Querying Wikipedia Documents and Relationships

Huong Nguyen Thanh Nguyen Hoa Nguyen Juliana Freire
School of Computing and SCI Institute, University of Utah
{huongnd,thanhh,thanhhoa,juliana}@cs.utah.edu

ABSTRACT

Wikipedia has become an important source of information which is growing very rapidly. However, the existing infrastructure for querying this information is limited and often ignores the inherent structure in the information and links across documents. In this paper, we present a new approach for querying Wikipedia content that supports a simple, yet expressive query interfaces that allow both keyword and structured queries. A unique feature of our approach is that, besides returning documents that match the queries, it also exploits relationships among documents to return richer, multi-document answers. We model Wikipedia as a graph and cast the problem of finding answers for queries as graph search. To guide the answer-search process, we propose a novel weighting scheme to identify important nodes and edges in the graph. By leveraging the structured information available in infoboxes, our approach supports queries that specify constraints over this structure, and we propose a new search algorithm to support these queries. We evaluate our approach using a representative subset of Wikipedia documents and present results which show that our approach is effective and derives high-quality answers.

1. INTRODUCTION

As the volume of information on Wikipedia increases, there is a growing need for techniques that allow users to query this information. Although users can pose keyword-based queries to search for information, these queries fail to capture both the structured information present in these documents and the connectivity among them. To illustrate this limitation, consider the following query, where a user wants to find *"all actors that starred in a movie with Brad Pitt"*. Answers to this query are available in Wikipedia data, but because they span multiple documents, they cannot be retrieved directly. A possible way for a user to find these answers is to first visit Brad Pitt's Wiki page, read the document, follow its internal links to the movies he played in, and then find all actors in those movies. This process, however, is complex and time-consuming. We posit that by leveraging the structure of the objects represented in Wikipedia documents and the relationships implied by links between documents, it is possible to answer the query above in an automated fashion. For example, by taking into account that there are document classes (or entity types) *ACTOR*, *MOVIE*, which are connected by a *star in* relationship, to answer above query, we can search for all movies that have Brad Pitt as a star and then find all actors that are stars in those movies.

One possible approach to represent Wikipedia documents and relationships is to capture them in a relational database. This approach allows sophisticated queries in the form of structured query languages such as SPARQL and SQL [2]. However, substantial effort is required to extract information from Wikipedia and store it in a database. Even if we consider only information in infoboxes, a global schema needs to be created for each document class, and each infobox needs to be mapped into the global schema. Due to the wide variability in infobox schemas and the large number of classes, this can be challenging. Another disadvantage of this approach is usability: it requires users to have a-priori knowledge of the schema and to be familiar with the query language.

In this paper, we propose WIKIQUERY, a new approach to query Wikipedia documents that addresses these limitations. WIKIQUERY supports simple, yet expressive queries, and returns answers that span multiple documents without requiring the information to be integrated in a database. Inspired by approaches for querying relational databases using keyword-based queries [6, 13, 1, 11], WIKIQUERY models Wikipedia as a graph and derives answers by traversing the graph—in essence, it *discovers* how the different documents are *joined*. An important challenge we address is that, unlike in relational databases where a few tables are connected exclusively through key-foreign key relationships, in Wikipedia, not only there is a large number of document types (*i.e.,* many distinct schemas) but also the documents are highly connected, by a potentially large number of links (*i.e.,* many distinct relationships). As a result, blindly following all the possible paths in this dense graph is likely to lead to several irrelevant answers. Based on the intuition that popular relationships are the most sought after, our approach aims to find the best answers ranked by popularity and compactness. As we discuss below, this assumption is practical and derives high-quality answers.

Besides supporting simple, keyword-based queries, WIKIQUERY also supports constraint-based queries (c-queries). These allow users to more precisely specify their information need. For example, while the query *Paris* returns all documents that contain this keyword, for users looking for touristic information, a better query would be `city(name=Paris)`, or for users interested in movies, `actor(name=Paris)` would be preferable. We leverage this additional information to develop a search algorithm that selects important entities and relationships to improve the quality of query results. In particular, we propose a novel weighting scheme that promotes important entities and relationships while de-emphasizing

WebDB '10 Indianapolis, IN USA

common ones. In addition to weights assigned to nodes and edges, we utilize the entity types specified in user query constraints to provide the search algorithm more context to help in answer selection.

