From Keywords to Semantic Queries — Incremental Query Construction on the Semantic Web

Gideon Zenz [a], Xuan Zhou [b], Enrico Minack [a], Wolf Siberski [a], Wolfgang Nejdl [a]

[a] *L3S Research Center, Appelstr. 9a, 30167 Hannover, Germany*

[b] *CSIRO ICT Centre, GPO Box 664, Canberra ACT 2601, Australia*

Abstract

Constructing semantic queries is a demanding task for human users, as it requires mastering a query language as well as the schema which has been used for storing the data. In this paper, we describe QUICK, a novel system for helping users to construct semantic queries in a given domain. QUICK combines the convenience of keyword search with the expressivity of semantic queries. Users start with a keyword query and then are guided through a process of incremental refinement steps to specify the query intention. We describe the overall design of QUICK, present the core algorithms to enable efficient query construction, and finally demonstrate the effectiveness of our system through an experimental study.

Key words: Semantic Web, Keyword Search, Query Construction

1 Introduction

With the advance of the Semantic Web, increasing amounts of data are available in a structured and machine understandable form. This opens opportunities for users to employ semantic queries instead of simple keyword based ones, to accurately express the information need. However, constructing semantic queries is a demanding task for human users [10]. To compose a valid semantic

Email addresses: `zenz@L3S.de` (Gideon Zenz), `xuan.zhou@csiro.au` (Xuan Zhou), `minack@L3S.de` (Enrico Minack), `siberski@L3S.de` (Wolf Siberski), `nejdl@L3S.de` (Wolfgang Nejdl).

query, a user has to (1) master a query language (e.g. SPARQL) and (2) acquire sufficient knowledge about the ontology or the schema of the data source. While there are systems which support this task with visual tools [20,25] or natural language interfaces [4,12,13,17], the process of query construction can still be complex and time consuming. According to [23], users prefer keyword search, and struggle with the construction of semantic queries although being supported with a natural language interface.

Several keyword search approaches have already been proposed to ease information seeking on semantic data [15,31,33] or databases [1,30]. However, keyword queries lack the expressivity to precisely describe the user's intent. As a result, ranking can at best put query intentions of the majority on top, making it impossible to take the intentions of all users into consideration.

In this paper, we introduce QUICK [1], a novel system for querying semantic data. QUICK internally works on pre-defined domain-specific ontologies. A user starts by entering a keyword query, QUICK then guides the user through an incremental construction process, which quickly leads to the desired semantic query. Users are assumed to have basic domain knowledge, but don't need specific details of the ontology, or proficiency in a query language. In that way, QUICK combines the convenience of keyword search with the expressivity of semantic queries.

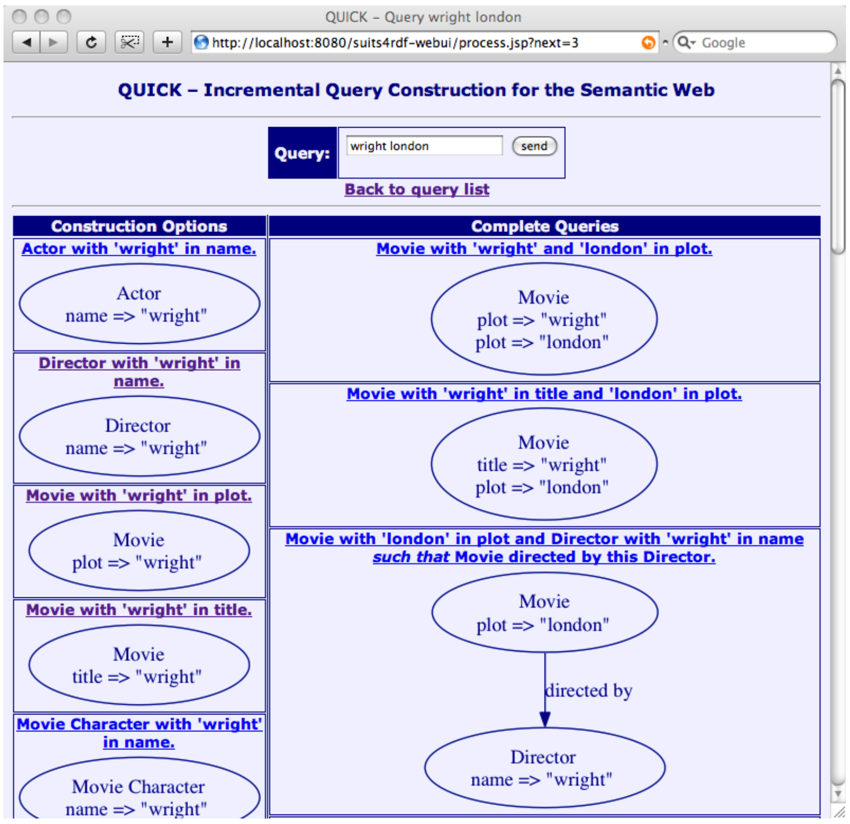
The paper presents a detailed realization of the QUICK system, including the following contributions: (1) we defined a framework for incrementally constructing semantic queries from keywords; (2) we devised algorithms to generate near-optimal query construction guides, which enable users to quickly construct semantic queries; (3) to support the QUICK system, we designed a scheme for optimizing the execution of full-text queries on RDF data; (4) we conducted experiments to evaluate the effectiveness of QUICK and the efficiency of the proposed algorithms.

The rest of the paper is organized as follows. Section 2 provides an overview on how to use QUICK. In Section 3 we present our framework for incremental query construction. Section 4 presents algorithms for generating near-optimal query guides. Section 5 introduces optimization techniques to improve query execution performance. In Section 6, we present the results of our experimental evaluation. Section 7 reviews the related work. We close with conclusions in Section 8.

[1] QUery Intent Constructor for Keywords

2 QUICK Overview

As illustrated in Fig. 1, the interface of QUICK consists of three parts, a search field (on the top), the construction pane showing query construction options (on the left), and the query pane showing semantic queries (on the right).


QUICK – Query wright london

http://localhost:8080/suits4rdf-webui/process.jsp?next=3

QUICK – Incremental Query Construction for the Semantic Web

Query: wright london send

Back to query list

Construction Options	Complete Queries
Actor with 'wright' in name.	Movie with 'wright' and 'london' in plot.
Director with 'wright' in name.	Movie with 'wright' in title and 'london' in plot.
Movie with 'wright' in plot.	Movie with 'london' in plot and Director with 'wright' in name *such that* Movie directed by this Director.
Movie with 'wright' in title.	
Movie Character with 'wright' in name.	

Fig. 1. QUICK User Interface

Suppose a user looks for a movie set in *London* and directed by *Egdar Wright*[2]. The user starts by entering a keyword query, for instance '*wright london*'. Of course, these keywords can imply a lot of other semantic queries than the intended one. For example, one possible query is about an actor called *London Wright*. Another one could search for a character *Wright*, who was performed by an actor *London*. QUICK computes all possible semantic queries and presents selected ones in the query pane. More importantly, it also generates a set of query construction options and presents them in the construction pane. If the intended query is not yet offered, the user can incrementally construct this query by selecting an option in the construction pane. Whenever the user makes a selection, the query pane changes accordingly, zooming into the subset of semantic queries that conform to the chosen options. At the

[2] Throughout this paper, we use the IMDB movie data set as an example to illustrate our approach.

same time, a new set of construction options is generated and presented in the construction pane. We call this series of construction options *query guide*, because it offers the user a path to the intended query. In the screenshot, the user has already selected that '*london*' should occur in the movie plot, and is now presented alternate construction options for '*wright*'. When the user selects the desired query, QUICK executes it and shows the results.

