GQBE: Querying Knowledge Graphs by Example Entity Tuples*

Nandish Jayaram† Mahesh Gupta† Arijit Khan§ Chengkai Li† Xifeng Yan§ Ramez Elmasri†
†*University of Texas at Arlington,* §*University of California, Santa Barbara*

***Abstract*—We present GQBE, a system that presents a simple and intuitive mechanism to query large knowledge graphs. Answers to tasks such as *"list university professors who have designed some programming languages and also won an award in Computer Science"* are best found in knowledge graphs that record entities and their relationships. Real-world knowledge graphs are difficult to use due to their sheer size and complexity and the challenging task of writing complex structured graph queries. Toward better usability of query systems over knowledge graphs, GQBE allows users to query knowledge graphs by example entity tuples without writing complex queries. In this demo we present: 1) a detailed description of the various features and user-friendly GUI of GQBE, 2) a brief description of the system architecture, and 3) a demonstration scenario that we intend to show the audience.**

I. Introduction

Consider the scenario where a computer historian is interested in preparing an article on university professors who have designed a programming language and also won an award in Computer Science. If the historian only knows of a few professor-university-award triples, to start writing the article she must have a more comprehensive list of such triples. In general, users are interested in finding entities of various types (e.g., persons, products, organizations) that are related in certain ways. Tasks like the above one are becoming increasingly common in several applications, including search, recommendation systems, and business intelligence.

The historian may have several means to get the professor-university-award list, including using a search engine and querying a knowledge base. Particularly, the historian may tap into *knowledge graphs*, which capture entities and their relationships. For example, Fig. 1 is an excerpt of a knowledge graph, in which the edge labeled *award* from node Donald Knuth to another node Turing Award captures the fact that the person has won the award. Instances of real-world knowledge graphs include DBpedia [2], YAGO [6], Probase [8] and Freebase [3] which powers Google's knowledge graph.[1]

Both users and application developers are often overwhelmed by the daunting task of understanding and using knowledge graphs, due to the sheer size and complexity of such data. Given its schema-less nature, a knowledge graph is typically represented as a set of (subject, property, object) triples. The Linking Open Data community has interlinked billions of RDF triples spanning over several hundred datasets, and these are stored in relational databases, graph databases and triplestores.[2]. In retrieving data from these databases, the norm is often to use structured query languages such as SQL, SPARQL and those alike. However, writing structured queries requires extensive experiences in query language and data model and good understanding of particular datasets [4].

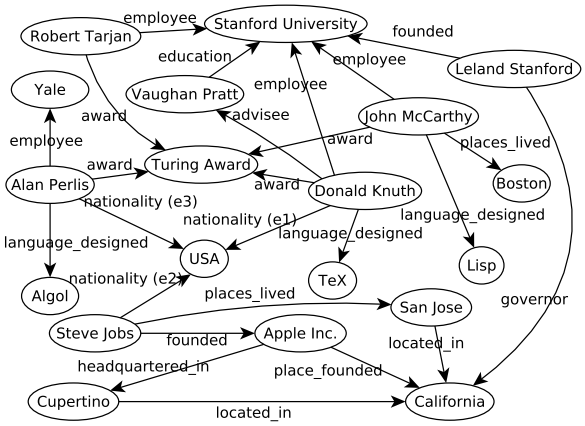

Fig. 1: An Excerpt of a Knowledge Graph

We present GQBE, a system that allows users to query large knowledge graphs by example tuples of entities without having to write complex structured queries. For instance, suppose the aforementioned computer historian knows a few professor-university-award triples that satisfy her requirement, e.g., ⟨Donald Knuth, Stanford University, Turing Award⟩. She can use them as example tuples to GQBE which will find similar answer tuples. Given the knowledge graph in Fig.1, GQBE may find answers such as ⟨John McCarthy, Stanford University, Turing Award⟩ and ⟨Alan Perlis, Yale, Turing Award⟩.

