Leibniz Universität Hannover

Fakultät für Elektrotechnik und Informatik

User Accessibility of Web Data

Der Fakultät für Elektrotechnik und Informatik der Gottfried Wilhelm Leibniz Universität Hannover zur Erlangung des akademischen Grades Doktor-Ingenieur (abgekürzt: Dr.-Ing.) genehmigte Dissertation

von M.Sc. Gideon Zenz,

geboren am 03.05.1979 in Neunkirchen/Saar

2013

Erstprüfer:	Prof. Dr. Wolfgang Nejdl
Zweitprüfer:	Prof. Dr.-Ing. Bernardo Wagner
Vorsitzender:	Prof. Dr.-Ing. Christian Müller-Schloer
Betreuer:	Dr. Wolf Siberski
Tag der Promotion:	04.11.2013

Kurzfassung

Im Zeitalter des Web 2.0 werden immer mehr Daten produziert, gemanagt, sowie öffentlich zugänglich gemacht. Die größte Menge dieser Daten befindet sich im so genannten „hidden web“, d.h. es ist in strukturierten Datensenken gespeichert und hauptsächlich über Web-Formulare oder ähnliche Möglichkeiten zugänglich. Das herausragende Problem für Endbenutzer bei solch riesigen Datenmengen ist das Auffinden relevanter Informationen, die sein Informationsbedürfnis befriedigen. Derzeit übliche Stichwortsuchverfahren arbeiten auf Dokumenten sehr effektiv, weniger jedoch auf strukturierten Daten. Unserer Erfahrung nach wird insgesamt weiterhin häufig die Wichtigkeit der Nutzerinteraktion unterschätzt, sowie der Komplexität und dem Umfang der zugrundeliegenden Daten nicht ausreichend Rechnung getragen. Diese Arbeit stellt, unter besonderer Beachtung der Benutzeranforderungen an den gesamten Suchprozess, einige Ansätze für den einfachen Zugriff auf strukturierte Web-Daten vor, die u.a. auch der Notwendigkeit, den Benutzer mit notwendigen Domänenwissen über die zugrundeliegenden Daten versorgen, Rechnung tragen.

Stichworte

H.2.4.h [Database Management] Query processing

H.3.3.e [Information Storage and Retrieval] Query formulation

H.5.2 [Information Interfaces and Presentation]: User Interfaces

Abstract

In the age of Web 2.0, ever more data is produced, managed, and made publicly accessible. Most of this data is in the so-called hidden web, which essentially implies it is stored in structured data stores and accessible mostly through forms or other means. While already the sheer amount of information seems to necessitate new innovations, and feeds the hope for an age of information and knowledge, finding relevant information satisfying the information need becomes increasingly difficult. While current approaches solve standard keyword search on documents quite effectively, they lack on structured data. Furthermore, we believe that the importance of user interaction is often either neglect, or the complexity and scale of the underlying data. With a strong focus on users' requirements of the whole search process, we present approaches for easy access to structured web data, as well as providing the user with domain knowledge needed to successfully understand the underlying data.

Keywords

H.2.4.h [Database Management] Query processing

H.3.3.e [Information Storage and Retrieval] Query formulation

H.5.2 [Information Interfaces and Presentation]: User Interfaces

An den Mistral

Mistral-Wind, du Wolken-Jäger,
Trübsal-Mörder, Himmels-Feger,
Brausender, wie lieb ich dich!
Sind wir zwei nicht Eines Schoßes
Erstlingsgabe, Eines Loses
Vorbestimmte ewiglich?

Hier auf glatten Felsenwegen
Lauf ich tanzend dir entgegen,
Tanzend, wie du pfeifst und singst:
Der du ohne Schiff und Ruder
Als der Freiheit freister Bruder
Über wilde Meere springst.

Kaum erwacht, hört ich dein Rufen,
Stürmte zu den Felsenstufen,
Hin zur gelben Wand am Meer.
Heil! da kamst du schon gleich hellen
Diamantnen Stromesschnellen
Sieghaft von den Bergen her.

Auf den ebnen Himmels-Tennen
Sah ich deine Rosse rennen,
Sah den Wagen, der dich trägt,
Sah die Hand dir selber zücken,
Wenn sie auf der Rosse Rücken
Blitzesgleich die Geißel schlägt, -

Sah dich aus dem Wagen springen,
Schneller dich hinabzuschwingen,
Sah dich wie zum Pfeil verkürzt
Senkrecht in die Tiefe stoßen, -
Wie ein Goldstrahl durch die Rosen
Erster Morgenröten stürzt.

Tanze nun auf tausend Rücken,
Wellen-Rücken, Wellen-Tücken -
Heil, wer neue Tänze schafft!
Tanzen wir in tausend Weisen.
Frei - sei unsre Kunst geheißen,
Fröhlich - unsre Wissenschaft!

Raffen wir von jeder Blume
Eine Blüte uns zum Ruhme
Und zwei Blätter noch zum Kranz!
Tanzen wir gleich Troubadouren
Zwischen Heiligen und Huren,
Zwischen Gott und Welt den Tanz!

Wer nicht tanzen kann mit Winden,
Wer sich wickeln muß mit Binden,
Angebunden, Krüppel-Greis,
Wer da gleicht den Heuchel-Hänsen,
Ehren-Tölpeln, Tugend-Gänsen,
Fort aus unsrem Paradeis!

Wirbeln wir den Staub der Straßen
Allen Kranken in die Nasen,
Scheuchen wir die Kranken-Brut!
Lösen wir die ganze Küste
Von dem Odem dürrer Brüste,
Von den Augen ohne Mut!

Jagen wir die Himmels-Trüber,
Welten-Schwärzer, Wolken-Schieber,
Hellen wir das Himmelreich!
Brausen wir ... o aller freien
Geister Geist, mit dir zu zweien
Braust mein Glück dem Sturme gleich. -

- Und daß ewig das Gedächtnis
Solchen Glücks, nimm sein Vermächtnis,
Nimm den Kranz hier mit hinauf!
Wirf ihn höher, ferner, weiter,
Stürm empor die Himmelsleiter,
Häng ihn - an den Sternen auf!

(Friedrich Nietzsche, An den Mistral, Ein Tanzlied. Aus: Die fröhliche Wissenschaft. („la gaya scienza“) Anhang: Lieder des Prinzen Vogelfrei, S. 348-350, Leipzig, 1887)

Danksagungen

Wolfgang Nejdl

Für die Möglichkeit, an diesem herausragenden Institut arbeiten zu können

Wolf Siberski

Für die hervorragende Betreuung und viele geistreiche Stunden

Thomas Risse

Für die stimulierende Zusammenarbeit an nicht immer einfachen Projekten

Allen, mit denen ich gearbeitet, geforscht, gelitten habe

Xuan Zhou, Daniel Wichert, Enrico Minack, Nina Tahmasebi, Kaweh Djafari Naini, Sergiu Chelaru, Ismail Sengor Altingövde, Berkant Barla Cambazoglu, Ingmar Weber, Przemyslaw Grabowicz, Luca Maria Aiello, Tu Ngoc Nguyen, Uwe Thaden

Unseren unersetzlichen Technikern und Sekretärinnen

Dimitar Mitev, Marco Schneider, Olaf Jansen-Olliges, Jan-Hendrik Zab, Sebastian Gerecke, Susanne Elsner, Michaela Kleiner, Iris Zieseniß

Meinen ewigen Mitstreitern

Fabian Cholewa und Sebastian Schildt

Nicht zuletzt meinen Eltern, für ihre immerwährende Unterstützung, sowie meinem Bruder Ivo und seiner Frau Julia.

Und zum Schluß: Ihnen, dem unverhofften Leser!

Contents

List of Figures

List of Tables

List of Definitions

List of Algorithms

Other Lists

1. User Intent on Retrieving Data

Finding relevant information is much more than just devising a keyword query and evaluating the retrieved information. This chapter introduces the cognitive process of information retrieval as a circular task, starting from the problem identification, via the articulation phase to the actual query formation and reformulation, followed by result evaluation, which can lead again to reformulating the problem, depending on the results. We discuss how this process can best be modelled to help a user in finding relevant information.

In the age of Web 2.0, ever more data is produced, managed, and made publicly accessible. Most of this data is in the so-called hidden web, which essentially implies it is stored in structured data stores and accessible mostly through forms or other means. This hidden part of the web is estimated to be 400 – 550 times larger than the plainly visible web [1], [2]. While already the sheer amount of information seems to necessitate new innovations, and feeds the hope for an age of information and knowledge, finding relevant information satisfying the information need becomes increasingly difficult. For the typical user, it often feels like having to find the proverbial needle in a haystack.

Keyword search is currently the state-of-the-art solution for easy information access, as keyword based search engines like Google impressively display. These technologies are prefect for document-based retrieval, which covers a huge part of the available information. While classical IR technologies work very well for text collections or web pages, they are not easily applicable to structured data. This is an important issue for accessing the web, as mentioned about 80% of the web are dynamically generated – and thus typically stored in a structured way in a RDBMS or Knowledge Base. Having a way to access these data in an as easy fashion as by keyword search is important, as only by this accessing this data will be feasible by the common end user, as "*Thirty years of research on query languages can be summarized by […] end users will not learn SQL;*" [3].

To make IR technologies applicable to structured data, the current state-of-the-art way is to interpret tuples as partial documents. A full document is formed by a series of joining tuples, which connect all query terms in a compact manner.

The challenge now is identifying the right series of tuples, as the space of possible combination of the tuples grows exponentially [4].

The usual approaches try to rank this result space, but don’t allow the user to control the actual search process. This is even more important with searches on structured data compared with search on documents, as structured data allows for many different semantic interpretations of a query; e.g. when searching in a movie database, the user typically knows which keyword is an actor name, and which should be contained in the movie’s title. If this semantic information is not controllable by the user, a search process can easily become frustrating, as there is no direct way in removing results that obviously don’t match the intended semantic meaning of the user’s keyword query.

To achieve an understanding of the needs of end users, this chapter will first introduce results from cognitive psychology [5], [6] about the search process in general, and then discuss how existing work supports users in this process.

We follow the model as presented in [5], which details four phases of the search process, as Fig. 1 shows.

The *problem identification phase* serves the identification of the information need. In this phase, the actual goal of the search is specified. Now, in the *need articulation phase*, the identified need is formulated on a conceptual level, and then (para-)phrased in a step-by-step manner. The concepts needed for this step can either originate from the user’s long-term memory, or from external sources. During the *query formulation phase*, these phrases, which still are in natural language, are now cast into formal queries as understood by the IR system. This implies e.g. choosing suitable query terms, and system specific search operators for controlling the retrieval process, depending on the IR system.

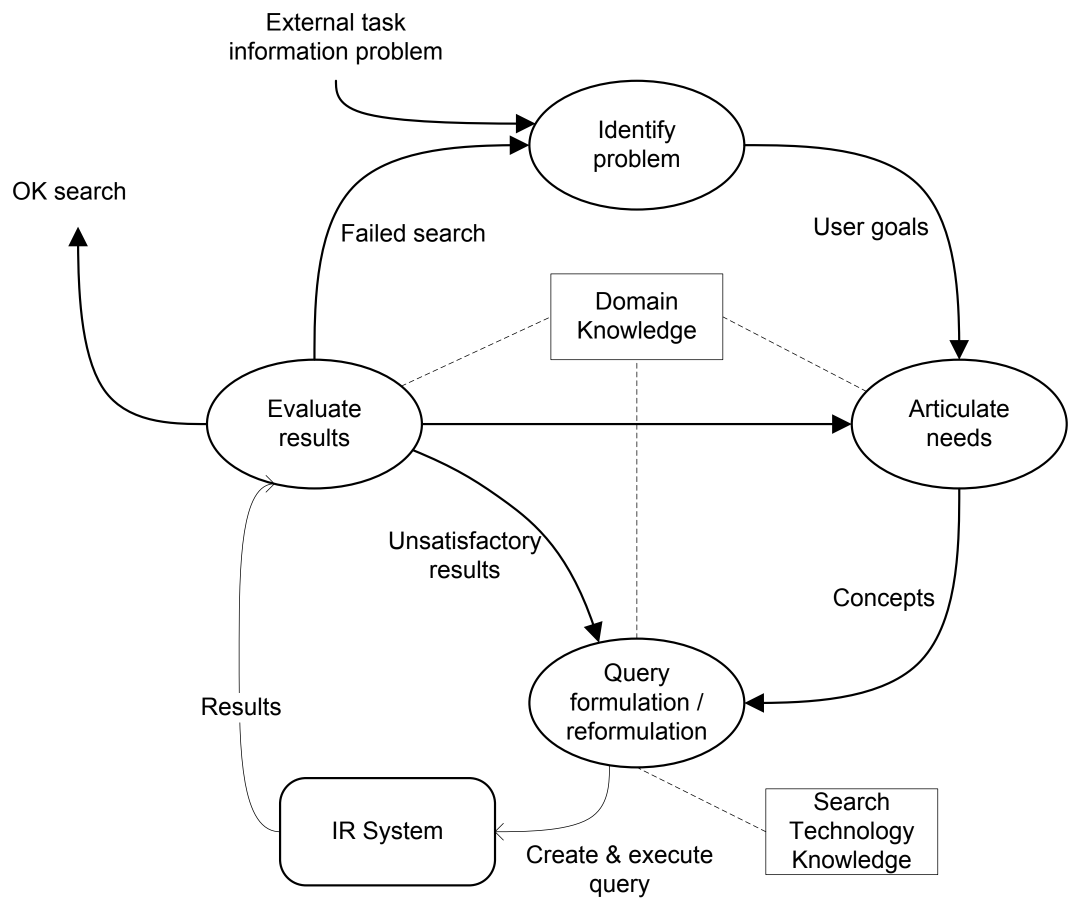


Fig. 1: Process model of information searching activities [5]

The *result evaluation phase* now follows after executing the search. Here, the user decides whether the results are satisfactory, or if the search needs to be continued. For evaluation, several approaches are feasible; foremost, *scanning* the results allows to quickly discriminating possibly interesting results from obviously negligible ones. Then, prospectus results are analysed more deeply by *sampling*. Finally, the most promising results can be *inspected* thoroughly.

If the results contain the needed information, the search process is successfully terminated. If not, the user needs to go back to the *need articulation phase,* or even to earlier phases, to concretize the search intent in a machine understandable way.

The underlying complexity of the whole query process makes it necessary for a good IR system to not only focus only on the technical execution of queries, but also on supporting the other phases of the process. To achieve this, methods specifically assisting query refinement and reformulation have shown to be helpful [7–9].

As data in structured form does at most only partially consist of text in natural language, it requires the user to bring forth a significantly higher cognitive effort for this than in classical text search. Specifically for a classical RDBMS, not only knowledge of the underlying query language, but also of the underlying schema of the data source is required. For the less experienced users, already this is an insurmountable obstacle. As specifically this group of users is currently growing [10–12], easy-to-use systems are increasingly indispensable in most contexts. This led in the recent years increasingly to research proposing methods allowing to intuitively formulating queries.

1.1 Keyword Search in Databases

The request for easy accessibility of databases is nearly as old as relational databases themselves. An early attempt to alleviating this issue is Universal Relation [13]. The relational model improved on previous technology as it removed the need to know physical paths, i.e. the file system structure. However, the relational model still needs logical navigation, as access paths and relations have to be exactly specified. Universal Relation allows a user to specify query without having to specify the concrete access path, while the system tries to figure out the intended path itself. Still, the user has to know the attribute names, and the schema restricts joins.

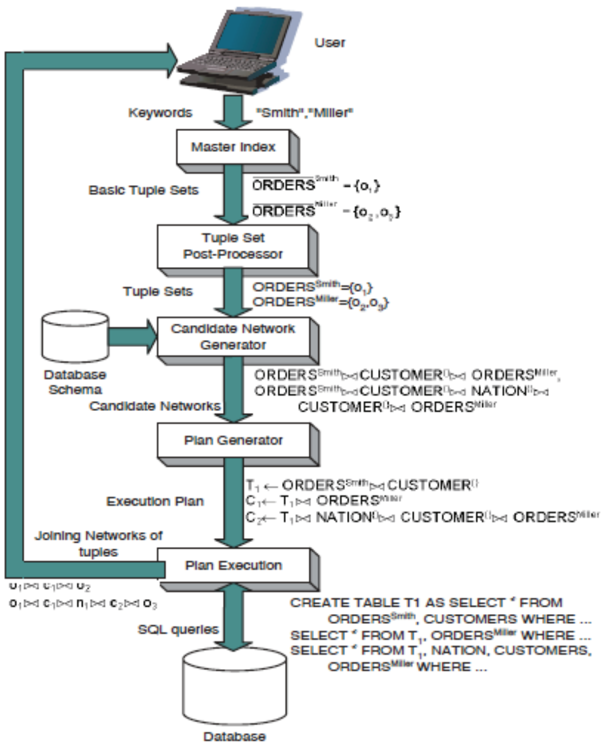

User
Keywords
"Smith","Miller"
Master Index
Basic Tuple Sets
Tuple Set Post-Processor
Tuple Sets
Database Schema
Candidate Network Generator
Candidate Networks
Plan Generator
Execution Plan
Joining Networks of tuples
Plan Execution
SQL queries
Database
CREATE TABLE T1 AS SELECT * FROM ORDERS^Smith, CUSTOMERS WHERE ...
SELECT * FROM T1, ORDERS^Miller WHERE ...
SELECT * FROM T1, NATION, CUSTOMERS, ORDERS^Miller WHERE ...