The generated construction options ensure that the space of semantic interpretations is reduced rapidly with each selection. For instance, by specifying that '*london*' refers to a movie and not a person, more than half of all possible semantic queries are eliminated. After a few choices, the query space comprises only a few queries, from which the user can select the intended one easily.

3 Query Construction Framework

In this section, we introduce the query construction framework of QUICK. We describe our model for transforming keyword queries to semantic queries using an incremental refinement process.

3.1 Preliminaries

QUICK works on any RDF knowledge base with an associated schema in RDFS; this schema is the basis for generating semantic queries. We model schema information as a *schema graph*, where each node represents either a concept or a free literal, and each edge represents a property by which two concepts are related. To keep Def. 1 simple, we assume explicit rdf:type declarations of all concepts.

Definition 1 (Knowledge Base) *Let L be the set of literals, U the set of URIs. A knowledge base is a set of triples $G \subset (U \times U \times (U \cup L))$. We use $R = \{r \in U \mid \exists (s\ p\ o) \in G : (r = s \vee r = o)\}$ to represent the set of resources*[3]*, $P = \{p \mid \exists s, o : \ (s\ p\ o) \in G\}$ to represent the set of properties, and $C = \{c \mid \exists s : \ (s\ rdf:type\ c) \in G\}$ to represent the set of concepts.*

Definition 2 (Schema Graph) *The schema graph of a knowledge base G is represented by $SG = (C, EC, EL)$, where C denotes a set of concepts, EC denotes the possible relationships between concepts, and EL denotes possible relationships between concepts and literals. Namely, $EC = \{(c_1, c_2, p) \mid \exists r_1, r_2 \in$*

[3] To keep the presentation clear we do not consider blank nodes; adding them to the model is straightforward.

$R,\ p \in P : (r_1, p, r_2) \in G \wedge (r_1\ rdf{:}type\ c_1) \in G \wedge (r_2\ rdf{:}type\ c_2) \in G\}$, and $EL = \{(c_1, p) \mid \exists r_1 \in R,\ p \in P, l \in L : (r_1,\ p, l) \in G \wedge (r_1\ rdf{:}type\ c_1) \in G\}$

The schema graph serves as the basis for computing query templates, which allow us to construct the space of semantic query interpretations, as discussed in the following.

3.2 From Keywords to Semantic Queries

When a Keyword Query $kq = \{t_1, \cdots, t_n\}$ is issued to a knowledge base, it can be interpreted in different ways. Each interpretation corresponds to a semantic query. For the query construction process, QUICK needs to generate the complete *semantic query space*, i.e., the set of all possible semantic queries for the given set of keywords.

The query generation process consists of two steps. First, possible query patterns for a given schema graph are identified, not taking into account actual keywords. We call these patterns *query templates.* Templates corresponding to the example queries from Section 2 are: 'retrieve movies directed by a director' or 'retrieve actors who have played a character'.

Formally, a query template is defined as composition of schema elements. To allow multiple occurrences of concepts or properties, they are mapped to unique names. On query execution, they are mapped back to the corresponding source concept/property names of the schema graph.

Definition 3 (Query Template) *Given a schema graph $SG = (C_{SG}, EC_{SG}, EL_{SG})$, $T = (C_T, EC_T, EL_T)$ is a query template of SG, iff (1) there is a function $\tau : C_T \rightarrow C_{SG}$ mapping the concepts in C_T to the source concepts in C_{SG}, such that $(c_1, c_2, p) \in EC_T \Rightarrow (\tau(c_1), \tau(c_2), p) \in EC_{SG}$ and $(c_1, L, p) \in EL_T \Rightarrow (\tau(c_1), L, p) \in EL_{SG}$; (2) the graph defined by T is connected and acyclic. We call a concept that is connected to exactly one other concept in T* leaf concept .

Figure 2 shows three query templates with sample variable bindings. QUICK automatically derives all possible templates offline from the schema graph (up to a configurable maximum size), according to Definition 3. This is done by enumerating all templates having only one edge, and then recursively extending the produced ones by an additional edge, until the maximum template size is reached.

Currently, we limit the expressivity of templates to acyclic conjunctions of triple patterns. Further operators (e.g., disjunction) could be added, however at the expense of an increased query space size. Investigation of this tradeoff

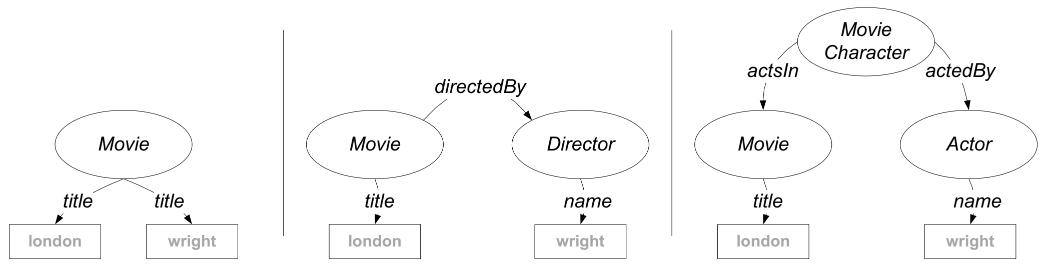


Fig. 2. Sample query templates for the IMDB schema; the terms in gray represent instantiations of these templates to semantic queries for *'wright london'*

is part of future work.

In the second step, *semantic queries* are generated by binding keywords to query templates. A keyword can be bound to a literal if an instance of the underlying knowledge base supports it. Alternatively, it can be bound to a concept or property, if the keyword is a synonym (or homonym) of the concept or property name. A full-text index on the knowledge base is used to efficiently identify such bindings. Fig. 2 shows some semantic queries for the keyword set '*wright london*', which bind the keywords to the literals of three different query templates. The left one searches for a movie with '*wright*' and '*london*' in its title. The middle one searches for a movie with '*london*' in its title directed by a director '*wright*'. The right one searches for an actor '*wright*' playing a character in a movie with '*london*' in its title. Furthermore, keywords can also be matched to properties and classes, such as '*name*' or '*Movie*'.

Definition 4 (SEMANTIC QUERY) *Given a keyword query kq, a semantic query is a triple* $sq = (kq, T, \theta)$*, where* $T = (C_T, EC_T, EL_T)$ *is a query template, and* θ *is a function which maps kq to the literals, concepts and properties in* T*.* $sq = (kq, T, \theta)$ *is a valid semantic query, iff for any leaf concept* $c_i \in C_T$*, there exists a keyword* $k_i \in kq$ *that is mapped by* θ *to* c_i *itself, or a property or a literal connected to* c_i*.*
The SEMANTIC QUERY SPACE *for a given query kq and schema graph SG is the set of all queries* $SQ = \{sq | \exists \theta, T : (kq, T, \theta) \text{ is a query template}\}$*.*

In our model, each term is bound seperately to a node of a template. Phrases can be expressed by binding all corresponding terms to the same property, c.f. the left-hand example in Fig. 2. Additionally, linguistic phrase detection could be performed as a separate analysis step; in this case, a phrase consisting of several keywords would be treated as one term when guiding the user through the construction process.

