as the number of edges in the entity-relationship graph which share the same edge label and at least one of the two end nodes of edge e. Formally,

$$\mathsf{p}(e) = |\{e'$=$(u', v') \mid label(e)$=$label(e'), u'$=$u \vee v'$=$v\}| \quad (3)$$

An edge is less important if there are other edges incident on the same node with the same label. For instance, compare edges representing the *employment* relationship between a company and many of its employees and edges for the *board member* relationship between the company and its board members. The later edges are more significant than the former edges. This is because most likely there are much more employees than board members in a company. Observe that, in contrast to $\mathsf{ief}(e)$ which captures the global frequency of edge labels, $\mathsf{p}(e)$ measures the local frequency of edge labels. Globally, *employment* might still be a relatively rare edge but not necessarily locally to a company.

Combining the above ideas, the weight $\mathsf{w}(e)$ of an edge e in H_t is proportional to $\mathsf{ief}(e)$ and inversely proportional to $\mathsf{p}(e)$, given by the following equation:

$$\mathsf{w}(e) = \frac{\mathsf{ief}(e)}{\mathsf{p}(e)} \quad (4)$$

We now define maximal query graph based on $\mathsf{w}(e)$.

Definition 8 (Maximal Query Graph) Given the neighborhood graph H_t for query tuple t, its reduced neighborhood graph H'_t, and a parameter m, the *maximal query graph* $QMax_t$ is a weakly connected subgraph of H'_t that maximizes total edge weight while satisfying (1) it contains all the query entities in t and (2) it has m edges. ■

There are two challenges in going after the above exact definition. First, a connected graph with exactly m edges, for a particular value of m, may not exist. However, it can be proved that there always exists an m such that a connected graph with that size exists.

Property 1 There exists at least one value for m that guarantees the existence of $QMax_t$. ■

By definition, the reduced neighborhood graph H'_t itself is weakly connected. So a trivial proof of Property 1 is to give m the size of H'_t, i.e., $|E(H'_t)|$. Furthermore, in H'_t, each query entity is connected to at least one other query entity within distance d, where d is the path length threshold in Definition 3. Hence the smallest tree that connects all query entities is of at most $(n-1) * d$ edges. So

a less trivial value of m guaranteeing the existence of $QMax_t$ is $(n-1)*d$.

Second, even if $QMax_t$ exists for an m, finding it requires maximizing the total edge weight, which is a hard problem as given in Theorem 2.

Theorem 2 The decision version of finding the maximal query graph $QMax_t$ for an m is **NP**-hard. ■

PROOF. Proof Omitted.

Based on the theoretical analysis, we design a greedy method to find a plausible sub-optimal maximal query graph. We denote the resulting sub-optimal graph MQG_t. Unless otherwise noted, we will simply call it the maximal query graph, to avoid excessive terminologies. The method sorts all edges of the reduced neighborhood graph H'_t in descending order of weights $\mathsf{w}(e)$. With a target size m', the method finds the smallest such s that $s \geq m'$ and among the top s edges by descending order of $\mathsf{w}(e)$, there exists a weakly connected subgraph of H'_t with exactly m' edges containing all query entities. That subgraph is the resulting MQG_t. The method initially sets m' to be an empirically chosen small m and finds the corresponding s and MQG_t, if feasible. If no such s can be found, it looks for the largest $m' < m$ such that the corresponding s exists. If that is still not achievable, the method settles on the smallest $m' > m$ such that the corresponding s exists.

The resulting MQG_t based on this method might not be a balanced graph, where each query entity gets a fair number of edges associated with it. Query entities in H'_t which have more neighbors will likely have a more prominent representation in the resulting MQG_t. Therefore we propose a divide-and-conquer mechanism to construct a balanced maximal query graph, as follows.

The idea is to break H'_t into $n+1$ components, where one special component, the *core graph*, connects all the n query entities in t and other n components are for the n query entities individually. The core graph combines all simple undirected paths between any pair of query entities. In the individual component for a query entity e, all entities connect to other query entities through e. The components are determined by running a depth first search (DFS) starting from each query entity. During the DFS from a query entity e, all edges on the paths reaching any other query entity within distance d belong to core graph, and other edges belong to e's individual component. We then apply the aforementioned greed method to find $n+1$ "maximal query graphs" (MQGs), one for each component. Since the core graph (thus its corresponding MQG) connects all n query entities, the $n+1$ MQGs altogether form the overall MQG_t. Using an empirically chosen small m as the target size of MQG_t, we set the target size for each individual MQG to be $\frac{m}{n+1}$, aiming at obtaining a fairly balanced MQG_t.

In constructing maximal query graph, the weighting scheme in Equation 4 considers the importance of an edge independent of the query tuple. The rationale behind the design is to obtain a balanced graph which includes not only edges incident on query entities, but also those in the larger neighborhood. By empirical observations, we find that, in capturing how well an answer graph matches the maximal query graph, it becomes imperative to further distinguish the importance of edges with respect to query entities. We consider that edges closer to the query entities convey more meaningful relationships than those farther away. This is captured by *edge depth*, as defined below.

