QUICK: Queries Using Inferred Concepts from Keywords

Technical Report CS-2009-18

Jeffrey Pound, Ihab F. Ilyas, and Grant Weddell
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada

Abstract

We present QUICK, an entity-based text search engine that blends keyword search with structured query processing over rich knowledge bases (KB) with massive schemas. We introduce a new formalism for structured queries based on keywords that combines the flexibility of keyword search and the expressiveness of structures queries.

We propose a solution to the resulting disambiguation problem caused by introducing keywords as primitives in a structured query language. We show how expressions in our proposed language can be rewritten using the vocabulary of a given KB. The rewritten query describes an arbitrary topic of interest for which corresponding documents are retrieved.

We also introduce a method for indexing that allows efficient search over large KB schemas in order to find sets of relevant entities. We then cross-reference the entities with their occurrences in a corpus and rank the resulting document set to produce the top-k relevant documents. An extensive experimental study demonstrates the viability of our approach.

1 Introduction

The World Wide Web hoards massive amounts of unstructured text data. Recent efforts to extract structured data sets from unstructured and semi-structured Web documents have resulted in the creation of massive knowledge bases: structured data sets encoding rich schematic information including millions of entities. As an example, both ExDB [2] and YAGO [21], a large knowledge base derived from Wikipedia and WordNet, have schema items numbering in the millions, and fact collections numbering in the hundreds and tens of millions respectfully. At this scale, writing structured queries can be a daunting task as the sheer magnitude of the information available to express the query is overwhelming. We call this the *information overload* problem.

$$
\begin{array}{lllll}
q(x) & \text{:-} & \text{“}german\text{”}(x), & \text{“}has\ won\text{”}(x,y), & \text{“}nobel\ award\text{”}(y). \\
q(x) & \text{:-} & \left\{\begin{array}{c}\texttt{German\_Language}\\ \texttt{GERMAN\_PEOPLE}\\ \texttt{German\_Alphabet}\end{array}\right\}(x), & \left\{\begin{array}{c}hasWeight\\ hasWonPrize\end{array}\right\}(x,y), & \left\{\begin{array}{c}\texttt{NOBEL\_PRIZE}\\ \texttt{AWARD}\\ \texttt{Turing\_Award}\end{array}\right\}(y).
\end{array}
$$

Figure 1: An ambiguous conjunctive query and the possible interpretations. The first and third predicates are an example of matching on keyword occurrence, while the second predicate is an example of matching on edit distance.

To deal with this problem, and also open a new form of exploratory search, recent work has brought keyword query processing to structured data models, such as keyword search over relational databases. While keyword search over structured data models alleviates the information overload problem caused by massive schemas, it comes at a loss of expressivity. Users can no longer express desired structure in the query, and can no longer take advantage of schema information.

As an example, consider a user who wants to find *all people of German nationality who have won a Nobel award*. The user may pose the following conjunctive query to an information system.

$$q(x) \text{ :- } \texttt{GERMAN\_PEOPLE}(x),\ hasWonPrize(x,y),\ \texttt{NOBEL\_PRIZE}(y). \tag{1}$$

If the user is not familiar with the schema of the underlying information system (i.e., they do not know the labels of the relations in order to formulate a well formed query), then they may alternatively issue the following keyword query.

$$\text{“}german\ has\ won\ nobel\ award\text{”} \tag{2}$$

This query searches for all data items with a syntactic occurrence of the given keywords. There are two important differences between the structured query and the keyword variant. First, the structured query will be processed by making use of explicit semantics encoded as an internal schema. In particular, the query will find data items that *are* German, rather than those which contain the keyword *“german”*. Second, the keyword query will look for data items with any syntactic occurrence of *“nobel award”*, while the structured query uses this an explicit selection predicate over the qualifying entities, exploiting the structure of the query to find qualifying results.

This example illustrates how query languages can vary in their expressiveness and ease-of-use. On one end of the spectrum, structured query languages, such as SQL, provide a mechanism to express complex information needs over a schema. A user of such a language must have intimate knowledge of the underlying schema in order to formulate well formed queries for arbitrary information needs. On the other end of the spectrum are free form query languages such as keyword queries. These languages allow users to express information needs with little to no knowledge of any schematic constructs in the underlying information system, giving ease-of-use as well as useful exploratory functionality to the user.

Consider the design of a query language that falls somewhere in the middle of this spectrum. We would ideally like to retain the expressive structure of structured queries, while incorporating the flexibility of keyword queries. One way to achieve this is to embed ambiguity into the structured query language by allowing keywords or regular expressions to take the place of entities or relations. In this setting the query from the previous example may be written using the structure in (1), but with keywords as in (2).

$$q(x) \text{ :- } \textit{``german''}(x),\ \textit{``has won''}(x,y),\ \textit{``nobel award''}(y).$$

In this model each keyword phrase will be replaced, by the query processor, with a set of candidate schema items based on some metric of syntactic matching, such as keyword occurrence or edit distance as depicted in Figure 1. The advantage of this approach is in the flexibility of how queries can be expressed. Users need not have intimate knowledge of the underlying schema to formulate queries as is necessary with traditional structured query languages. At the same time, the query retains explicit structure that gives greatly increased expressiveness over flat keyword queries.

The problem with this approach is that the number of possible (disambiguated) queries is exponential in the size of the query. This is problematic as many of the possible matchings may be meaningless with respect to the underlying schema, or may not represent the users actual intention. Processing every possible match is not only inefficient, as many of the possible query interpretations may have empty result sets, it can overload the user with many unwanted results from unintended interpretations. For example, the query for all `GERMAN_PEOPLE` who have won an `AWARD` (the super class of all awards) will produce many results that obstruct the user from finding those of interest (those who have won a `NOBEL_PRIZE`). Note that syntactic matching alone could not possibly be sufficient as a disambiguation solution in all cases, as the keyword *"german"* is a natural choice to express both a nationality and a language. This phenomena, known as *polysemy*, is just one of the challenges of disambiguation. Similar problems arise with synonymy.

In this paper, we investigate a solution to the problem of querying over rich massive schemas by introducing a structured query language that builds upon keywords as its most basic operator, while taking disambiguation steps before query evaluation in order to avoid the exponential blow-up caused by keyword ambiguity. Our query language allows a natural keyword-based description of entity-relationship queries over large knowledge bases without intimate background knowledge of the underlying schema. We also show how inference over a rich schema defined by a reference knowledge base, such as those extracted by YAGO or ExDB, can be efficiently incorporated into the entity search system by introducing a method for indexing that allows query time inference. Our index incorporates the traditional notions of inverted indexing over documents, allowing us to extend our entity search functionality to document retrieval, and incorporate full text search.

1.1 Contributions

In this work, we make the following contributions.

- We propose a keyword-based structured query language that allows users to create expressive descriptions of their information need without having detailed knowledge of the underlying schema.
- We analyze the consequences of introducing ambiguity into a structured query language (by means of keywords as base level constructs), and propose a model for disambiguation. We also show how query terms can be used as keywords to integrate full text search in the cases where full disambiguation can not be achieved.
- We introduce a compact scalable index for inference enabled entity search that is more space efficient than the current state-of-the-art approaches, with competitive performance in query processing.

We also demonstrate the viability of our proposed approach with an experimental evaluation of a prototype implementation of the complete system.

1.2 Outline

The remainder of the paper is organized as follows. Section 2 reviews relevant background information. Section 3 gives an overview of our problem and proposed architecture. Section 4 presents our structure keyword query language, with the disambiguation problem explored in Section 5. Section 6 then presents our index structure and details query processing of disambiguated queries. In Section 7 we show experimental results of a prototype implementation. Section 8 surveys and compares related work with our approach and Section 9 concludes.

2 Background

Entity Search

Entity search is a variant of traditional information retrieval (IR) in which *entities* (e.g., Albert Einstein, University of Waterloo, etc..) extracted from text documents form the most basic data-level construct, as opposed to *documents* in traditional IR. This view of the search problem opens up a variety of new query processing opportunities, in particular, the ability to process more structured types of queries over the entities and their relations to one another becomes feasible (e.g., [2, 4, 10, 11]). In contrast to traditional IR, query processing in this model may aggregate data from multiple documents to determine which entities are part of a query result. In these models, structured or unstructured query languages can be used to describe entities of interest, and data can reside in a structured model or in text documents. The common facet is that entities form the primitives for search.