The QUICK user interface prototype shows the queries as graphs as well as in textual form. The query text is created by converting graph edges to phrases. For each edge connecting a concept with a bound property, we create the phrase "<*concept*> **with** <keyword> **in** <*property*>", using the respective concept and property labels. If an edge connects two concepts, the relation is translated to the phrase "**this** <*concept1*> <*property*> ***this*** <*concept2*>". For the second query in Figure 2, the following text is be generated: "*Movie* **with** 'london' **in** *title* **and** *Director* **with** 'wright' **in** *name* **such that** *Movie directed by* **this** *Director*".

As shown in Section 5, a semantic query corresponds to a combination of SPARQL triple pattern expressions, which can be directly executed on an RDF store.

3.3 Construction Guides for Semantic Queries

QUICK presents the user with query construction options in each step. By selecting an option, the user restricts the query space accordingly.

These options are similar to semantic queries, except they don't bind all query terms. Therefore, the construction process can be seen as the step-wise process of selecting partial queries that subsume the intended semantic query.

To describe precisely how a query guide is built, we introduce the notions of partial query and of sub-query relationship. Our notion of query subsumption relies on the RDF Schema definition of concept and property subsumption. Note that our algorithms are not dependent on a specific definition of query subsumption, it would work equally well with more complex approaches, e.g., concept subsumption in OWL.

Definition 5 (Sub-query, Partial query)
$sa = (q_a, T_a, \theta_a)$ is a sub-query of $sb = (q_b, T_b, \theta_b)$, or sa subsumes sb, iff:
(1) $q_a \subset q_b$;
(2) there exists a sub-graph isomorphism ϕ between T_a and T_b, so that each concept $a_1 \in T_a$ subsumes $\phi(a_1)$ and each property $p \in T_a$ subsumes $\phi(p)$;
(3) for any $k_1 \in q_a$, $\phi(\theta_a(k_1)) = \theta_b(k_1)$.
A partial query is a sub-query of a semantic query.

For example, in Fig. 3, the partial queries pq_1, pq_2 and pq_3 are sub-queries of sq_1, sq_2 and sq_3 respectively.

The construction options of QUICK are modeled as a *Query Construction Graph* (QCG), as illustrated in Fig. 3. While the example shown is a tree, in general the QCG can be any directed acyclic graph with exactly one root node. Given a set of semantic queries SQ, a QCG of SQ satisfies:
(1) the root of QCG represents the complete set of queries in SQ;
(2) each leaf node represents a single semantic query in SQ;
(3) each non-leaf node represents the union of the semantic queries of its

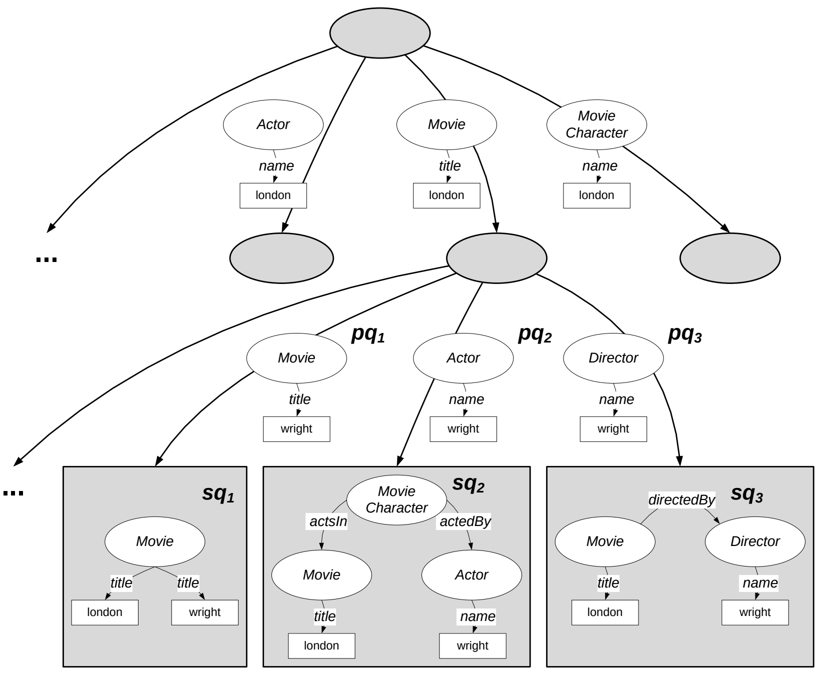


Fig. 3. Part of a query guide for '*wright london*'

children;
(4) each edge represents a partial query;
(5) the partial query on an incoming edge of a node subsumes all the semantic queries represented by that node.

Definition 6 (Query Construction Graph) *Given a set of semantic query SQ and its partial queries PQ, a Query Construction Graph is a graph* $QCG = (V, E)$, *where* $V \subset SQ^*$, $E \subset \{(v_1, v_2, p) | v_1, v_2 \in V,\ v_2 \subset v_1,\ p \in PQ, p \text{ subsumes } v_2\}$ *and* $\forall v \in V, |v| > 1 : v = \{\bigcup v_i | \exists p : (v, v_i, p) \in E\}$

If SQ is the complete query space of a keyword query, a QCG of SQ is a *Query Guide* of the keyword query.

Definition 7 (Query Guide) *A* query guide *for a keyword query is a query construction graph whose root represents the complete semantic query space of that keyword query.*

With a query guide, query construction can be conducted. The construction process starts at the root of the guide, and incrementally refines the user's intent by traversing a path of the graph until a semantic query on a leaf is reached. In each step, the user is presented the partial queries on the outgoing edges of the current node. By selecting a partial query, the user traverses to the respective node of the next level. In the example of Figure 3, after having chosen that '*wright*' is part of the actor's name and '*london*' part of the movie's title, QUICK can already infer the intended query. The properties of the query construction graph guarantee that a user can construct every semantic query in the query space.

Every query guide comprises the whole query space, i.e., covers all query intentions. However, these guides vary widely in features such as branching factor and depth. A naïvely generated guide will either force the user to evaluate a huge list of partial query suggestions at each step (width), or to take many steps (depth) until the query intention is reached. E.g. for only 64 semantic queries, a naïve algorithm produces a guide which requires the user to evaluate up to 100 selection options to arrive at the desired intention, while an optimal guide only requires 17. Therefore, we aim at a query guide with minimal interaction cost.