Fig. 2: Architecture of Discover [11]

Later, several approaches to keyword search in databases emerged [11], [12], which do not necessitate the user to know the schema. All were driven by the idea to bring the easiness of keyword search from the web to databases. The idea is that a user only has to specify keywords, while the system tries to guess the intended join path (in this context called *tuple tree*) and the intended attribute for each keyword. This approach is more complex than Universal Relation, as here the schema may be walked through several times and the system has to also guess the intended attributes. As there are usually several tuple trees matching the query terms, these systems usually employ ranking to deal with this ambiguity. These algorithms have been continuously developed in the last years [9], [14–16].

An inherently different approach is proposed by [17], which first transforms the data into a graph, and by this reduces the problem of keyword search to finding the Minimum Steiner Tree. Continuations of this idea are [17–19], who develop a coherent framework for keyword search in structured data. Further approaches are introduced in [8], [20], [21].

By this approaches, the *need articulation phase* is made considerably easier, and allows an intuitive access. Unfortunately, the complete query process as discussed above is only partly supported. E.g., the user cannot specify the interpretation of the given query terms, which would help ranking tremendously. A reformulation of the query intention is not supported, if the algorithm in use does not recognize the intention of the query, as only the terms of a query can be changed.

Therefore, the user has to rely on the ranking heuristics identifying the preferred intention in order to find the needed information fast. As we observed, this is often not the case.

1.2 Form Based Keyword Search

The most common method for accessing structured data is by employing forms. Here, the system architect specifies the query intentions during the system's design phase, and accordingly designs query forms. Commonly, these forms are static and usually only allow to switch to an advanced version if a more refined query specification is needed. Forms therefore are a very easy and intuitive methodology for accessing structured data, but for the price of a very narrowly specified expressivity. A system designed in such a way prevents the ordinary user from issuing queries of an intent not previously foreseen.

A noteworthy exception is [22], which allows the user during the *query formulation phase* to dynamically extend and modify the query, in order to match it more closely the information need. A continuation of this approach defines in [23] a ranking on automatically generated forms, based on the data's schema. This allows for choosing the most useful search attributes from a given data source. The limitation here is that only queries referring to one table are supported, joins can therefore not be expressed.

Other, similar systems allow for a more dynamic form composition [24–26], but struggle to hide the inherent complexity in an intuitive user interface. The expressed target audiences for these tools are developers, for who constructing queries in comparison to using formal query languages is made comparably easier.

1.3 Graphical Query Specification

Visual query tools are another possibility to formulate complex queries, as discussed by [27]. Usually, a graphical notation of the chosen query language is defined, and a user interface is proposed, which guides in constructing a syntactical correct query [28–30]. Such tools are also commercially available, but the requirements for using these are steep. For satisfactory results, the user needs to have a general understanding of the underlying query language, and the *query formulation process* still requires a high degree of abstraction from the semantic view on the information need, which is the starting point for a normal search process. These techniques therefore require less knowledge of the employed search technology in the *query formulation phase*. Still, understanding the data sources' schema remains the more difficult part of this phase [30].

1.4 Faceted Search

Faceted search is an approach that supports the complete query process [31], [32]. The main idea is to create facets out of the available document properties by defining a set of orthogonal categories. By selecting values or ranges of values, the search space is gradually limited. Additionally, query terms can be entered to refine the search. An evolved representative of this approach is mSpace [33], which represents facets by ordered columns, allowing for incremental refinement of each column.

Although this concept is confusing at first, it reveals its power after a modest settling period. A long term study by [34] revealed, that users in the beginning prefer the keyword search capabilities of the system, but quickly adopt to the faceted system, and start to preferably use it.

Means to dynamically extract facets is proposed by [35], which effectively supports search in big collections of pictures. An algorithm for automated extraction and evaluation of facets from structured data is introduced by [36].

Facetedpedia [37] automatically extracts facets from Wikipedia, in order to support search and exploration in such text corpora. Further developments are described by [38], [39].

In general, the main advantage of faceted search is supporting the complete search process. The result set is already previewed during the *need articulation phase*, and conditions can be dynamically added to and removed from facets. A drawback shared with form-based systems is the limitation to flat data, i.e. joins are not supported.

A specifically interesting approach here is Feldspar [40], which allows for a so-called associative search. The data is represented as a graph, and the user interface allows to specifying associative connections, e.g. “where's the webpage related to Email related to Person related to Event related to Date xy”. Such paths can be arbitrarily long, and go far beyond the expressivity allowed by classical faceted search, as they allow specifying unambiguous queries by this approach.

The limitation here is that only paths, not graphs can be specified. Therefore, only a small part of the query space can be reached.

1.5 Visualising Result Sets

Queries on structured data often return a result set so big that returning the complete set to the user is not feasible. In order to allow users to get an overview of the result, methods like clustering or hierarchisation are of help. These aggregate sets sharing similar attributes and visualise them by only one attribute. The success of these methods depends strongly on the choice of this aggregated attribute, as a user has to recognize it as being relevant for the query [41] issued. Here, multimedia previews are employed when the user hovers the mouse above a facet. A user study showed this to significantly help users in their search.

Another opportunity to help the interactivity of a user interface is employing animations [34], [42]. A further effective means are soft transitions, e.g. by using zooming. An interesting approach is presented by WaveLens [43], which allows to zoom into particular results. By doing so, the preview originally shown is extended. An in-depth overview of this field is e.g. given in [44].

1.6 Open Issues

The discussed groups of approaches exhibit several different advantages and disadvantages; tools for graphically specifying classical database queries allow for the full expressivity, but still require expert knowledge with regard to the formal query language and schema of the data source employed. As of this, the usability for typical users is limited.

Faceted search does offer sufficient support of the query process, but at the expense of expressivity of the queries. Relations and joins are either not supported, or in a very restricted way. Understandably, work here focuses first and foremost on the user-interface, and leave aspects like scalability to other lines of research.

Ranking based approaches on the other hand support complex schemas without requiring expert knowledge by the user. The drawback here is that the query process as a whole is not considered; if the intended information is not among the highest-ranking results, the user is left with no options of effectively refining the query, i.e. the semantic of queries is systematically ignored.

1.7 Outline

In the following chapters we will discuss several approaches, which aim to tackle the issue of supporting the user during the search process[1], as outlined in Fig. 1. In Chapter 2, we present an approach for keyword search based ranking in relational database systems and RDF stores, called SUITS. This approach supports mainly the *need articulation phase*, but also partially the *query formulation phase*. In Chapter 4, we discuss QUICK, which aims to fully support all possibly user intents with a keyword query on RDF stores. Accordingly, it fully supports the *need articulation phase,* as well as the *query formulation phase*, and finally also the *result evaluation phase*. Although this approach offers the best support for the query process, it does not offer ranking support, which would make frequent queries considerably easier to construct, and also makes no use of domain knowledge. Chapter 5 we present a ranking based approach for facetted search on relational database systems, which makes use of external domain knowledge. This system supports the *result evaluation phase*, as well as the *query formulation phase*, but offers only partial support for the *need articulation phase.* In Chapter 6, we introduce a system aimed for mobile use, that automatically generates the needed domain knowledge, and fully supports the *need articulation phase* and the *query formulation phase.* Finally, in Chapter 7, we conclude this thesis.

[1] No technical system can (currently) directly support the *problem identification phase,* because current systems need the user to articulate (i.e. cast in words) the problem and enter (i.e. using a keyboard) these words in order to help finding information.

2. Publications

1. Enrico Minack, Wolf Siberski, Gideon Zenz, Xuan Zhou, SUITS4RDF: Incremental Query Construction for the Semantic Web, International Semantic Web Conference (Posters & Demos) 2008
2. Xuan Zhou, Gideon Zenz, Elena Demidova, Wolfgang Nejdl, SUITS: Constructing Structured Queries from Keywords, Technical Report, 2008
3. Elena Demidova, Xuan Zhou, Gideon Zenz, Wolfgang Nejdl, SUITS: Faceted User Interface for Constructing Structured Queries from Keywords, The 14th International Conference on Database Systems for Advanced Applications (Best Demo Award), 21-23 April 2009, Brisbane, Australia
4. Gideon Zenz, Xuan Zhou, Enrico Minack, Wolf Siberski, Wolfgang Nejdl, From keywords to semantic queries - Incremental query construction on the semantic web, J. Web Sem. 7, 3 (2009), 166-176 (Most Cited Article Award 2006-2010)
5. Kerstin Denecke, Gideon Zenz, Wladimir Krasnov, Semantic Web Technologies to Improve Customer Service, International Semantic Web Conference 2009, October 2009, Washington, USA
6. Nina Tahmasebi, Gideon Zenz, Tereza Iofciu, Thomas Risse, Terminology Evolution Module for Web Archives in the LiWA Context, In Proc. of 10th International Web Archiving Workshop in conjunction with iPRES in Vienna, Austria, 2010
7. Gideon Zenz, Nina Tahmasebi, Thomas Risse, Language Evolution On The Go, SAME 2010 - 3rd International Workshop on Semantic Ambient Media Experience (NAMU Series) November, 10th-12th November 2010 in conjunction with AmI-10 in Malaga, Spain
8. Gideon Zenz, Xuan Zhou, Enrico Minack, Wolf Siberski, Wolfgang Nejdl, Interactive Query Construction for Keyword Search on the Semantic Web, chapter in book “Semantic Search over the Web”, Data-Centric Systems and Applications, Springer 2012, pp 109-130
9. Gideon Zenz, Nina Tahmasebi, Thomas Risse, Towards mobile language evolution exploitation, J Multimed Tools Appl., Springer, 2012

3. SUITS: Constructing Structured Queries from Keywords

Keyword queries are meanwhile the most common and effective way to accessing unstructured web data, but it fails to be effective to accessing structured web data. We devised SUITS, an interactive mechanism to construct the intended query for structured data using a ranking based approach, and amends this by offering query construction options for guiding the ranking algorithm. As can be seen in Fig. 3, SUITS offers direct support for the *need articulation phase*, as well as the *query formulation phase*. The query construction options, which allow specifying the schema of query terms, i.e. 'hanks' as actor name, it also allows limited support for the *query reformulation* and *result evaluation phase*.

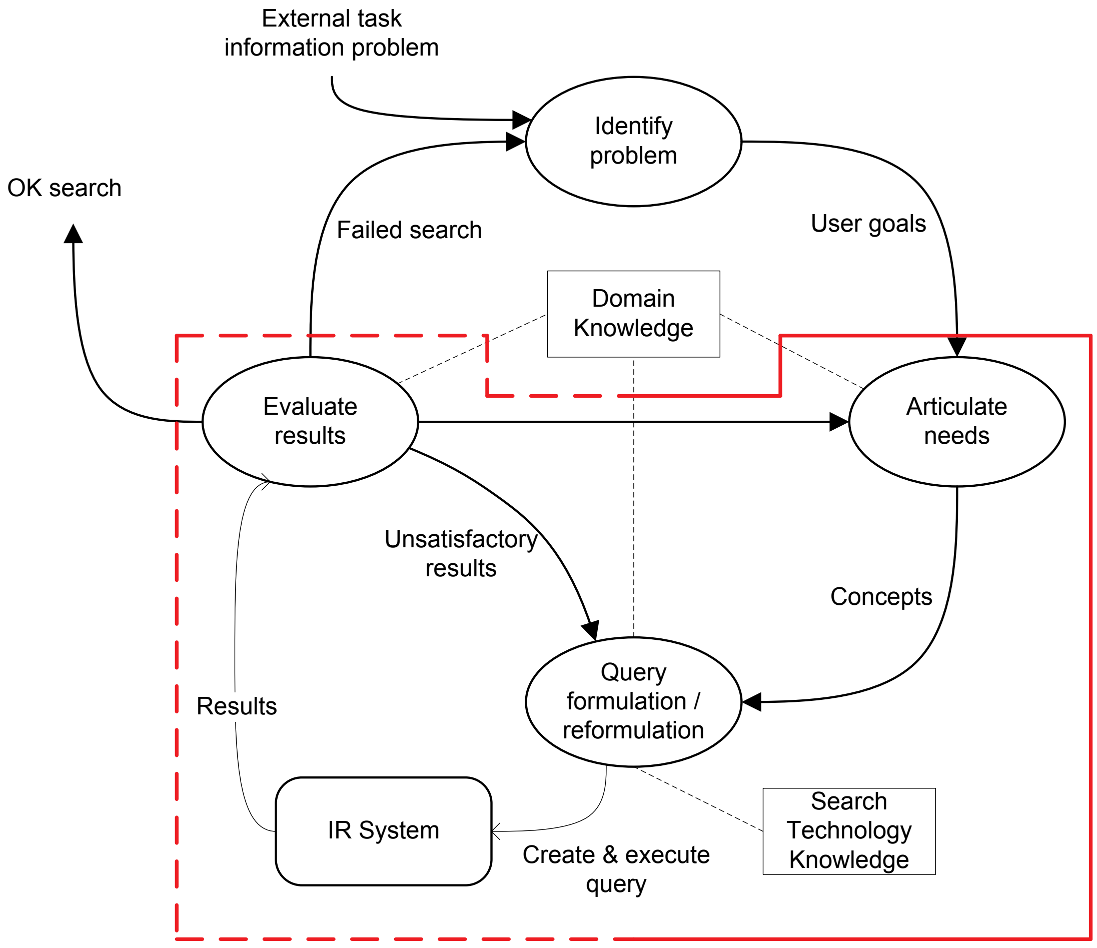


Fig. 3: SUITS support of the query process

Relevant publications for this chapter:

- Enrico Minack, Wolf Siberski, Gideon Zenz, Xuan Zhou, SUITS4RDF: Incremental Query Construction for the Semantic Web, International Semantic Web Conference (Posters & Demos) 2008
- Elena Demidova, Xuan Zhou, Gideon Zenz, Wolfgang Nejdl, SUITS: Faceted User Interface for Constructing Structured Queries from Keywords, The 14th International Conference on Database Systems for Advanced Applications (Best Demo Award), 21-23 April 2009, Brisbane, Australia

3.1 Introducing SUITS

Today’s heterogeneous data management environments demand search interfaces that are not only sufficiently expressive to exploit structured queries, but also as intuitive and easy to use as keyword search. Furthermore, the system should not force users to study and memorize schemas in advance. We demonstrate this requirement within the following scenario:

Alice is searching for the movie “Hot Fuzz”[2] in a video database. Unfortunately, she forgot the movie title and only remembers one word “Fuzz”. In addition, she knows that the director’s surname is “Wright” and that the story takes place in London. Therefore, Alice issues a keyword query “Fuzz Wright London” to the database. However, there are too many occurrences of these keywords in the database, as these words are often used in movie titles, story descriptions, person names, reviews and other attributes. Alice can thus obtain many results but almost none of them are related to the movie “Hot Fuzz”. To better express her intent, she starts to form the following structured query:

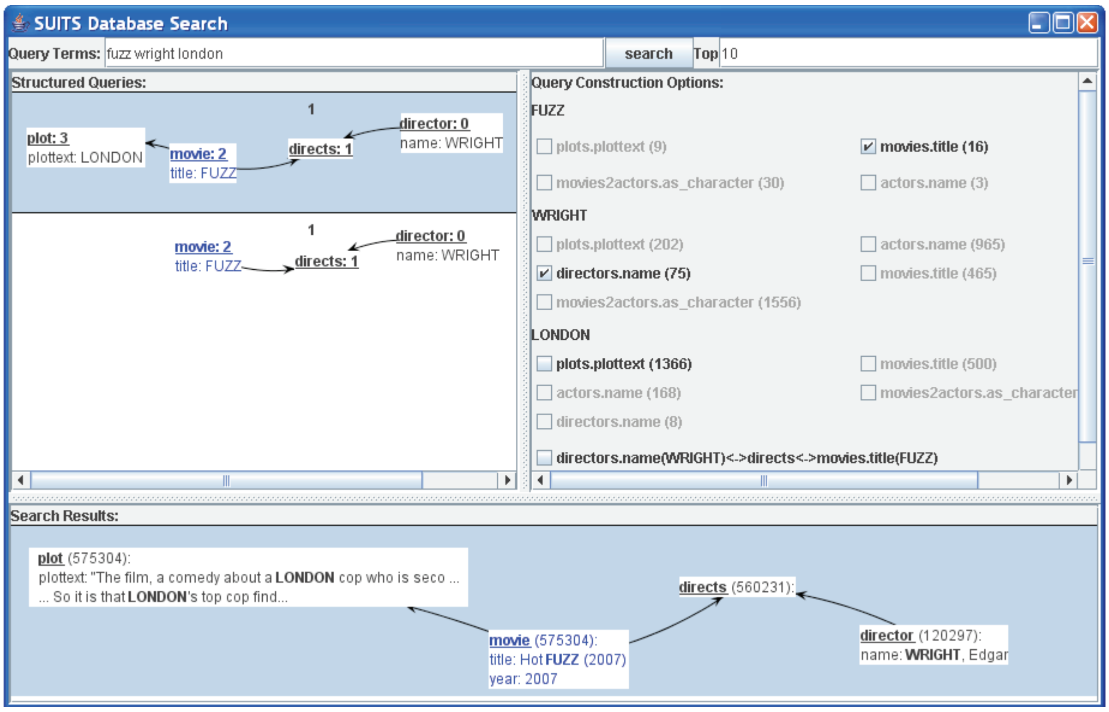
```
SELECT     * FROM movie
WHERE      movie.title CONTAIN “Fuzz”
AND        movie.director.name CONTAIN “Wright”
AND        movie.contents CONTAIN “London”
```

[2] http://www.imdb.com/title/tt0425112/

However, it is very unlikely that the query is regarded as valid by the database, as both table and attribute names do not match. To successfully issue this query, Alice must first examine the database schema in order to specify the correct tables and attributes, and then structure her query carefully. This information finding process is troublesome and time consuming, especially if Alice is used to Google-like interfaces to find appropriate information.