Our main contributions can be summarized as follows: We define a layered data model that captures both schema and data-level features of Wikipedia documents; we propose a query model that supports both keyword-based and c-queries over Wikipedia documents that returns multi-document answers; we design a weighting scheme for documents and relationships between documents to guide the search for answers and propose a new algorithm to support c-queries that takes the input query into account; we perform an experimental evaluation and present results which show that our approach derives high-quality answers.

2. SOLUTION OVERVIEW

In what follows, we give a brief overview of WIKIQUERY and discuss how it relates to and extends previous approaches to supporting keyword-based queries over relational databases.

2.1 Linked Answers for Wikipedia Queries

An important goal of our work is to derive multi-document answer to queries over Wikipedia. To do so, we model Wikipedia as a graph. Each node in this graph corresponds to a document which contains an infobox. An infobox is a table that summarizes, in a structured way, the important attributes of the entity represented in a document.[1] Each item in an infobox consists of an attribute name, which describes its meaning, and a set of values for the attribute. For example, Figure 1 shows the infobox for `Steven Spielberg`'s page. It contains four attributes: `Born`, `Occupation`, `Years Active`, and `Spouse`. Attribute values of infoboxes may be atomic or contain links to other Wikipedia documents, which represent *references* (or relationships) among entities. These references correspond to the edges in our graph. For example, in Figure 1, the attribute `Years active` contains only one atomic value `1984 - present`, while the attribute `Spouse` contains 2 reference values: `Amy Irving` and `Kate Capshaw`.

Definition 1. An *infobox* I contains a set of attribute-value pairs $\langle A,V \rangle = \{\langle a_i,V_i \rangle\}$ where a_i is an attribute name and V_i is a set of values $\{v_1, v_2, ..., v_k\}$. Each value $v_j \in V_i$ is either an *atomic* value or a *reference* to another infobox (*i.e.,* a value that contains a hyperlink to another entity).

Each entity belongs to one or more *entity types* which identify document classes with similar semantics. Consider the example in Figure 1. The entity "Steven Spielberg" belongs to three different entity types: *Director, Producer*, and *Screenwriter*. We note that the problem of recognizing entity types for Wikipedia document is out of the scope of this paper. This information can be retrieved from Wikipedia categories, infobox templates, or provided by some systems such as DBpedia [2] and Yago [3]. In this paper, we focus on exploiting infoboxes (entities) and their relationships to derive high-quality results for user queries.

Given an input query, be it a keyword-based query or a structured query, our goal is to identify a set of connected infoboxes that together provide the answer to the query.

Definition 2. Let $G = \{I, R\}$ be the graph induced by a set of infoboxes $I = \{i_1, i_2, ..., i_{n_i}\}$ and a set of references $R = \{r_1, r_2, ..., i_{n_r}\}$ that connect these infoboxes. Given an input query Q, the answers to this query consist of a set S of minimal trees $G' \subset G$ of related infoboxes $I' \subset I$ and references $R' \subset R$ that satisfy the query:

[1]In the remainder of the paper, we refer to infobox and entity interchangeably.

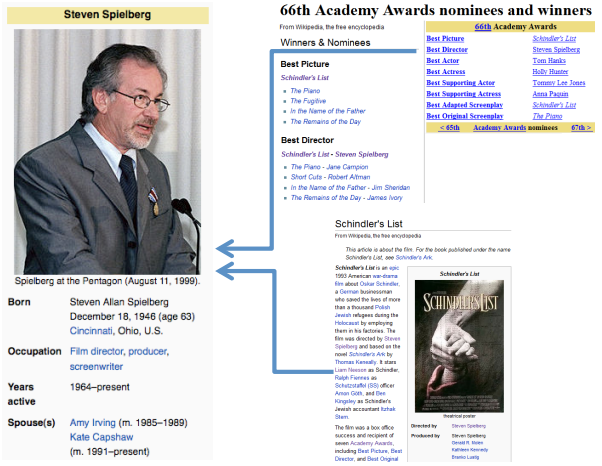


Figure 1: A query answer consisting of infoboxes from multiple Wikipedia documents.

$$G' = \{I', R'\} \mid G' \vDash Q \text{ and } \nexists G'' \subset G' \text{ such that } G'' \vDash Q$$

Figure 1 shows a multi-document answer returned by WIKIQUERY. To derive these multi-document answers, we have devised an approach that was inspired by BANKS [6]. Below, we briefly review the important components of BANKS and discuss its limitations in the context of Wikipedia.