Query by example has a long positive history in relational databases. Some user-friendly querying paradigms for graphs include keyword-based, interactive and visual interfaces, but require patterns, query graphs [9] or meta-paths [7] to be presented as input to the system. GQBE proposes to query knowledge graphs by example entity tuples, where the users do not have to construct any query graph. Instead, GQBE automatically discovers a hidden weighted query graph to capture the query intent. To find approximate answer graphs,

*This work of Li is partially supported by NSF IIS-1018865, CCF-1117369, 2011 and 2012 HP Labs Innovation Research Awards, and the National Natural Science Foundation of China Grant 61370019. The work of Yan was partially supported by the Army Research Laboratory under cooperative agreement W911NF-09-2-0053 (NSCTA). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

[1]http://www.google.com/insidesearch/features/search/knowledge.html

[2]http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

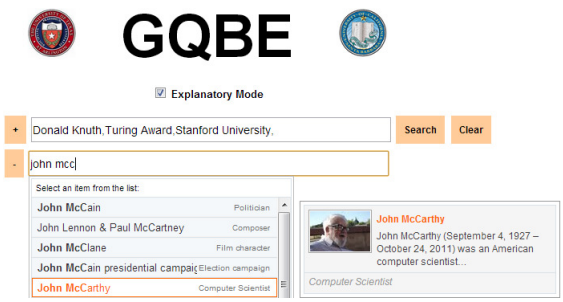

Fig. 2: GQBE's Input Interface

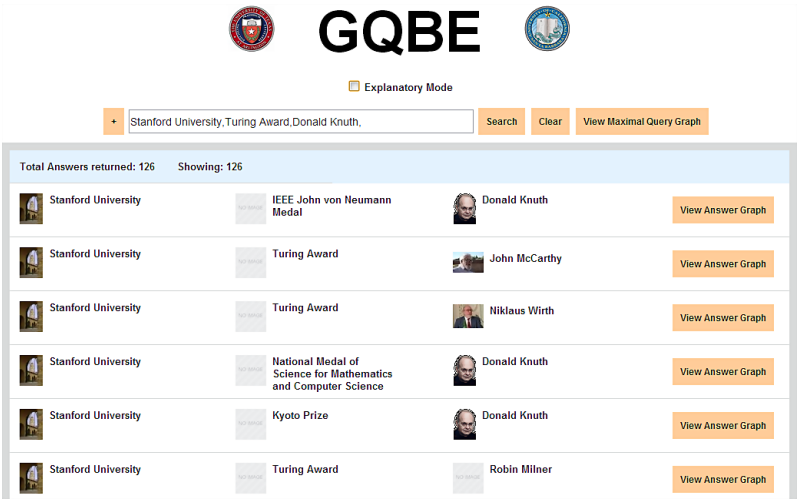

Fig. 3: Interface Displaying Answer Tuples

GQBE models the solution space as a query lattice, which can be prohibitively large. An efficient top-k lattice exploration algorithm is thus used to evaluate the lattice and find similar answer tuples. GQBE obtains high accuracy and good execution time as shown by the extensive experiments in [5].

In this paper, we focus on presenting the functionality and user interface of GQBE (Section II) and demonstrating its usage scenarios (Section IV). Section III provides a brief overview of its underlying query processor, which is detailed in [5]. Its query processor can be used to query different knowledge graphs.

II. GQBE's User Interface and Functionality

GQBE provides several functions that aid in convenient query experience: 1) a simple search box for entering example entity tuple, 2) auto completion of entity names that helps a user find the exact entity she is typing for, 3) provision to provide multiple example tuples, 4) display of the query graph discovered by the system for capturing user intent, 5) display of a ranked list of answer tuples with their corresponding answer graphs that justify the ranking, and 6) an option of *explanatory mode* that further helps users understand the rationale behind the results. The rest of this section provides the details.

GQBE features a simple keyword-based input interface (Fig.2), in which a user enters example tuples of entities known to her. For instance, in Fig.2, the user is in the middle of typing John McCarthy. GQBE offers auto completion, powered by Freebase API. Specifically, when the user partially enters the name of an entity ("john mcc" in Fig.2), GQBE shows a list of suggested entities whose names match the keywords. Hovering the mouse pointer over one suggested entity (John McCarthy in Fig.2) will bring out a summary of the entity from its corresponding Freebase page. This summary can be used to resolve ambiguity among multiple entities with similar names.

Deriving user intent based on a single example entity tuple is a hard task. GQBE allows multiple example tuples to better capture the user intent. As shown in Fig.2, the user has entered the first tuple ⟨Donald Knuth, Stanford University, Turing Award⟩ and is in the middle of entering the second tuple ⟨John McCarthy, Stanford University, Turing Award⟩. More example tuples can be entered by clicking the '+' sign preceding the first entered tuple. Entered example tuples can be removed by clicking the individual '−' signs preceding them. Entered tuples can also be altered by directly changing the keywords in the corresponding search boxes.