As an example, consider an entity-based search system over text documents. Given a query for documents containing the entity `Max_Planck`, the system would retrieve all documents mentioning `Max_Planck`, and *not* the documents which mention the `Max_Planck_Institute` or the `Max_Planck_Medal` (an award in physics). Having entities as primitives reduces ambiguity in the semantics of the search, which avoids problems common to traditional keyword search systems. Also, information can be aggregated from different sources to answer a query, such as a search for all documents containing entities *bornIn*(`Canada`). Documents about qualifying entities will be returned independent of the location of the information that encodes where they were born.

Semantic Search

Ontology-based semantic search is a similar problem to entity search in which entities and relations form the base level data constructs. In these systems, rich semantics defined by ontologies are taken into consideration during query processing (or as a preprocessing phase). However, semantic search systems generally operate over structured semantic data, such as RDF or OWL [7, 17], fact triples [13], or XML [6] (with explicit references to the ontology concepts). A variant of the semantic search problem is to support searching over the semantics of entities found in text documents. Again these models may make use of structured or unstructured query languages. The common thread in semantic search systems is the support for inference over a reference ontology.

As an example, consider a semantic search system over text documents. A search for documents containing entities of type `GUITARIST` would return documents about `B._B._King`, `Jimi_Hendrix`, and other entities known to be guitarists. These documents are query results independent of whether or not they contain a syntactic occurrence of the word "guitarist." Thus, the search is over the semantics of the entities, rather than the syntax.

Knowledge Representation

The field of knowledge representation is concerned with formalisms for encoding knowledge with clear semantics to support inference over the encoded information. A collection of schematic information, which forms general rules about a domain of interest, is called an ontology. Concrete instance information, or entities, describe explicit information about particular entities in the domain, such as `Albert_Einstein`. A *knowledge base* is a general term for a domain ontology and instance data.

As an example, an ontology might define relationships among abstract primitive concepts, such as the relationship `PHYSICIST` *is-a* `SCIENTIST`. Ontologies may also include arbitrary relations, such as *hasWonPrize*, to encode information like `NOBEL_LAUREATE` *is-a* `PERSON` that has a *hasWonPrize* relation to some `NOBEL_PRIZE`. The encoding of the knowledge base information and the associated semantics rest on an underlying knowledge representation formalism.

3 Problem Definition and Solution Overview

3.1 Preliminaries

We assume access to a domain related knowledge base and document corpus. These data sources are preprocessed to: (1) extract entities from the text (used for entity search over documents), (2) compute statistics over the knowledge base (used in disambiguation), and (3) index the knowledge base's content and structure (used in query processing), as well as the document text (used to integrate full text search).

The schema language for the underlying knowledge bases assumed in our work includes entities, primitive concepts, conjunctions, existential relations, and acyclic concept hierarchies. This is roughly equivalent to the $\mathcal{EL}$ dialect of description logics, and is sufficient to capture many real world knowledge bases and ontologies such as YAGO [21] and SNOMED CT[1], an ontology of medical terms.

The basic knowledge base terminology used in our discussion, as well as our notational conventions, are as follows.

- *Entities* are written in a typed font with leading capitals. Entities denote constant terms such as people, places, and organizations (e.g., `Albert_Einstein`, `Canada`).
- *Primitive Concepts* are written in a typed font in all capitals. Primitive concepts are types or classes to which entities may belong (e.g., `SCIENTIST`, `COUNTRY`).
- *Relations* are written in italic camel caps. Relations are binary relationships that can exist between entities and/or primitive concepts (e.g., *hasWonPrize* or *bornIn*). There is also a transitive *is-a* relation used to encode hierarchies among concepts.
- A *general concept* is any number of entities or primitive concepts joined by logical connectives such as conjunctions and relations (note that relations are implicitly existentially qualified). We use "," to denote a conjunction. (e.g. `SCIENTIST`, *hasWonPrize*(`Nobel_Prize_in_Physics`)).

We also adopt the convention of writing *keywords* in quoted italic lower case (e.g., *"scientist"*).

3.2 Problem Definition

The problem we address is to provide expressive yet flexible access to document collections. We aim to make use of entity-based retrieval and incorporate inference over a rich schema defined by a reference KB. We address this problem in the following way.

[1]SNOMED CT actually uses an extended version of $\mathcal{EL}$ called $\mathcal{EL}$++ which includes hierarchies among relations and transitive relations. Our proposed system would need to be extended to handle inference over these constructs.

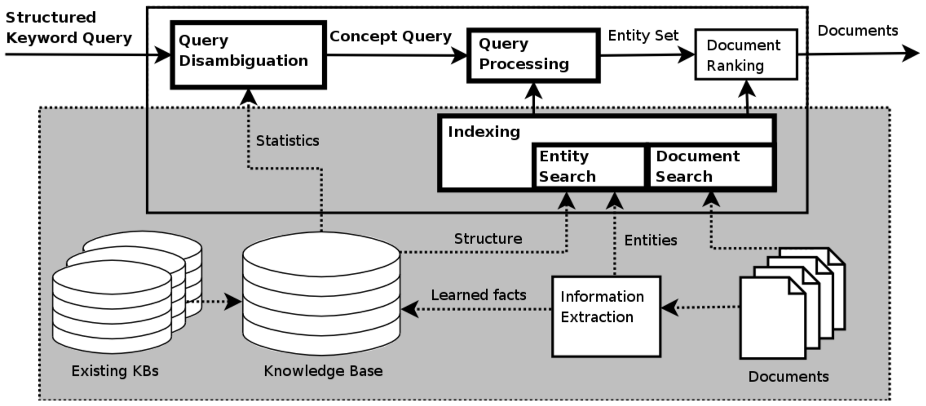

Figure 2: System architecture.

- We develop a keyword-based structured query language that blends the expressiveness of structured queries with the flexibility of keyword queries (Section 4).
- Next, we address the problem of ambiguity in our query language caused by introducing keywords as primitives. We propose an efficient approach for disambiguating queries in our language to produce a ranked set of possible query interpretations using the vocabulary of an underlying schema (Section 5).
- We then focus our attention to the problem of efficiently evaluating the disambiguated queries over large rich schemas. To achieve this, we encode KB structure and content in a compact index structure that allows us to efficiently find all entities that satisfy a disambiguated query in our language. Our index builds on traditional notions of inverted indexing, allowing us to easily incorporate entity-based document retrieval and full text search (Section 6).
- Lastly, we address the issue of ranking relevant documents by incorporating established mechanisms from the IR community for document ranking based on relevance of keywords to documents.

3.3 System Overview

Our system architecture is illustrated in Figure 2. The solid lined box on top shows the run-time components, with data flow represented as solid lined arrows. The grey dotted lined box shows the preprocessing phase, with preprocessor data flow represented as dotted lined arrows. The bold typed components in the diagram show the parts of the system which we focus on in this paper, namely, the query language, query disambiguation, indexing, and query processing. We use generic solutions to support the remaining components of the system (document ranking and information extraction).

The preprocessor consists of two components, the information extraction system and the indexing module. These two components make use of two domain related data sources, a knowledge base and a document corpus. The preprocessing phase proceeds as follows. First the information extraction tool is run over the document corpus to mine structured information from the text. This information includes extracted named entities as well as relationships among the entities. Facts that were previously unknown to the knowledge base can be contributed (note that we omit this learning phase in our implementation and discussion as the focus of this paper is not on the extraction component). The named entities are then sent to the indexing mechanism to build an inverted index over entities and documents. The documents' text is also indexed directly to support full text search.

In the second preprocessing phase, the structure of the knowledge base is indexed to support efficient entity search within the knowledge base. It is important to note that the entity search module must be capable of handling the expressivity of the constructs used in the underlying knowledge base. The final preprocessing step is to compute statistics over the knowledge base which will be used during query disambiguation.

At run time, an arbitrary keyword-based structured query is taken as input to the system. Because the query is based on keywords, there is an inherent ambiguity as to the semantics of the query (i.e., the user's intention). We compute a ranked set of possible query disambiguations based on the vocabulary of the knowledge base. Each disambiguated query is thus a general concept over the underlying knowledge base. One or more of these disambiguated queries can then be evaluated to find relevant entities and their corresponding documents. The documents are then returned to the user ranked by some relevance metric.

4 Keyword-based Queries

We now introduce our keyword-based structured query language.

Definition 1 (Document Query) *A* document query *DQ is given by the following.*

$$\begin{array}{rcl} DQ & ::= & Q \\ & \mid & (Q_1), (Q_2), ..., (Q_n) \end{array}$$

where Q_i is a structured keyword query.