2.2 BANKS: Overview

With BANKS, Bhalotia et al. [6] introduced the idea of supporting keyword queries over relational databases. They model a database as a directed graph whose *nodes* are tuples and *edges* between these nodes correspond to key-foreign key relationships between the tuples. Both edges and nodes are associated with a pre-defined *weight*. Each edge is also associated with a *backward edge* which reverses it to enable the graph to be traversed in two directions. Using this graph, BANKS casts the problem of keyword searching over relational database as a problem of finding connected nodes that contain the keywords provided in the query, more specifically, query answers are rooted Steiner trees where the leaf nodes contain the keywords. The algorithm performs a bidirectional search by using two iterators: an *incoming iterator*, which starts from the leaf nodes and travels to root nodes using backward edges, and an *outgoing iterator*, which starts from the root node and travels to the leaf nodes until they find a common node.

BANKS relies on the node and edge weights to guide the search. The node weight is defined as a logarithm scale function of the in-degree of the node—this is based on the assumption that the more references a node has, the more important it is. Edge weights are set to 1 to reflect the fact that, initially, all nodes are equally close. Although node weights remain unchanged during the search, BANKS adopts an *activation model* that uses edge weights to reorder nodes while expanding the search process to help find the shortest path. The activation model works by spreading a proportion of the activation of a node being visited to its parents, i.e., the closer to its children a node is, the more activation it gets.

BANKS also uses node and edge weights to compute the edge score and the node score of the answer tree during ranking. The edge score of an answer tree is the total weight of all edges in that tree. The node score is the summarization of the root node's weight and all the leaf nodes' weight. To promote concise answers, their ranking function assigns a smaller score to large trees, and they prefer trees with higher weight on the root and the leaves.

Limitations of BANKS for Querying Wikipedia. Although the

BANKS framework was shown to be effective in the context of relational databases, it has important limitations when it comes to querying Wikipedia documents. The framework relies on key-foreign key relationships to build the data graph, and this information is not readily available on Wikipedia. While any tuple in a database belongs to a single relation, a Wikipedia document may belong to many classes, consequently, two entities may be connected by multiple relationships. Furthermore, information in Wikipedia can be heterogeneous—even entities belonging to the same class may have different schemas. Last, but not least, BANKS ranking algorithm depends only on the structure of the data graph and is independent of the context and semantics of input queries. As we discuss later, higher-quality results are derived when the context where the keywords sought appear is specified in queries. As we discuss below, to address these limitations, WIKIQUERY makes use of a two-layer data model and weighting scheme that takes into account specific features of Wikipedia and introduces a query-dependent search algorithm.

3. THE LAYERED GRAPH MODEL

Because the Wikipedia document graph is highly-connected, and even within a fixed document class, the structure of infoboxes is heterogeneous, to derive high-quality answers, WIKIQUERY needs to automatically assess the importance of the references between the infoboxes. It does so by using a two-pronged approach. First, it models the Wikipedia contents using two graphs: the *schema graph*, which captures the *references between entity types*; and the *data graph* which captures the *references between entities*. And second, it applies a weighting scheme that assigns values to nodes and edges in these graphs that reflect their importance.

Data Graph. We model *entities* and *references* using a labeled graph. A data graph D is a labeled weighted directed graph with *entities* as nodes and *references* as edges. We model each reference as two edges: a *forward edge* and a *backward edge*. For each reference λ from entity v to v', we create a forward edge from node v to node v' and a backward edge from v' to v, both with label λ. We denote forward edge e from v to v' with label λ as $e = v \xrightarrow{\lambda} v'$ and its corresponding backward edge as $e^b = v \xleftarrow{\lambda} v'$. Figure 2(a) illustrates a data graph, where we omit the backward edges for simplicity of presentation.