Once the user provides example tuples and clicks the "Search" button, GQBE's back-end query processor kicks in. It discovers a hidden weighted query graph, termed the *maximal query graph* (MQG), to capture the user's query intent. GQBE evaluates the MQG to find similar answer graphs and corresponding answer tuples, and ranks them by how well they match the input tuples (details in Section III). Fig.3 shows the result interface displaying the ranked answer tuples. The user can further explore the entities in the answer tuples by clicking on them which opens their corresponding Freebase pages in new Web browser windows.

GQBE assists an advanced user who may be interested in understanding the rationale behind the answer tuples and their ranking. To this end, if the user clicks the "View Maximal Query Graph" button beside the first example tuple's search box (Fig.3), GQBE displays the MQG in a pop-up window. This helps the user find out if her query intent was captured or not. For example, the left graph in Fig.4 is the MQG for input tuple ⟨Donald Knuth, Stanford University, Turing Award⟩. If the user clicks the "View Answer Graph" button to the right of an answer tuple (Fig.3), GQBE displays the corresponding answer graph in another pop-up window. For example, the right graph in Fig.4 shows the answer graph for answer tuple ⟨Robin Miller, Stanford University, Turing Award⟩. The nodes in these graphs can be rearranged for better readability. A user can compare the two graphs to better understand the matching.

If an user is interested in knowing the weights of edges in an MQG and the scores of its answer tuples, the "Explanatory Mode" check-box above the search box (see Fig.2 and Fig.3) can be selected. As shown in Fig.4, this displays a new column in the result interface to show the score of each answer tuple. It also displays the edge weights of the two graphs.

III. GQBE's Query Processor

The architecture of GQBE's underlying query processor is shown in Fig.5. The input to GQBE is a query tuple t instead of