A document query specifies documents of interest based on entities that qualify for the structured keyword queries using "AND" semantics. A structured keyword query describes a set of entities as defined below.

Definition 2 (Structured Keyword Query) *Let k be a keyword phrase (one or more keywords), then a* structured keyword query *Q is defined by the follow-*

ing grammar.

$$\begin{array}{rcl} Q & ::= & k \\ & | & k(Q) \\ & | & Q_1, Q_2, ..., Q_n \end{array}$$

The first construct allows a primitive concept in a query to be described by a set of one or more keywords (e.g., "*nobel prize*"). The second construct allows one to describe an entity in terms of the relationships it has to other entities (e.g., "*born in(Germany)*"). The third constructor allows a set of queries to describe a single class of entities in conjunction (e.g., "*harmonica player, songwriter*"). The comma operator is used to explicitly label the conjunctions, reducing the ambiguity among multi-keyword phrases that occur in conjunction.

Note that the relation query constructor allows an arbitrary query to describe the entity to which the relation extends, for example, "*born in(Country, has official language(Spanish))*" could be used to query all entities born in spanish speaking countries.

More formally, the resulting entity set $\{e\}$ for a structured keyword query Q denoting general concept C is given by the following: $\{e \mid e \in C\}$. We use the notation $e \in Q$ to say that entity e is in the result set of query Q. Section 5 discusses how to find the concept C which is denoted by Q.

To illustrate how structured keyword queries are used in document queries, consider the following example document query.

(person, born in(Germany)), (participated(World War II))

The query indicates that qualifying documents should mention both a person born in Germany and some entity that participated in World War II. More formally, the resulting document set $\{D\}$ for a document query $DQ = (Q_1), (Q_2), ..., (Q_n)$ is given by the following:

$$\{D \mid \forall_{Q \in DQ} \exists_{e \in Q} \text{ such that } e \in D\}$$

where $e \in D$ denotes entity e occurring in the text of document D. Intuitively, the document retrieval uses "OR" semantics per entity set denoted by a subquery, and "AND" semantics across subqueries in the DQ.

5 Disambiguating Query Intent

Because our query language is built on keywords as primitives, there is an inherent ambiguity in the semantics of the query. The user's intentions are unknown since each keyword in the query could map to one of many items from the knowledge base, and there is an exponential number of ways in which we could select an item for each keyword to create a general concept.

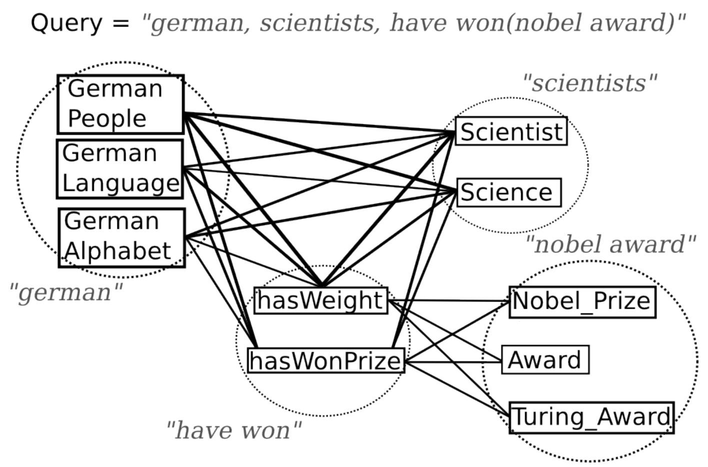

Figure 3: An example query disambiguation graph.

5.1 The Disambiguation Model

Consider a structured keyword query Q. Each keyword in Q could have many possible schema item matches based on some measure of syntactic similarity. Let $M(k)$ denote the set of possible schema item (entity/concept/relation) mappings of keyword k.

The goal is to choose one schema item for each keyword in the query, such that the resulting disambiguation is a meaningful representation of the query with respect to the knowledge base, and is also representative of the user's intention. There is a tradeoff that must be considered when choosing concept interpretations for each keyword: we want mappings that represent the users intentions as best as possible in terms of syntactic similarity, but which also have a meaningful interpretation in terms of the underlying knowledge base by means of semantic relatedness.

We encode our disambiguation problem as a graph which represents the space of all possible interpretations that bind one entity, concept, or relation to each keyword. In the discussion that follows, we refer to a concept or the subgraph that denotes a concept interchangeably.

Definition 3 (Disambiguation Graph) *Let Q be a structured keyword query. Then a* disambiguation graph $G = \langle V, E\rangle$ *with vertex set V, edge set E, and P a set of disjoint partitions of the nodes in V, are defined by the following.*

$$\begin{aligned} V &= \bigcup_{k\in Q} M(k) \\ P &= \bigcup_{k\in Q} \{M(k)\} \\ E &= edges(Q) \end{aligned}$$

where the edge generating function edges(Q) is defined as follows.

$$\begin{aligned}
edges(k) &= \{\} \\
edges(k(Q)) &= \left(\bigcup_{\substack{n_1 \in M(k), \\ n_2 \in root(Q)}} \langle n_1, n_2 \rangle \right) \cup\ edges(Q) \\
edges(Q_1, Q_2) &= \left(\bigcup_{\substack{n_1 \in root(Q_1), \\ n_2 \in root(Q_2)}} \langle n_1, n_2 \rangle \right) \cup\ edges(Q_1) \\
&\qquad \cup\ edges(Q_2)
\end{aligned}$$

where root(Q) denotes the vertices in the root level query of Q as follows.

$$\begin{aligned}
root(k) &= M(k) \\
root(k(Q)) &= M(k) \\
root(Q_1, Q_2) &= root(Q_1)\ \cup\ root(Q_2)
\end{aligned}$$

(Note that we have abstracted n-ary conjunctions as binary conjunctions for ease of presentation, arbitrary conjunctions can be formed by considering all pairwise conjunctions in the n-ary expression.)

Figure 3 depicts an example of a disambiguation graph for the query *"german, scientists, have won(nobel award)"*. The boxes denote vertices (corresponding to concepts or entities in the underlying knowledge base), the dotted circles denote partitions (the sets M(k_i)), the italic font labels denote the keywords for which the partition was generated (k_i), and the interconnecting lines denote edges which represent semantic similarity. Observe that each level of any subquery is fully connected t-partite, while edges do not span query nesting boundaries. This is because nested concepts do not occur in conjunction with concepts at other nested levels. They are related only through the vertices that denote the explicit knowledge base relations.

For a disambiguation graph G generated from a structured keyword query Q, any induced subgraph of G that spans all partitions in P corresponds to a concept interpretation of Q. For example, the induced subgraph spanning the nodes

$$\{\texttt{GERMAN\_PEOPLE}, \texttt{SCIENTIST}, \mathit{hasWonPrize}, \texttt{NOBEL\_PRIZE}\}$$

corresponds to the concept

$$\texttt{GERMAN\_PEOPLE},\ \texttt{SCIENTIST},\ \mathit{hasWonPrize}(\texttt{NOBEL\_PRIZE}).$$

Figure 4 illustrates a disambiguation graph for a more complex query with multiple nesting branches and multiple levels of nesting. In this example, a meaningful interpretation could be the following concept.

$$\texttt{GERMAN\_PEOPLE},\ \texttt{SCIENTIST},\ \mathit{hasWonPrize}(\texttt{NOBEL\_PRIZE}),$$

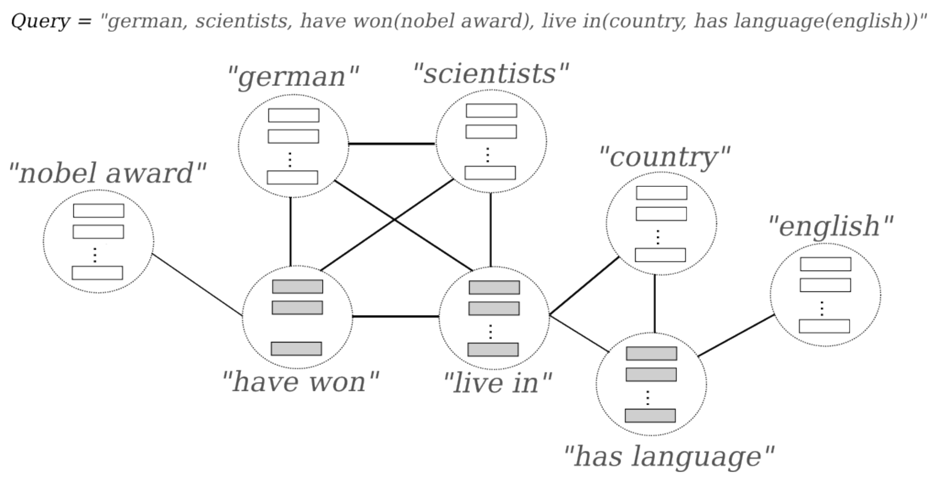
Query = "german, scientists, have won(nobel award), live in(country, has language(english))"
"german"
"scientists"
"nobel award"
"country"
"english"
"have won"
"live in"
"has language"