We define the interaction cost as the number of partial queries the user has to view and evaluate to finally obtain the desired semantic query. As QUICK does not know the intention when generating the guide, the worst case is assumed: the interaction cost is the cost of the path through the guide incurring most evaluations of partial queries. This leads to the following cost function definition:

Definition 8 (Interaction Cost of a Query Construction Graph) *Let* $cp = (V', E')$ *be a path of a query construction graph* $QCG = (V, E)$*, i.e.,* $V' = \{v_1, ..., v_n\} \subset V$ *and* $E' = \{(v_1, v_2, p_1), (v_2, v_3, p_2), ..., (v_{n-1}, v_n, p_{n-1})\} \subset E$*. Then:*
$cost(cp) = \sum_{v \in V'} |\{p : (v, v_1, p) \in E\}|$*, and* $cost(QCG) = \max(cost(cp) : cp)$*.*

Definition 9 (Minimum Query Construction Graph) *Given a set of semantic queries SQ, a query construction graph QCG is a* minimum query construction graph *of SQ, iff there does not exist another query construction graph* QCG' *of SQ such that* $cost(QCG) > cost(QCG')$*.*

A query guide which satisfies Definition 9 leads the user to the intended query with minimal interactions. In the following section, we show how to compute such guides efficiently.

4 Query Guide Generation

For a given keyword query, multiple possible query guides exist. While every guide allows the user to obtain the wanted semantic query, they differ significantly in effectiveness as pointed out in Section 3.3. It is thus essential to find a guide that imposes as little effort on the user as possible, i.e., a minimum query guide. Query construction graphs have several helpful properties for constructing query guides:

Lemma 10 *(Query Construction Graph properties)*
(i) Given a node in a query construction graph, the complete sub-graph with this node as root is also a query construction graph.
(ii) Suppose QCG is a query construction graph, and A is the set of children of its root. The cost of QCG is the sum of the number of nodes in A and the maximum cost of the sub-graphs with root in A, i.e. $Cost(T) = |A| + MAX(Cost(a) : a \in A)$.
(iii) Suppose QCG is a minimum query construction graph and A is the set of children of its root. If g is the most expensive sub-graph with root in A, then g is also a minimum query construction graph.

4.1 Straightforward Guide Generation

Lemma 10 can be exploited to construct a query construction guide recursively. Based on Property (ii), to minimize the cost of a query construction graph, we need to minimize the sum of (1) the number of children of the root, i.e., $|A|$ and (2) the cost of the most expensive sub-graph having one of these children as root, i.e., $MAX(Cost(a) : a \in A)$. Therefore, to find a minimum construction graph, we can first compute all its possible minimum sub-graphs. Using these sub-graphs, we find a subset A with the minimum $|A| + MAX(Cost(a) : a \in A)$. The method is outlined in Algorithm 1.

Algorithm 1: Straightforward Query Guide Generation

```
**Simple_QGuide()**
**Input**: partial queries $P$, semantic queries $S$
**Output**: query guide $G$
**if** $|S| = 1$ **then**
    **return** $S$ ;
**end**
**for** *each* $p \in P$ **do**
    $p.sg$ := **Simple_QGuide**($P - \{p\}$, $S \cap p.SQ$) ;
    // p.SQ denotes the semantic queries subsumed by p
**end**
$G.cost := \infty$ ;
**for** *each* $p \in P$ **do**
    $Q(p) := \{(p' \in P) : p'.sg.cost \leq p.sg.cost\}$ ;
    $min\_set$ := minSetCover($S - p.SQ, Q(p)$) ;
    **if** $G.cost > |min\_set| + p.sg.cost + 1$ **then**
        $G := min\_set \cup \{p\}$ ;
        $G.cost := |min\_set| + p.sg.cost$ ;
    **end**
**end**
**return** $G$
```

According to Lemma 10, this algorithm always finds a minimum query guide. However, it relies on the solution of the SetCover problem, which is NP-complete [11]. Although there are polynomial approximation algorithms for minSetCover with logarithmic approximation ratio, our straightforward algorithm still incurs prohibitive costs. Using a greedy minSetCover, the complexity is still $O(|P|! \cdot |P|^2 \cdot |S|)$. In fact, we can prove that the problem of finding a minimum query guide is NP hard.

Definition 11 minSetCover *Given a universe U and a family S of subsets of U, find the smallest subfamily $C \subset S$ of sets whose union is U.*

Theorem 12 *The* minConstructionGraph *problem is NP hard.*

PROOF. *We reduce* minSetCover *to* minConstructionGraph*:*
$M_S : U \leftrightarrow U_S$ is a bipartite mapping between U and a set of semantic queries U_S. $M_P : S \leftrightarrow S_P$ is a bipartite mapping between S and a set of partial queries S_P, such that each partial query $p \in S_P$ subsumes the semantic queries $M_S(M_P^{-1}(p))$. Create another set of semantic queries A_S and a set of partial queries A_P. Let $|A_S| = 2 \times |M_S|$. Let A_P contain two partial queries, each covering half of A_S. Therefore, the cost of the minimum query construction graph of A_S is $|M_S| + 1$, which is larger than any query construction graph of M_S. Based on Lemma 10, if we solve minConstructionGraph*$(U_S \cup A_S, U_P \cup A_P)$, we solve* minSetCover*$(U, S)$.*

4.2 Incremental Greedy Query Guide Generation

As shown, the straightforward algorithm is too expensive. In this section, we propose a greedy algorithm, which computes the query construction graph in a top-down incremental fashion.

Algorithm 2 starts from the root node and estimates the optimal set of partial queries that cover all semantic queries. These form the second level of the query construction graph. In the same fashion, it recurses through the descendants of the root to expand the graph. Thereby, we can avoid constructing the complete query construction graph; as the user refines the query step-by-step, it is sufficient to compute only the partial queries of the node the user has just reached.

The algorithm selects partial queries one by one. Then, it enumerates all remaining partial queries and chooses the one incurring minimal *total estimated cost.* It stops when all semantic queries are covered. The complexity of the algorithm is $O(|P| \cdot |S|)$.

The formula for the *total estimated cost* of a query construction graph is given in Definition 13.

Algorithm 2: Incremental Greedy Query Guide Generation

```
Input: partial queries $P$, semantic queries $S$
Output: query guide $G$
if $|S| = 1$ then
    return $S$ ;
end
$G := \emptyset$
while $|S| \neq 0$ do
    select $p \in P$ with min. $TotalEstCost(S, G \cup \{p\})$
    if no such p exists then
        break ;
    end
    $G := G \cup \{p\}$ ;
    $S := S - p.SQ$ ;
    // p.SQ denotes the semantic queries subsumed by p
end
if $|S| \neq 0$ then
    $G := G \cup S$ ;
end
return $G$
```

Definition 13 (Total estimated cost) *Let S be the semantic queries to cover, SP the set of already selected partial queries, and p the partial query to evaluate, the estimated cost of the cheapest query construction graph is:*

$$TotalEstCost(S, SP) = |S| \frac{|SP|}{|S \cap \bigcup SP|} + \max(minGraphCost(|p|) : p \in SP),$$

where

$$minGraphCost(n) = \begin{cases} n = 1 : & 0 \\ n = 2 : & 2 \\ n > 2 : & e \cdot \ln(n) \end{cases}$$

Here, $minGraphCost(|p|)$ estimates the minimum cost of the query construction graph of p. Suppose f is the average fan-out for n queries, then the cost is approximately $f \cdot \log_f(n)$, which is minimal for $f = e$. The first addend of $TotalEstCost$ estimates the expected number of partial queries that will be used to cover all semantic queries. This assumes the average number of semantic queries covered by each partial query does not to vary.