Schema Graph. The schema graph S is a labeled weighted directed graph that summarizes the information contained in a data graph D. Recall that an entity may belong to multiple entity types. Let T_v be the set of entity types that an entity v belongs to, we construct S from D as follows:

- For each node $v \in D$, add node t to the set of nodes in S for each entity type in $t \in T_v$.
- For each edge $e \in D$, $e = v \xrightarrow{\lambda(e)} v'$, add to S an edge with same label $\lambda(e)$ from each entity type $t \in T_v$ to each entity type $t' \in T_{v'}$.

Figure 2(b) shows the schema graph for the data graph of in Figure 2(a). Note that since the entity `Steven Spielberg` belongs to the types `producer` and `director`, and it is connected to film `Schindler's List` via edges `Produced By` and `Directed By`, corresponding edges are created in the schema graph to connect the entity type `Film` to `Director` and `Producer`.

3.1 Weighted Schema Graph

Based on the observation that entities with high frequency in Wikipedia, on average, have a large number of incoming edges, we assign lower weights to such nodes. Similar to information retrieval systems which assign low importance to common terms [4], we penalize nodes that are less likely to contribute important information to answers. For every node $t \in S$, let $\{E_t\}$ be the set of entities v with type t. The weight of t is defined as follows:

$$\eta_S(t) = \frac{|E_t|}{\sum_{v \in E_t} Indegree(v)} \quad (1)$$

We use two measures to assign weights to the edges in a schema graph: *node dependence* and *edge strength*. The *dependence* between nodes (or entity types) in a schema graph measures the probability of observing a node t' given a node t. For example, given an `Actor`, the probability of observing `Film` is higher than that of observing `Politician`. More formally, the dependence δ between schema nodes t and t' is defined as $\delta(t', t) = \frac{\sum_{v \in E_t} C(v, t')}{|E_t|}$ where E_t and $E_{t'}$ are the sets of entities with types t and t', respectively, and

$$C(v, t') = \begin{cases} 1, & \text{if } \exists\ e = v \rightarrow u,\ u \in E_{t'},\ e \in \text{D} \\ 0, & \text{otherwise.} \end{cases}$$

Note that $\delta(t', t)$ different from $\delta(t, t')$.

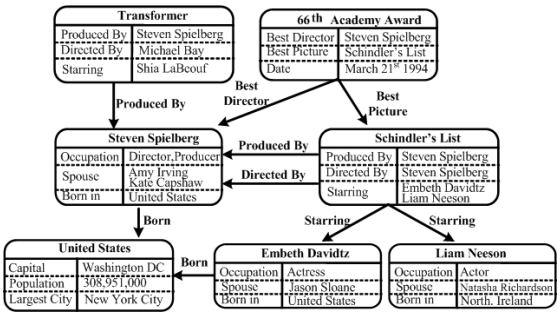
Besides taking nodes into account, WIKIQUERY also considers the relationships between nodes to rank answers. Since there can be many relationships between two entities (documents) in Wikipedia, we use *edge strength* to capture how 'close' entity types are so that we can distinguish relationships regarding their importance. For example, the edge strength of `Starring` given `Film` and `Actor` should be higher than the strength of `Directed by` given the same types. The *edge strength* of an edge $e = t \xrightarrow{\lambda(e)} t'$ is defined as the normalized likelihood of the reference $t \rightarrow t'$ given the edge label $\lambda(e)$:

$$P(t, t'|\lambda(e)) = \frac{\sum_{v \in E_t,\ v' \in E_{t'}} (v \xrightarrow{\lambda(e)} v')}{\sum_{\forall v, v'} (v \xrightarrow{\lambda(e)} v')}$$

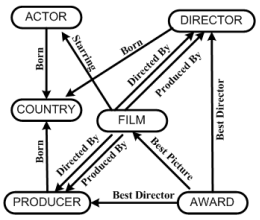
The strength of edge e is denoted as $\chi(t, t', \lambda(e))$:

$$\chi(t, t', \lambda(e)) = \frac{P(t, t'|\lambda(e))}{\max_{\lambda(e_i)} P(t, t'|\lambda(e_i))}$$

Similar to δ, χ is not symmetric.



(a) Data Graph



(b) Schema Graph

Figure 2: The layered graph model.