In this chapter, we present SUITS[3], a novel search interface, which provides a layer of abstraction on top of relational databases to smoothly integrate the flexibility of keyword search and the precision of database queries. As shown in Fig. 4, the SUITS interface consists of four parts: a search field for the user to input keyword queries, a result window to present search results (at the bottom), a query window to present structured queries (on the left) and a faceted query construction panel providing query construction options (on the right). To perform search, Alice first issues a keyword query, for instance “Fuzz Wright London”. Besides returning a ranked list of results like standard keyword search, the system will suggest to Alice a list of structured queries in the query window to clarify her intent. The suggested queries assign different semantics to the keywords. For example, some queries may search for movies with the actor “Wright”, while others search for actors who incorporate a character named “London”. If Alice identifies the query that represents her intent, she can click on it so that the result window will zoom into the results of that particular query. If she cannot identify the intended query and neither is satisfied with the displayed results, she can go to the faceted construction panel to select some query construction options suggested by the system and construct the intended query incrementally. For example, she can specify that “Fuzz” must appear in the movie title and “Wright” must be a director’s name. The query window will change accordingly, to show only the queries satisfying the options she has chosen. Interaction between Alice and the system continues iteratively until she obtains the right query and/or satisfactory results.

[3] SUITS: Structuring User’s Intent Towards Search



Query Terms: fuzz wright london

Top 10

FUZZ

- [ ] plots.plottext (9)
- [✓] movies.title (16)
- [ ] movies2actors.as_character (30)
- [ ] actors.name (3)

WRIGHT

- [ ] plots.plottext (202)
- [ ] actors.name (965)
- [✓] directors.name (75)
- [ ] movies.title (465)
- [ ] movies2actors.as_character (1556)

LONDON

- [ ] plots.plottext (1366)
- [ ] movies.title (500)
- [ ] actors.name (168)
- [ ] movies2actors.as_character
- [ ] directors.name (8)

- [ ] directors.name(WRIGHT)<->directs<->movies.title(FUZZ)

plot (575304):
plottext: "The film, a comedy about a **LONDON** cop who is seco ...
... So it is that **LONDON**'s top cop find...

movie (575304):
title: Hot **FUZZ** (2007)
year: 2007

directs (560231):

director (120297):
name: **WRIGHT**, Edgar

Fig. 4: Interface of SUITS

With the SUITS interface, users can issue database queries in an ad-hoc way without any prior knowledge of database schemas. They start with some keywords, which they believe to be sufficiently descriptive and then structure their query progressively by following the system's suggestions. Using the interface, they can either: create a completely structured query by selecting one from the query window (top left) or a partially structured query (top right) by specifying appropriate query construction options. The type of interaction depends on the degree to which users want or are able to clarify their intent.

Recently, a number of approaches [9], [11], [12], [14], [17], [45], [46] have been proposed to realize keyword search over databases. Most of them adopt IR style mechanisms, and return a long list of virtual documents as results (usually graphs that connect the query terms). In contrast, SUITS attempts to help user construct more expressive structured queries. For instance, a user may search for all movies having Tom Hanks as an actor in 2001. Using traditional keywords search, he has to go through the entire list of results to find the answers. Using SUITS he gets the answers by simply choosing the relevant structured query. In the areas of IR and the semantic web, there are some emerging techniques [15], [16], [47] for interpreting users' keyword queries into structured queries. SUITS goes beyond these approaches by allowing users to construct more general database queries in a progressive way, and by optimizing query processing.

We present a detailed realization of the SUITS interface and the underlying algorithms. Specifically, we realize the following contributions: (1) we propose an architecture for the SUITS system; (2) we define a framework for incrementally constructing relational database queries from keywords; (3) we devise statistical methods for predicting user intended structured queries, which exhibits better accuracy than previous work [9], [45] in ranking query result; (4) we propose a method that exploits the relationship between queries to optimize the performance of SUITS; (5) we conduct extensive experiments to evaluate the effectiveness of SUITS and its various components.

The current SUITS system is designed for databases that contain mainly text, as such databases are very common in many enterprise information systems and web-based applications. Therefore, SUITS uses only limited types of predicates in its structured queries; more complex structured queries will be investigated in future work. When designing SUITS, we assume that the end users are able to correctly understand structured queries presented by the system, although they are unable to form valid structured queries without investigating the database schema in detail. In practice, the assumption relies on a well-defined database schema that is easily understandable as well as an intuitive way for presenting structured queries (e.g. in figures or in natural language). However, as it is not the goal of this work to design a presentation layer for relational databases, we assume that these facilities are available.

3.2 Architectural Overview

The architecture of SUITS is shown in Fig. 5. Processing steps can be split into two phases: an offline pre-computing phase and an online query phase. In the pre-computing phase, SUITS creates inverted indices for all text columns in the database, which will be used in both query generation and query execution. It also generates query templates that are potentially employed by users when forming structured queries. For example, users sometimes search for movies with a certain character, and sometimes for actors who played in a certain movie. These are all meaningful query templates to be generated in the pre-computing phase.

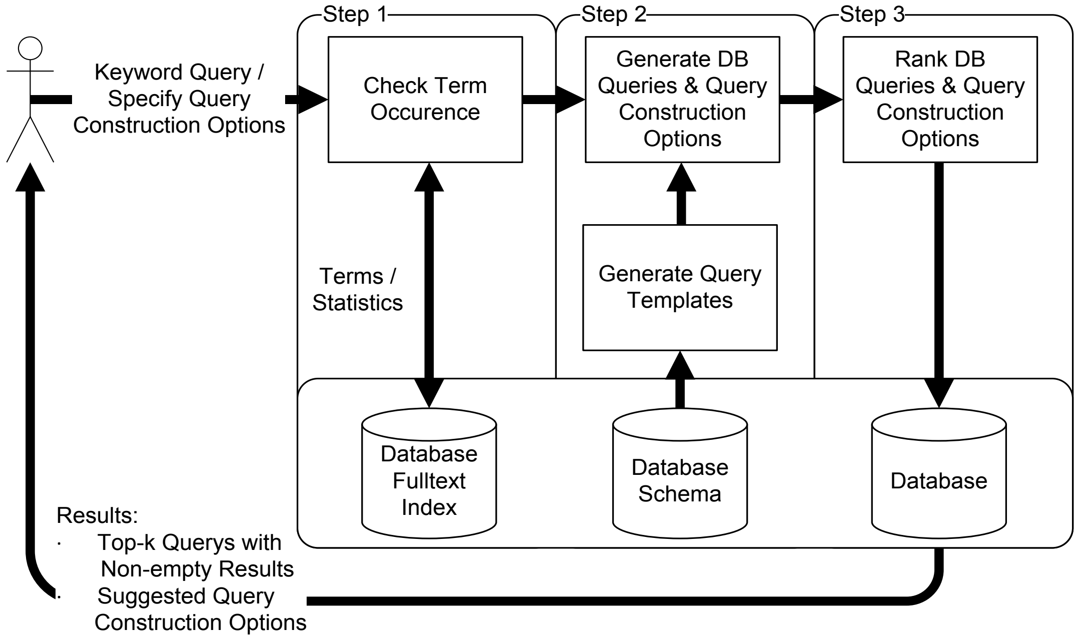

Step 1
Step 2
Step 3
Keyword Query / Specify Query Construction Options
Check Term Occurence
Generate DB Queries & Query Construction Options
Rank DB Queries & Query Construction Options
Terms / Statistics
Generate Query Templates
Database Fulltext Index
Database Schema
Database
Results:
· Top-k Querys with Non-empty Results
· Suggested Query Construction Options

Fig. 5: Architecture of SUITS

The online query phase consists of three steps. In step 1, the system receives the user’s keyword query and passes it to the full-text indices to check for occurrences of the query terms in all tables and attributes. In step 2, it combines these term occurrences with the pre-computed query templates to generate meaningful structured queries. In step 3, the system ranks the structured queries according to their likelihood of matching the user’s intent and returns the top-k queries with non-empty result-sets. When generating structured queries in step 2, the system also generates query construction options that the users can use later in incrementally refining their keyword queries. These options are ranked in step 3 and returned to the user. If the user selects some of these options, the selected options are passed together with the keywords to the query generation step (step 2) to filter out queries that do not satisfy the selected options.

The success of the SUITS system relies on a number of steps: proper generation of query templates, structured queries and query construction options, effective ranking of queries and construction options, and efficient query processing. We describe the detailed implementation of these steps in the next three sections.

3.3 Query Generation and Ranking

Let us first describe the generation of structured queries using a set of keywords and the assessment of their likelihood for matching user's real intent. Similar to previous approaches [11], [12], [14], we treat a result of keyword search on relational databases as a joining network of tuples (also known as tuple tree). In addition, we introduce definitions for schema graph, query template and structured query. The examples given in this section are based on a database of movies, which contains information about movies, actors, directors and etc.

3.3.1 Query Template Generation

A query template is a structural pattern the user uses to query a database. For example, users sometimes search for movies with a certain character, and sometimes search for actors who have played in a certain movie. Both are commonly used query templates. SUITS creates such query templates using only the database schema.

Definition 1: Schema Graph

A schema graph is a directed graph SG. Each node in *SG* corresponds to a table R_i in the database and vice versa. Each edge in *SG* corresponds to a primary-to-foreign key relationship (R_i, R_j) in the database and vice versa. Fig. 6 gives the schema graph of the movie database.

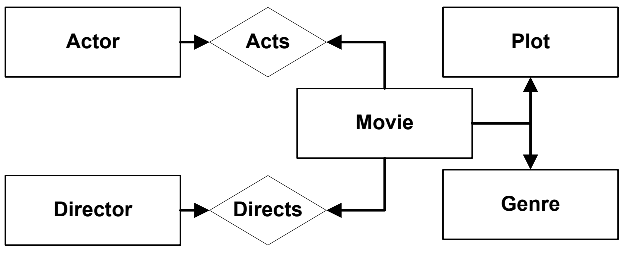


Fig. 6: The Schema Graph of a Movie Database

Query Template: a query template is a connected non-cyclic graph QT, such that each node in *QT* is a replica of a node in *SG* and each edge in *QT* is a replica of an edge in *SG*. Each node in *QT* is labelled as *non-free* or *free*, indicating whether the corresponding node should contain query predicates or not.

As we consider databases that contain only text, the predicates used in structured queries are limited to checking occurrence of terms. A predicate has the form "$k \in a$", specifying that each result must contain keyword *k* in attribute *a*.

Fig. 7 shows an example of a query template, which searches for movies with two particular actors. We can see that a query template consists of tables and foreign key relationships in the schema graph, which may be used repeatedly. Both actor nodes are labelled to be non-free, indicating that a query created from the template should specify predicates on both actor nodes.

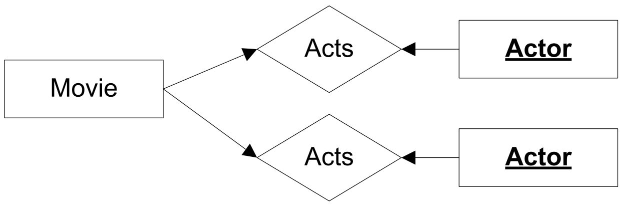


Fig. 7: A Query Template (non-free nodes bold)

Let e_k be a node in *QT* and let *(ei,ej)* be an edge in *QT*. We use *R(ek)* to denote the relational table e_k corresponds to, and accordingly use *(R(ej),R(ej))* to denote the primary-to-foreign key relationship *(ej,ej)* corresponds to.

As described in the previous section, SUITS generates query templates to cover all possible queries issued by users. Although these templates could be manually generated by a database administrator, this would be a very time consuming task especially when the schema graph is big. SUITS automatically generates templates, following a set of rules, which are able to enforce usefulness of the templates most of the time. These rules restrict templates by:

- maximum number of nodes
- maximum number of non-free nodes
- all leafs in a template must be non-free nodes

As we observed in a real-world query log described later in the experimental section, user queries have limited complexity. A query usually involves only a limited number of tables and predicates, which can be enforced by the maximum numbers of nodes and the maximum number of non-free nodes in the query template. For the movie database in Fig. 6, if we set the maximum number of nodes to 7 and the maximum number of non-free nodes to 4, we obtain most of the useful query templates. The third rule has been used in previous work [9], [12], [14] for determining meaningful tuple trees, and it is used by SUITS to avoid redundant query templates.

Theorem 1: Connectivity of a schema graph
If the connectivity of a schema graph *SG*, i.e. the maximum number of foreign keys per table, is limited, and the maximum number of nodes in a query template is limited, then the number of possible query templates based on *SG* is linear in the number of nodes in *SG*.
As the connectivity of most real world database schemas is limited, we can conclude from Theorem 1 that the number of query templates generated by SUITS is only linear in the size of the input database schema.

3.3.2 Query Generation

As mentioned, given a keyword query issued by a user, SUITS first checks for occurrences of the keywords in all tables and attributes to create all possible predicates, and then applies these predicates to the available query templates to generate all possible queries.

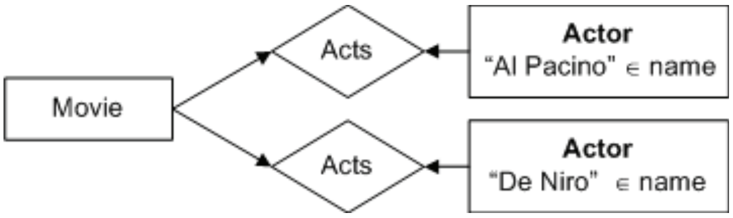


Fig. 8: Query Searching for a Movie acted by Al Pacino and De Niro

Definition 2: Structured Query
Given a keyword query $K=\{k_1,\ldots,k_n\}$ and a query template *QT*, *Q* is a possible structured query based on *QT*, iff: (1) *Q* is a graph that contains the same set of nodes and edges as *QT*; (2) each non-free node in *Q* contains at least one

predicate; (3) for each predicate "$k_i \in a$" in *Q*, k_i is contained by *K* and there is at least one occurrence of "$k_i \in a$" in the database. Fig. 8 shows an example query, which is based on the query template in Fig. 6.

Definition 3: Query Result

A query result is a joining network of tuples in the database (also known as tuple tree). A tuple tree *T* is a result of a query *Q,* if there is a bijective map between the nodes in *Q* and the tuples in *T*, i.e. $f{:}Q \leftrightarrow T$, which satisfies: (1) for each node e_k in *Q*, the corresponding tuple $f(e_k)$ is contained by table $R(e_k)$; (2) for each edge (e_j, e_j) in Q, the corresponding pair of tuples $(f(e_i),f(e_j))$ is contained by the natural join $R(e_i)\bowtie R(e_j)$; (3) for each non-free node e_t in *Q*, the corresponding tuple $f(e_t)$ satisfies all the predicates e_t contains.

In practice, each structured query can be expressed using an SQL query that is able to retrieve the complete set of results from the database. For a large database such as IMDB, query terms usually occur in many tables, which may increase the number of queries exponentially with the number of keywords and thus can result in poor performance for long keyword queries. We will address this issue in the query processing section.

3.3.3 Ranking Queries

Using the query generation algorithm of SUITS, a small number of keywords may result in a large number of structured queries, whereas the user typically intends only a specific one. Hence, queries generated by SUITS are ranked based on their likelihood of being intended by the user before being executed in the database, to improve the performance of query processing as well as the appropriate presentation of results.

Most of the previous systems [12], [14] treat tuple tree based results as virtual documents and rank them using IR methods, such as TF×IDF scores. Some approaches [7], [9], [45] involve additional factors, such as tuple tree size, distance between keyword occurrences, as well as term frequency normalization based on attribute lengths. However, as their ranking functions are designed for actual query results, they cannot be directly applied to structured queries. Instead, we exploit the fact that the structure of database schemas provides rich information that enables effective ranking of both structured queries and results.

The SUITS ranking function is composed of three factors. The first factor, Standardized Expected Results (SER), is a measure of whether a query would retrieve a reasonable number of results. Typical users in an online session intend their queries to be sufficiently representative and descriptive, so that they can retrieve a small number of results. In other words, if a query returns too many results, it is less probable to be intended by the user. This heuristic is similar to inverse document frequency (IDF) in IR, which prefers documents containing query terms of higher selectivity. Let Q_r be an estimated number of results to be returned by the query Q. We use the following function to compute the SER score of a query *Q*:

$$SER(Q) = \frac{p_1}{\sqrt{{Q_r}^2 + {p_1}^2}} \qquad (1)$$

SUITS estimates Q_r by counting the number of tuples in the first table in *Q*, multiplied by the cardinality of every join operation, and multiplied again by the selectivity of every predicate. p_1 is a tuning parameter that can be understood as the maximum number of results normally intended by the user. While Formula (1) is decreasing monotonously, its value remains quite large ($0.7<SER<1$) as long as $Q_r<p_1$. Only when Q_r is much larger than p_1, the value drops quickly. This characteristic is important, as users usually only intend the number of results to be within a reasonable margin, but do not prefer smaller results in each case. Our experiments show that $p_1=10$ is a reasonable number for generating good rankings.

The second factor we use is Attribute Completeness (AC), which measures how completely each attribute in the predicates is covered by query terms. In contrast to unstructured documents, a structured database is composed of attributes of rather short length, such as names of persons or titles of movies. According to our observation of real world query sets, users tend to use short attributes more frequently and they often specify an attribute as completely as possible in keyword queries. For example, users type in the full name of an actor or the complete title of a movie. Although they are not able to explicitly indicate these attributes in their keyword queries, they always have these basic concepts in mind. We may therefore assume that the more complete an attribute is covered by query terms, the more likely it is used in the intended structured query. Let *terms(a)* be the number of query terms in a attribute *a,* and *attrlen(a)* the average length of the attribute *a* in the database. Then the AC of a query *Q* is calculated as follows:

$$AC(Q) = \left(\prod_{a \in Q} \min\left(\frac{p_2 + \ln(terms(a))}{p_2 + \ln(attrlen(a))}, 1 \right) \right)^{\frac{1}{|a \in Q|}} \quad (2)$$

The function calculates the geometric mean of term coverage for all attributes in the predicates of the query *Q*. We use *ln(.)* because smaller attributes such as actor name need to be covered more completely than larger attributes such as plots. In order to prevent attributes from being overly crowded by query terms, term coverage of each attribute is limited to 1. p_2 is a tuning parameter, which we found to have good characteristics when set to 0.5.