Figure 4: An example of a query disambiguation graph. Each edge represents complete bipartite connectivity between all vertices in the partitions.

$$\textit{livesIn}(\texttt{COUNTRY},\ \textit{hasOfficialLanguage}(\texttt{English\_Language}))$$

Note the difficulty in disambiguating this query. In one part of the query the keyword *"german"* is used to denote a nationality, while in another part of the query the keyword *"english"* is used to denote a language. Both of these keywords are legitimate choices to denote both nationalities and languages, and consequently purely syntactic matching alone could not possibly handle both situations correctly. This phenomena is known to linguists as *polysemy*. Similar issues arise with synonymy (for example, the phrases *"guitarist"* and *"guitar player"* both denote the same concept).

It is evident that the space of possible query interpretations is exponential in the size of the query. Finding the "best" or top-k interpretations will depend on how we compute a score for a candidate subgraph corresponding to a query interpretation.

5.2 The Scoring Model

Now that we have established a model for generating a space of candidate query interpretations, we need to define the notion of a score in order to compute the top-k interpretations. We start by reviewing notions of semantic and syntactic similarity which will form the basis of our scoring function.

Semantic Similarity

The first factor of our scoring model is *semantic similarity*, which we denote generically using $semanticSim(A, B)$. The problem of computing the similarity between two concepts has a long history in the computational linguistics and artificial intelligence fields. Early works focused on similarities over taxonomies, often including the graph distance of concepts [12, 14, 23], and probabilistic notions of information content of classes in the taxonomy [12, 15, 19]. More recent works have looked at generalizing semantic similarity metrics from taxonomies to general ontologies (i.e., from trees to graphs) [5, 16, 20]. As an example, Lin defines semantic similarity between two concepts as the following [15].

$$Sim_{Lin}(A, B) = \frac{2 \cdot log(P(LCS(A, B))}{log(P(A)) + log(P(B)))}$$

where $LCS(A, B)$ denotes the least common subsumer of A and B, i.e., the first common ancestor in an "is-a" graph, and $P(A)$ denotes the probability of a randomly drawn entity belonging to concept A.

Lastly, there are the traditional notions of set similarity which can be applied, since concepts in a taxonomy or ontology denote sets of entities. Two commonly used metrics are the Dice coefficient and the Jaccard index, the latter illustrated below.

$$Sim_{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

One common trait to all of these similarity measures is that they are binary; they express similarity over pairs of concepts. We will see that efficiently accommodating n-ary similarities using preprocessed statistics poses challenges, though our model allows a natural approximation based on aggregates of binary metrics.

We now introduce the notion of knowledge base support, which characterizes the semantic similarity of all primitive concepts and relations occurring in a complex general concept. In the definition of knowledge base support, we appeal to n-ary similarity, with our approximation technique to be introduced in Section 5.3.

Definition 4 (Knowledge Base Support) *Let C be a general concept expressed using the primitive concepts, relations, and entities in the knowledge base KB. Then the* support *of C by KB is given by the following.*

$$\begin{aligned}
&support(C, KB) = semanticSim(C_1^0, \cdots, C_{n_0}^0, R_1^0, , ..., R_{m_0}^0) \\
&\quad + semanticSim(R_1^0, ..., R_{m_0}^0, C_1^1, \cdots, C_{n_1}^1, R_1^1, ..., R_{m_1}^1) \\
&\quad + \cdots \\
&\quad + semanticSim(R_1^{p-1}, ..., R_{m_{p-1}}^{p-1}, C_1^1, \cdots, C_{n_p}^p, R_1^p, ..., R_{m_p}^1)
\end{aligned}$$

where each C_j^i (R_j^i) is the j^{th} primitive concept (relation) in a conjunction occurring in C at nesting branch i.

Intuitively, this is the similarity according to the structure of the disambiguation graph (which corresponds to the structure of the query). We compute the similarity of primitive concepts occurring in conjunction, along with the incoming relations from the previous nesting level, and the outgoing relations to the next nesting level.

For our example in Figure 4, with the previously suggested disambiguation, the support would correspond to the following.

$$\begin{aligned}
&\mathit{semanticSim}(\texttt{GERMAN\_PEOPLE}, \texttt{SCIENTIST}, \mathit{hasWonPrize}, \mathit{livesIn})\\
&+ \mathit{semanticSim}(\mathit{hasWonPrize}, \texttt{NOBEL\_PRIZE})\\
&+ \mathit{semanticSim}(\mathit{livesIn}, \texttt{COUNTRY}, \mathit{hasOfficialLanguage})\\
&+ \mathit{semanticSim}(\mathit{hasOfficialLanguage}, \texttt{English\_Language})
\end{aligned}$$

Syntactic Similarity

The second component to our scoring model is the syntactic matching of the concept label (or one of its synonyms) to the keyword. Indeed, the user entering queries has an idea of the entities they are trying to describe. It is simply the terminology of the user and the knowledge base that may differ, and the massive scale of the knowledge base impedes learning on the user's part. The closer we can match a query syntactically while maximizing semantic similarity, the better our query interpretation will be with respect to both the user's intentions and the knowledge encoded in our knowledge base. We use the function label $syntaxSim(a, b)$ to denote a measure of syntactic similarity. This could be simple keyword occurrence in the knowledge base item's label, or a more relaxed measure such as edit distance.

Score Aggregation

Our final scoring model is thus some combination of knowledge base support, which quantifies semantic support for the candidate disambiguation, and syntactic similarity, which reflects how closely the candidate concept interpretation matches the user's initial description of their query intention. We can tune how important each factor is by how we combine and weight the two similarity metrics. In general, we allow any aggregation function to be plugged into our system. However, as we will see in later sections, we can improve efficiency if the aggregation function is monotonic by making use of threshold algorithms for top-k search. We denote such an aggregation function using $\oplus$.

Definition 5 (Concept Score) *Let Q be a structured keyword query, C a concept, $\oplus$ a binary aggregation function, and KB a knowledge base. Then the* score *of concept C, with respect to Q, KB, and $\oplus$ is given by the following.*

$$score(C, Q, KB) = support(C, KB) \oplus syntaxSim(C, Q)$$

Given a disambiguation graph G with partition set P and a parameter integer k, the goal of disambiguation is to find the top-k maximum scoring subgraphs of G that span all partitions in P. Intuitively, we want to find the best k interpretations of the query in terms of some balance between knowledge base support and syntactic similarity.

The difficulty in solving the disambiguation problem lies in the nature of the scoring function. Looking back to our disambiguation graph model, the score of a general concept representing a candidate query interpretation depends on *all* components of the concept (all conjunctions of primitive concepts and relations). From the disambiguation graph point of view, this means that the score (or weight) of each edge *changes* depending on which other edges are part of the subgraph forming the candidate concept interpretation. The amount of statistics that would have to be (pre)computed in order to support queries with n terms would be very large, one would need to have access to the semantic similarity of all n-way compositions of entities, primitive concepts, and relations occurring in the knowledge base.

5.3 Approximating the Scoring Model

In most cases, having access to n-way semantic similarity would require precomputing an infeasible number of statistics, or incurring the expensive cost of computing similarity at query time. Because there are an exponential number of candidate query interpretations, computing similarity at query time could add unacceptable costs to query performance. For example, the previously described Sim_{Lin} metric requires a least common subsumer computation over a large graph, while the Jaccard and Dice metrics require intersections of (possibly large) sets. Thus, we move to an approximation model for any binary metric of semantic similarity.

We approximate the score of a candidate query interpretation by aggregating the pairwise support of all components of the candidate concept. (Note that pairwise support reduces Definition 4 to a single binary similarity computation.) In terms of the disambiguation graph, this means that each edge weight in the graph can represent the support of having the two primitive concepts or relations denoted by the edge's vertices in conjunction.