Edge Depth. The depth $d(e)$ of an edge e=(u,v) is its smallest distance to any query entity $v_i \in t$, i.e.,

$$d(e) = \min_{v_i \in t} \min_{u,v} \{\mathsf{dist}(u, v_i), \mathsf{dist}(v, v_i)\} \quad (5)$$

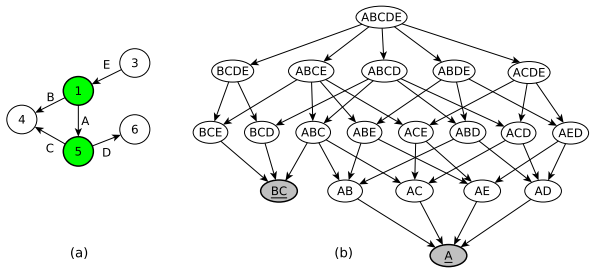


Figure 7: Query Graph and Query Lattice.

Here, $\mathsf{dist}(.,.)$ is the shortest-path distance between the two nodes considering undirected edges. The more the depth of an edge is, the lower is its importance.

Finally, we define edge-weight of edges in MQG_t by combining $\mathsf{ief}(e)$, $\mathsf{p}(e)$ and $\mathsf{d}(e)$, as follows. The query processing component in Section 5 ranks answer graphs based on this measure.

$$\mathsf{w}(e) = \frac{\mathsf{ief}(e)}{\mathsf{p}(e) \times \mathsf{d}^2(e)} \quad (6)$$

5. QUERY PROCESSING

Given an entity-relationship graph G and an n-tuple t, the query processing component of GQBE takes the maximal query graph MQG_t produced in Section 4 and finds answer graphs matching its subgraphs. MQG_t and its subgraphs are organized in a directed acyclic graph, called the *query lattice* $\mathcal{L}$ and defined below.

5.1 Query Lattice

Definition 9 (Query Lattice) Given a maximal query graph MQG_t, the query lattice $\mathcal{L}$ is a directed acyclic graph in which the root node of $\mathcal{L}$ is MQG_t, and the other nodes of $\mathcal{L}$ are the subgraphs of MQG_t. Each node of the lattice represents a *query graph* defined in Section 3, and we shall use the terms lattice node and query graph interchangeably. The *children* and *parents* of a lattice node Q are defined following the subgraph-supergraph relationships, i.e.,

$$\mathsf{Children}(Q) = \{Q' : Q' \preceq Q, |E(Q)| - |E(Q')| = 1\}$$
$$\mathsf{Parents}(Q) = \{Q' : Q \preceq Q', |E(Q')| - |E(Q)| = 1\}$$

The leaf nodes of $\mathcal{L}$ constitute of the *minimal query trees*, which are those query graphs that cannot be made any simpler and still keep all the query entities connected.

Definition 10 (Minimal Query Tree) A query graph Q is a *minimal query tree* if it does not have a subgraph that is also a query graph, that is, removing any edge from Q will disqualify it from being a query graph – the resulting graph either is not weakly connected or does not contain all the query entities. Note that such a query graph must be a tree. ■

The importance of the *minimal query trees* comes from the fact that if an n-tuple t' is an answer to the query tuple t, then t' must have an answer graph corresponding to at least one of the *minimal query trees* in the query lattice of t.

Example 3 (Query Lattice) Figure 7(a) shows a maximal query graph MQG_t, which contains two query entities in shaded circles and five edges A, B, C, D, and E. Its corresponding query lattice $\mathcal{L}$ is in Figure 7(b). The root node of $\mathcal{L}$, denoted by $ABCDE$, represents the maximal query graph itself. The two shaded lattice nodes, denoted by A and BC, are the two minimal query trees.

Each lattice node is a distinct subgraph of the maximal query graph. For example, the node denoted as ABD represents a query graph with only edges A, B and D. It can also be noted that there is no lattice node such as BDE, which is not a valid query graph since it is not connected. ■

The query lattice is constructed by first generating all minimal query trees, and then going all the way up to the root of the lattice. Every new lattice node can be generated by adding exactly one appropriate edge to the query graph represented by one of its children. Note that, in our algorithm, we integrate the actual lattice construction with its exploration, i.e., the the lattice is built in a "lazy" manner based on how we require to explore the lattice nodes. In the following section, we first provide a brief overview of our minimal query tree generation method.

5.2 Computing Minimal Query Trees

As mentioned in Section 4, all the edges used in forming the undirected paths of maximum length d between any two query entities constitute of our *core graph*. In generating the maximal query graph, a connected subgraph of the core graph is included into it. In our following discussion, we will simply refer to it as the core graph, without causing confusion. The strategy we employ to generate all minimal query trees is to first enumerate all distinct spanning trees of the core graph by technique in [9], and then *trim* them. Specifically, all non-query entities (nodes) of degree one along with its edge are deleted from the spanning trees. This deletion is performed iteratively until all the remaining nodes of degree one correspond to query entities only, and thereby generating all possible minimal query trees. Only the distinct minimal query trees are maintained.