The last factor we use is Term Completeness (TC), which measures the percentage of terms in the keyword query included in the structured query. As all keywords issued by the user should be related to the desired search results, they should be included into the structured query as completely as possible. Given a structured query *Q* and a set of query terms *K*, TC is defined as:

$$TC(Q, K) = \left(\frac{terms(Q)}{|K|} \right)^{P_3} \quad (3)$$

where *terms(Q)* is the number of keywords used in query Q, and p_3 is a tuning parameter to adjust the importance of this factor. Our experiments showed that P_3 = 8 is sufficient to enforce AND semantics in keyword queries.

The total score of a query is calculated by multiplying the partial scores.

$$Score(Q, K) = SER(Q) \cdot AC(Q) \cdot TC(Q, K) \quad (4)$$

As all scores can be calculated without any access to the actual database, the formula allows all queries to be ranked prior to their execution on the database. While the information about schema and data structure as in the SER and AC measures has never been used by previous approaches to rank results of keyword search, our experimental evaluation shows that these factors are very effective in ranking both structured queries and results.

3.4 Query Construction Options

Besides returning a ranked list of structured queries, SUITS suggests appropriate query construction options to support users in incrementally creating the intended structured query from their keywords. Choosing a query construction option is equivalent to specifying a fragment of a structured query. As illustrated in Fig. 4, SUITS allows users to choose the attribute a keyword should occur in, which confirms one table and one predicate that should appear in a query. Therefore, the query construction options returned by SUITS are partial queries. Partial queries have the same definition as structured queries. The only distinction is that a partial query is not regarded as a stand-alone query but a fragment included in a complete query.

Definition 4: Sub-query Relationship

A query Q is a sub-query of another query Q', i.e. $Q \subset Q'$, iff (1) $Q \neq Q'$, (2) the nodes and edges in Q are a subset of the nodes and edges of Q', (3) each node in Q has the same free/non-free label as the corresponding node in Q', and (4) the predicates used in each node in Q are also used in the corresponding node in Q'.

The sub-query relationship is transitive. We say that Q is a *direct sub-query* of Q' if (1) $Q \subset Q'$ and (2) there is no query Q'' such that $Q \subset Q''$ and $Q'' \subset Q'$.

3.4.1 Constructing a Query using the Partial Query Hierarchy

If we connect all possible partial queries using direct sub-query relationships, we obtain a hierarchy of partial queries. At the bottom of the hierarchy are the smallest partial queries that do not have sub-queries. These smallest partial queries are also known as *term-attribute combinations*, which simply assign a query term to a certain attribute in a certain table. At the top of the hierarchy we have complete structured queries that can be utilized by the user to retrieve intended results. SUITS lets users start with term-attribute combinations, and gradually evolve them into larger partial queries by climbing the query hierarchy, until they end up with a complete query.

Fig. 9 shows part of a partial query hierarchy for constructing the structured query to search for the movie “Hot Fuzz”. A user issues a keyword query “Fuzz London Wright”. For each of the keywords, SUITS provides a list of attributes for the user to choose. For example, the user can specify whether “Wright” should appear in the actor name, director name or movie title. After the user specifies term-attribute combinations the system offers larger partial queries that contain the selected combinations. For instance, after the user specifies the

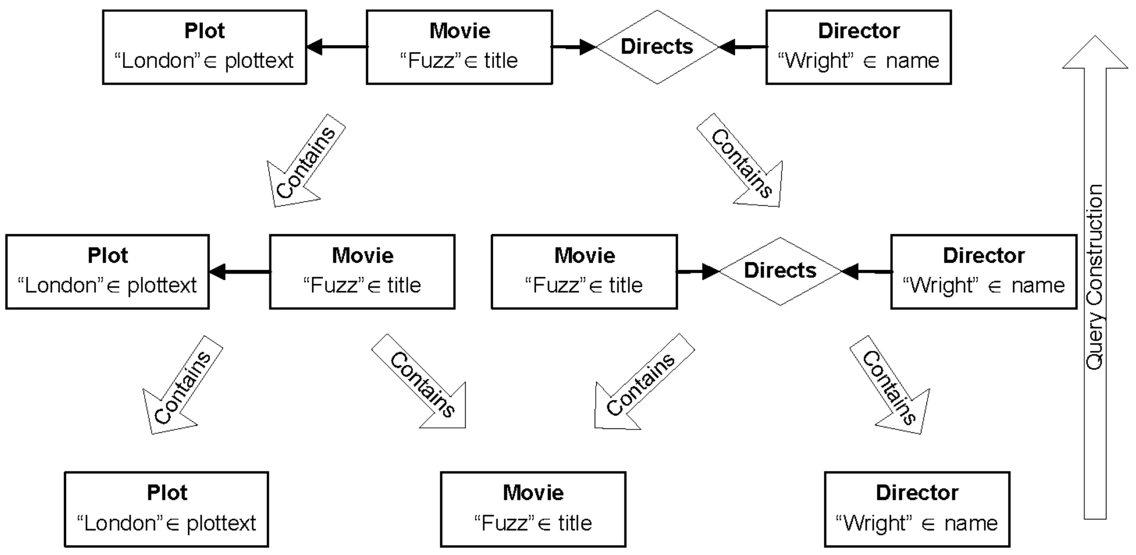


Fig. 9: Hierarchy of Partial Queries

director name “Wright” and movie title “Fuzz”, the system can suggest the partial query that connects these two term-attribute combination using the *directs* relation, as shown in the middle left of Fig. 9. Afterwards, the user can assign “London” to the plot-text, and the system can suggest the partial query at the top of Fig. 9, which is already a complete structured query.

As the example shows, when suggesting query construction options SUITS starts from the bottom of the partial query hierarchy, i.e. term-attribute combinations. A partial query is then suggested when the user has specified:

- one of its direct sub-queries
- all the term-attribute combinations it comprises

These conditions attempt to suggest partial queries as small as possible to ensure that the user can construct a query incrementally. To alleviate the burden on the user, SUITS ensures that each suggested partial query and all already selected partial queries could result in complete queries that return non-empty result-sets.

After the user selects a partial query, the ranked query list presented is reconsidered to include only the top-k queries comprising all selected partial queries. With the introduced ranking function, users usually do not need to completely construct a structured query from scratch. Our experiments showed that, with smaller schemas, a desired query could be obtained specifying only one or two term-attribute combinations.

3.4.2 Ranking Partial Queries

Depending on the size of the database schema as well as on the keyword query given by the user, there could be too many partial queries that can be offered to the user. Therefore, partial queries need to be ordered based on their likelihood of being chosen by the user. For ranking partial queries, we use a similar but slightly different formula from the one for ranking complete queries, as follows:

$$Score(Q) = SEL(Q)^{ps} \cdot AC(Q)^{pa} \quad (5)$$

SEL(*Q*) denotes the selectivity of the partial query *Q*, which measures the percentage of tuple trees instantiated from *Q*'s template can be selected by *Q*. We use SEL instead of SER in Formula (4), because SER measures the number of final results desired by users, which does not apply to partial queries. AC(*Q*) is the previously introduced attribute completeness of *Q*. We do not consider term completeness (TC), as partial queries do not need to contain all queries terms. *ps* and *pa* are tuning parameters to adjust the weight of these two factors.

When applying the ranking function to a term-attribute combination $t \in a$, SEL($t \in a$) is equivalent to the inverse document frequency of term *t* in attribute *a*. So the formula changes to:

$$Score(t \in a) = IDF(t \in a)^{ps} \cdot AC(t \in a)^{pa} \quad (6)$$

Our experiments show that an instantiation of these parameters as *ps*=0.5 and *pa*=0.1 performs well for term-attribute combinations. Database usage statistics can be used to further improve the ranking of partial queries. However, in this work we concentrate on a generic ranking function that works without knowledge of query statistics.

3.5 Query Processing

As stated earlier, the number of query templates generated by SUITS grows linearly with the size of the database schema graph; the number of structured queries for a keyword query can, in the worst case, grow exponentially with the number of keywords provided by the user. Therefore, with a big database schema and long keyword queries, SUITS will have to generate many structured queries that can possibly be desired by the user. It is obviously infeasible for SUITS to pre-execute all structured queries before presenting results to the user. SUITS employs a top-k query approach to ease the burden on the database and exploits the sub-query relationships to further optimize query processing performance.

3.5.1 Top-k Queries

Instead of evaluating and executing all structured queries at once, SUITS first executes the k highest ranked queries that return non-empty result-sets and presents them to the user. Only when the user requests more results or specifies some query construction options, SUITS proceeds to test additional structured queries. Compared with usual top-k processing in databases, finding top-k queries in SUITS is much simpler. As our ranking function does not require any information about the query results, structured queries can be ranked completely before accessing the database. Hence SUITS can rank all queries in memory, and execute one query after another until it obtains k queries that return non-empty result-sets. This ensues that SUITS achieves better performance than recent approaches [14], [45], which have to retrieve the results before ranking them correctly.

If a user issues too many keywords, the sorting of the complete set of structured queries can become expensive[4]. In these cases the system can adopt strategies of the Threshold Algorithm [48] or skyline query processing [45] to avoid calculating the scores of many lowly ranked queries. As our experiments showed that sorting is not a bottleneck of SUITS's performance, we do not discuss this issue further in this paper.

3.5.2 Optimization Using the Query Hierarchy

Even when using top-k queries the system can still be slow. As term occurrences are usually distributed sparsely in a database, the chance for them to be connected by tuple trees is small. Therefore, many structured queries generated by SUITS, especially the highly selective ones, will not obtain results. We observed that the proportion of structured queries that retrieve non-empty result-sets decreases exponentially with the number of terms (see Appendix), forcing the number of queries the system has to execute before it obtains top-k non-empty queries to grow exponentially with the number of keywords. We believe this is the potentially most significant performance bottleneck for SUITS.

Fortunately, we can utilize the sub-query relationship between structured queries to constrain the number of executed queries to non-exponential order, and thus significantly improve SUITS's performance for processing long keyword queries. With the definition of sub-query relationship, we can easily prove the following theorem:

Theorem 2: Empty Result Set Transitivity
For any two structured queries Q and Q', such that Q is a sub-query of Q', if Q returns an empty result-set then Q' returns an empty result-set.
Using the sub-query relationship, SUITS can construct a hierarchy of structured queries as shown in Fig. 9. When processing structured queries, SUITS starts

[4] The complexity of sorting top k queries out of a set of n queries is n×log(k).

from the bottom of the query hierarchy, so that some structured queries can be skipped if any of their sub-queries return empty result. The detailed algorithm for processing top-k queries is given in Algorithm 1. The algorithm sequentially executes the structured queries in the ranked list, until it finds k queries that return a non-empty result-set. Whenever SUITS tests a query, it starts to execute its sub-queries. Only if all the sub-queries return non-empty result-sets, it executes the current query on the database, otherwise it skips the query. Although the algorithm sometimes needs to execute additional queries that are not included in the top-k queries returned to the user, it avoids executing a lot of complex queries. Our experiments show that we can improve the performance of SUITS significantly, especially when a user issues a long keyword query (more than 3 keywords in our experiments).

We prove that, with this algorithm, the number of the structured queries executed on the database will not grow exponentially with the number of keywords. Therefore, we analyse why a query hierarchy can optimize the performance for median and long keyword queries.

For simplicity, we consider only one query template and assume that each node in the template contains only one attribute. We do not distinguish between free and non-free nodes, and allow each node to contain zero or multiple predicates. Suppose there are *n* nodes in the template. Given a keyword query with *k* terms, if each term occurs in every table of the databases, the number of the possible structured queries generated by SUITS will be $\sum_{i=1}^{k}\binom{n}{i} * n^{i}$, which is equivalent to $(n+1)^{k}-1$. This explains why the number of queries can grow exponentially with the number of query terms.

```
//Q[..] is a ranked list of structured queries.
//Initially Q[i].if_exe()=false for any i,
//because all queries have not been executed.

begin func top_k_queries(Q[1..n],k)
     nquery ::= 0;
     for i=1 to n
          if Q[i].if_exe()= false, then
               execute(Q[i]);
          end if
          if Q[i].if_empty()= false, then
               output(Q[i]);
               if ++nquery=k then
                    return;
               end if
          end if
     end for
end func

begin func execute(q)
     QC[] = q.sub-query(); //all sub-queries of q
     for i=1 to n
          if QC[i].if_exe()= false, then
               execute(QC[i]);
          end if
          if QC[i].if_empty()= true, then
               q.if_empty()::= q.if_exe()::= true;
               return;
          end if
     end for
     DB.execute(q);     //execute q on database
     q.if_exe()::= true;
     q.if_empty()::= DB.if_empty();
end func
```

Algorithm 1: Algorithm for Top-k Queries

Suppose there are *m* tuple trees instantiated from the query template and the average selectivity of a term in a table is *p (p<<1)*. Then the probability that a structured query with *k* predicates will obtain non-empty result-set is $1-\left(1-p^{k}\right)^{m}$. This explains why the non-empty queries become more and more rare when the number of query terms *k* increases.

By using a query hierarchy, a query is executed only when all its sub-queries returns non-empty result. It can be proved that the expected total number of queries to be executed is $\sum_{i=1}^{k}\left(1-\left(1-p^{i-1}\right)^{m}\right)^{i} * \binom{k}{i} * n^{i}$. This value is much smaller than the number of possible queries, and it does not grow exponentially with *k*. This explains why a query hierarchy can improve the performance of query processing significantly.

3.6 Evaluation

We have implemented the SUITS system and conducted extensive experiments using real world data and query sets for evaluating its performance. First, we give an overview of our experiment setup. Next, we evaluate the precision of SUITS in predicting user intended structured queries, followed by evaluating the quality of query construction options suggested by SUITS. Finally, we study the efficiency of SUITS.

3.6.1 Experiment Setup

3.6.1.1 Datasets

In our experiments, we used two popular real-world datasets: a crawl of the Internet Movie Database (IMDB) and a crawl of a lyrics database from the web. The IMDB dataset contains 7 tables as shown in Fig. 6, and more than 10 million records. The Lyrics dataset contains 5 tables, such as artists, albums and songs, and around 400.000 records. The two datasets have different characteristics. The IMDB database is larger and has a more complex schema, it therefore allows for more complex structured queries. The keywords used in different attributes of IMDB are more distinctive, such as movie titles and person names. In contrast, the Lyrics database is smaller, and the songs and albums contain a lot of common keywords, such “love”, “me”, “you”, which also appear quite often in user queries. Evaluating our approach over these two different data sets allows us to obtain a more complete view of SUITS’s usefulness.

3.6.1.2 Query Sets

In order to estimate the performance of SUITS in real-world settings, we used a real-world query load created from the AOL query log, containing 35 million queries. We filtered the queries by their visited URLs to obtain 3000 sample queries for movie web pages and 2000 sample queries for lyrics web pages. We observed that most of the queries in the samples used rather simple semantics, i.e. only containing a movie title or an actor name, which cannot fully reflect the advantages of a more complex approach like SUITS. We therefore restricted our experiments to queries containing at least two attributes, such as movie-actor and artist-lyrics, and finally including 108 queries for IMDB and 81 queries for Lyrics. These queries range from 2-6 keywords, with an average length of 4 terms. For each of the selected queries we manually assessed its meaning and intended results by searching on the Web, in order to determine the relevance of search results, and we always stick

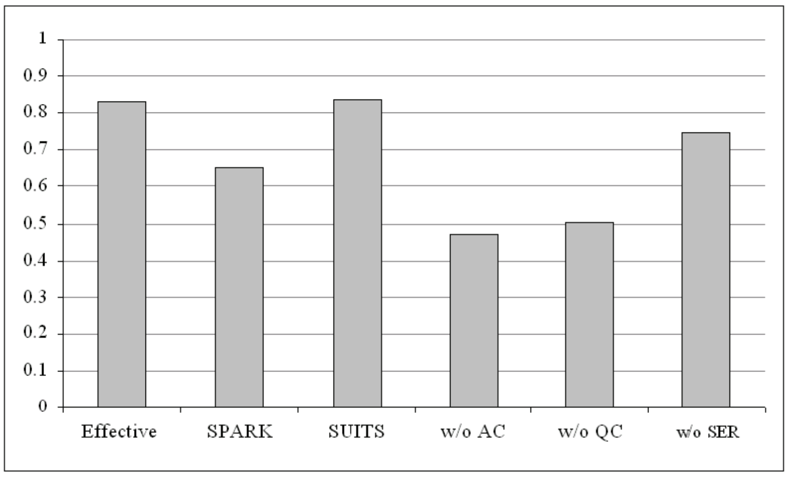


Fig. 10: Mean Reciprocal Rank, IMDB

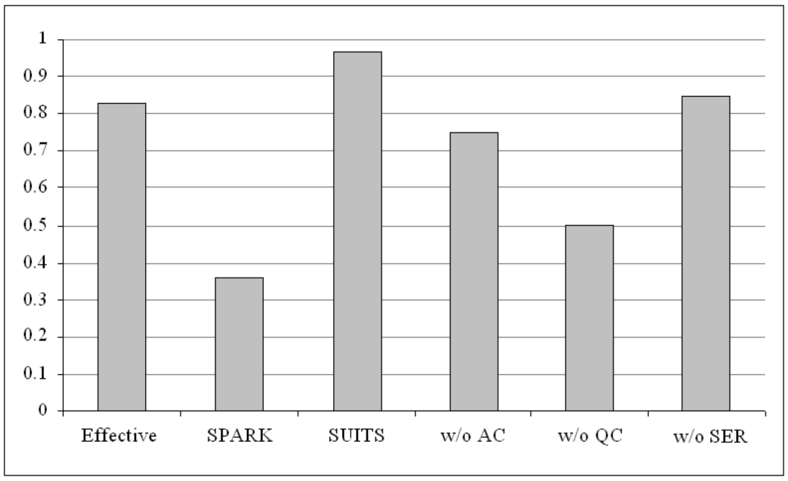


Fig. 11: Mean Reciprocal Rank, Lyrics

to the same interpretations during evaluation. As we did not implement schema term or phrase recognition support, we removed schema terms from the queries and manually phrased about 30% of the queries. We did not observe a significant change in effectiveness from phrased queries to non-phrased ones.