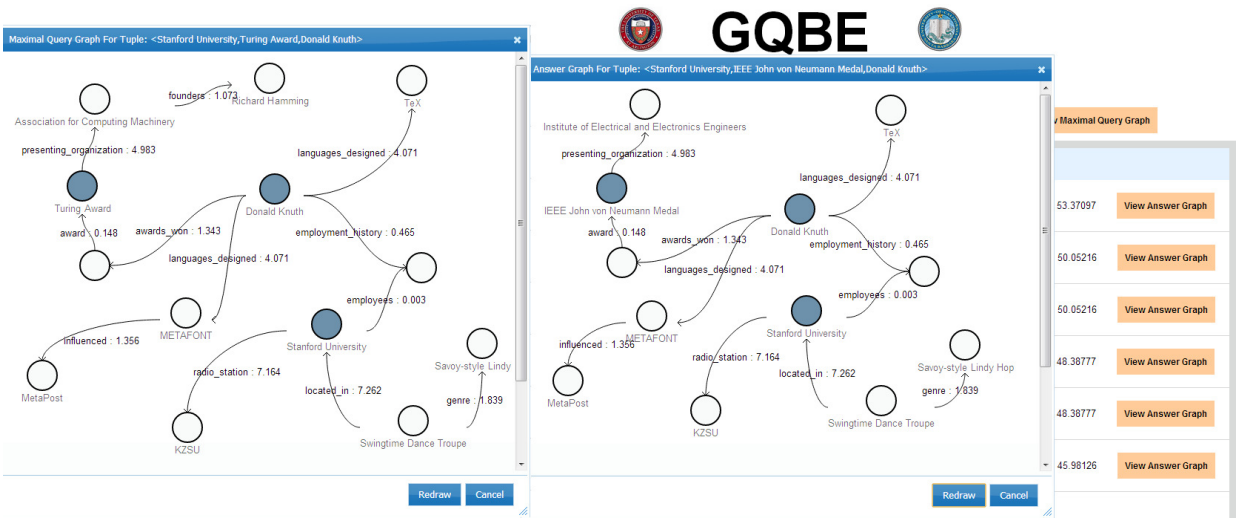

Fig. 4: Interface Displaying a Maximal Query Graph and an Answer Graph

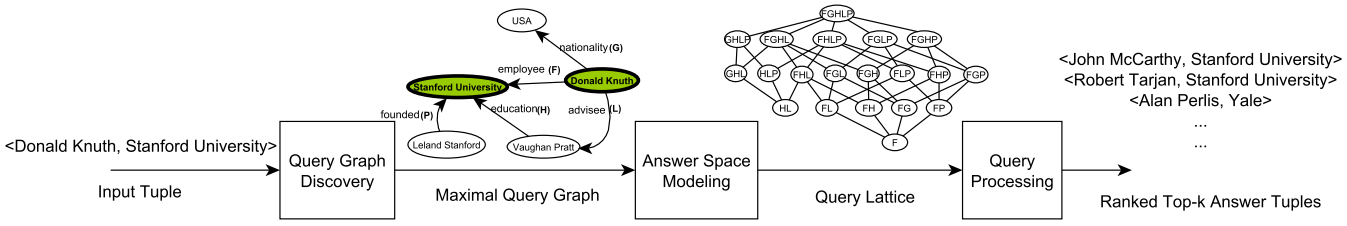

Fig. 5: The Components of GQBE's Query Processor

an explicit query graph. The *query graph discovery* component of GQBE derives a *maximal query graph* (MQG_t) to capture the user's query intent. GQBE further supports multiple query tuples as input which collectively better capture the user intent. There can be a large space of approximate answer graphs since it is unlikely to find answer graphs exactly matching MQG_t. The *answer space modeling* component of GQBE models the space of answer graphs by a *query lattice* formed by the subsumption relation between all possible subgraphs of MQG_t. For efficiently evaluating the large lattice, the *query processing* module employs a top-k lattice exploration algorithm that only partially evaluates the lattice nodes in the order of their corresponding query graphs' upper-bound scores. Various algorithms and other details of the query processor can be found in [5], while we only provide a brief overview here.

A. Maximal Query Graph Discovery

Consider the knowledge graph in Fig.1 and let $\langle$Donald Knuth, Stanford University$\rangle$ be the input query tuple. Neighbors of query entities up to length d are captured to form a *neighborhood graph* (H_t), from which *unimportant* edges are removed as a preprocessing step. In Fig.1, edge e_1=(Donald Knuth, USA) labeled *nationality* represents an important relationship between USA and query entity Donald Knuth. But other edges labeled *nationality* and incident on USA (edges e_2 and e_3 in Fig.1) are deemed unimportant (and thus removed) with regard to query entity Donald Knuth since there can be many citizens of the USA in the knowledge graph. H_t can still be large and its edges must thus be further pruned. We hence rank the remaining edges by weighting them using several distance-based and frequency-based heuristics:

$$\mathsf{w}(e) = \mathsf{ief}(e) \;/\; (\mathsf{p}(e) \times \mathsf{d}^2(e)) \qquad (1)$$

The weight $\mathsf{w}(e)$ of an edge e=(u, v) is 1) directly proportional to its inverse edge frequency, $\mathsf{ief}(e)$, that captures how rare a relationship is globally in the data graph, 2) inversely proportional to its participation, $\mathsf{p}(e)$, that determines the number of edges in the data graph that share the same label and one of e's end nodes (u or v), and 3) inversely proportional to the distance, $\mathsf{d}(e)$, that captures the distance of edge e from the query entities. A greedy heuristic is used to choose the MQG_t (example MQG_t in Fig. 5), that is an m-edged subgraph of H_t containing all query entities, while maximizing the total edge weight.

B. Answer Space Modeling

We model the space of possible query graphs by a lattice. The lattice in Fig. 5 corresponds to the maximal query graph of query tuple $\langle$Donald Knuth, Stanford University$\rangle$. Each query graph in the lattice is a connected subgraph of MQG_t and contains all query entities. The bottom-most nodes in the lattice are called the *minimal query trees* (nodes F and HL in Fig. 5) which together capture all relationships between the input entities. The top-most node (FGHLP in Fig. 5) is the MQG_t, and other lattice nodes have exactly one edge more than its children. Answer graphs to these query graphs are also subgraphs of the data graph and are structurally isomorphic to the query graph. The score of a query graph Q is equal to the sum of all its

edges' weights. Given an answer graph, nodes corresponding to the query tuple entities are projected as its answer tuple. Thus the answer tuples are approximate answers to MQG_t.

C. Query Processing

The data graph can be represented as a set of RDF triples (source, property, object). We use relational database techniques to store and query them. We particularly adopt the vertical partitioning method [1] and maintain a table for each property with two columns $(subj, obj)$, for the edges' source and destination nodes, respectively. For efficient query processing, two in-memory hash tables are created on each table, using *subj* and *obj* as the hash keys, respectively.