As discussed above, the algorithm runs in polynomial time with respect to the number of partial and semantic queries. Although the greedy algorithm can still be costly when keyword queries are very long, our experimental results of Section 6.4 show that it performs very well if the number of keywords is within realistic limits

5 Query Evaluation

When the user finally selects a query that reflects the actual intention, it will be converted to a SPARQL query and evaluated against an RDF store to retrieve the results. The conversion process is straight forward: For each concept node in the query or edge between nodes, a triple pattern expression of SPARQL is generated. In the first case, it specifies the node type, in the second case it specifies the relation between the nodes. Finally, for each search term, a filter expression is added. See Fig. 4 for an example [4].

```
SELECT * WHERE { ?movie rdf:type imdb:Movie.
   ?movie rdf:type imdb:Director.
   ?movie imdb:directedBy ?director.
   ?movie imdb:title ?term1 FILTER regex(?term1, ".*london.*").
   ?director imdb:name ?term2 FILTER regex(?term2, ".*wright.*"). }
```

Fig. 4. SPARQL query for the right-hand sample in Fig. 2.

To make QUICK work, these SPARQL queries need to be processed efficiently. Recent research in the area of SPARQL query evaluation has made significant progress [22,24,29,27], mainly based on dynamic programming, histograms and statistical methods for estimating cardinality and selectivity. However, available mature SPAQL engines either do not employ these techniques or do not offer them in combination with the full-text querying capabilities QUICK requires. In [19], we evaluated the performance of the most popular RDF engines with respect to these features, using an extended Lehigh University Benchmark (LUBM). The results show that the evaluated engines do not perform well on the type of hybrid queries needed by QUICK, as they still use rather simple heuristics to optimize the order of join operations.

Statistics of join cardinality can be employed to optimize the join order effectively. We adopted this approach in QUICK to improve the semantic query execution performance. In the following, we show how to estimate the cardinality of each triple pattern, and explain how to exploit these estimates for join ordering.

Cardinality Estimation Triple patterns in SPARQL queries can be categorized into four types. For each type, we use a different method to estimate

[4] In our actual implementation, we use the custom language extension of LuceneSail to express full-text filters.

the cardinality. Let `s p o` be a triple pattern, where `p` is always a constant. For patterns of type `s p ?o`, where `!s` is a constant and `?o` is a free variable, we exploit the B-Tree subject index, which is employed in most RDF stores [5,8,22]. We estimate the cardinality as the distance between the leftmost and rightmost matching element in the B-Tree. For patterns of type `?s p o`, the B-Tree object index is used. For patterns of type `!s p ?o`, where `s` is a variable already bound in the query plan, and `o` is a free variable, the cardinality can – assuming uniform distribution of values – be estimated as $card(s, p) = |triples(p)| / |subjects(p)|$, where $triples(p)$ are the triples containing p, and $subjects(p)$ are the distinct subjects occurring in these triples. The same technique is applied to the pattern `?s p !o`. These statistics can be precomputed, and only need to be updated on significant knowledge base changes.

Join Order Optimization Among the set of SPARQL algebra operators, i.e. *select*, *join* and *triple pattern*, query execution costs are dominated by *join* in most cases. This makes join ordering crucial for efficient query processing. For a query containing n join operators, they cannot be exhaustively checked, as the number of possible join orders is factorial. Dynamic programming is thus used to identify a near-optimal join order [26]. Our implementation works as follows: It starts with a pool of (distinct) join sequence candidates, where each one initially contains only one join. Then, the join sequence with the minimum cost is chosen, and the join with smallest cardinality that is not yet in this sequence is added to the pool. The plan generation is complete as soon as the join sequence contains all joins.

To compute the join operator cost, we rely on the fact that RDF stores use the *Index Nested Loop Join* (INLJ) (e.g., [5,9]). Let *INLJ* be a index nested loop join. Let $INLJ.left$ ($INLJ.right$) represent the left (right) child of the join. Then the join's cost can be calculated as $cost(INLJ) = cost(INLJ.left) + (card(INLJ) * cost(INLJ.right))$, where $card(INLJ)$ represents the cardinality estimation of the join. The join cardinality is estimated as $card(join) = card(join.left) * card(join.right)$, where $card(join.left)$ and $card(join.right)$ are obtained using our statistics.

As shown in Section 6.4.1, this approach ensures efficient evaluation of semantic queries.

6 Experimental Evaluation

We implemented the QUICK system using Java. The implementation uses Sesame2 [5] as RDF Store and the inverted index provided by LuceneSail [18] to facilitate semantic query generation. Parts of the described query optimization approaches have been integrated to Sesame2 version 2.2. We have used this implementation to conduct a set of experiments to evaluate the effectiveness and efficiency of the QUICK system and present our results in this section.

6.1 Experiment Setup

Our experiments use two real world datasets. The first one is the Internet Movie Database (IMDB). It contains 5 concepts, 10 properties, more than 10 million instances and 40 million facts. The second dataset (Lyrics, [2]) contains songs and artists, consists of 3 concepts, 6 properties, 200 thousand instances and 750 thousand facts. Although the vocabulary of the datasets is rather small, they still enable us to show the effectiveness of QUICK in constructing domain-specific semantic queries.

To estimate the performance of QUICK in real-world settings, we used a query log of the AOL search engine. We pruned the queries by their visited URLs to obtain 3000 sample keyword queries for IMDB and 3000 sample keyword queries for Lyrics web pages. Most of these queries are rather simple, i.e., only referring to a single concept, such as a movie title or an artist's name, and thus cannot fully reflect the advantages of semantic queries. We therefore manually went through these queries and selected the ones referring to more than two concepts. This yielded 100 queries for IMDB and 75 queries for Lyrics, consisting of 2 to 5 keywords. We assume that every user has had a clear intent of the keyword query, implying that each one can be interpreted as a unique semantic query for the knowledge base. We manually assessed the intent and chose the corresponding semantic query as its interpretation. It turned out that most keyword queries had a very clear semantics. These interpretations served as the ground truth for our evaluation.

The experiments were conducted on a 3.60 GHz Intel Xeon server. Throughout the evaluation, QUICK used less than 1 GB memory.

6.2 Effectiveness of Query Construction

Our first set of experiments is intended to assess the effectiveness of QUICK in constructing semantic queries, that is, how fast a user can turn a keyword query into the corresponding semantic query. At each round of the experiment, we issued a keyword query to QUICK and followed its guidance to construct the corresponding semantic query. We measured the effectiveness using the following two metrics:

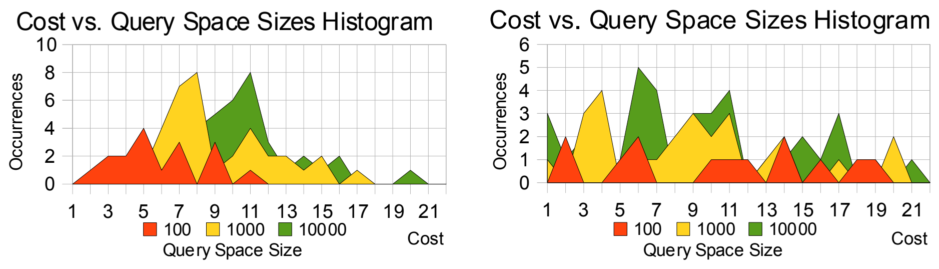


Fig. 5. Query construction cost histograms for IMDB (left) and Lyrics (right) for three different query space sizes

(1) the interaction cost of each query construction process, i.e., the total number of options a user had evaluated to construct each semantic query;
(2) the number of selections a user performed to construct each semantic query, i.e., the number of clicks a user had to make.
The results of the experiments are presented in Table 1 and Figures 5 and 6.