Recall that edge weight is computed by combining *dependence* and *edge strength*. However, when the sink node belongs to a prevalent type, it may represent a relationship that is also common, and as a result, its strength should be attenuated. Edge strength is thus computed as follows:

$$Importance(t, t', \lambda(e)) = \delta(t', t) * \chi(t, t', \lambda(e)) * \eta(t')$$

Because our search algorithm (Section 4) finds the shortest paths among the candidate nodes, we annotate e with weight $\mu_S(t, t', \lambda(e))$ as the inverse of $Importance$ with intuition that the shorter path is the path with the higher relevance:

$$\mu_S(t, t', \lambda(e)) = \frac{1}{Importance(t, t', \lambda(e))} \quad (2)$$

3.2 Weighted Data Graph

A node in data graph D is assigned a weight to determine its importance in the dataset. While in the schema graph node weights determine the importance of the entity types, in the data graph they capture the relative importance among entities of the same type. Intuitively, entities that have more references are more *important*, *i.e.,* popular entities are favored among entities with the same type. Driven by this intuition, we compute the weight of a node v in D as a function of its indegree and its entity type weight. Formally, the weight of a node v with entity types T_v is

$$\eta_D(v) = log(1 + \frac{indegree(v)}{\max_{t \in T_v} \eta_S(t)}) \quad (3)$$

Forward Edge Weight. We determine the edge weights in D using μ_S of the edges in the corresponding schema graph S. Since an entity may belong to multiple entity types, an edge in D may be associated to multiple edges in the S. Therefore, we assign edge weight in the data graph using the weight of the most important edge among associated entity types in S. Let $e = v \xrightarrow{\lambda(e)} v'$ be an edge in the D and T_v and $T_{v'}$ be the set of entity types of v and v', we assign edge weight to e as follows. For each entity type $t \in T_v$ and $t' \in T_{v'}$, we find corresponding edge weight w' of edge $e' = t \xrightarrow{\lambda(e)} t'$ in S and assign this weight to e if current weight of e is smaller than w'. The forward edge weight is thus defined as:

$$\mu_D(e) = \min_{t \in T_v, t' \in T_{v'}} \mu_S(t, t', \lambda(e)) \quad (4)$$

Backward Edge Weight. For each forward edge, the weight of the corresponding backward edge depends on the number of edges incident on v' [6], so that the backward edges to nodes with prevalent types are penalized (*e.g.,* `Film` ← `Country`). We assign the weight to the backward edge of a forward edge e as:

$$\mu_D(e^b) = \mu_D(e) * log(1 + indegree(v')) \quad (5)$$

Note that if the backward edge already exists in the graph as a forward edge, we assign its weight as the minimum value between its forward and the backward edges weight calculated as above.

4. QUERYING AND SEARCHING

Keyword Queries. A keyword query K consists of a set of keywords $k_1, k_2, ..., k_n$. Without loss of generality, we assume these keywords form a conjunction.[2] For example, to "find actors who worked with director `Steven Spielberg`" we could construct the following query: {"Actor","Director Steven Spielberg"}.

To support keyword queries, we index all values of entities in the data graph. A traditional way to construct this index is to build an inverted list that connects terms to entities. But since users may enter metadata as terms in the query, *e.g.,* "director Steven Spielberg", we index entity types and attribute names. To answer queries using this indexing structure, for each keyword k_i, we find a set of entities E_i in the data graph that match k_i. Let $\mathcal{E} = E_1 \cup ... \cup E_n$ be the set of candidate leaf nodes of the answer tree. We apply the Bidirectional Expanding Search algorithm [13], using the set $\mathcal{E}$, to find a common node connecting at least one node in each E_i and construct the answer trees.

Constraint-Based Queries. A keyword query does not require users to have knowledge of the data schema, but it limits their ability to precisely specify an information need. To address this problem, WIKIQUERY supports conjunctive constraint-based queries (c-queries).

Definition 3. A c-query consists of a conjunction of constraints $q = c_1, c_2, ..., c_n$. Each constraint c_i is of the form *Entity Type(AttributeName op Value)*, where an *op* $\in \{=, !=, >, <>=, <=\}$.