5.3 Answer Tuple Scoring Function

Given a query tuple t, the set of answer tuples are $\{t' | t' \leftarrow A, A \in \mathcal{A}_Q, Q \in \mathcal{Q}_t\}$. The score of an answer tuple t' comes from its best answer graph, as given in Equation 1. The score of a particular answer graph A, $\mathsf{score}_{Q,H_t}(A)$, captures the similarity between A and the query graph Q that A matches (thus the neighborhood graph H_t). Note that our algorithm operates on the maximal query graph MQG_t instead of the original large H_t. Hence we give the detailed equation for $\mathsf{score}_{Q,MQG_t}(A)$ below. It sums up two values–the total weight of edges in Q and the extra credits given to matching nodes in A and Q. Note that $f : V(Q) \rightarrow V(A)$ is the bijection between A and Q, and the edge weight $\mathsf{w}(e)$ was defined in Equation 6.

$$\mathsf{score}_{Q,MQG_t}(A) = \sum_{e \in E(Q)} \mathsf{w}(e) + \sum_{\substack{e=(u,v) \in E(Q) \\ e'=(f(u),f(v)) \in E(A)}} match(e, e') \tag{7}$$

where $match(e, e')$ is for giving extra credit to matching nodes in A and Q, defined as follows. Note that it does not award a matching node excessively. Instead, only a fraction of $\mathsf{w}(e)$ is awarded, since when one node matches, all its neighbors will also likely match other nodes of the query graph.

$$match(e, e') = \begin{cases} \frac{\mathsf{w}(e)}{|E(u)|} & \text{if } u{=}f(u) \\ \frac{\mathsf{w}(e)}{|E(v)|} & \text{if } v{=}f(v) \\ \frac{\mathsf{w}(e)}{|min(|E(u)|,|E(v)|)|} & \text{if } u{=}f(u), v{=}f(v) \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Devising a top-*k* algorithm based on the above exact equation increases the overall size and complexity of the lattice, due to consideration of node matching. It can also make upper-bound scores (introduced below) very loose and an early termination of lattice exploration might be hard. Hence, our algorithm first finds the top-k' (where k' is a heuristic-based number and is larger than the target answer size k) answers based on only edge weights, i.e., the first component in Equation 8. After we obtain the top-k' answer tuples, we further compute the scores of all answers by the full scoring function in Equation 8 to re-rank them and thus obtain the final top-*k* answer tuples. Therefore, our following discussion of the algorithm assumes a simplified scoring function $\mathsf{score}(Q) = \mathsf{score}_{Q,MQG_t}(A) = \sum_{e \in E(Q)} \mathsf{w}(e)$. Note that in this simplified function all answer graphs A to a query graph Q have the same score $\sum_{e \in E(Q)} \mathsf{w}(e)$.

5.4 A Brute-Force Approach

We employ a breadth-first search (BFS) based bottom-up lattice exploration method as the brute-force mechanism of evaluating the query lattice. We first start off with the minimal query trees and use the results of each lattice-node to find the answer graphs of its parents. We complete the evaluation of every lattice node in a level (all query graphs of the same size) before evaluating any lattice node in the upper level. If traversing a lattice node does not yield any answer graph, all its ancestors are pruned out. This process continues until there is no more unpruned node to evaluate. When the algorithm terminates, the scores of all answer graphs and answer tuples are obtained.

5.5 Best-First Exploration of the Lattice

The brute-force approach traverses many lattice nodes to find all answer tuples, while we are only seeking top-*k* answer tuples. We now propose a top-k lattice exploration algorithm to overcome this obvious shortcoming. We traverse the lattice starting from the leaf nodes, i.e., minimal query trees, and follow a *best-first search* method, that is to always expand the most promising lattice node Q_{cur} during our traversal. The following definitions will be helpful for selecting the most promising node Q_{cur}.

Definition 11 (Upper Boundary) Given a node Q in lattice $\mathcal{L}$, we define its *upper boundary* as the set of lattice nodes Q', such that Q' is a supergraph of Q, and Q' has no parent in the lattice, i.e.,

$$\mathcal{U}(Q) = \{Q' : Q \preceq Q', \mathsf{Parents}(Q') = \phi\} \tag{9}$$ ■

Clearly, at the beginning of our method, $\mathcal{U}(Q) = \{MQG_t\}$ for any lattice node Q. However, we dynamically update the lattice during our best-first traversal by deleting various lattice nodes. Thus, the upper boundary of some lattice node also changes, which will be discussed shortly. Next, we define an *upper-bound score* and a *lower-bound score* for each lattice node, which provide bounds on the best-possible and worst-possible scores on answer tuples that have an answer graph corresponding to that lattice node.