Table 1 shows that the size of the semantic query space grows very fast with the number of terms in a keyword query. Because the datasets are large, a term usually occurs in more than one place of the schema graph. As the size of the query space is usually proportional to the occurrences of each term in the schema graph, it grows exponentially with the number of terms. Furthermore, even for less than five terms, the size of the query space can be up to 9,000 for IMDB and up to 12,000 for Lyrics – such a huge query space makes it difficult for any ranking function to work effectively. For comparison purposes, we applied the SPARK [33] ranking scheme to the semantic queries generated by QUICK, but experiments showed that SPARK could not handle it in a satisfactory manner. In most cases, a user needed to go through hundreds or thousands of queries to obtain the desired one. In contrast, QUICK displays steady performance when confronted with such big query spaces. As shown in Table 1, the maximum number of options a user needs to examine until obtaining the desired semantic query is always low (33 for IMDB rsp. 22 for Lyrics). On average, only 9 rsp. 7 options have to be examined by the user. The cost of the query construction process grows only linearly with the size of the keyword queries. This verifies our expectation that QUICK helps users in reducing the query space exponentially, enabling them to quickly construct the desired query.

Fig. 5 shows the cost distribution of the query construction for different query space sizes. We can see that for most queries, the user only has to inspect between 4 and 11 queries. Only in rare cases more than 20 queries had to be checked. The cost of the query construction process shows a similar trend, growing only logarithmically with the size of the query space. As shown on the left-hand side of Fig. 6, in most cases, only 2 to 5 user interactions were needed. On the right, we show the average position of the selected partial queries. These were almost always among the first 5 presented options.

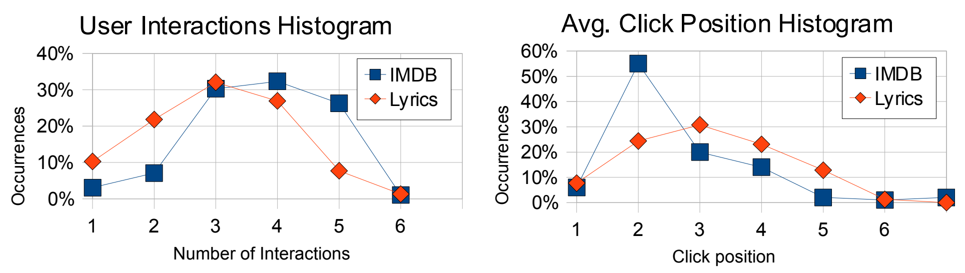


Fig. 6. Histograms of the number of interactions and average click position

To summarize, this set of experiments shows that QUICK effectively helps users to construct semantic queries. In particular, the query construction process enables users to reduce the semantic query space exponentially. This indicates the potential of QUICK in handling large-scale knowledge bases.

6.3 Efficiency

To demonstrate that QUICK is feasible for real world applications, we conducted experiments to measure the computation time for generating query guides and for retrieving the final intended semantic queries.

6.3.1 Query Guide Generation

We recorded the response times for each interaction of the query construction process. The initialization time as shown in the 2nd and 3rd column of Table 2 comprises (1) the creation of the semantic query space, (2) computation of subquery relationships, and (3) keyword lookup in the full-text index. These tasks are fully dependant on the RDF store. The response time to user interactions were very short (4th and 5th column), enabling a smooth construction process.

In implementing QUICK, we focused on efficient algorithms for query guide

No. of	Avg. Query	Cost	
Terms	Space Size	Avg	Max
2	60.5	5.08	7
3	741	8.69	15
4	7,365	10.17	32
> 4	9,449	14.88	33
all	4,535	9.46	33

No. of	Avg. Query	Cost	
Terms	Space Size	Avg	Max
2	23.9	3.21	7
3	235	7.96	13
4	1,882	12.30	19
> 4	12,896	14.44	22
all	2,649	7.50	22

Table 1
Effectiveness of QUICK for IMDB (left) and Lyrics (right)

generation and query evaluation. We are confident that for the initialization tasks the performance can be improved significantly, too, e.g., by adapting techniques from [16] and by introducing special indexes.

6.4 Quality of the Greedy Approach

To evaluate the quality of the query guides generated by the greedy algorithm discussed in Section 4.2, we compared it against the straightforward algorithm of Section 4.1. As the latter is too expensive to be applied to a real dataset, we restricted the experiments to simulation. We generated artificial semantic queries, partial queries and sub-query relationships. The semantic queries subsumed by each partial query were randomly picked, while the number of these was fixed. Therefore, three parameters are tunable when generating the queries, *n_complete* – the number of semantic queries, *n_partial* – the number of partial queries, and *coverage* – the number of semantic queries subsumed by each partial query.

In the first set of experiments, we fixed *n_complete* to 128 and *coverage* to 48, and varied *n_partial* between 4 and 64. We run both algorithms on the generated queries, and recorded the cost of the resulting query guides and their computation time. To achieve consistent results, we repeatedly executed the simulation and calculated the average.

As shown in the left-hand side of Fig. 7, the chance to generate cheaper query guides increases with the number of partial queries. Guides generated by the greedy algorithm are only slightly worse (by around 10%) than those generated by the straightforward algorithm, independent of the number of partial queries. The computation time of the straightforward algorithm increases exponentially with the number of partial queries, while that of the greedy algorithm remains almost linear. This is consistent with the complexity analysis of Section 4.

No. of	Init time (ms)		Response time (ms)	
terms	IMDB	Lyrics	IMDB	Lyrics
2	98	664	2	0.5
3	993	384	19	4
4	16,797	4,313	1,035	107
> 4	31,838	120,780	3,290	7,895
all	3,659	17,277	314	1,099

Table 2
QUICK Response Time for IMDB (left) and Lyrics (right)