We extend a disambiguation graph G to include a weight function w that assigns weights to the vertices (v) and edges ($\langle v_1, v_2 \rangle$) of G as follows. The weights are computed with respect to some knowledge base KB and the keywords k from a structured keyword query Q.

$$\begin{aligned} w(v) &= syntaxSim(label(v), k), \ \ {\scriptstyle where\ v \in M(k)} \\ w(\langle v_1, v_2 \rangle) &= support((item(v_1),\ item(v_2)),\ KB) \end{aligned}$$

where $item(v)$ denotes the knowledge base schema item (primitive concept, relation, or entity) represented by vertex v and $label(v)$ denotes the string representation of the schema item represented by v.[2]

The approximate score of an induced subgraph denoting a candidate concept interpretation is given by the following.

Definition 6 (Approximate Score) *Let Q be a structured keyword query, KB a knowledge base, and G a subgraph of the disambiguation graph of Q representing a concept C. Then the* approximate score *of C represented by G*

[2]In practice, it would be beneficial to consider the syntactic similarity of the keyword to the closest synonym of the concept label.

with respect to Q and KB is given by the following.

$$s\hat{core}(G, Q, KB) \; = \; \left(\sum_{\langle v_1, v_2 \rangle \in E} w(\langle v_1, v_2 \rangle) \; \oplus \; \sum_{v \in V} w(v) \right)$$

5.4 Solving the Disambiguation Problem

The goal of the disambiguation problem is to find the top-k maximally scoring subgraphs (corresponding to concept interpretations of the original query) which are single connected components. Note that simply taking the maximum scoring subgraph for each subquery will not necessarily result in a connected graph, and thus does not map to a concept interpretation of the query. If our scoring function is monotonic, we can implement a threshold algorithm (TA) to efficiently find the top scoring subgraphs, so long as we have access to vertices and edges in sorted order. A global ordering over all edges can be precomputed and indexed, and vertices denoting concepts can be sorted on the fly since these sets fit in memory and can be kept relatively small in practice. Alternatively, we can implement a TA if we have access to only the vertices in sorted order and allow random access to edges.

The disambiguation problem can be viewed as a relational rank-join, a TA which implements a special case of A* search over join graphs. In this view of the problem, our disambiguation graph is a join graph, with each keyword in the query providing an input to a rank-join operation based on the partition corresponding to the keyword. Each partition is ordered by syntactic similarity.

Consider each possible concept or relation match (partition $M(k)$), ordered by syntactic similarity, as an input. We "join" two concepts if they have knowledge base support, and rank the join based on the value of the approximate score. Simultaneously, we join across nested subqueries based on equivalence of the vertex representing relation keyword. This ensures the resulting joins form connected graphs in terms of the underlying disambiguation graph.

As an example, consider the disambiguation graph from Figure 3, we illustrate a rank-join approach to disambiguation of the query in Figure 5. The illustration shows each partition as an input to the rank join. Conjunctive components are joined based on the existence of edges in the disambiguation graph, while the entire nested queries are joined based on equality of the binding of the relation keyword to a schema relation.

For brevity, we omit a review of the threshold algorithm for rank-join. The challenge here is in the modeling of the problem as a rank-join as we have presented. With this model in mind, the actual execution does not differ from that of a traditional rank-join. We refer the reader to [9] for an overview of the algorithm.

While solving the disambiguation problem involves possibly exploring an exponential search space in the worst case, our experiments show that we can still find meaningful solutions quickly in practice (see Section 7).

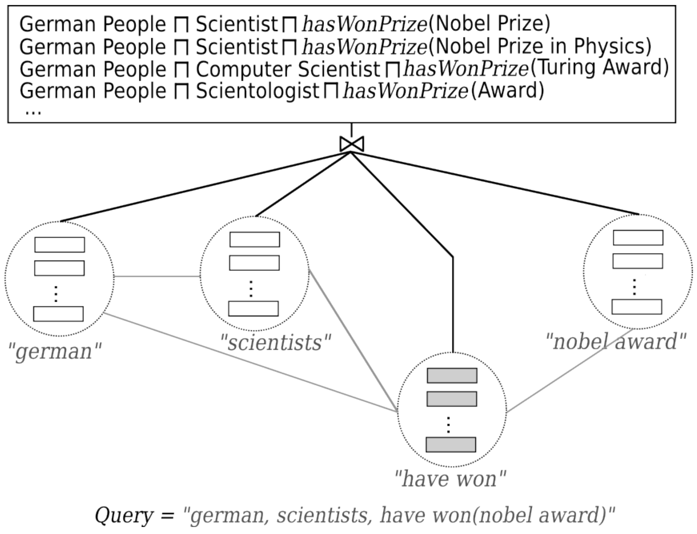
German People ⊓ Scientist ⊓ *hasWonPrize*(Nobel Prize)
German People ⊓ Scientist ⊓ *hasWonPrize*(Nobel Prize in Physics)
German People ⊓ Computer Scientist ⊓ *hasWonPrize*(Turing Award)
German People ⊓ Scientologist ⊓ *hasWonPrize*(Award)
...
"german"
"scientists"
"nobel award"
"have won"
Query = "german, scientists, have won(nobel award)"

Figure 5: Encoding disambiguation as a rank-join with random access to edges. The join conditions are on the existence of edges in the disambiguation graph for each subquery, and on the equality of relation bindings to join across all subqueries.

5.5 Integrating Full Text Search

As a final consideration, it may be the case that no subgraph can be found that spans all partitions in a disambiguation graph for some query. This happens when we can not find a meaningful interpretation for one or more keywords. In this case we proceed as follows:

- We relax the problem definition to not require spanning all partitions when finding a subgraph of the disambiguation graph. This allows us to build partial interpretations.
- We augment our scoring model to add a constant penalty factor for each unmapped keyword. This allows us to tune preference for complete disambiguations.
- Lastly, we include all keywords that are unmapped as part of the document retrieval process. In this setting, we will retrieve documents that contain one or more entities from the concept interpretation of the query (as before), and favor those which contain the additional keywords in the ranking.

For example, a partial interpretation for the query *"german, scientists, have won (nobel award)"* could be the concept `SCIENTIST`, *hasWonPrize*(`NOBEL_PRIZE`) which would then be parameterized with the keyword *"german"* during document retrieval. Note that in the worst case (in terms of disambiguation), our system degrades to a regular keyword search.

6 Indexing and Query Evaluation

One way to support inference over an ontology or type hierarchy in a semantic search system is by expanding the knowledge into the underlying data [1, 2, 3]. In the document retrieval model, this amounts to computing the deductive closure over each entity discovered in the text, and expanding the full closure into the document for indexing. For example, if one finds the entity `Albert_Einstein` in a document, they would add identifiers for `PHYSICIST`, `SCIENTIST`, `PERSON`, `MAMAL`, `LIVING_ORGANISM`, etc..., to the document. This way a query for `PHYSICIST` will retrieve documents that contain the entity `Albert_Einstein`. The benefit of this approach is in its simplicity, one need not make any special modifications to the indexing procedure or the search algorithm in order to support inference over the hierarchy. A drawback of this method is that it greatly reduces the capabilities for inference during query processing. For example, even if we expand the full closure of each entities relations into the document, we still couldn't answer a complex query such as *"born in(located in(has language(German)))"* to find documents about `Albert_Einstein` (an entity who was born in a city that is located in a country with German as the spoken language).

The other major drawback to this approach is the redundancy in the resulting inverted index. Sets of entities will be repeated over and over again for each concept of related type. For example, the index entry for `SCIENTIST` will repeat the entire index entry for `PHYSICIST`, since every document mentioning a physicist will also be mentioning a scientist by transitivity. This can potentially cause an infeasible amount of redundancy, even for relatively simple ontologies. Current approaches to this problem restrict the amount of the closure that is expanded into the document, and require that any query can then be mapped to one of a small number of pre-defined top level classes.

6.1 Hierarchical Inverted Index

We propose a family of combined document and ontology index structures referred to as a *Hierarchical Inverted Index*, or HII, that build on traditional notions of graph representations and inverted indexing. The motivation behind our index structure is to avoid redundancy in the index to be space efficient, and support inference over the ontology at run time through query expansion. We detail the design of the index below, and explore the impact on performance in Section 7.