Definition 12 (Upper-Bound Score) The *upper-bound score* of a lattice node Q is defined as the maximum score of any node in its upper boundary. Formally,

$$U(Q) = \arg\max_{Q' \in \mathcal{U}(Q)} \mathsf{score}(Q') \tag{10}$$

Definition 13 (Lower-Bound Score) The *lower-bound score* of a lattice node Q is defined as its own score, i.e.,

$$L(Q) = \mathsf{score}(Q) \tag{11}$$

In our best-first search, we follow an *optimistic strategy*, that is we always traverse the lattice node Q_{cur} with the highest upper-bound score [1] from the candidate pool $\mathcal{C}$. We then insert all its

Algorithm 1: Lattice Exploration

Input: Query lattie $\mathcal{L}$, query tuple t, and an integer k
Output: Top-k answer tuples
candidate pool $\mathcal{C} \leftarrow$ leaf nodes of $\mathcal{L}$
priority queue $\mathcal{Q} \leftarrow \phi$; *Terminate* $\leftarrow$ **false**
while (**not** *Terminate*) **do**
  $Q_{cur} \leftarrow$ node with highest upper-bound score in $\mathcal{C}$
  if (Q_{cur} is *Null*) **then**
    prune all predecessors of Q_{cur} in $\mathcal{L}$
    recompute upper-bound scores of nodes in $\mathcal{C}$ (Alg. 2)
  else
    insert Parents(Q_{cur}) in $\mathcal{C}$
    if (top-k answer tuples found [Th. 3]) **then**
      Terminate $\leftarrow$ **true**
    end if
  end if
end while

Algorithm 2: Recomputing Upper-Bounds

Input: Query lattice $\mathcal{L}$, null node Q_{cur}, and candidate pool $\mathcal{C}$
Output: $U(Q)$ for all Q in $\mathcal{C}$
for all $Q \in \mathcal{C}$ **do**
  for all $Q' \in \mathcal{U}(Q) \cap$ Predecessor(Q_{cur}) **do**
    $\mathcal{U}(Q) \rightarrow \mathcal{U}(Q) \setminus \{Q'\}$
    $m \leftarrow |E(Q_{cur} \setminus E(Q)|$
    for all $i = 1$ **to** $m - 1$ **do**
      $E(Q''_i) \leftarrow E(Q)$
      $E(Q''_i) \leftarrow E(Q''_i) \cup (E(Q) \setminus E(Q_{cur}))$
      insert all but one edges of $E(Q_{cur}) \setminus E(Q)$ in $E(Q''_i)$
      find Q_{sub}, maximum weakly-connected subgraph of Q''_i, containing all query entities
      $\mathcal{U}(Q) \leftarrow \mathcal{U}(Q) \cup \{Q_{sub}\}$, if Q_{sub} not subgraph of any nodes in $\mathcal{U}(Q)$
    end for
  end for
  recompute $U(Q)$ using Eq. 10
end for

parent nodes, Parent(Q_{cur}) in the candidate pool. The candidate pool $\mathcal{C}$ initially contains all the leaf nodes of lattice $\mathcal{L}$.

When we traverse a lattice node Q_{cur} during our best-first exploration, we also find all the answer graphs A that can be matched with Q_{cur}. A history of all the answer graphs corresponding to its immediate children nodes is maintained which helps us to quickly find the answer graphs for the currently traversed lattice node. As mentioned in Definitions 6 and 7, each answer graph in A can be projected to an answer tuple which is assigned a score equal to the lower bound of Q_{cur} if it is higher than its current score. The score of its *best answer graph* is used for each answer tuple and a top-k priority queue $\mathcal{Q}$ is maintained to store the top-k answer tuples and their scores. The top-k priority queue is updated accordingly every time a new lattice node is traversed.

Note that we may not always find an answer graph that can be matched with a lattice node. Such lattice nodes are referred to as the *null nodes*. Null nodes satisfy the following *upward-closure* property in our algorithm.

Property 2 (Upward Closure) All the ancestor lattice nodes of a null node are also null nodes. ■

[1] If there are multiple nodes in $\mathcal{C}$ with the same upper-bound score, then the node with the most number of edges is selected as Q_{cur}. If there are multiple nodes with the same upper-bound score and edge counts, then the node with the highest lower-bound score is selected. If there are multiple nodes with the same values for each of the above three criteria, any one of those nodes is randomly selected. Intuitively, these heuristic criteria help to quickly identify the top-k answers.

Property 2 follows from the construction of our lattice $\mathcal{L}$. Hence, if the currently traversed lattice node is a null node, there is no need to traverse its ancestor lattice nodes, which in turn allows us to delete all such ancestor nodes from the lattice. However, the deletion of the ancestor nodes of a null node might change the upper boundaries and upper-bound scores for some lattice nodes in our candidate pool $\mathcal{C}$. Therefore, we need to recompute these variables every time the currently traversed lattice node becomes a null node. An efficient method to recompute these variables will be discussed in Section 5.6.

Finally, we terminate our best-first search method when the lowest score stored in the top-k priority queue $\mathcal{Q}$ is greater than or equal to the upper-bound score for the next node to be traversed from the candidate pool $\mathcal{C}$. The correctness of our termination criteria is justified below.

Theorem 3 The termination condition guarantees a correctly ranked top-k answer tuples list.

PROOF. Proof Omitted.

5.6 Recomputation of Upper Boundaries and Upper-Bound Scores

Recall that when the currently traversed lattice node Q_{cur} becomes a null node, we delete all its ancestor nodes from the lattice, and this might change the upper boundaries and upper-bound scores for some lattice nodes in the candidate pool. Hence, there are two issues we need to resolve here. First, what are the lattice nodes in the candidate pool for which the upper boundaries and upper-bound scores change? We refer to such nodes as the *dirty nodes*. Second, for these dirty nodes, how can one recompute the new upper boundaries and upper-bound scores. In this section, we propose efficient methods to answer both these question.