Note that although more expressive than keyword queries, c-queries are still simple to specify and they do not require users to have knowledge of how different entities are connected (joined). For example, a c-query to answer the question "find actors born after 1980 who worked with director `Steven Spielberg`" could be constructed using the following two constraints: *Director(name = Steven Spielberg) and Actor(born>1980)*. One possible answer to this query includes joins between `Director`, `Film`, and `Actor`. We leverage the additional information provided in a c-query to derive a search algorithm which is able to select entities and relationships that best match the query and lead to better result ranking.

Query-Dependent Search Algorithm. As discussed in Section 3, our approach to result ranking is based on the intuition that a user is most likely interested in popular documents and relationships. Although this assumption is practical and reflects the requirements of many applications that use keyword queries (see Section 5), it has an important limitation: it does not take into account the context in which the keywords appear. To address this problem, we compute node and edge weights in data graph based on metadata information provided by the users in c-query: instead of building a data graph with fixed weights, we leverage semantics provided in c-queries to compute query-dependent weights. We propose search algorithm which combines navigation using query-dependent weights with the expanding iterators of the Bidirectional Search algorithm (Section 2.2), whereby the iterators are made to focus on the most relevant nodes and edges with respect to the query.

To each node v with entity type t in the data graph that satisfies a constraint in c-query q, we assign a query-dependent weight:

$$\eta^q_D(v) = log(1 + \frac{indegree(v)}{\eta_S(t)}) \quad (6)$$

Let e be a forward edge from v to some node v', $e = v \xrightarrow{\lambda(e)} v'$, then e is assigned a query-based edge weight $\mu^q_D(e)$:

$$\mu^q_D(e) = \min_{t' \in T_{v'}} \mu^q_S(t, t', \lambda(e)) \quad (7)$$

where $T_{v'}$ is the set of possible entity types of v', t is the type of v which is given by the query, and $\mu^q_S(t, t', \lambda(e))$ is the edge weight of e (Equation (2)).

The query-dependent weights are integrated into the bidirectional search as follows. We start with the sets of nodes that match the query constraints. When an expanding iterator visits the successor node v' of a keyword node v, we determine the best type for v' together with the best relationship between the two nodes based on the type of v. This navigation process iterates to explore the next level of entities simultaneously with the expansion process. When the navigator encounters a node that has already been visited for which it derives a conflicting decision upon the node type,

[2]Disjunctive queries can be obtained as an union of the results of multiple conjunctive queries.

Algorithm 1 Query-Dependent Search Algorithm

1: **Input:** Keyword node sets $\{S_1, ..., S_n\}$, Entity Types $\{E_1, ..., E_n\}$
2: **Output:** Set of answers
3: v = Expanding_Iterator.pop()
4: **if** v creates an answer **then**
5: Construct an answer with root v
6: Add the answer to output heap
7: **else**
8: **for all** v' in $surrounding[v]$ **do**
9: Navigate(v,v')
10: Update_Path(v,v')
11: **end for**
12: **end if**

Algorithm 2 Navigate(v,v')

1: **if** $v'.type == null$ **then**
2: **for all** $t' \in type[v'], l \in label[v, v']$ **do**
3: $e = v \xrightarrow{l} v'$, where v' has type of t'
4: Find l, t' that minimize $\mu_D^q(e)$ using Equation (7)
5: **end for**
6: $v'.type = t', e(v, v').label = l$
7: **else**
8: **for all** $t' \in type[v'], l \in label[v, v']$ **do**
9: $s = \mu_D^q(e), e = v \xrightarrow{l} v'$ where v' has type of t'
10: **for** each keyword i **do**
11: p = previous node of v' in the path from v' to S_i
12: $e' = p \xrightarrow{l'} v'$, where v' has type of t'
13: $s+ = \mu_D^q(e')$
14: **end for**
15: $v'.type = t', e(v, v').label = l$ in which (t', l) minimize s
16: **end for**
17: **end if**

we choose the type with higher total relevance score to its predecessors. This procedure is described in Algorithm 1. The bidirectional search algorithm is executed simultaneously with our algorithm, which is represented by Procedure Expanding_Iterator.pop() and Update_Path(v,v'). Expanding_Iterator.pop() in line 3 returns node v with the highest priority for exploring. If this node has paths to all the keyword sets, we build an answer with this node as the root of the tree and then insert it into a heap for reordering by the relevance score of the whole tree (line 4-6). Procedure Update_Path(v, v') updates the path for v' if v' has a better path to any keyword k_i via v, and insert v' into the fringe waiting for the iterator to pop out. More details of Expanding_Iterator.pop() and Update_Path(v,v') can be found in BANKS paper[13]. Before the search algorithm expands the new surrounding nodes $\{v'\}$ of v, Algorithm 2 determines the most relevant type for v' and most relevant relationship of (v, v'), given the type of v using the query-based weights presented above.