3.6.1.3 Experiment Settings

We installed our databases on a dedicated MySQL 5.0.22 dual Xeon server with 4 GB RAM. We manually increased the join-cache of MySQL server to allow for longer queries. The SUITS system was implemented using JDK 1.5 and JDBC and installed on a laptop with a 2.0 GHZ C2D and 2 GB RAM. As all experiments were conducted within our intranet, the performance of SUITS was mainly limited by disk I/O.

3.6.2 Query Ranking Effectiveness

Our first set of experiments evaluated the precision of SUITS in predicting user intended structured queries. Given a keyword query, we assess how well SUITS can rank the corresponding structured query intended by the user. In order to better assess SUITS's ranking function for structured queries, we compared SUITS against two state-of-the-art approaches for keyword search, Effective[5] [9] and SPARK [45]. Both approaches treat tuple trees as search results of relational databases and order results using IR-style ranking functions.

In the experiments, we retrieved for each keyword query the top-30 structured queries returned by SUITS, and executed each structured query to retrieve at most 10 valid tuple trees as query results from the database (30 is sufficiently large to include most relevant results). We then ranked the tuple trees (query results) using the three approaches. As the ranking functions of SPARK and Effective were intentionally designed for tuple trees, we applied them directly. The parameters we chose for these two algorithms are the ones suggested by

[5] While we implemented all normalizations proposed, we did not implement phrase or schema-term support.

the original papers. As SUITS's ranking function is designed for structured queries, it does not consider actual results, and thus uses less information in ranking. For SUITS, we set the score of each tuple tree to the score of its corresponding structured query, and ranked the tuple trees using the query scores. The parameters we used for SUITS are p_1=10, p_2=0.5 and p_3=8.

For measuring the effectiveness of ranking, we employ the reciprocal rank. Given a query, the reciprocal rank is the inverse of the rank of the first correct answer in the result list. Mean reciprocal rank is an average reciprocal rank over a set of queries.

Fig. 10 and Fig. 11 give the mean reciprocal rank for the three approaches on both datasets. In addition, the figures show the mean reciprocal ranks of SUITS when not using one of the three factors, i.e. AC, TC and SER, in the ranking function, to measure the importance of these factors.

We can see from the figures that SUITS and Effective exhibit comparable effectiveness on IMDB, whereas SPARK performs slightly worse. On the Lyrics dataset, SUITS outperforms both Effective and SPARK. We observed different ranking behaviour for the tested approaches. Effective preferred long results, i.e. those containing many tuples with many query terms. However, we also saw that if the query terms do not appear in the right places, those results are not necessarily relevant to the queries. In contrast, SPARK preferred short trees. Although SPARK's algorithm includes a factor adapted directly from the IR literature [45] to enforce term completeness, i.e. all query terms being contained by a result, it seems that this factor does not work very well for our structured data, especially for the Lyrics dataset. The SUITS ranking algorithm was able to rank the best result in a very high position for most cases. As SUITS's ranking function works on the query level, without knowing the actual results, it sometimes may fail to give correct estimations of some factors. For example, SUITS sometimes scored the AC factor too high, because the actual size of an attribute in a particular tuple-tree can be much longer than the average size of the attribute in the database. Even without knowing the actual tuples, though, SUITS still achieved better effectiveness than the other two approaches. This implies that information about schema and data structure, such as AC and SER, can be very effective in ranking search results.

Fig. 10 and Fig. 11 also show that the three factors used in SUITS's ranking function are all important for effectively predicting structured queries intended by users. Term completeness (TC) seems to be the most effective, because most keyword queries issued by users assume AND semantics. Attribute completeness (AC) seems to be equally important. It works especially well for the IMDB dataset, because most IMDB queries refer to short attributes that represent real-world entities, such as movie title and actor names, while many Lyrics queries search for lyrics content. The number of expected results (SER) is less important than the other two factors, although it still improved rankings significantly.

To summarize, the experiments on IMDB and Lyrics showed that SUITS is able to predict the structure behind a user's keyword query with good precision.

3.6.3 Option Ranking Effectiveness

With a large database for which many query templates can be generated, it is more difficult for SUITS to predict the right structured query in one step. In these cases, the progressive query construction by choosing system suggested partial queries becomes important to obtain the user intended queries. Our second set of experiments thus evaluated the effectiveness of ranking query construction options in SUITS.

As discussed in during introducing the query construction options, when a user issues a keyword query, SUITS first suggests term-attribute combinations as query constructions options. When the user specifies more than one term-attribute combination, SUITS starts to suggest more complex partial queries. However, as the IMDB and Lyrics datasets use rather small schemas, most of the time users can already identify the intended structured query after selecting one or two term-attribute combinations. Therefore, our experiments only evaluated the ranking of term-attribute combinations. We considered the rankings for different terms separately. For each term in a keyword query we recorded the rank of the correct attribute it should appear in and calculated its reciprocal rank. Finally, we computed the mean reciprocal ranks for both Lyrics and IMDB datasets. As discussed, the ranking functions for term-attribute combinations include both database specific and query specific factors, i.e. attribute completeness (AC) and term selectivity (IDF). We evaluated the importance of these two factors as well.

Fig. 12 presents our experiment results using only AC, only IDF, as well as the proposed ranking function combining both factors. The results show that both factors are positively correlated to the ranking of the correct term-attribute combination. IDF seems to be the most effective factor, achieving 66-70% correctness. In comparison, attribute completeness had a smaller impact, achieving 48-58% correctness. The combination of both factors using the formula 6 achieved 82-89% correctness, which is a 16-19% improvement compared to using only IDF. The experiment shows that our attribute ranking function is able to predict correct term-attribute combination quite accurately.

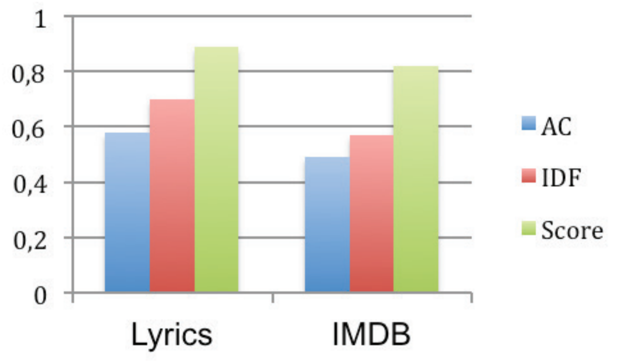


Fig. 12: Term-Attribute Ranking Parameters

3.6.4 Performance Study

Our third set of experiments studied the efficiency of SUITS. We used 2 measures in the evaluation: (1) number of SQL queries that need to be executed on the database in order to obtain top-k structured queries that have non-empty result-sets and (2) running time. We conducted experiments on both IMDB and Lyrics. For each dataset we picked queries from the query log that contain 2 to 6 terms, and grouped them by the number of terms. Then we selected 20 queries from each group. In the experiments, we measured the number of executed SQL queries and time required to obtain top-1, top-5 and all non-empty queries for each group of keyword queries.

As Fig. 13 (a) and (b) show, SUITS provides very good performance for queries containing 2 to 5 keywords. For almost all queries with less than 4 keywords, SUITS could retrieve the top-5 non-empty queries within 1 second. For most queries with 4 or 5 keywords, SUITS could retrieve the top-5 non-empty queries within 10 seconds. Running times were less satisfactory when the number of keywords increased to more than 5. However, as we observed in our query log, almost all real-world queries contain less than 6 keywords, which implies that SUITS is able to handle the majority of queries for large datasets such as IMDB and Lyrics. We also observe that SUITS displayed better performance on Lyrics than on IMDB, because Lyrics has a smaller schema that produces less structured queries to be tested.

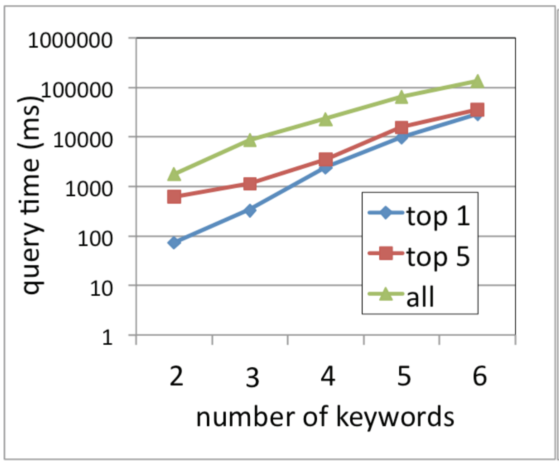
In Fig. 13, (c) and (d) show the average number of SQL queries SUITS had to execute in order to obtain the top-k non-empty structured queries. We can see that many structured queries generated by SUITS return empty results. For example, for an IMDB keyword query with 4 terms, SUITS had to try 100 structured queries on average to obtain the first non-empty one. This is the most expensive overhead in query processing of SUITS, and it is actually a common challenge for keyword search on relational databases.

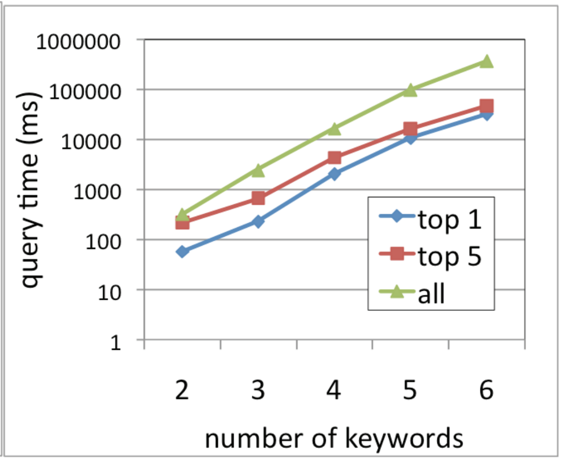
We can see from the charts in Fig. 13 that the cost of query processing grows fast with the number of terms in the keyword query. This increase is slowed down by using the query hierarchy of SUITS, as shown by our next set of experiments. The performance displayed in Fig. 13 shows that the growth of overhead slows down when the number of terms reaches 5.

Our last set of experiments evaluated how the optimization based on the query hierarchy can improve the performance of SUITS. We took the keyword queries used in the previous experiment and executed them on two versions of the SUITS system, where one version used the query hierarchy for optimization and the other did not. For each run we retrieved top-5 non-empty structured queries. We recorded the average number of executed SQL queries and the average query execution time for both versions.

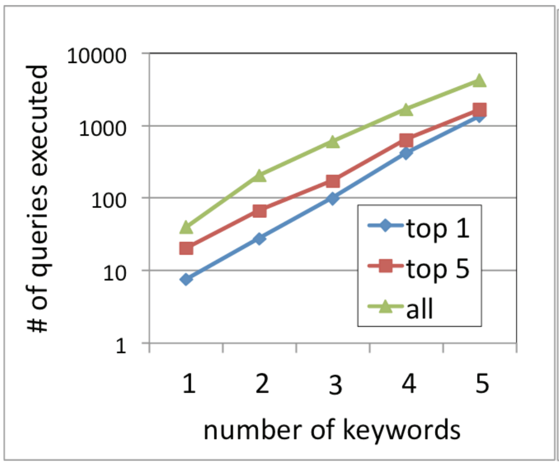
Table 1 and Table 2 compare the performance of the two versions of SUITS and give the percentage of improvement.

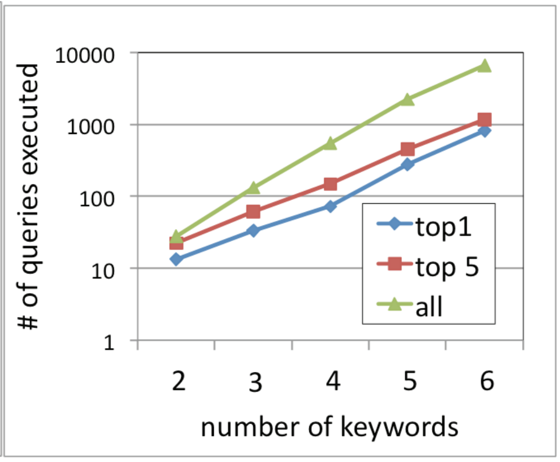
We observe that for two keyword queries query hierarchy did not improve performance and even slightly decreased it. This is because the structured queries generated using two keywords are much more likely to obtain non-empty result-sets, and by using the query hierarchy SUITS has to execute some extra queries at the bottom of the hierarchy. However, when the number of keywords grows, the improvement caused by utilizing the query hierarchy





(a) Execution Time, IMDB (b) Execution Time, Lyrics





(c) # of Queries, IMDB (d) # of Queries, Lyrics

Fig. 13: Performance Measurement

increases exponentially. As shown in Table 1, for an IMDB query with 4 terms, query hierarchy saves around 50% of time in query processing on average. When the number of terms grows to 6, query hierarchy speeds up the whole process by a factor of 8. The improvement using the query hierarchy for Lyrics was smaller, because the Lyrics dataset allows for much less query templates, which result in much smaller query hierarchies. However, the trend of improvement is similar to that of IMDB.

Table 1: Query Hierarchy Influence on the IMDB Database

Nr. terms	Nr. SQL queries			Time (ms.)		
	w/o H	w/s H	ratio	w/o H	w/s H	ratio
2	14.5	20.5	0.7	613.7	643.8	0.9
3	85.7	66.9	1.3	1339.5	1279.0	1.0
4	467.3	177.8	2.6	6888.3	3765.6	1.8
5	4590.0	643.0	7.1	67276.0	15519	4.3
6	26908	1670.1	16.1	312028.2	35809	8.7

Table 2: Query Hierarchy Influence on the Lyrics Database

Nr. terms	Nr. SQL queries			Time (ms.)		
	w/o H	w/s H	ratio	w/o H	w/s H	ratio
2	16.6	22.6	0.7	194.7	204.6	0.9
3	69.3	61.9	1.1	636.7	624.5	1.0
4	283.2	151.1	1.9	5143.8	4188.8	1.2
5	1237.4	458.3	2.7	24332.0	16813	1.4
6	4621.1	1223.9	3.8	95129.0	50022	1.9

Our experiments showed that SUITS offers good performance for short and medium keyword queries, and successfully employs query hierarchies to significantly reduce the overhead of query processing for medium and long keyword queries.

3.7 Related Work

In recent years, conducting keyword search over relational data and XML documents has been investigated extensively [7], [9], [11], [12], [14], [17], [45], [46], [49]. Most of this work focuses on how to improve the efficiency in processing keyword queries. As information relevant to a user query is distributed in different tables and attributes in the database, the search engine has to try many different ways to connect the information in order to obtain the optimal search results, which incurs a large overhead. As shown by our performance study, SUITS has to cope with the same problem as well. As SUITS uses a ranking function for structured queries, this eases the burden of top-k processing. SUITS' query hierarchy also helps to improve performance significantly. There is much less work on how to improve search quality in databases. Most existing approaches treat search results as virtual documents and ignore the rich semantics existing in their structures. SUITS goes a step further to allow users to express their intents through structured queries, so that they can utilize the structured information in the database to improve the capability and the quality of information seeking. Moreover, SUITS also shows that the structured information available in the database can enhance ranking significantly.

Mapping keyword queries to structured queries have recently been investigated in the areas of IR and the Semantic Web. In [15], the authors proposed to use structured queries to interpret users' intents in document retrieval. As its purpose is not constructing structured queries for databases, the queries are document centric and less general than SUITS' queries. In [16], [47], the authors proposed techniques for transforming keywords to structured semantic queries. However, the queries considered are quite limited. As they do not filter out the queries with empty results, users may have to choose from too many possible queries. Moreover, all these previous approaches do not allow incremental query constructions as SUITS does, which reduces their usability when confronted with big data schemas.

Database usability [50] is an issue that has been studied for many years. Natural Language Query Interfaces [51–53] for databases are intended to allow users to specify structured queries in human language. Although this provides certain flexibility for accessing a database, state-of-the-art natural language interfaces still require users to use terminology compatible with the database schema and form grammatically well formed sentences. SUITS offers users the same expressivity for structuring their queries, but much more flexibility, accepting simple keyword queries and requiring no a-priori schema knowledge. Query Auto-completion [54], [55] is a technique that helps users form appropriate structured queries by suggesting possible structures or terms based on the partial query the user has already entered. By using the suggestions, users can make correct database queries without complete knowledge about the schema. However, as this technique still requires users to form complete structured queries in order to access the database, it is less flexible than SUITS, which allows arbitrary keyword queries and multiple user-system interactions.

Recent work on XML retrieval [56], [57] and XML approximate queries [58], [59] aims to provide interfaces that allow users to access XML data with only partial schema knowledge. The expressivity of those interfaces is, however, bounded by users' schema knowledge. SUITS goes beyond this by allowing users to structure their queries on the fly using schema knowledge suggested a posteriori by the system.