Finding the Dirty Nodes. We verify all the lattice nodes in the candidate pool whether any of its upper boundary nodes are in the deleted nodes. If so, we consider this lattice node as a dirty node.

Recompute Upper-Bound Scores for Dirty Nodes. Let us consider all the pairs $\langle Q, Q' \rangle$, such that Q is a dirty node in the candidate pool, and Q' is one of its upper boundary nodes among the deleted nodes. For each such pair $\langle Q, Q' \rangle$, we create a set of $m-1$ distinct graphs Q'' as follows, where m is the number of edges in Q_{cur} that are not present in Q. (1) Each Q'' contains all the edges of Q. (2) Each Q'' also contains all the edges of Q' that are not in Q_{cur}. (3) Next, consider the list of m edges in Q_{cur}, which are not present in Q. Each Q'' contains all but one edges from this list of m edges.

For each of these $m-1$ newly generated graphs Q'', we find a subgraph Q_{sub}, such that (a) $Q_{sub} \preceq Q''$, (b) Q_{sub} contains all the input entities, (c) Q_{sub} is weakly connected, and (d) Q_{sub} has the maximum number of edges among all graphs satisfying (a), (b), (c). Now, Lemma 1 and 2 will be useful to show that Q_{sub} must be a node in the modified lattice $\mathcal{L}$ (i.e., after one deletes the ancestor nodes of Q_{cur} from $\mathcal{L}$).

Lemma 1 For each Q'', Q_{sub} must be at least as large as Q. ■

Lemma 2 Q_{sub} is a *query graph* and does not belong to the deleted nodes of lattice $\mathcal{L}$. ■

PROOF. Proof Omitted.

Therefore, we can locate Q_{sub} in the modified lattice $\mathcal{L}$. If Q_{sub} has no parents in the set of all upper boundary nodes (which is essentially the union of upper boundary of all candidate nodes in $\mathcal{C}$), then we assign Q_{sub} as a new boundary node of Q. The above

method is performed for all pairs $\langle Q, Q' \rangle$. Finally, we recompute the upper-bound score of Q as the maximum score among all its upper boundary nodes. Theorem 4 justifies the *correctness* of our approach for recomputing the upper-bound score of all dirty nodes.

Theorem 4 All new upper boundary nodes for every dirty node is identified using the aforementioned technique. ■

PROOF. Proof Omitted.

6. EXPERIMENTS

We present experimental results to demonstrate the (1) effectiveness (Section 6.2 and 6.3) and (2) efficiency (Section 6.4) of GQBE over the *Freebase* dataset [4], which is a large-scale real-life entity relationship graph. The effectiveness of our results are measured using four well-defined metrics: Precision-at-k (P@k), Normalized Discounted Cumulative Gain (nDCG), Mean Average Precision (MAP), and Pearson Product-Moment Correlation Coefficient (PCC), while we consider both *ground truths* and *user study* for the accuracy of our results. We have shown several case studies over 20 carefully selected queries. We also compare our results with that of a Baseline method (described in Section 5.4), and also with NESS, a graph querying framework that finds matches with query graphs having unknown node labels.

6.1 Experiment Setup

6.1.1 Graph Dataset

Freebase is a collaborative knowledge base created using various sources including Wikipedia. We used the Freebase dataset from September 2011 after cleaning it to keep only named entities (e.g., Stanford University) and abstract concepts (e.g., Jewish people). Every edge in Freebase also has a direction and a label. For instance, *inventions* is an edge from Sergey Brin to Google, representing that Google is an invention of Sergy Brin. Our knowledge graph contains around 47M edges, 28M nodes, and $5,428$ distinct edge labels.

6.1.2 Queries, Ground Truths and User Study.

We use a set of 20 queries to evaluate the effectiveness and efficiency of GQBE. We have a range of queries involving one, two and three query entities in the input tuples. The queries used and their ground truth size is summarized in Table 1. Observe that our query tuples are chosen from diverse domains such as MOVIES, FOUNDERS, COMPANY, SPORTS, AWARDS, RELIGION, UNIVERSITY, AUTOMOBILE, MUSIC and ENTERTAINMENT to name a few. The ground truth for queries *Q1* and *Q6* were manually obtained from *Wikipedia* and the rest from *Freebase*. Besides, we also have another set of ground truths from *user study* with the Amazon Mechanical Turks.

6.1.3 Evaluation Metrics.

We consider the following four metrics [22] to measure the effectiveness of our system.

Precision-at-k. P@k for some query is calculated as the ratio of the number of top-k answers that belong to ground truth over k.