To support querying over simple hierarchies (taxonomies) we describe the simplest instantiation of our index, HII_α, depicted in Figure 6. For each primitive concept and entity, the general idea is to index the *inverse direction* of the "is-a" relations in order to support finding more specific concepts and entities

$$\begin{array}{lcl} C_1 & : & \langle \{D_a, D_b, ...\}, \{\textit{is-a}^{-}\{C'_1, C'_2, ...\}\rangle \\ C_2 & : & \langle \{D_i, D_j, ...\}, \{\textit{is-a}^{-}\{C''_1, C''_2, ...\}\rangle \\ & & \vdots \\ C_n & : & \langle \{D_l, D_p, ...\}, \{\textit{is-a}^{-}\{C^n_1, C^n_2, ...\}\rangle \end{array}$$

Figure 6: HII$_\alpha$: an inverted index mapping entities and concepts to documents with links in the inverse direction of the hierarchical “is-a” relationship

$$\begin{array}{lcl} C_1 & : & \langle \{D_a, D_b, ...\}, \{R^-_1\{C'_1, C'_2, ...\}, ..., R^-_m\{C'_1, C'_2, ...\}\}\rangle \\ C_2 & : & \langle \{D_i, D_j, ...\}, \{R^-_1\{C''_1, C''_2, ...\}, ..., R^-_m\{C''_1, C''_2, ...\}\}\rangle \\ & & \vdots \\ C_n & : & \langle \{D_l, D_p, ...\}, \{R^-_1\{C^n_1, C^n_2, ...\}, ..., R^-_m\{C^n_1, C^n_2, ...\}\}\rangle \end{array}$$

Figure 7: HII$_\beta$: an inverted index mapping entities and concepts to documents with links in the inverse direction of all relations R (including the hierarchical “is-a” relationship).

of a specified entity type. This is a subset of the adjacency list based graph encoding. For each index entry, the document collection directly associated with that concept or entity (but not its entire closure) is listed. Note that we may also populate strictly keyword based entries to support full text search over the same inverted index.

The problem with the simple HII$_\alpha$ is that querying by relations can not efficiently be supported (for example, “*born in(Canada)*”). We extend HII$_\alpha$ to HII$_\beta$ in Figure 7 to support these types of queries by also indexing the inverse direction of all relations R. This also gives us the ability to index ontologies with more complex concept definitions which include both conjunctions and relations. For example, `NOBEL_LAUREATE` *is-a* (`PERSON`, *hasWonPrize*(`NOBEL_PRIZE`)). This index does not introduce any extra redundancy, but rather each entry references the related entries via the appropriate relations. HII$_\beta$ is also encodes sufficient information to efficiently support the disambiguated structured keyword queries described in Section 4. Figure 8 shows an example index for a portion of the KB in our running example.

Lastly, for completeness, we note that our index family can be extended to include a full ontology index by also indexing inverse relations. In our index structure, this amounts to indexing the forward direction of relations (the inverse of the inverse) to support querying over inverse relations (e.g., *bornIn*$^-$(`Albert_Einstein`)).

$$
\begin{array}{rcl}
\texttt{NOBEL\_PRIZE} & : & \langle\{\},\{\textit{is-a}^-\{\texttt{Nobel\_Prize\_In\_Physics},\\
 & & \quad \texttt{Nobel\_Prize\_In\_Chemistry, Nobel\_Peace\_Prize},\ldots\}\}\rangle\\
\texttt{SCIENTIST} & : & \langle\{\},\{\textit{is-a}^-\{\texttt{PHYSICIST, CHEMIST, COMPUTER\_SCIENTIST},\\
 & & \quad \ldots,\texttt{Isaac\_Newton},\ldots\},\ldots\}\rangle\\
\texttt{PHYSICIST} & : & \langle\{\},\{\textit{is-a}^-\{\texttt{Albert\_Einstein, Max\_Planck, Galileo\_Galilei},\\
 & & \quad \ldots\},\ldots\}\rangle\\
\texttt{CHEMIST} & : & \langle\{\},\{\textit{is-a}^-\{\texttt{Marie\_Curie},\ldots\},\ldots\}\rangle\\
\texttt{Nobel\_Prize\_In\_Physics} & : & \langle\{D_4\},\{\textit{hasWonPrize}^-\{\texttt{Albert\_Einstein, Max\_Planck},\\
 & & \quad \texttt{Marie\_Curie},\ldots\},\ldots\}\rangle\\
\texttt{Nobel\_Prize\_In\_Chemistry} & : & \langle\{D_9\},\{\textit{hasWonPrize}^-\{\texttt{Marie\_Curie, Otto\_Wallach},\ldots\\
 & & \quad \},\ldots\}\rangle\\
\texttt{Albert\_Einstein} & : & \langle\{D_1,D_4,D_{12}\},\{\textit{isMarriedTo}^-\{\texttt{Mileva\_Maric}\},\ldots\}\rangle\\
\texttt{Max\_Planck} & : & \langle\{D_4,D_7\},\{\textit{hasAcademicAdvisor}^-\{\texttt{Gustav\_Ludwig\_Hertz},\\
 & & \quad \texttt{Walther\_Bothe, Max\_von\_Laue}\},\ldots\}\}\rangle
\end{array}
$$

Figure 8: Part of an example HII$_\beta$ index.

6.2 Query Evaluation

The procedure illustrated in Figure 9 details the general search procedure for query processing over the hierarchical inverted index. We use "index-lookup(C)" to denote retrieving an index entry for entity or primitive concept C, and use "get(*relation*)" to denote retrieving the entity and concepts associated with *relation* from the index entry. For notational convenience, we also use the function *getEnts*() to denote retrieving the entities associated hierarchical "is-a" relation, and use *getSubClasses*() to denote retrieving the primitive concepts associated with the hierarchical "is-a" relation.

Consider a search for the entities described by the following concept over the index shown in Figure 8.

$$\texttt{PHYSICIST},\ \textit{hasWonPrize}(\texttt{NOBEL\_PRIZE})$$

The first call to our **eval** procedure would compute the intersection of the recursive calls to **eval** over PHYSICIST and *hasWonPrize* (NOBEL_PRIZE). The call to **eval**(PHYSICIST) would run by collecting the entities in the index entry, and would then recursively walk down the hierarchy by following any references (in the example index there are none). The call to **eval**(*hasWonPrize* (NOBEL_PRIZE)) would look up the index entry for NOBEL_PRIZE and collect any entities in the corresponding *hasWonPrize* set (in the example, this set is again empty). For primitive concepts occurring in this set, we would evaluate them and collect all entities by descending the hierarchy in the same way as was done for PHYSICIST. We then descend the hierarchy for NOBEL_PRIZE by following the *is-a*$^-$ references, collecting all entities in the *hasWonPrize* sets in each recursive call. In the example index, we will find all entities corresponding to the *hasWonPrize* relation for the Nobel_Prize_In_Physics and Nobel_Prize_In_Chemistry

```
eval($Q_1, Q_2$):
    return eval($Q_1$) $\cap$ eval($Q_2$)

eval($C$):
    entry = index-lookup($C$)
    res = entry.getEnts()
    sc = entry.getSubClasses()
    for each ($C'$ in sc)
        res = res $\cup$ eval($C'$)
    end for
    return res
```

```
eval($R(C)$):
    entry = index-lookup($C$)
    related = entry.get($R$)
    for each($C'$ in related)
        res = res $\cup$ eval($C'$)
    sc = entry.get($is\text{-}a$)
    for each($C'$ in sc)
        res = res $\cup$ eval($R(C')$)
    end for
    return res
```

Figure 9: Hierarchical Index traversal procedure.

entries. This set includes

{`Albert_Einstein, Max_Planck, Marie_Curie, Otto_Wallach`}

which we then intersect with our set of `PHYSICIST`s,

{`Albert_Einstein, Max_Planck, Galileo_Galilei`}

yielding the query result, {`Albert_Einstein,Max_Planck`} and their corresponding documents $\{D_1, D_4, D_7, D_{12}\}$.

6.3 Query Results and Ranking

There are a number of ways one could envision presenting and ranking results from our system. One approach could have entities as results, each with a list of associated documents. The entities could be ranked by their relationships to the query in the ontology, or based on statistics over the document corpus. Alternatively, one may want a document list as the result, ranked by the relevance of the corresponding entity to the document using a form of tf-idf measure. A third option could combine these two approaches, ranking groups of documents per entity, and ranking the groups based on an overall ranking of entities. This would be analogous to a "GROUP BY" statement in relational query processing, in this case grouping by entity. Various works have looked at entity specific ranking techniques [4, 8, 18].