3.8 Discussion

In this chapter we introduced SUITS, a novel interface allowing flexible access to text databases with little knowledge about database schema or formal query languages. SUITS lets users start with arbitrary keyword queries and then allows them to structure / refine them incrementally, following suggestions given by the system. In this way, the SUITS approach integrates the flexibility of keyword search with the expressivity of database queries. We presented the architecture and interface of the SUITS system, implementation of all its components, as well as several new and important optimizations. We conducted extensive experiments using real-world data sets and query loads which showed performance and practicality of our approach.

From the query process perspective (c.f. Fig. 3, page 11), the ranking-based SUITS approach nicely supports the *need articulation* and *query formulation phase.* The offered query construction options offer limited *query reformulation* support. The main limitation of these options is they only support 1:1 joins through the RDBMS schema. Trees, or even more complicated structures are not constructible. Therefore, not the full possible space of queries is reachable, and not all possible user intended queries can be directly specified. While ranking helps alleviating this issue for the most commonly intended results, reaching less common ones becomes increasingly difficult, as it requires increasingly more user interface interactions (e.g. scrolling in the result list) to reach the intended answers.

4. QUICK: QUery Intent Constructor for Keywords

With QUICK[6] we introduce a system for structured web data that directly supports all query phases (c.f. Fig. 14). This is a big improvement from the previously discussed SUITS system, as QUICK guides the user through an incremental construction process "quickly" to the desired query.

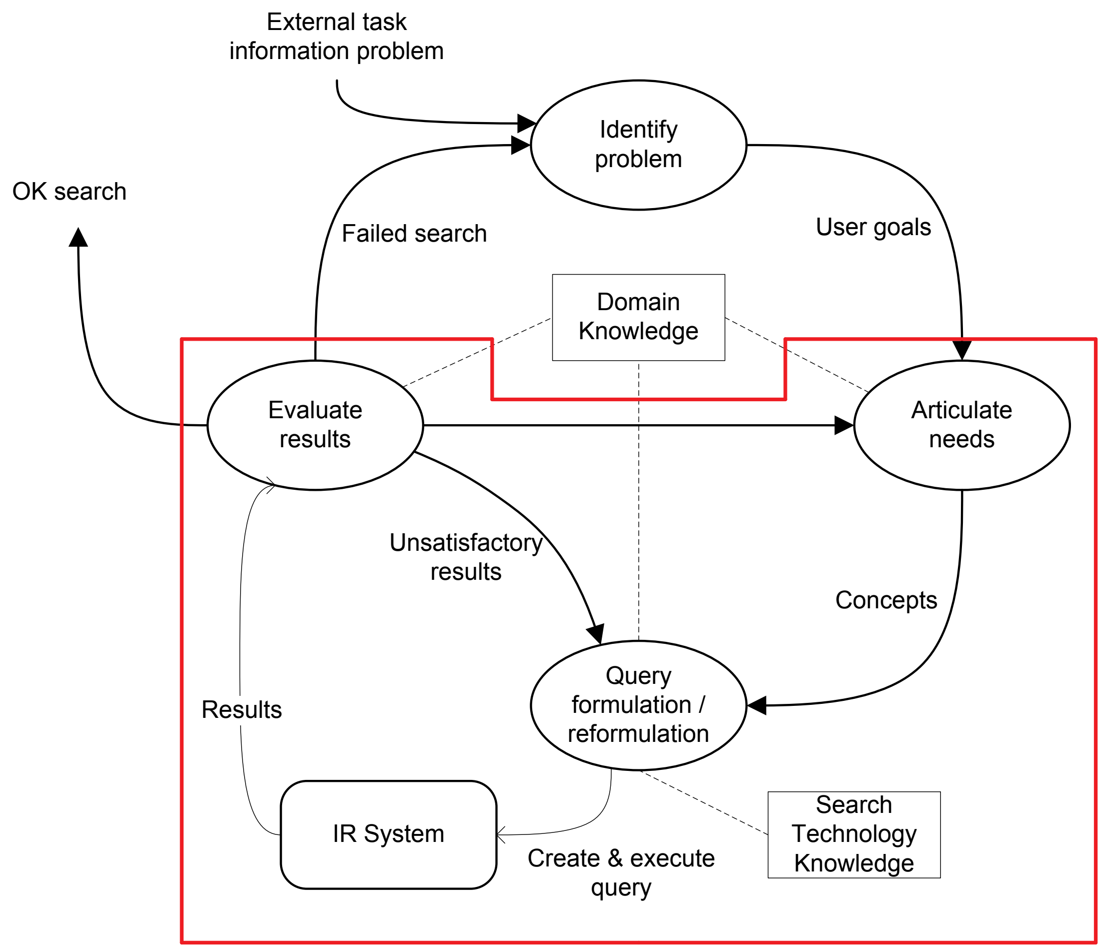


Fig. 14: QUICK support of the query process

It is mostly suitable for environments with very diverse and specific query needs, where the user has basic knowledge of the underlying data's domain. Still, specific knowledge of details of the ontology, or proficiency in a query

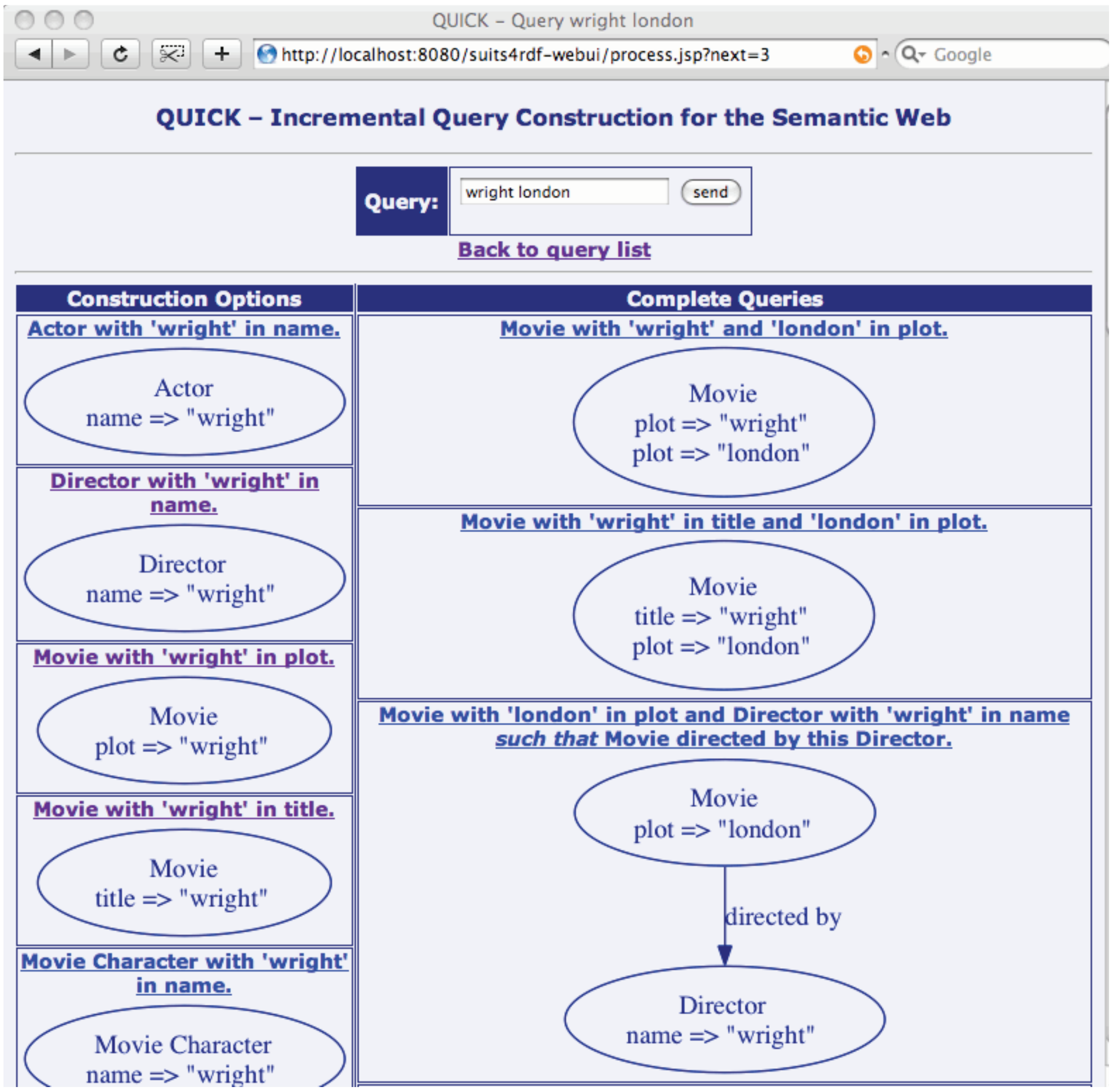
[6] QUery Intent Constructor for Keywords

language are not needed. In that way, QUICK combines the convenience of keyword search with the expressivity of structured, semantic, queries.

This chapter 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.

Relevant publications for this chapter:

- Gideon Zenz, Xuan Zhou, Enrico Minack, Wolf Siberski, Wolfgang Nejdl, From keywords to semantic queries - Incremental query construction on the semantic web, J. Web Sem. 7, 3 (2009), 166-176 (Most Cited Article Award 2006-2010)
- Gideon Zenz, Xuan Zhou, Enrico Minack, Wolf Siberski, Wolfgang Nejdl, Interactive Query Construction for Keyword Search on the Semantic Web, chapter in book “Semantic Search over the Web”, Data-Centric Systems and Applications, Springer 2012, pp 109-130



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. 15: QUICK User Interface

4.1 QUICK Overview

As illustrated in Fig. 15, 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). Suppose a user looks for a movie set in *London* and directed by *Egdar Wright*[7]. 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*.

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

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 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.

4.2 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.

4.2.1 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 Definition 5 simple, we assume explicit rdf:type declarations of all concepts.

Definition 5: 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, $P = \{p \mid \exists s, o : (s\ p\ o) \in G\}$ 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 6: 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, $EC = \{(c_1, c_2, p) \mid \exists r_1, r_2 \in R, p \in P :$ $(r_1, p, r_2) \in G \wedge (r_1 \text{ rdf:type } c_1) \in G \wedge (r_2 \text{ rdf:type } c_2) \in G\}$, and $EL = \{(c_1, p) \mid$ $\exists r_1 \in R, p \in P, l \in L : (r_1, p, l) \in G \wedge (r_1 \text{ rdf:type } c_1) \in G\}$

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.

4.2.2 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 our example queries 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 7: 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.

Fig. 16 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 7. 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.

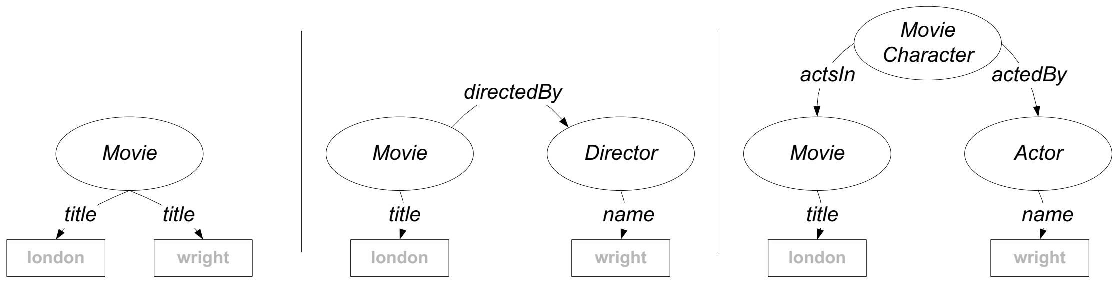


Fig. 16: Sample query templates for the IMDB schema; the terms in gray represent instantiations of these templates to semantic queries for ‘wright london’

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.

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. 16 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 8: 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 $SQ = \{sq \mid \exists \theta, T : (kq, T, \theta) \text{ is a query template}\}$.

In our model, each term is bound separately 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. 16. 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 Fig. 16, 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 the evaluation, a semantic query corresponds to a combination of SPARQL triple pattern expressions, which can be directly executed on an RDF store.

4.2.3 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 9: Sub-query, Partial query
$sa = (q_a, T_a, \theta_a)$ is a subquery 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.

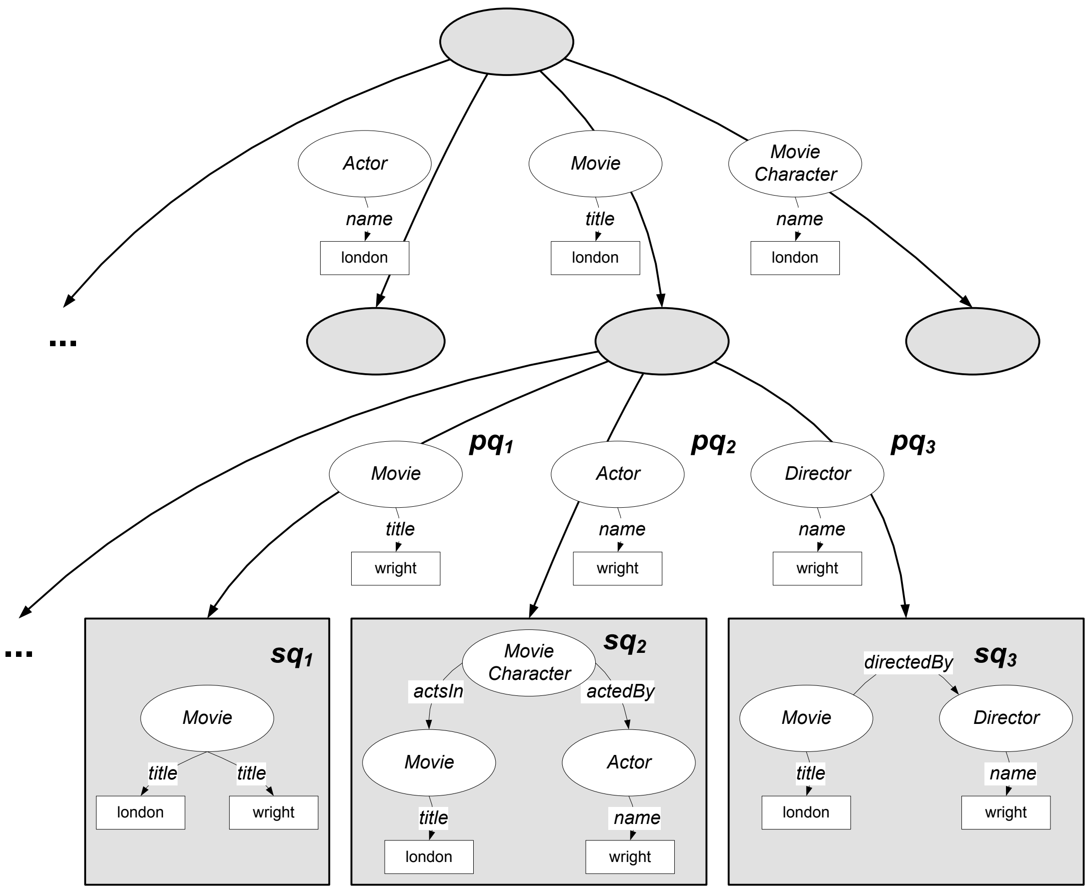


Fig. 17: Part of a query guide for '*wright london*'

For example, in Fig. 17, 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 modelled as a *Query Construction Graph* (QCG), as illustrated in Fig. 17. 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 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.

Which leads us to:

Definition 10: 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 $V \subset SQ^*$, $E \subset \{(v_1, v_2, p) \mid v_1, v_2 \in V,\ v_2 \subset v_1,\ p \in PQ, p \text{ subsumes } v_2\}$ and $\forall v \in V, |v| > 1 : v = \left\{\bigcup v_i \mid \exists p : (v, v_i, p) \in E\right\}$

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

Definition 11: 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 Fig. 17, 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 12: 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: $cost(cp) = \sum_{v \in V'} \left| \{p : (v, v_1, p) \in E\} \right|$, and $cost(QCG) = \max(cost(cp) : cp)$.

Definition 13: 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 who satisfies Definition 13 leads the user to the intended query with minimal interactions. In the following section, we show how to compute such guides efficiently.

4.3 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 the previous. 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 1: 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.

4.3.1 Straightforward Guide Generation

Lemma 1 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 2.

```
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
```

Algorithm 2: Straightforward Query Guide Generation

According to Lemma 1, this algorithm always finds a minimum query guide. However, it relies on the solution of the *SetCover* problem, which is NP-complete [60]. 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 14: 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 3: 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 1, if we solve $minConstructionGraph(U_S \cup A_S, U_P \cup A_P)$, we solve $minSetCover(U, S)$.

4.3.2 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 3 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.

```
Input: partial queries P, semantic queries S
Output: query guide G

if |S|=1 then
    return S ;
end
G := $\varnothing$
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
```

Algorithm 3: Incremental Greedy Query Guide Generation

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 15.

Definition 15: 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 show that it performs very well if the number of keywords is within realistic limits

4.4 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 RDFstore to retrieve the results. The conversion process is straightforward: 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.

4.5 Experimental Evaluation

We implemented the QUICK system using Java. The implementation uses Sesame2 [61] as RDF Store and the inverted index provided by Lucene Sail to facilitate semantic query generation. 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.

4.5.1 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, [9]) 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.