5. PRELIMINARY EXPERIMENTS

We evaluate WIKIQUERY using keyword and c-queries derived from the same set of user questions. To assess the effectiveness of our approach, we compare the quality of the results it derives against results derived by BANKS. We note that although there are other approaches to querying Wikipedia [2, 15], their work is orthogonal to ours. In particular, it is not possible to perform a fair comparison against these since they require user to specify the join paths between entities.

Dataset. Our data set consists of a graph with $22,000$ nodes, corresponding to infoboxes, and $230,000$ edges connecting these nodes. We obtained these infoboxes by crawling Wikipedia documents under the top category `Film`.

Metrics. Since the expected number of results returned varies from query to query, using a top-K precision evaluation can be misleading. Instead, we use the normalized discounted cumulative gain

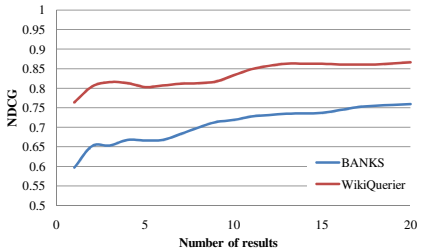


Figure 3: Keyword query NDCG using top-K results.

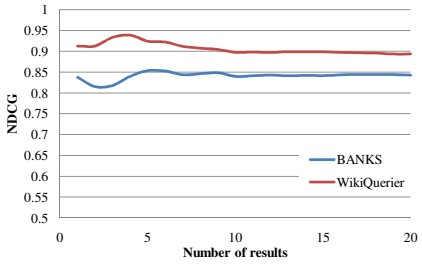


Figure 4: Constraint-based query NDCG using top-K results.

(NDCG) as a measure [12], which has been widely used in IR benchmarking [12] as well as in other search and ranking problems [15]. NDCG is independent of the number of results returned by the system for a given query and it allows us to incorporate different levels of relevance. For each query, we select the top 20 results and presented them to 5 evaluators. Evaluators were required to indicate whether each result is "highly relevant", "relevant", "somewhat relevant", "unclear" or "not relevant". We define the discounted cumulative gain (DCG) as follows:

$$DCG(i) = \begin{cases} G(i) \text{ if } i = 1 \\ DCG(i-1) + \frac{G(i)}{log(i)} \text{ otherwise} \end{cases}$$

where i is the i^{th} result in the result, and $G(i)$ is the relevance of that result, *i.e.,* the average relevance of that result over all evaluators. The *NDCG* is obtained by dividing DCG by the DCG of the ideal ranking.

Results. Figure 3 presents evaluation results for keyword queries. WIKIQUERY always outperforms BANKS, from 16% in NDCG at $K = 1$ to 10% at $K = 20$. However, in this configuration, both BANKS and WIKIQUERY have lower NDCG at smaller K. A possible explanation for this behavior is that keyword queries fail to accurately express the users' intentions. For example, consider a keyword query "List movies whose actors are politicians", a possible encoding for this query as a set of keywords is "movie, politician actor"; and the results for this may contain a list of movies that are produced by some producer who is a politician. Without the ability to provide contextual information, many interpretations of a query are possible which may not reflect the information need of a user.

c-queries try to overcome this limitation by allowing users to specify entity types as well as constraints involving entity attributes and values. Figure 4 shows the NDCG values obtained by WIKIQUERY and BANKS for c-queries. The results show that WIKIQUERY is not only more effective than BANKS, but it also obtains high NDCG at higher rank results. Given a c-query "Movie(name=harry potter) AND Actor(born=England)", WIKIQUERY will assign a higher rank to actors who acted in Harry Potter movies than producers or directors. This is in contrast to BANKS, which adopts a fixed structure of the data graph and does not consider the input query to rank results.