While ranking is not the focus of this paper, it is a necessary component to a deployable system. We take the simple approach of ranking the resulting document set as a whole, ranking based on the tf-idf score of the corresponding entities that qualified the document as a result.

7 Experimental Evaluation

We have implemented the entire QUICK system as depicted in Figure 2, with the exception of the feedback mechanism for learning new facts from text.

	QUICK	PostgreSQL
KB Representation Size	1.2 GB	41 GB
Average CPU Usage	100%	29.2%

Figure 10: Statistics for experimental runs.

The focus of our experimental evaluation is on performance. The quality of query disambiguations and quality of document results is an important issue, but its relevance is diminished if the system does not perform at a reasonable standard. While additional user studies are required to evaluate the the quality of our system, we are only concerned with the performance aspects in this paper and leave quality evaluation as future work. Informally however, we find in practice that our system does return relevant documents and generally finds intended query interpretations.

7.1 System Descriptions

QUICK is built on a variety of open-source software. For the information extraction component, we built a simple text processor using the OpenNLP library.[3] This tool will parse and annotate person, location, and organization named entities. For simplicity in document retrieval, we build the document side of our hierarchical index separate from the KB information. We built a simple text and named entity indexer using the Lucene library.[4] The hierarchical inverted index was written in Java and uses Berkeley DB[5] as a back-end data store. Our query disambiguation system was also written in Java and uses a separate Berkeley DB store to maintain the knowledge base semantic similarity measures. All semantic similarity values used for disambiguation are pre-computed. We use the Jaccard index discussed in Section 5 as our semantic similarity measure, and the Levenshtein edit distance for syntactic similarity with addition as our aggregation function.

We compare our system to a more traditional approach for semantic search, a relational back-end with a pre-computed closure of the knowledge base. In this system, all inference that could possibly be done over the knowledge base is pre-computed and stored. We use the tools distributed with YAGO to create the database image (both the closure and indexing). We also use the query processor distributed with YAGO to generate SQL equivalents of our workload. We use PostgreSQL[6] as our relational database.

7.2 Data and Workload

We use the 2008 version of YAGO [21] as our knowledge base (note that this is approximately three times larger than the initial instance reported in [21]).

[3]http://opennlp.sourceforge.net/
[4]http://lucene.apache.org/
[5]http://www.oracle.com/technology/products/berkeley-db/je/index.html
[6]http://www.postgresql.org/

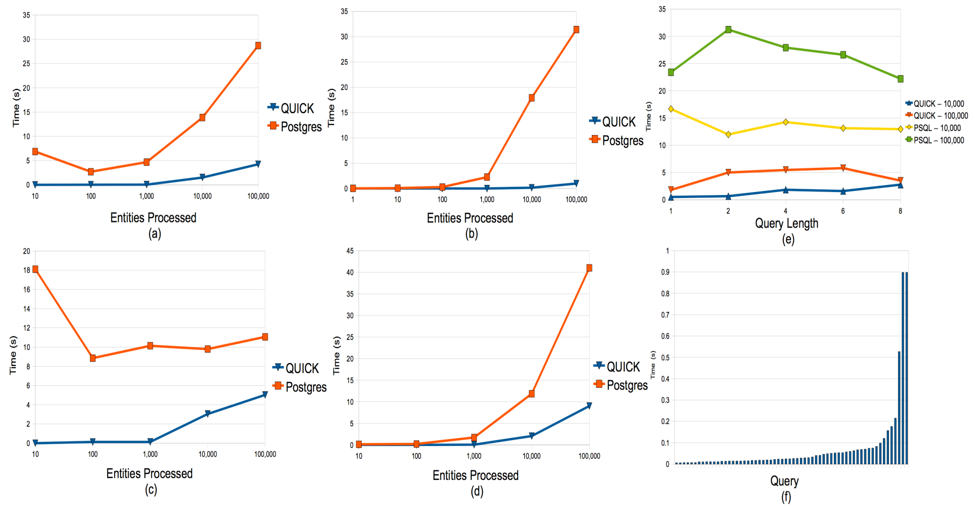

Figure 11: Performance results for (a) mix workload, (b) flat queries, (c) chain queries, and (d) star queries. Graph (e) shows query times as a function of query length for fixed values of entity processing size. Graph (f) shows disambiguation times for all queries in the workload.

YAGO consists of about 6 GB of raw ontology and entity data. It contains over 2 million entities, about 250,000 primitive concepts, 100 relations, and over 20 million facts about these entities, concepts, and relations. We use a snapshot of Wikipedia consisting of 27 GB of individual articles as our corpus. Our KB index is 1.2 GB in total, and when combined with the document index is still under 6 GB.

To evaluate our system, we created a number of queries of varying length and generality. These queries vary in length from one to eight terms, and vary in generality of the terms. We characterize the overall generality of a query by the total number of entities that are processed at intermediate stages in order to evaluate the query. Our workload varies in generality from 1 to 100,000 intermediate entity results. Queries also vary in their shape. Our workload includes flat queries (conjunctions with no relations), chain queries (a nesting of relations), and star queries (a conjunction of relations) of each length and generality.

7.3 Experimental Results

Figure 10 summarizes two important characteristics of our approach vs. PostgreSQL. QUICK makes use of a compact 1.2 GB representation of the knowledge base, while the relational encoding of the closure consumes 41 GB. Conversely, our approach utilizes 100% of the CPU during query processing, while Post-

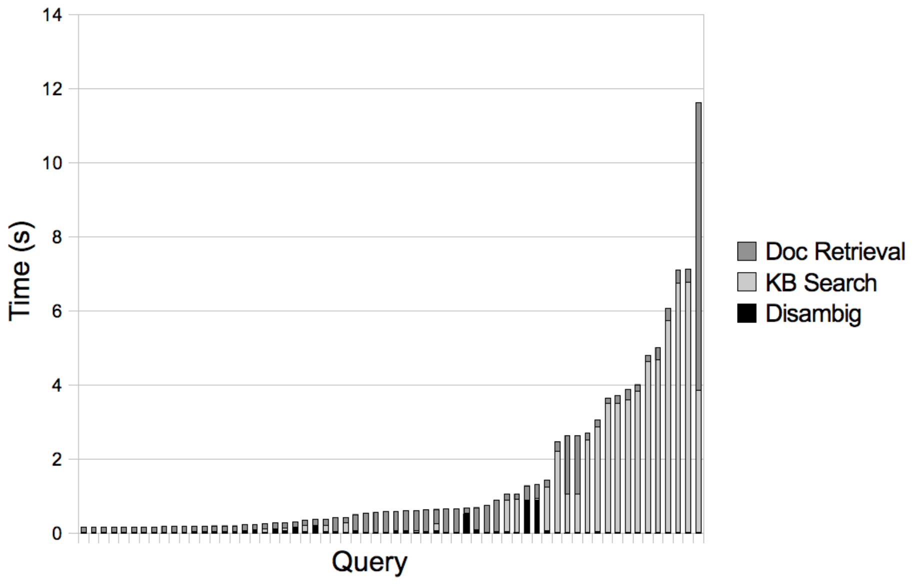

Figure 12: Total run times for all queries.

greSQL stays under 30%. This tradeoff of CPU cycles for space allows us to keep a sufficiently compact KB index facilitating in memory KB query processing. Conversely, the full closure approach with PostgreSQL must necessarily be disk resident. The tradeoff is now that QUICK must support inference at query time, a CPU intensive task, while the relational approach will retrieve the pre-computed inferred data from disk, an I/O intensive task.

Figure 11 (a) shows KB query run-times for QUICK and PostgreSQL for our entire workload with each separate query type shown in Figures 11 (b) to (d). In this graph, queries are grouped based on the number of entities touched by our system during query processing, and the average run-time is graphed. (Note that PostgreSQL may choose query plans that are substantially different from how our system evaluates queries. This is why PostgreSQL does not necessarily follow the same trends as QUICK in the graphs.) It is evident from these graphs that our approach of materializing inferred data on-demand performs better on average than the stored approach. This is because our KB representation is compact enough to maintain in memory. Also, our query processing algorithm is specially designed for our class of nested concept queries. PostgreSQL on the other hand uses repeated joins of a large *facts* table, a general relational solution that is not tailored to this application. Figure 11 (e) shows run-time as a function of query length for two fixed sizes of query generality (10,000 and 100,000). It is evident that query length has little effect on performance, query performance is largely driven by query generality.