4.5.2 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:

(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 3 and Fig. 18 and Fig. 19. Table 3 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

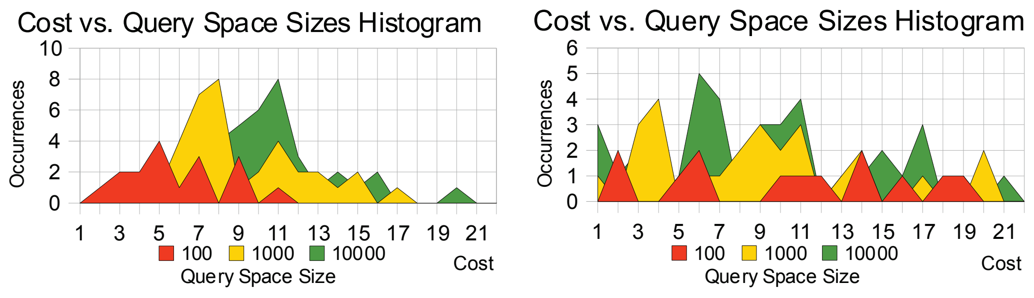


Fig. 18: Query construction cost histograms for IMDB (left) and Lyrics (right) for three different query space sizes

purposes, we applied the SPARK [16] 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 3, the maximum number of options a user needs to examine until obtaining the desired semantic query is always low (33 for IMDB resp. 22 for Lyrics). On average, only 9 resp. 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.

No. of	Init time (ms.)		Response time	
	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 3: Effectiveness of QUICK for IMDB and Lyrics

Fig. 18 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. 19, 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.

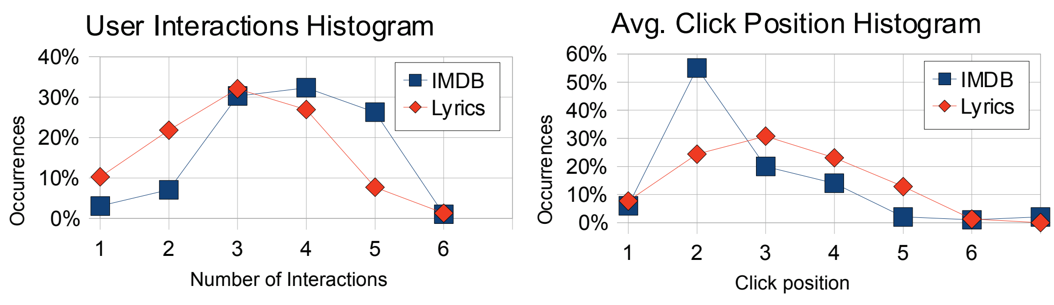


Fig. 19: Histograms of the number of interactions and average click position

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.

In implementing QUICK, we focused on efficient algorithms for query guide 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 [8] and by introducing special indexes.

4.5.3 Quality of the Greedy Approach

To evaluate the quality of the query guides generated by the greedy algorithm, we compared it against the straightforward algorithm. 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 tuneable 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.

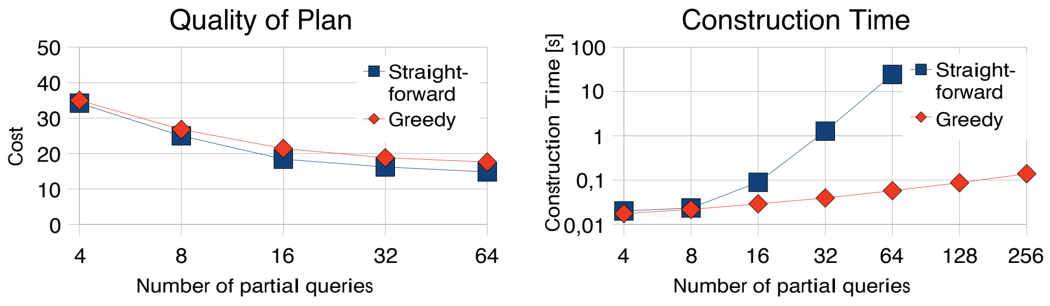


Fig. 20: Varying number of partial queries

As shown in the left-hand side of Fig. 20, 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, which is consistent with our complexity analysis.

In the second set of experiments, we varied *n_complete* between 32 and 256, fixed *n_partial* to 32, and set *coverage* to ¼ of *n_complete*. The results are shown in Fig. 21.

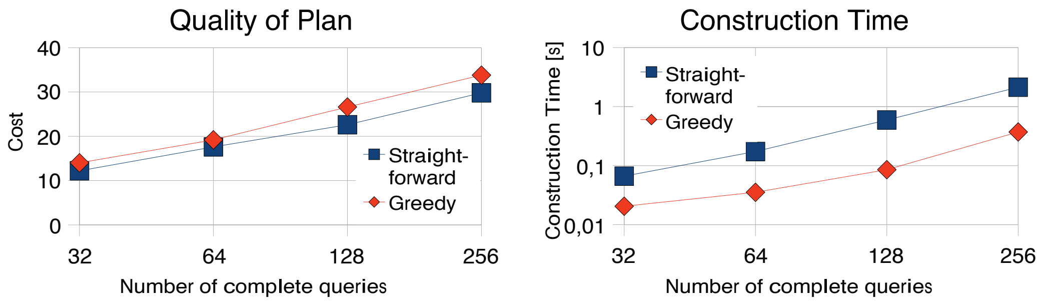


Fig. 21: Varying number of semantic queries

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 our complexity analysis.

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. 22 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 15.

Fortunately, in real data, most partial queries have a relatively small coverage (less than 20%), where this effect is less noticeable, justifying our assumptions.

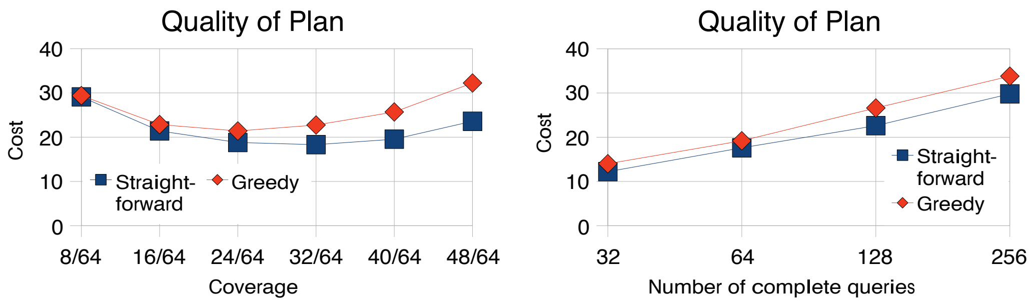


Fig. 22: Varying coverage of partial queries

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.

4.6 Discussion

In this chapter, 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 polynomial runtime behaviour, 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, and therefore supports the complete query process, as can be seen in Fig. 14, on page 42. 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:

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, making use of concept hierarchies to aggregate query construction options is desirable. 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.

(i) While the current approach allows reaching all possible intended queries in the query space, the effort for every such query is the same. While this is desirable for rarely used or complicated queries, helping users with often asked queries by introducing a ranking mechanism would help. A useful combination of the QUICK query construction and the SUITS ranking approach is therefore desirable.

(ii) A user study to verify the suitability for non-expert users and its effectiveness on a larger scale would help assessing the benefits of this system.

(iii) The current system still requires domain knowledge in order to make good use of the system. Therefore, using background knowledge for guiding the user would be helpful.

5. A Facetted Query Interface for Customer Service

In this chapter, we present an interface for facetted access to customer service data, which additionally supports the user by providing domain knowledge. As already discussed in general Chapter 1, and as detailed in Fig. 23, our facetted interface supports specifically the *query formulation/reformulation phase*, as well as the *result evaluation phase*. Additionally, we make use of a lexical knowledge base for providing relevant domain knowledge, which helps the user by automatically matching semantically similar service documents, even if different vocabulary is employed, allowing for easy access even for the less proficient user.

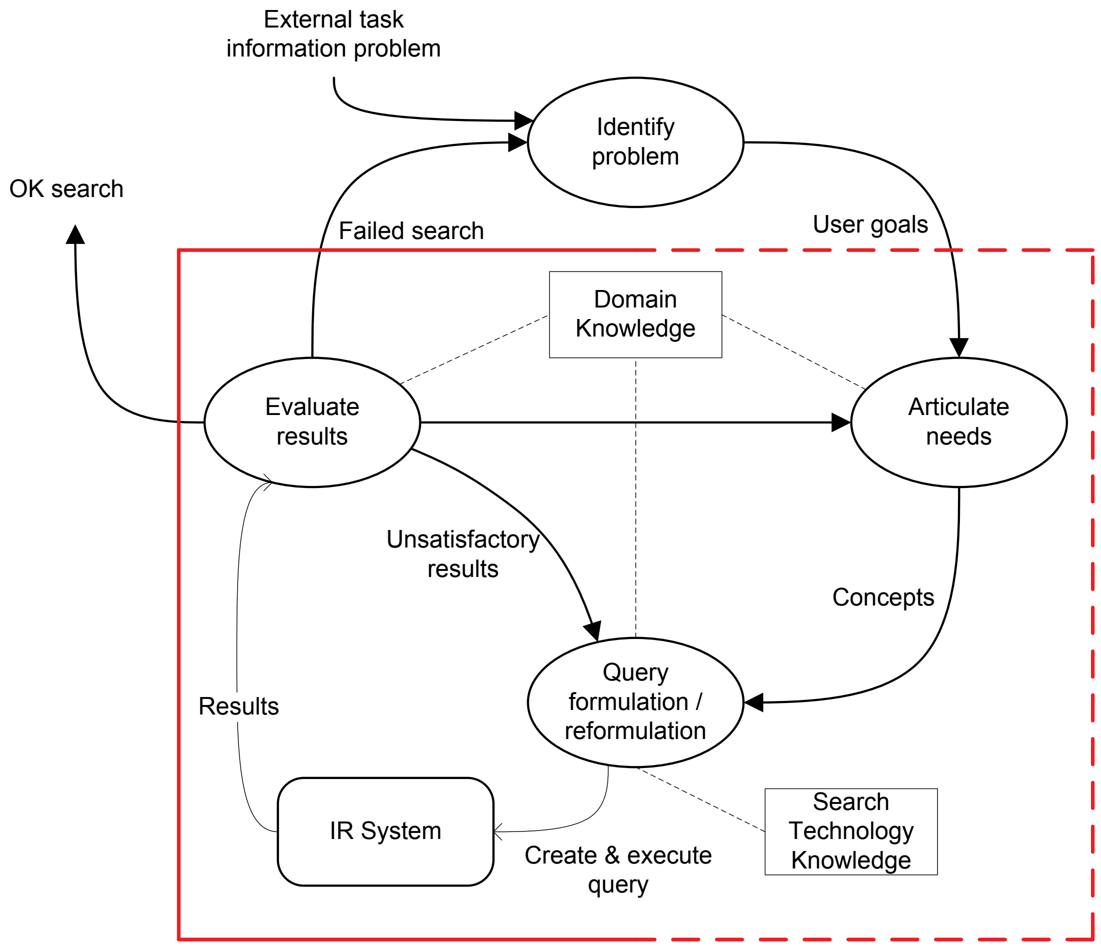


Fig. 23: Facetted Interface support of the query process

Devising query applications for business use pose specific challenges on the usability of a query system. The presented system specifically aims for customer service needs. As business users are usually are focused in their information need, we stressed on the query reformulation part, amending this refinement facility by providing relevant, and focused, domain knowledge, through a knowledge base specifically tailored for technical and business needs.

Relevant publications for this chapter:

- Kerstin Denecke, Gideon Zenz, Wladimir Krasnov, Semantic Web Technologies to Improve Customer Service, International Semantic Web Conference 2009, October 2009, Washington, USA

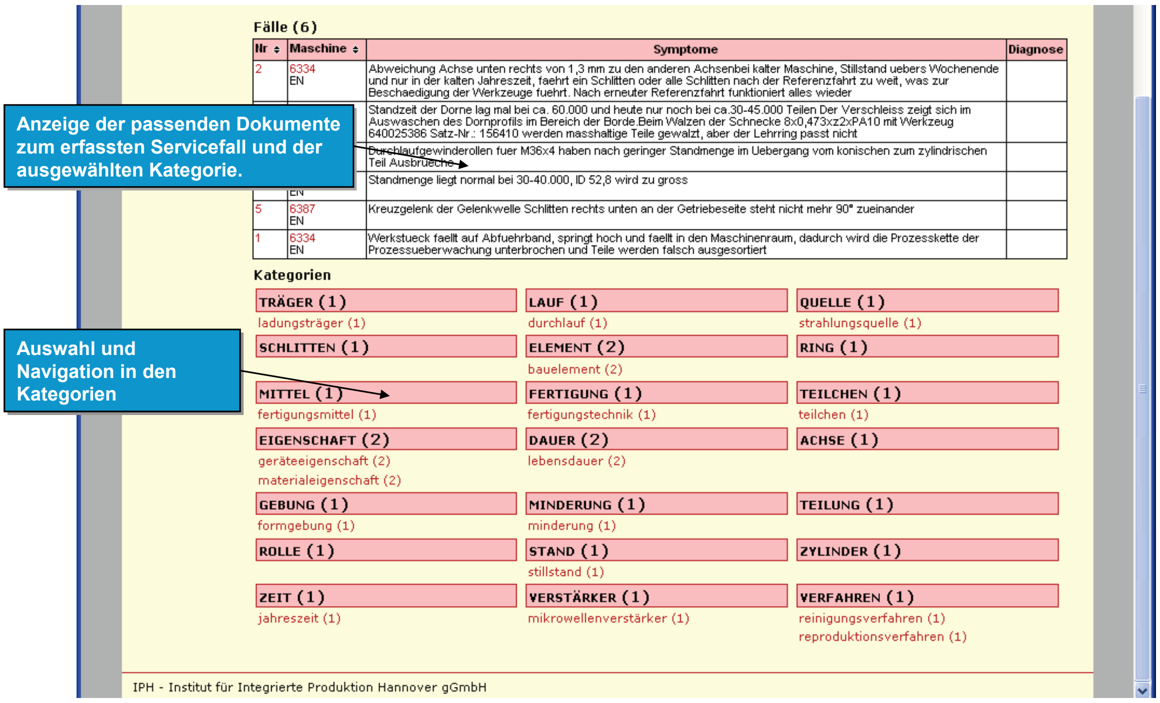
5.1 Devising an Interface for Business Use

It is crucial for service departments to find relevant information that helps to answer customer requests in time to avoid long process. But, relevant knowledge is often stored in non-retrievable form, which causes huge difficulties in knowledge discovery for employees. Further, a long learning period is needed for new employees to go into the depth with existing (company internal) knowledge and to learn how to formulate the “right” request for finding relevant information. Even with existing retrieval systems, formulating the "right" request that results in an appropriate number of suited results is difficult, as there are many different ways to describe a problem with natural language. Service messages are documented in natural language and are therefore stored in unstandardized, unstructured manner. To allow efficient access and reuse of this data, we describe an approach to store service messages in a standardized way and to structure the content of a text collection hierarchically into facets. By visualizing and navigating this hierarchy, a user is able to specify constraints on the items selected from the repository. The facets help to discover similar service messages even if they use different terms to describe similar issues. Furthermore, the service message database can be browsed in an intuitive manner. Currently, two methods are quite popular to group search results appropriately, namely clustering and faceted categorization. The Latent Dirichlet Allocation (LDA, [62]) algorithm generates a probabilistic model to describe the content of texts and clusters them based on this model. Scatter/Gather offers a navigation system based on document clustering [63]. In faceted classification, a set of category hierarchies is built, rather than a single large category hierarchy [64]. These capture the different facets, i.e., dimensions or features, relevant to a collection. In [65], an unsupervised approach for facet extraction is presented relying upon WordNet and Wikipedia, to identify useful facet terms. The approach presented in this paper can mainly be seen as a modification and extension of the Castanet algorithm presented by Stoica et al. [66]. The castanet algorithm works on the textual description of items like images or documents. It then generates automatically hierarchical faceted metadata from text based on WordNet and WordNet Domains within five major steps, c.f. Algorithm 4. We modified and adapted the original algorithm to fit the given scenario.

```
1. Select target terms from textual descriptions of information items.
2. Build the Core Tree:
    a. For each term, if the term is unambiguous (see below), add its
       Wordnet synset's IS-A path to the Core Tree.
    b. Increment the counts for each node in the synset's path with the
       number of documents in which the target term appears.
3. Augment the Core Tree with the remaining terms' paths:
    a. For each candidate IS-A path for the ambiguous term, choose the
       path for which there is the most document representation in the Core
       Tree.
4. Compress the augmented tree.
5. Remove top-level categories, yielding a set of facet hierarchies.
```

Algorithm 4: The Castanet Algorithm as of [66]

In particular, this algorithm is customized to the domain of mechanical engineering by using a domain specific thesaurus. Furthermore, a more sophisticated text pre-processing comprising stemming of words and resolution of compound nouns extends it. The Castanet approach only considers nouns to generate the structure while our extension considers all relevant words. Finally, the algorithm is applied and tested on a collection of service messages of the engineering domain.