Mean Average Precision. MAP for a set of queries is the mean of the average precision scores for each query. Formally,

$$\text{AveP}(q) = \frac{\sum_{i=1}^{k} \text{P @k(q,i)}}{\#Ground\text{-}Truths}$$
$$\text{MAP} = \frac{\sum_{q} \text{AveP}(q)}{\#Queries} \quad (12)$$

Query ID	Query Tuple	Ground Truth Size
Q1	Donald Knuth, Stanford University, Turing Award	18
Q2	Ford Motor, Lincoln, Lincoln MKS	25
Q3	Nike, Tiger Woods	20
Q4	Michael Phelps, Sportsman of the Year	55
Q5	Gautam Buddha, Buddhism	621
Q6	Manchester United, Malcolm Glazer	40
Q7	Boeing, Boeing C-22	89
Q8	David Beckham, A C Milan	94
Q9	Beijing, 2008 Summer Olympics	41
Q10	Microsoft, Microsoft Office	200
Q11	Jack Kirby, Ironman	25
Q12	Apple Inc, Sequoia Capital	300
Q13	Beethoven, Symphony No. 5	600
Q14	Uranium, Uranium-238	26
Q15	Microsoft Office, C++	300
Q16	Dennis Ritchie, C	163
Q17	Steven Spielberg, Minority Report	40
Q18	Jerry Yang, Yahoo!	8349
Q19	C	1240
Q20	TomKat	16

Table 1: Queries and Ground Truth Size

Normalized Discounted Cumulative Gain. nDCG measures the effectiveness of a ranked list by first computing the cumulative gain of relevant answers in the list. Relevance in our case is either 0 or 1 at each position. We then penalize the lower ranked relevant answers and aggregate the total gain. This is then normalized by the gain obtained for an ideal ranking (obtained from Wikipedia and Freebase ground truths).

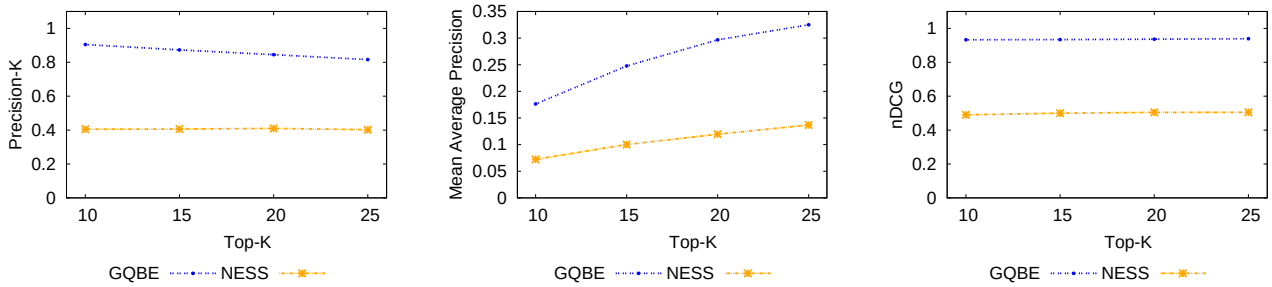
Pearson Product-Moment Correlation. PCC determines the effectiveness of our ranking with respect to the user study. We generate 50 random answer tuple pairs from the top-30 answers generated by GQBE and 20 users specify their preference of ranking for each pair. We construct two ranking lists for every query, one representing GQBE, and the other, user evaluation. The system's ranking list is computed by taking the difference between each pairs' ranks and the user-evaluation ranking list is computed by the difference of number of users agreeing and disagreeing to the ranking list proposed by the system. PCC is then computed on these two ranking lists.

Implementation. The followings were implemented in Java: (1) GQBE, (2) Baseline. We obtained the executables of NESS from the authors [19]. All the experiments were performed using a dual core 24 GB Memory, 2.0 GHz Xeon server.

6.2 Effectiveness

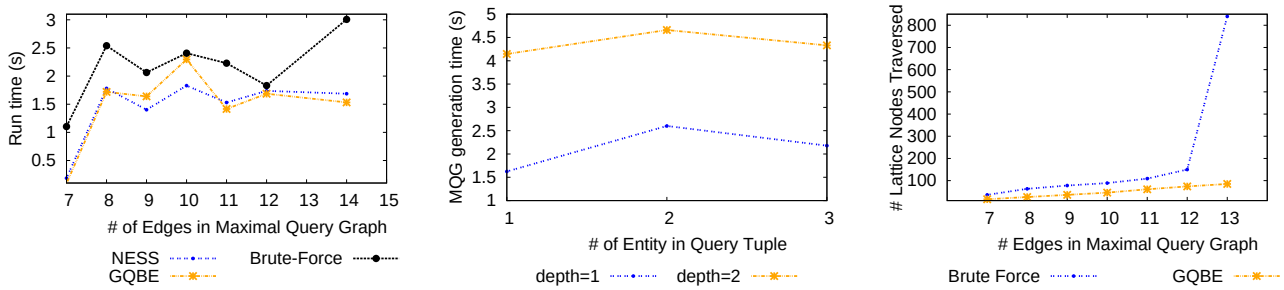
In this section, we present the efficiency results of GQBE using three metrics: P@k, MAP, and nDCG. We consider the 20 queries in Table 1, and the corresponding ground truths are obtained from Wikipedia and Freebase. We also compare our accuracy with that of NESS using the three metrics.