6. RELATED WORK

Although approaches for supporting keyword queries in RDBMS model a database as a graph, in some, nodes correspond to relations [1, 11, 10] while in others, nodes represent tuples [5, 7, 9, 6, 13]. Relation-based approaches have important limitations in the context of querying Wikipedia documents: not only do they need the database schema to process keyword queries using SQL, but they also require users to specify structured queries (and join paths), which can be challenging in the presence of a large number of related entities with heterogeneous schemas. Tuple-based approaches evaluate a keyword query by utilizing the weights associated with nodes and edges in the data graph. BANKS is the work most closely related to ours and it has been discussed in Section 2.2. The goal of DBPF [7] is to find the optimal solution and thus, query evaluation is very computationally expensive making it unsuitable for large data. BLINKS [9] utilizes the backward search strategy of BANKS, but it focuses on finding the root node of the answer tree, not relationships between the nodes of the tree. In contrast to our work, the focus of BLINKS is on the search efficiency, not answer quality. ObjectRank [5] uses an authority-based ranking strategy and similarly to BLINKS, returns individual nodes instead of trees as answers. Similar to our approach, SphereSearch [8] and [16] also return multiple-document answers, however, they do not take into account specific labeled relationships between entities.

Also related to our work are approaches to querying Wikipedia. Freebase, Powerset and Faceted Wikipedia Search (FWS)[3] allow keyword queries over structured information extracted from Wikipedia documents, and similar to our approach they also return different entities that are related to the terms in the query. DBpedia [3] and NAGA [15, 14] model structured information extracted from Wikipedia documents as RDF triples and use the SPARQL query language for querying the data. They allow sophisticated queries but the users need not only to know the exact data schema and specify join paths, but they must also be familiar with SPARQL. This negatively impacts the usability of the query interface, since even simple queries can become too complicated for naïve users [2]. Similar to our approach, NAGA [15] and STAR [14] employ a ranking model based on informativeness, confidence, and compactness. However, they require users to explicitly specify the join paths.

Yu and Jagasish [17] studied the problem of schema summarization and proposed a metric to assess importance of schema elements. Although there are commonalities in our approach assigning weight to the graph elements, their goal is different: they aim to help users explore a schema. In addition, they not consider the importance of *relationships* between elements which play a role in our search and ranking algorithms.

7. CONCLUSION

In this paper we introduced WIKIQUERY, a new approach to querying Wikipedia documents that returns results in the form of linked documents. Inspired by approaches that support keyword-based queries over structured data, WIKIQUERY models documents as a graph. But in contrast to those, it utilizes a weighting scheme that takes into account specific features of Wikipedia documents to identify important entities and relationships. In addition to keyword queries, WIKIQUERY supports the more expressive c-queries, and we proposed algorithms which use the weighting scheme to search for and rank query results for both kinds of queries. We present the results of a preliminary experimental evaluation which shows that WIKIQUERY is effective and obtains answers that are of higher quality than those derived by a state-of-the-art system for keyword search over relational databases.

Acknowledgments. Our research has been funded by National Science Foundation grants IIS-0905385, IIS-0844546, IIS-0746500, CNS-0751152, IIS-0713637, and the Department of Energy.

[3] www.freebase.com,www.powerset.com, and dbpedia.neofonie.de

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, page 5, 2002.

[2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In *ISWC/ASWC*, pages 722–735, 2007.

[3] S. Auer and J. Lehmann. What have Innsbruck and Leipzig in common? Extracting semantics from wiki content. In *ESWC*, pages 503–517, 2007.

[4] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.

[5] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[6] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, S. Sudarshan, and I. Bombay. Keyword searching and browsing in databases using banks. In *ICDE*, page 431, 2002.

[7] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[8] J. Graupmann, R. Schenkel, and G. Weikum. The spheresearch engine for unified ranked retrieval of heterogeneous XML and Web documents. In *VLDB*, pages 529–540, 2005.

[9] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[10] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[11] V. Hristidis and Y. Papakonstantinou. Discover: keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[12] K. Järvelin and J. Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *SIGIR*, pages 41–48, 2000.

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

[14] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. Star: Steiner-tree approximation in relationship graphs. In *ICDE*, pages 868–879, 2009.

[15] G. Kasneci, F. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.

[16] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[17] C. Yu and H. V. Jagadish. Schema summarization. In *VLDB*, pages 319–330, 2006.