Figure 11 (f) shows the performance of our query disambiguation system for all queries. In each case we find the top 10 disambiguations for a simple text representation of the query. In most cases disambiguation can be achieved very efficiently, however some cases can be more challenging. These cases correspond to situations in which many of the candidate concepts have semantic relations to one another, and thus non-zero edges in our disambiguation graph.

Figure 12 shows the breakdown of performance over the entire workload for the three phases of query processing (query disambiguation, KB search, and document retrieval). KB search is generally the most expensive task, while document retrieval occasionally dominates. This corresponds to situations in which there are a large number of qualifying entities for the query, and thus the document retrieval must make a large number index look-ups.

8 Related Work

The problem of entity/relation search over extracted data sets has recently received increasing attention, with various efforts for structured query processing in this setting [2, 4, 10, 11, 13]. Our system differs from these approaches in that they generally do not consider rich semantics of the entities and relations, or do so using a pre-computed closure.

The other differing factor is our integration of ambiguous terms into queries. NAGA [13] also supports ambiguity in queries via regular expression matching and keyword matching against strings in a special *"means"* relation that is part of YAGO. Query processing is done by finding all matches and evaluating the constraints in the query against candidates. This is in contrast to our heuristic based disambiguation as a preprocessing step to query evaluation.

With respect to the semantic search side of our system, our approach differs from current works in a number of ways. First, semantic search systems generally assume documents are annotated with structured semantic data, such as RDF or OWL [7, 17], or XML [6] (with explicit references to the ontology concepts) and often assume a structured query language. Our system uses a mix of extracted structure from text, like the fact triples in NAGA, and the free text itself, like the text-based semantic search engine ESTER [1].

A few works have considered semantic search over text databases [1, 3] making use of keyword search interfaces, the most successful of which is ESTER [1], an ontology-assisted wikipedia search engine. In ESTER, the keyword query is matched against known entities and a dynamic user interface allows users to disambiguate among possible matches to primitive concepts. However, keyword mappings are based solely on primitive concepts and no attempts to compose general concepts from sets of keywords are made.

In these works inference is supported by expanding the deductive closure into the text, potentially causing large amounts of redundancy. The expressivity of queries making use of inference is also limited in these systems and restrictions on the ontologies are imposed, generally requiring a small number of pre-defined top-level concepts as starting points for queries [1, 3]. Our proposed system does not rely on these types of assumptions and can process a broad class of nested concept queries. The EntityRank system [4] also incorporates keywords into structured queries as so called context keywords. Our approach of including parts of queries which can not be disambiguated as residual keywords for ranking is similar. The TopX system [22] has also been designed to support inference over ontologies, however the current version of this system only includes support for WordNet synonyms, and ontology integration is listed as an avenue for future work.

9 Conclusions and Future Work

We have proposed a keyword-based structured query language for entity-based document retrieval, and shown how rich semantics from a reference knowledge base can be integrated. We have explored ambiguity issues with basing a structured query language on keywords and proposed a solution for disambiguation. We have also proposed a compact representation for knowledge base information that facilitates efficient semantic search for nested concept queries.

While we have demonstrated the competitive performance of our system, we have not yet experimentally validated the quality of results. Our future work will include a user study to identify the quality of our query disambiguation scheme as well as the accuracy of our entity-based retrieval.

There are also a number of other avenues for future work which focus on the issue of scalability in different contexts. While this work explored scaling the size of the metadata and expressiveness of the query language, one could look at other variants of the scalability problem. These problems include further increasing the expressiveness of the query language, increasing the expressiveness of the logic dialect that underlies the knowledge base, and scaling the corpus size. The first two problems further the challenges of efficient query processing and compact indexing, while the third problem poses issues in information extraction likely to require an on-the-fly approach rather than full preprocessing.

Another problem for future work is ranking. There is an abundance of information available at ranking time: the qualifying entities, the qualifying documents, statistical metrics of entities to documents (e.g., tf-idf scores), the terms mentioned in the query, and so on. More sophisticated approaches may exploit this information to improve ranking.

References

[1] H. Bast, A. Chitea, F. Suchanek, and I. Weber. ESTER: efficient search on text, entities, and relations. In *SIGIR '07: Proc. of the 30th intl. conf. on information retrieval*, pages 671–678. ACM, 2007.

[2] M. J. Cafarella, C. Re, D. Suciu, and O. Etzioni. Structured Querying of Web Text Data: A Technical Challenge. In *CIDR*, pages 225–234, 2007.

[3] P. Castells, M. Fernandez, and D. Vallet. An adaptation of the vector-space model for ontology-based information retrieval. In *IEEE Transactions on Knowledge and Data Enginering 19(02)*, pages 261–272, 2007.

[4] T. Cheng, X. Yan, and K. C.-C. Chang. EntityRank: Searching Entities Directly and Holistically. In *VLDB*, pages 387–398, 2007.

[5] F. M. Couto, M. J. Silva, and P. M. Coutinho. Measuring semantic similarity between Gene Ontology terms. *Data & Knowledge Engineering*, 61(1):137 – 152, 2007.

[6] F. Farfan, V. Hristidis, A. Ranganathan, and M. Weiner. XOntoRank: Ontology-Aware Search of Electronic Medical Records. In *To appear in 25th Intl. Conf. on Data Engineering (ICDE 2009)*, 2009.

[7] R. Guha, R. McCool, and E. Miller. Semantic search. In *WWW '03: Proc. of the 12th Intl. Conf. on World Wide Web*, pages 700–709. ACM, 2003.

[8] H. Hwang, V. Hristidis, and Y. Papakonstantinou. Objectrank: a system for authority-based search on databases. In *SIGMOD '06: Proc. of 2006 Intl. Conf. on Management of data*, pages 796–798. ACM, 2006.

[9] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, 2004.

[10] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. Towards a query optimizer for text-centric tasks. *ACM Trans. Database Syst.*, 32(4):21, 2007.

[11] A. Jain, A. H. Doan, and L. Gravano. Optimizing SQL Queries over Text Databases. *Data Engineering, 2008. ICDE 2008. IEEE 24th intl. conf. on*, pages 636–645, April 2008.

[12] J. J. Jiang and D. W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. In *Intl. Conf. on Computational Linguistics*, 1997.

[13] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and Ranking Knowledge. In *24th Intl. Conf. on Data Engineering (ICDE 2008)*, 2008.

[14] C. Leacock and M. Chodorow. Combining local context with wordnet similarity for word sense identification. In *WordNet: A Lexical Reference System and its Application*, 1998.

[15] D. Lin. An information-theoretic definition of similarity. In *ICML '98: Proc. of the Fifteenth Intl. Conf. on Machine Learning*, pages 296–304. Morgan Kaufmann Publishers Inc., 1998.

[16] A. G. Maguitman, F. Menczer, H. Roinestad, and A. Vespignani. Algorithmic detection of semantic similarity. In *WWW '05: Proc. of the 14th Intl. Conf. on World Wide Web*, pages 107–116. ACM, 2005.

[17] J. Mayfield and T. Finin. Information retrieval on the Semantic Web: Integrating inference and retrieval. In *Workshop on the Semantic Web at the 26th International SIGIR Conference on Research and Development in Information Retrieval*, 2003.

[18] Z. Nie, Y. Zhang, J.-R. Wen, and W.-Y. Ma. Object-level ranking: bringing order to web objects. In *WWW '05: Proc. of the 14th Intl. Conf. on World Wide Web*, pages 567–574. ACM, 2005.

[19] P. Resnik. Semantic Similarity in a Taxonomy: An Information-Based Measure and its Application to Problems of Ambiguity in Natural Language. *Journal of Artificial Intelligence Research*, 11:95–130, 1999.

[20] M. A. Rodríguez and M. J. Egenhofer. Determining Semantic Similarity among Entity Classes from Different Ontologies. *IEEE Trans. on Knowledge and Data Engineering.*, 15(2):442–456, 2003.

[21] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge - Unifying WordNet and Wikipedia. In *16th Intl. World Wide Web Conference (WWW 2007)*, pages 697–706, 2007.

[22] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. Topx: efficient and versatile top-k query processing for semistructured data. *The VLDB Journal*, 17(1):81–115, 2008.

[23] Z. Wu and M. Palmer. Verb semantics and lexical selection. In *32nd. Annual Meeting of the Association for Computational Linguistics*, pages 133 –138, 1994.