Figures 8(a), 8(b) and 8(c) show that our answers and their rankings are very accurate with respect to the ground truths; for example, P@k, MAP, and nDCG are 0.8, 0.35 and 0.9, respectively, with $k = 25$. Moreover, GQBE outperforms NESS significantly in all three metrics, over different values of k. This is because GQBE gives priority to query entities and other important edges, while NESS gives equal importance to all nodes and edges in the maximal query graph. The other reason is that the top-k answer graphs reported by NESS is with respect to only one of the query entities. In fact, NESS does not work well when a query graph has multiple nodes with unknown labels. In contrast. GQBE is modeled to handle such cases, by allowing all the query entities to be replaced by answer entities.



(a) P@k (b) MAP (c) nDCG

Figure 8: Effectiveness Results



(a) Run Time (b) Maximum Query Graph Finding Time (c) #Lattice Nodes Traversed

Figure 9: Efficiency Results

Case Study. In Table 2, we list the top-5 results found by GQBE for 3 selected queries from Table 1. We observe that all the top-5 answer tuples are very relevant with respect to the corresponding query tuple.

Query Tuple	Top-5 Answer Tuples
Donald Knuth, Stanford, Turing Award	D. Knuth, Stanford, V. Neumann Medal J. McCarthy, Stanford, Turing Award N. Wirth, Stanford, Turing Award D. Knuth, Stanford, Nat. Medal, Sc. D. Knuth, Stanford, Kyoto Prize
Jerry Yang, Yahoo!	David Filo, Yahoo! Bill Gates, Microsoft Steve Wozniak, Apple Inc. Steve Jobs, Apple Inc. Paul Allen, Microsoft
C (programming language)	Java C++ C Sharp Python Visual Basic .NET

Table 2: Case Study: Top-5 Results for some Queries

6.3 User Study

We conducted a extensive user study using Amazon Mechanical Turks to measure the effectiveness of GQBE in the real world. We chose top-30 answers in each of the 20 queries and created 50 random pairs per query. These pairs were broken down into smaller tasks, and 2,000 users were asked to rank the better answer in each pair with respect to the input query. We measured the effectiveness of the ranking of answers by GQBE using PCC for each of these queries. The range of PCC is from -1 to 1, and a value greater than 0 indicates positive correlation between the ranking produced by GQBE and the user feedback.

The PCC value for each query and the entire system is summarized in Table 3. The correlation between the ranking of our system and the user feedback for queries like *Q1*, *Q2* and *Q4* is very high indicating a very good ranking of those answers by GQBE. It can be observed that the ranking of answers is better for 3-tuple queries, suggesting that more query entities probably captures a better maximal query graph. It can also be observed that the PCC value of *Q12* and *Q13* is 0. This is because the scores of all the top-30 answer tuples were the same, indicating that all those answers were projected from the same query graph. Since the top-30 answers were ranked the same, the PCC value is 0. The overall PCC value for the ranking produced by GQBE across all the 20 queries is *0.497*, which was calculated without considering *Q12* and *Q13*. Having a Pearson correlation coefficient over 0.3 is generally considered a strong positive ranking correlation. A PCC of *0.497* indicates that GQBE produces a good ranking of answers which in line with what users expect in real world.

Query ID	PCC	Query ID	PCC	Query ID	PCC
Q1	0.79	Q2	0.78	Q3	0.60
Q4	0.80	Q5	0.34	Q6	0.27
Q7	0.06	Q8	0.26	Q9	0.33
Q10	0.770	Q11	0.578	Q12	0
Q13	0	Q14	0.620	Q15	0.43
Q16	0.29	Q17	0.64	Q18	0.300
Q19	0.40	Q20	0.65		

Table 3: Perason's Correlation Coefficient (Average = 0.497)

6.4 Efficiency Results

In this section, we compare the efficiency of GQBE with respect to the Baseline approach and NESS. Figure 9(a) compares the total running times of the three methods with respect to the size (number of edges) in the maximal query graph. The running time includes both *Query Graph Discovery* and *Query Processing* times. One can observe that the running time of the Baseline suffers as compared to GQBE, and this is due to the more number of lattice nodes the Baseline requires to evaluate. It is worth noting that the running time of GQBE and Baseline do not increase consistently with increase in the query graph size. This is because the lattice evaluation time is not solely dependent on the number of edges, but it is al-

so dependent on the particular edges chosen in the maximal query graph. The running times of GQBE and NESS is comparable in most cases and NESS performs a little better in some cases. This can be explained by the fact that NESS is an approximate graph querying system. It does not always try to find the best score for an answer tuple with respect to a given maximal query graph. GQBE, on the other hand, always guarantees to fetch the best score for an answer tuple.

We next study the time taken to discover the maximal query graph as a function of number of query entities in the query tuple, while also varying the depth threshold d. Recall that the maximal query graph generation involves first getting a neighborhood graph using d, and then using a greedy heuristic to obtain a much smaller subgraph. Figure 9(b) shows that the maximal query graph generation time increases when we have a higher d, since this increases the size of the neighborhood graph.