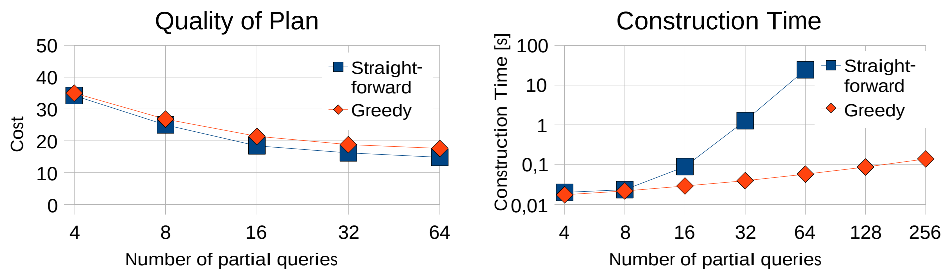


Fig. 7. Varying number of partial queries

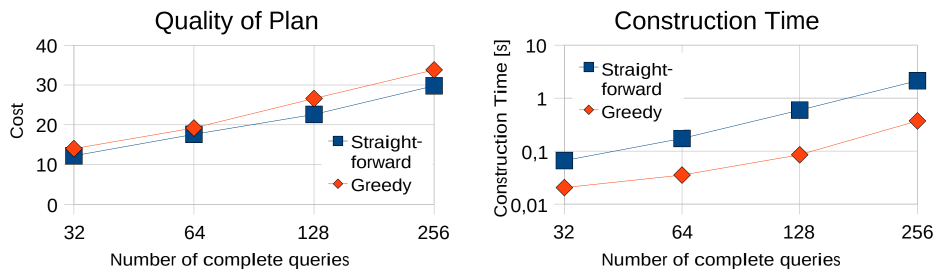


Fig. 8. Varying number of semantic queries

In the second set of experiments, we varied *n_complete* between 32 and 256, fixed *n_partial* to 32, and set *coverage* to $^1/_4$ of *n_complete*. The results are shown in Fig. 8.

As expected, as the number of semantic queries increases exponentially, the cost of query construction increases only linearly. The guides generated by the greedy algorithm are still only slightly worse (by around 10%) than those generated by the straightforward algorithm. This difference does not change significantly with the number of semantic queries. The performance conforms to the complexity analysis of Section 4.

In the third set of experiments, we fixed *n_complete* to 64 and *n_partial* to 16, and varied *coverage* between 8 and 48.

Fig. 9 shows that as the *coverage* increases, the cost of resulting query guides first decreases and then increases again. This confirms that partial queries with an intermediate coverage are more suitable for creating query construction graphs, as they tend to minimize the fan-out and the cost of the most expensive sub-graph simultaneously. The difference between the greedy algorithm and the straightforward algorithm increases with the *coverage*. This indicates that the greedy algorithm has a non-constant performance with respect to *coverage*, which was to be expected, as result of the logarithmic performance rate of MINSETCOVER and the assumptions of Definition 13. Fortunately, in real data, most partial queries have a relatively small coverage (less than 20%), where this effect is less noticeable, justifying our assumptions.

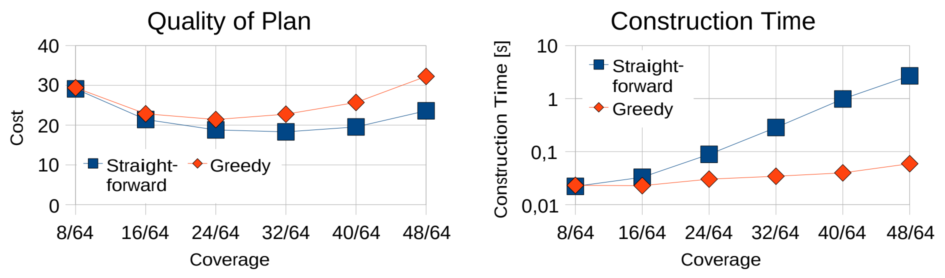


Fig. 9. Varying coverage of partial queries

In summary the experiments showed the greedy algorithm to have the desired properties. In comparison to the straightforward algorithm, the generated guides are just slightly more costly for the user, but are generated much faster, thereby demonstrating the applicability of the QUICK approach.

6.4.1 Query Evaluation

To assess the impact of the query evaluation techniques discussed in Section 5, we randomly selected 14 keyword queries from the IMDB query set. Using QUICK, we generated the semantic query space for these queries with a maximum size of 7. The 5305 semantic queries which returned more than 100 results were used to assess the query evaluation performance.

We computed the improvement factor in terms of query acceleration. Fig. 10 shows the distribution of this factor. In comparison with the un-optimized RDF store, some queries did not run faster or even took slightly longer. However, the majority of the queries improved significantly (factor 4 to 40), demonstrating the benefits of our join order optimizations.

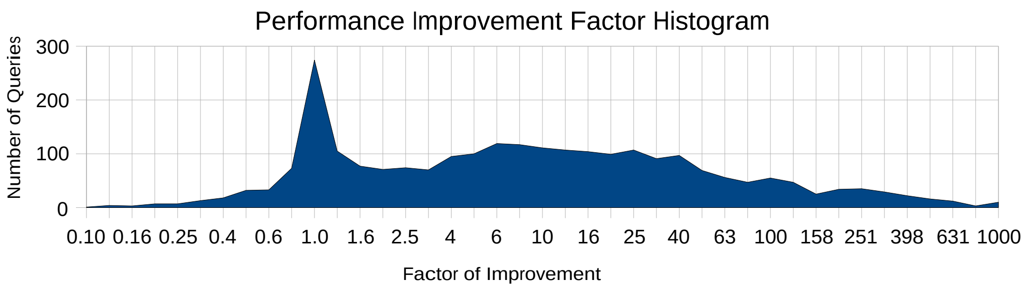


Fig. 10. Histogram of performance improvements

7 Related Work

In recent years, a number of user interfaces have been proposed to facilitate construction of semantic queries. These interfaces can be mainly classified into visual graphic interfaces [20,25] and natural language interfaces [13,4,17]. Natural language interfaces are potentially more convenient for end users [12], as they require little prior knowledge of the data schema or a query language. However, the state-of-the-art natural language interfaces still require users to use a terminology that is compatible with the data schema and form grammatically well-formed sentences [12]. To overcome this limitation, a number of techniques have been used to help users construct valid natural language queries. For instance, the interface of [13] provides a dialog for users to align natural language terms to the ontology. The interface of [4] utilizes auto-completion to guide users to create meaningful sentences. In comparison with these methods, QUICK offers more flexibility in issuing queries. It accepts arbitrary keyword queries and allows users to incrementally structure the query through an optimized process.

Extensive investigations have recently been conducted on how to perform keyword search on the Semantic Web [15,31–33] and databases [1,30]. The majority of work treats each data set as a graph, and regards the query results as sub-graphs, e.g. Steiner trees [14], which contain the query terms. They have in common that relevance ranking of these graphs is performed. As identifying the top-k query results would incur prohibitive I/O and computational cost, most of the approaches have been focusing on improving the performance of query processing. In contrast, QUICK is designed to help users to construct semantic queries. As the query results are retrieved after the semantic query is constructed, query processing is not a critical issue for QUICK. Work that is closest to QUICK includes [31,33,30]. Instead of retrieving search results directly, these approaches attempt to translate a keyword query into a set of semantic queries that are most likely to be intended by users. However, these approaches can at best rank the most popular query intentions on top. As the number of semantic query interpretations is very high, it is impossible to take the intentions of all users into consideration. As QUICK allows users to clarify the intent of their keyword queries, it can satisfy diverse user information needs to a much higher degree.

Automatic query completion [3,21,7] and refinement [28] are techniques that help users forming appropriate structured queries by suggesting possible structures, terms or refinement options based on the partial queries the user has already entered. By using these suggestions, the user constructs correct database queries without completely knowing the schema. However, as this technique still requires users to form complete structured queries, it is less flexible than QUICK, which allows users to start with arbitrary keyword queries. Moreover, they do not tackle the issue of minimizing the users' interaction effort.