A query graph can be evaluated using a multi-way join query. For instance, the MQG_t in 5 corresponds to SELECT F.subj, F.obj FROM F,G,H,L,P WHERE F.subj=G.sbj AND F.obj=H.obj AND F.subj=L.subj AND F.obj=P.obj AND H.subj=L.obj. We use right-deep hash-joins to process such a query. Consider the topmost join operator in a join tree for query graph Q. Its left operand is the *build relation* which is one of the two in-memory hash tables for an edge e. Its right operand is the *probe relation* which is a hash table for another edge or a join subtree for Q'=$Q-e$ (i.e., the resulting graph of removing e from Q). GQBE uses a *best-first* exploration strategy of the lattice to obtain top-*k* answers. It explores the query lattice in a *bottom-up* way, starting with the minimal query trees. After a query graph is processed, its answers are materialized in files. To process a query Q, at least one of its children Q'=$Q-e$ must have been processed. The materialized results for Q' form the probe relation and a hash table on e is the build relation. The best-first strategy always chooses to evaluate the most promising lattice node Q_{best} from a set of candidate nodes. Q_{best} is the candidate with the highest upper-bound score. If processing Q_{best} does not yield any answer graph, Q_{best} and all its ancestors are pruned and the upper-bound scores of other candidate nodes 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.

IV. Demonstration Plan

We use the running example of the computer historian to demonstrate the system. In this demo we focus on Freebase as the knowledge graph that contains around 28M nodes, 47M edges and 5,428 distinct edge labels. A demonstration video of GQBE can be found at http://www.youtube.com/watch?v=4QfcV-OrGmQ .

Scenario A The historian only knows the name of a professor but not the professor's university and award. Thus she uses a single-entity example tuple ⟨Donald Knuth⟩ as the input:
(A1) Type in Donald Knuth in the search box. As she starts typing, the auto completion feature displays the plausible options.
(A2) Choose the correct entity by clicking the mouse on the presented entity after reading through the summary.
(A3) Click the "Search" button to see the ranked answer tuples.

Scenario B The historian knows of a more complex example tuple with two entities—a professor-university tuple ⟨Donald Knuth, Stanford University⟩:
(B1) Click the "Clear" button to clear out the input and output of the previous search.
(B2) Repeat steps (A1)-(A2) for the two entities Donald Knuth and Stanford University, in turn.
(B3) Click the "Search" button to see the ranked answer tuples.

Scenario C This scenario shows how to provide a three-entity example tuple, ⟨Donald Knuth, Stanford University, Turing Award⟩, that tries to better capture the user intent by explicitly listing the award won too:
(C1) Click the "Clear" button next to the first search box to clear out the previous search scenario.
(C2) Repeat steps similar to (A1)-(A2) for the three entities Donald Knuth, Stanford University and Turing Award, in turn.
(C3) Click the "Search" button to see the ranked answer tuples.

Scenario D If ⟨John McCarthy, Stanford University, Turing Award⟩ is another known example tuple satisfying her query requirement, the historian can provide multiple example tuples as the input:
(D1) Click the "Clear" button next to the first search box to clear out the previous search scenario.
(D2) Enter ⟨Donald Knuth, Stanford University, Turing Award⟩, the first tuple, by following steps (C1)-(C2).
(D3) Click the '+' button to the left of the first search box. A new search box is created below it.
(D4) Enter tuple ⟨John McCarthy, Stanford University, Turing Award⟩ in the second search box by following steps similar to (C1)-(C2).
(D5) Click the "Search" button to see the ranked answer tuples.

Viewing Results Users can check the rationale behind the answer tuples and their ranking by performing the following steps after any of the previous 4 scenarios.
(E1) Click the "View Maximal Query Graph" button. The maximal query graph discovered by the system is displayed.
(E2) Click the "View Answer Graph" button next to any answer tuple to see its answer graph. The degree of matching can be observed by comparing the maximal query graph and the answer graph.
(E3) Select the "Explanatory Mode" check-box to view the details of edge weights and answer tuple scores.
(E4) Click any answer entity to find out more details about it from its Freebase page.

References

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB'07*.

[2] 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.

[3] 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.

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

[5] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. *CoRR*, abs/1311.2100, 2013.

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

[7] Y. Sun, J. Han, X. Yan, P. S. Yu, , and T. Wu. PathSim: Meta path-based top-k similarity search in heterogeneous information networks. *VLDB'11*.

[8] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. In *SIGMOD*, pages 481–492, 2012.

[9] X. Yu, Y. Sun, P. Zhao, and J. Han. Query-driven discovery of semantically similar substructures in heterogeneous networks. In *KDD'12*.