We also compare the number of lattice-nodes traversed by GQBE and Baseline, in order to justify the efficiency of our system. The lower the number of lattice nodes we evaluate, the better it is. Figure 9(c) shows the total number of lattice-nodes evaluated as a function of the maximal query graph size. As we can observe, the number of lattice-nodes evaluated by GQBE is lesser than the Brute Force approach. This is because in the best-first strategy the choice of the next lattice-node to evaluate depends on the highest upper bound which forces the lattice traversal to greedily reach the top of the lattice. Evaluating fewer nodes has a direct impact on the running time of the algorithm which is shown in the running time comparison of the two methods.

7. CONCLUSION

We introduce GQBE, a system that queries entity-relationship graphs by example entity tuples. The query graph discovery component of GQBE derives a hidden maximal query graph based on an input query tuple. The query graph lattice based on the maximal query graph may contain a large number of query graphs. Our top-*k* query algorithm guarantees to only evaluate query graphs that are necessary for obtaining the top-*k* answers. Our experiments on the large Freebase dataset show that GQBE clearly outperforms an adaptation of a related system NESS in query answer accuracy and its execution efficiency is comparable to NESS and outperforms a baseline method by orders of magnitude.

As an initial step toward better usability of graph query systems, GQBE saves users the burden of forming explicit query graphs, by allowing querying graphs by example entity tuples. To the best of our knowledge, there has not been such proposal in the past. As we see an unprecedented proliferation of entity data graphs in the real world, we hope this work will have profound impact on many applicants.

8. REFERENCES

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, , and Z. Ives. DBpedia: A nucleus for a Web of open data. In *ISWC*, 2007.

[2] A. Birkl and G. Yona. Biozon: a hub of heterogeneous biological data. *Nucleic Acids Research*, 34, 2006.

[3] C. Böhm, F. Naumann, M. Freitag, S. George, N. Höfler, M. Köppelmann, C. Lehmann, A. Mascher, and T. Schmidt. Linking open government data: What journalists wish they had known. In *I-SEMANTICS*, 2010.

[4] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250, 2008.

[5] L. Chang, J. X. Yu, L. Qin, Y. Zhu, and H. Wang. Finding information nebula over large networks. In *CIKM*, 2011.

[6] D. H. Chau, C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, and T. Eliassi-Rad. GRAPHITE: A visual query system for large graphs. In *ICDM Workshops*, pages 963–966, 2008.

[7] E. Demidova, X. Zhou, and W. Nejdl. FreeQ: an interactive query interface for Freebase. In *WWW*, demo paper, 2012.

[8] L. Fang, A. D. Sarma, C. Yu, and P. Bohannon. REX: explaining relationships between entity pairs. In *PVLDB*, pages 241–252, 2011.

[9] H. N. Gabow and E. W. Myers. Finding all spanning trees of directed and undirected graphs. *SIAM J. Comput.*, 7(3):280–287, 1978.

[10] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS.*, 2006.

[11] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked keyword searches on graphs. *SIGMOD*, pages 305–316, 2007.

[12] Y. He and D. Xin. SEISA: set expansion by iterative similarity aggregation. In *WWW*, pages 427–436, 2011.

[13] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, pages 13–24, 2007.

[14] M. Jarrar and M. D. Dikaiakos. A query formulation language for the data web. *TKDE*, 24:783–798, 2012.

[15] C. Jin, S. S. Bhowmick, X. Xiao, J. Cheng, and B. Choi. GBLENDER: Towards blending visual query formulation and query processing in graph databases. In *SIGMOD*, pages 111–122, 2010.

[16] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: a scalable and general graph management system. In *KDD*, 2011.

[17] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, pages 681–692, 2011.

[18] G. Kasneci, S. Elbassuoni, and G. Weikum. MING: mining informative entity relationship subgraphs. In *CIKM*, 2009.

[19] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, pages 901–912, 2011.

[20] A. Khan, Y. Wu, and X. Yan. Emerging graph queries in linked data. In *ICDE*, pages 1218–1221, 2012.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[22] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[23] J. Pound, I. F. Ilyas, and G. E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, pages 423–434, 2010.

[24] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, pages 939–948, 2010.

[25] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge unifying WordNet and Wikipedia. In *WWW*, pages 697–706, 2007.

[26] Y. Tian and J. M. Patel. TALE: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[27] H. Tong and C. Faloutsos. Center-piece subgraphs: Problem definition and fast solutions. In *KDD*, pages 404–413, 2006.

[28] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. *KDD*, 2007.

[29] 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*, pages 405–416, 2009.

[30] R. C. Wang and W. W. Cohen. Language-independent set expansion of named entities using the web. In *ICDM*, pages 342–350, 2007.

[31] M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum. Deep answers for naturally asked questions on the web of data. In *WWW*, demo paper, pages 445–449, 2012.

[32] J. Yao, B. Cui, L. Hua, and Y. Huang. Keyword query reformulation on structured data. *ICDE*, pages 953–964, 2012.

[33] X. Yin and S. Shah. Building taxonomy of web search intents for name entity queries. In *WWW*, pages 1001–1010, 2010.

[34] M. M. Zloof. Query by example. In *AFIPS*, 1975.