A main advantage of QUICK is its ability to allow users to clarify the intent of their keyword queries through a sequence of steps. In the area of Information Retrieval, document clustering and classification have been used for similar purposes, especially in connection with faceted search interfaces. For example [6] classifies document sets based on pre-defined categories and lets users disambiguate their search intent by selecting the most preferred categories. However, Information Retrieval approaches do not work on complex structured data, because joins over different concepts are not supported.

8 Conclusion

In this paper, we introduced QUICK, a system for guiding users in constructing semantic queries from keywords. QUICK allows users to query semantic data without any prior knowledge of its ontology. A user starts with an arbitrary keyword query and incrementally transforms it into the intended semantic query. In this way, QUICK integrates the ease of use of keyword search with the expressiveness of semantic queries.

The presented algorithms optimize this process such that the user can construct the intended query with near-minimal interactions. The greedy version of the algorithm exhibits a polynomial runtime behavior, ensuring its applicability on large-scale real-world scenarios. To our knowledge, QUICK is the first approach that allows users to incrementally express the intent of their keyword query. We presented the design of the complete QUICK system and demonstrated its effectiveness and practicality through an extensive experimental evaluation.

As shown in our study, QUICK can be further improved and extended in the following directions. (i) While QUICK currently works well on focused domain schemas, large ontologies pose additional challenges with respect to usability as well as efficiency. To improve usability, we plan to make use of concept hierarchies to aggregate query construction options. In that way, we keep their number manageable and prevent the user from being overwhelmed by overly detailed options. To improve further on efficiency, we are working on an algorithm that streamlines the generation of the semantic query space. (ii) A further field of improvement is making the user interface more intuitive, especially for users without expertise in semantic web or databases. (iii) Based on the improved user interface, we will conduct a user study to verify its suitability for non-expert users and its effectiveness on a larger scale.

References

[1] S. Agrawal, S. Chaudhuri, G. Das, DBXplorer: a system for keyword-based search over relational databases, in: ICDE, 2002.

[2] F. L. andClement T. Yu andWeiyi Meng andAbdur Chowdhury, Effective keyword search in relational databases, in: Proceedings of the ACM SIGMOD International Conference onManagement of Data, Chicago, Illinois, USA, 2006.

[3] H. Bast, I. Weber, The CompleteSearch engine: Interactive, efficient, and towards IR& DB integration, in: CIDR, 2007.

[4] A. Bernstein, E. Kaufmann, Gino - a guided input natural language ontology editor, in: ISWC, 2006.

[5] J. Broekstra, A. Kampman, F. van Harmelen, Sesame: A generic architecture for storing and querying RDF and RDF Schema, in: ISWC, 2002.

[6] V. Broughton, L. Heather, Classification schemes revisited: Applications to web indexing and searching, Journal of Internet Cataloging 2 (3/4) (2000) 143–155.

[7] H. Haller, QuiKey – the smart semantic commandline (a concept), in: Poster and extended abstract presented at ESWC2008, 2008.

[8] A. Harth, S. Decker, Optimized index structures for querying RDF from the Web, in: Proceedings of the 3rd Latin American Web Congress, 2005.

[9] A. Harth, J. Umbrich, A. Hogan, S. Decker, YARS2: A federated repository for querying graph structured data from the Web, in: ISWC/ASWC, 2007.

[10] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, C. Yu, Making database systems usable, in: SIGMOD, 2007.

[11] R. M. Karp, Reducibility among combinatorial problems, in: R. E. Miller, J. W. Thatcher (eds.), Complexity of Computer Computations, Plenum Press, 1972.

[12] E. Kaufmann, A. Bernstein, How useful are natural language interfaces to the semantic web for casual end-users, in: ISWC/ASWC, 2007.

[13] E. Kaufmann, A. Bernstein, R. Zumstein, Querix: A natural language interface to query ontologies based on clarification dialogs, in: ISWC, 2006.

[14] B. Kimelfeld, Y. Sagiv, Finding and approximating top-k answers in keyword proximity search, in: PODS, 2006.

[15] Y. Lei, V. S. Uren, E. Motta, Semsearch: A search engine for the Semantic Web, in: EKAW, 2006.

[16] G. Li, B. C. Ooi, J. Feng, J. Wang, L. Zhou, Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data, in: SIGMOD Conference, 2008.

[17] V. Lopez, V. Uren, E. Motta, M. Pasin, Aqualog: An ontology-driven question answering system for organizational semantic intranets, Journal of Web Semantics 5 (2) (2007) 72–105.

[18] E. Minack, L. Sauermann, G. Grimnes, C. Fluit, J. Broekstra, The Sesame LuceneSail: RDF queries with full-text search, Tech. Rep. 2008-1, NEPOMUK (Feb 2008).

[19] E. Minack, W. Siberski, W. Nejdl, Benchmarking Fulltext Search Performance of RDF Stores, in: Proceedings of the 6th European Semantic Web Conference (ESWC 2009), Heraklion, Greece, 2009.

[20] K. Möller, O. Ambrus, L. Josan, S. Handschuh, A visual interface for building SPARQL queries in Konduit, in: International Semantic Web Conference (Posters & Demos), 2008.

[21] A. Nandi, H. V. Jagadish, Assisted querying using instant-response interfaces, in: SIGMOD, 2007.

[22] T. Neumann, G. Weikum, RDF-3X: a RISC-style engine for RDF, Proceedings of the VLDB Endowment 1 (1) (2008) 647–659.

[23] M. Reichert, S. Linckels, C. Meinel, T. Engel, Student's perception of a semantic search engine, in: IADIS CELDA, Porto, Portugal, 2005.

[24] E. Ruckhaus, M.-E. Vidal, E. Ruiz, OnEQL: An ontology efficient query language engine for the semantic web, in: ALPSWS, 2007.

[25] A. Russell, P. R. Smart, NITELIGHT: A graphical editor for SPARQL queries, in: International Semantic Web Conference (Posters & Demos), 2008.

[26] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access path selection in a relational database management system, in: SIGMOD, 1979.

[27] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, D. Reynolds, Sparql basic graph pattern optimization using selectivity estimation, in: Proceedings of the 17th International Conference on World Wide Web, Beijing, China, 2008.

[28] N. Stojanovic, L. Stojanovic, A logic-based approach for query refinement in ontology-based information retrieval systems, Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on (2004) 450–457.

[29] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, J. Broekstra, Index structures and algorithms for querying distributed RDF repositories, in: WWW, 2004.

[30] S. Tata, G. M. Lohman, SQAK: doing more with keywords, in: SIGMOD, 2008.

[31] T. Tran, P. Cimiano, S. Rudolph, R. Studer, Ontology-based interpretation of keywords for semantic search, in: ISWC, 2007.

[32] H. Wang, K. Zhang, Q. Liu, T. Tran, Y. Yu, Q2semantic: A lightweight keyword interface to semantic search, in: ESWC, 2008.

[33] Q. Zhou, C. Wang, M. Xiong, H. Wang, Y. Yu, SPARK: Adapting keyword query to semantic search, in: ISWC, 2007.