Anzeige der passenden Dokumente zum erfassten Servicefall und der ausgewählten Kategorie.
Auswahl und Navigation in den Kategorien

Fälle (6)

Nr	Maschine	Symptome	Diagnose
2	6334 EN	Abweichung Achse unten rechts von 1,3 mm zu den anderen Achsenbei kalter Maschine, Stillstand uebers Wochenende und nur in der kalten Jahreszeit, faehrt ein Schlitten oder alle Schlitten nach der Referenzfahrt zu weit, was zur Beschaedigung der Werkzeuge fuehrt. Nach erneuter Referenzfahrt funktioniert alles wieder	
		Standzeit der Dorne lag mal bei ca. 60.000 und heute nur noch bei ca.30-45.000 Teilen Der Verschleiss zeigt sich im Auswaschen des Dornprofils im Bereich der Borde.Beim Walzen der Schnecke 8x0,473xz2xPA10 mit Werkzeug 640025386 Satz-Nr.: 156410 werden masshaltige Teile gewalzt, aber der Lehrring passt nicht	
		Durchlaufgewinderollen fuer M36x4 haben nach geringer Standmenge im Uebergang vom konischen zum zylindrischen Teil Ausbrueche	
	EN	Standmenge liegt normal bei 30-40.000, ID 52,8 wird zu gross	
5	6387 EN	Kreuzgelenk der Gelenkwelle Schlitten rechts unten an der Getriebeseite steht nicht mehr 90° zueinander	
1	6334 EN	Werkstueck faellt auf Abfuehrband, springt hoch und faellt in den Maschinenraum, dadurch wird die Prozesskette der Prozessueberwachung unterbrochen und Teile werden falsch ausgesortiert	

Kategorien

TRÄGER (1)	LAUF (1)	QUELLE (1)
ladungsträger (1)	durchlauf (1)	strahlungsquelle (1)
SCHLITTEN (1)	ELEMENT (2)	RING (1)
	bauelement (2)	
MITTEL (1)	FERTIGUNG (1)	TEILCHEN (1)
fertigungsmittel (1)	fertigungstechnik (1)	teilchen (1)
EIGENSCHAFT (2)	DAUER (2)	ACHSE (1)
geräteeigenschaft (2)	lebensdauer (2)	
materialeigenschaft (2)		
GEBUNG (1)	MINDERUNG (1)	TEILUNG (1)
formgebung (1)	minderung (1)	
ROLLE (1)	STAND (1)	ZYLINDER (1)
	stillstand (1)	
ZEIT (1)	VERSTÄRKER (1)	VERFAHREN (1)
jahreszeit (1)	mikrowellenverstärker (1)	reinigungsverfahren (1)
		reproduktionsverfahren (1)

IPH - Institut für Integrierte Produktion Hannover gGmbH

Fig. 24: Facetted Semantic Interface for Retrieving Customer Service Documents

5.2 Making Data Semantic

Similar to the CastaNet Algorithm, our method requires a lexical knowledge base. Since our system targets at processing documents of the engineering domain, a domain unspecific lexical resource such as WordNet or GermaNet is unsuited since relevant domain-specific terms are missing. Therefore, we decided to base our algorithm to the FIZ Thesaurus.

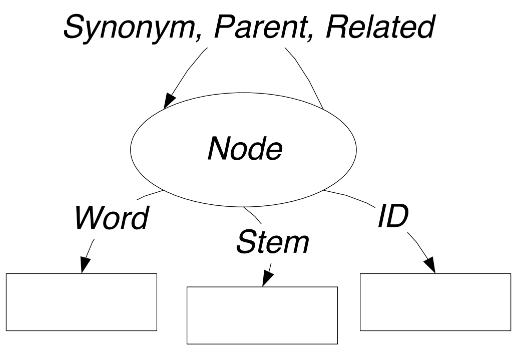


Fig. 25: Schema of the FIZ Thesaurus

The FIZ thesaurus (Fig. 25) provides 58,300 technical terms in German and 63,500 technical terms in English, which are related through links indicating synonymy, hierarchy and semantic relation. The thesaurus covers the vocabulary of different engineering subfields and was originally created to extend online literature repositories and to improve the document retrieval.

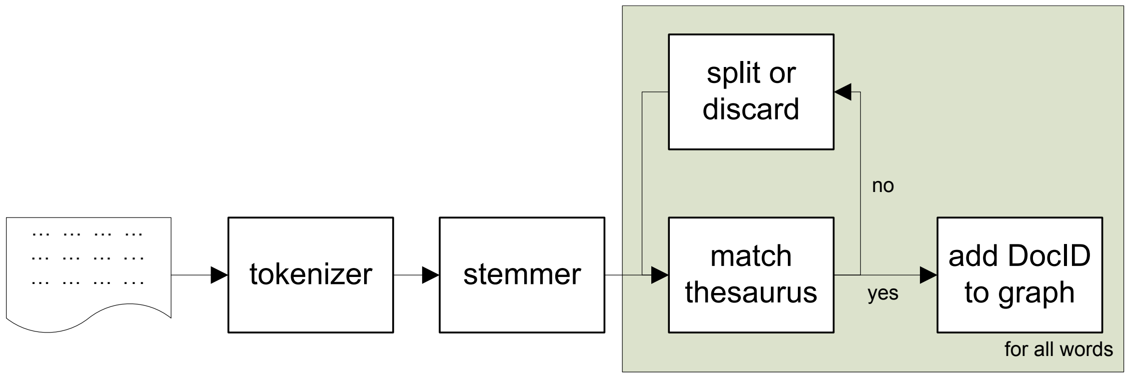


Fig. 27: Facet generating algorithm

For our algorithms, the thesaurus is stored into RDF, which allows for its manual extension and its efficient use for creating hierarchies. Each word is represented by a node with a unique id, the actual word, and the stemmed version of the word. Nodes can be connected as synonyms, parents, or other semantic relations. A term A is considered as parent of term B if A is a generic term to B.

Natural language text, in our case service messages of the engineering domain, is mapped to concepts of this thesaurus. In this way, the service requests become comparable and automatically interpretable. Relations between terms as provided by the thesaurus are then used to generate document hierarchies. The hierarchical

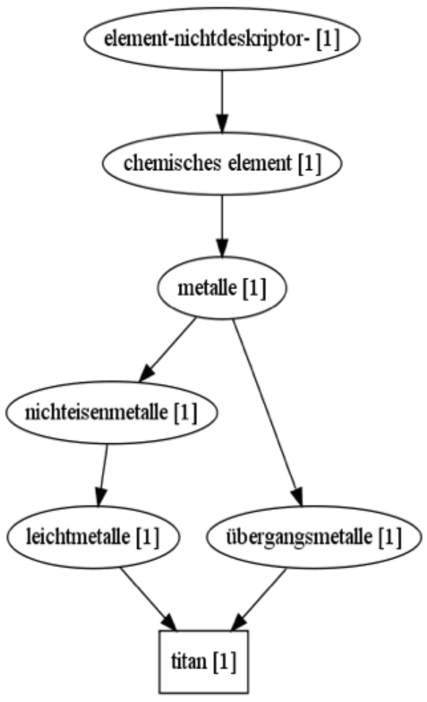


Fig. 26: Graph generated for the term *Titan*

facets allow navigation through search results and improved document retrieval.

To represent a document through categories of the underlying thesaurus the following processing steps are conducted, as can be seen in Fig. 27. First, the document is pre-processed, i.e. special characters and other punctuation marks are removed and the words are split into their linguistic segments. This is necessary to be able to identify compound nouns, which occur very often in service messages due to the reduced length of these messages and the compact language. In addition, the words are stemmed using the Stemmer presented in [67] and looked up in the FIZ thesaurus. The matched words are assembled in a domain specific document term list, which in turn is extended by synonyms of the terms that are collected using the synonym relationship provided by the thesaurus.

Finally, a directed graph is constructed where the terms of the expanded term list are the leaves, c.f. Fig. 26. Starting from the term, the hyperonomy relations provided by the thesaurus are iteratively used for constructing a hierarchical tree. In more detail, for each term (leaf) parent nodes are collected from the thesaurus and are inserted into the graph. The result is a connected graph that resembles a hierarchical semantic representation of the document.

Furthermore, all paths are attributed with a count that specifies the usage frequency of each concept. This frequency information is used to select facets as categories for the document under consideration. In particular, the most probable generic concepts are collected for categorizing the document.

5.3 Evaluation

The introduced method is evaluated on service messages of the mechanical engineering domain. The collection consists of 4,884 documents written in German. In average, each message comprises 20-30 words and is a short statement, where a hotline employee summarizes the error description or request of a customer. From this collection, 200 service messages are randomly selected. Four persons were involved in the evaluation. For each test document, the evaluating person was confronted with the system generated hierarchical graph and had to select the best-suited categories describing the document from this graph.

The categories chosen by the algorithm remained hidden to the evaluators. In particular, the evaluation examines whether the system assigns the test documents to the same categories as the evaluators.

The system achieved a precision of 0.93 and a recall of 0.86. Furthermore, the time to categorize manually was measured. The evaluators needed in average 2.5 minutes to select relevant categories, obviously due to the complexity of the domain and the shortness of the service message. Compared to this, the proposed method takes only 0.5 seconds for classifying a service message. The hierarchical graphs for the complete data set were generated in 130 seconds on a 1.6 GHz Pentium Processor with 2 GB RAM. Since the presented system performs very well in terms of accuracy and time, we can conclude, that this method can help to reduce the time for categorization significantly.

Errors occur when abbreviations were used in a text or terms could not be found in the lexical database. A manual or semi-automatically extension of the lexical resource with relevant abbreviations could help to improve the system's accuracy. The system also fails when confronted with terms with writing errors. Technologies for error correction or soundex- or metaphone technologies could help to deal with this problem.

In contrast to existing algorithms that allow for the generation of hierarchical facets for general texts [66], we showed how domain-specific knowledge could be exploited efficiently to create a faceted representation of technical texts. Our representation contains only technical terms, which is crucial for document retrieval purposes. Through the pre-processing of the natural language text messages by means of stemming and analysis of compound nouns, the system is able to identify morphological variants and to identify terms even if they are hidden within compound nouns. We tested the algorithm on German texts only since similar texts in English were unavailable in this project. Since the FIZ thesaurus already contains terms in English, the system can be also applied to this language. An adaption of the linguistic pre-processing is required.

5.4 Discussion

In this chapter, an approach to generate hierarchical facets to highly specialized texts of the engineering domain was introduced. We showed that domain-specific knowledge could be successfully used to generate such facet representation. By exchanging the domain knowledge, the approach can be transferred to other domains.

In contrast to the previous approaches, although giving users further support by providing domain knowledge through the lexical knowledge base, this system does not support structured data. Furthermore, the knowledge base has to be pre-produced and suitable for the document collection. This poses serious challenges on prospectus users, as generating such a knowledge base is time consuming and error prone. In the next chapter, we will discuss methods for automatically generating such domain knowledge needed to easily navigate document collections using polysemous vocabulary.

6. Querying with Generated Domain Knowledge

In this chapter, we focus on realizing a query system that automatically generates relevant domain knowledge for given data, in order to help users with relevant hints, and aid the understanding of the underlying corpus. As can be seen in Fig. 28, the application fully supports the *need articulation* and *query formulation phase*. Although it currently offers no direct support for *reformulations* or structured data, in contrast to the method discussed in the previous chapter, it automatically generates the needed background domain knowledge to provide the user easily with a deeper understanding of the underlying data.

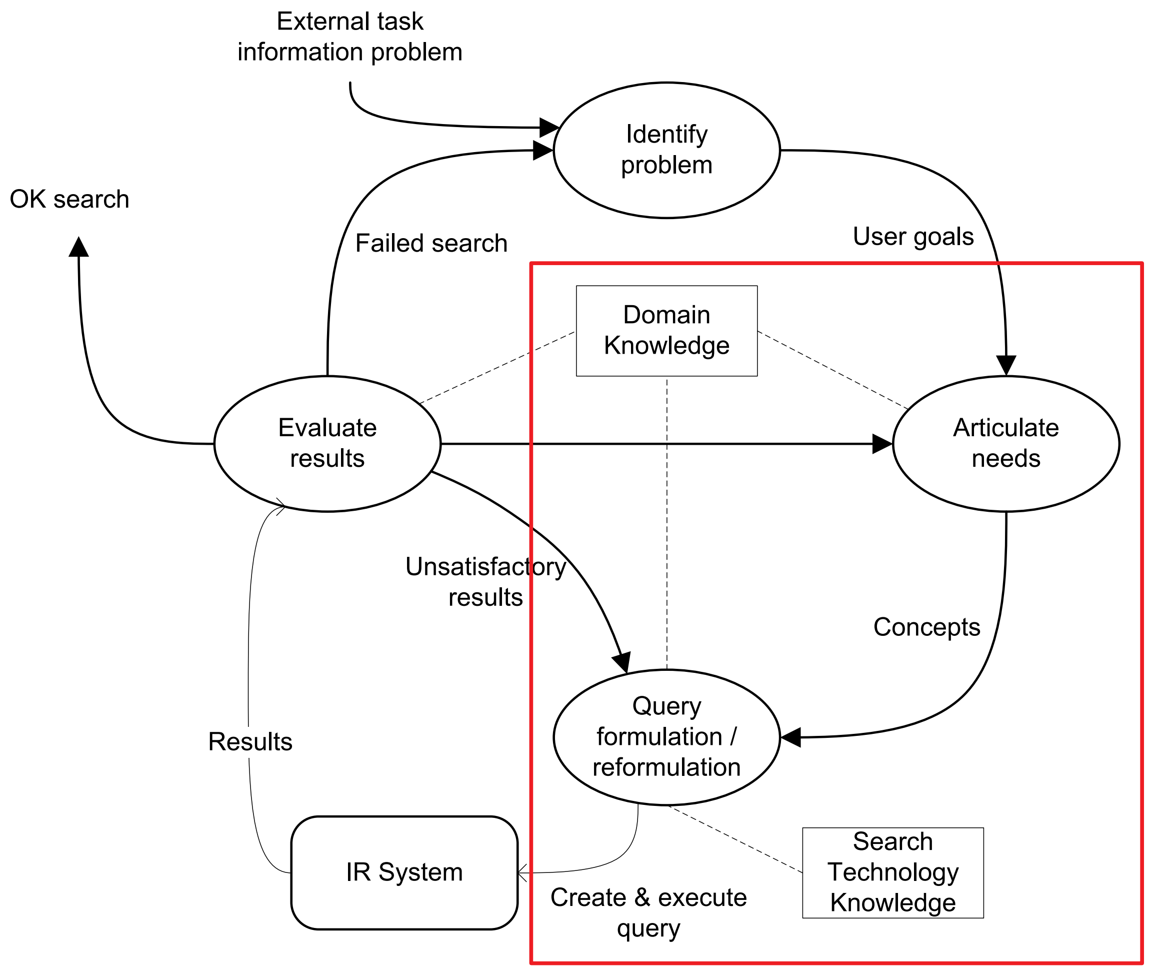


Fig. 28: Mobile Interface support of the query process

Furthermore, we want to provide an application that satisfies the ever-growing demand for immersive, just-in-time information needs on web data. Mobile and smart devices provide for excellent facilities for giving for such activities. Given the limited interaction possibilities and screen sizes, as well as being mobile usually implies an urgent information need, efficient and effective means to query are important. The usefulness and quality of this application is evaluated with a user study.

Relevant publications for this chapter:

- Nina Tahmasebi, Gideon Zenz, Tereza Iofciu, Thomas Risse, Terminology Evolution Module for Web Archives in the LiWA Context, In Proc. of 10th International Web Archiving Workshop in conjunction with iPRES in Vienna, Austria, 2010
- Gideon Zenz, Nina Tahmasebi, Thomas Risse, Language Evolution On The Go, SAME 2010 - 3rd International Workshop on Semantic Ambient Media Experience (NAMU Series) November, 10th-12th November 2010 in conjunction with AmI-10 in Malaga, Spain
- Gideon Zenz, Nina Tahmasebi, Thomas Risse, Towards mobile language evolution exploitation, J Multimed Tools Appl., Springer, 2012

6.1 Introduction

The usage and meaning of terminology is changing throughout time. Depending on the point in time considered, different connotations will be relevant, which is also important to consider when searching for information. As example, consider the term *anthrax*, which can be a rather famous band, or, as mostly only experts used to know, a disease. This unexpectedly changed in 2001, when letters containing anthrax were mailed in the U.S. All of the sudden, the disease relation of the term became well known to the general public; we present a terminology evolution application, which aims to detect such connections, and educates the user how and when meanings have shifted.

In order to automatically detect different word senses or meanings given a collection of terms, we employ word sense discrimination as pre-processing step. Word sense discrimination allows dividing term collections into consistent groups of terms.

In order to allow users to use this knowledge in an ambient and intuitive way at the very point in time it is needed, we devised two mobile user interfaces. One is more aimed for professional users, and allows permeating the knowledge space in depth, but also requires more sophisticated knowledge of languages and web applications. The second user interface is more concise and aimed towards fast, every-day usage.

The following paragraphs will give an overview on the state of the art, followed by a description of our methods to derive the perception of a term in a given corpus. Thereafter, the two user-interfaces are discussed and evaluated by a user study. Finally, we conclude this chapter and discuss future work.

6.2 Related Work

Detecting word senses inherent in a text corpus is the aim of word sense discrimination. Several methods based on co-occurrence analysis and clustering have been studied in [68–71]. Seminal in our context is the work of Schütze [71], who discusses this idea in the context of group discrimination. A word space is constructed by mapping ambiguous words from a training set, using cosine similarity as metric. Using context vectors constructed from terms occurring in context, this set is clustered into a set of coherent clusters. The representation of a sense is the centroid of a cluster.

An alternative approach was introduced by Lin [72], who employs a word similarity measure to automatically create a thesaurus. The disadvantage of this approach is the use of hard clustering, which lacks the flexibility needed to cover ambiguity and polysemy of terms. A further clustering algorithm is proposed by Pantel [73], named Clustering By Committee. It consistently outperforms previous algorithms like Buckshot, K-means, and Average Link in both recall and precision. A further contribution is a method for automatically evaluating the output using WordNet, which has seen broad usage [74], [75].

6.3 From Text to Ambient Perception

In order to arrive at a user understandable representation of word senses, we employ a processing pipeline. It consists of three major steps, natural language processing for extracting relevant terms, co-occurrence graph creation, and graph clustering.

As discussed, soft clustering is the more appropriate methodology to deal with the ambiguous and polysemous nature of words. Furthermore, due to the size of our text corpus, we also need an un-supervised approach to clustering. Here, curvature clustering [76] is the current state-of-art.

6.3.1 The Processing Pipeline for Word Sense Discrimination

For achieving discriminated word senses, we employ the processing pipeline proposed in [77], c.f. Fig. 29. This pipeline employs text cleaning, natural language processing, creation of the co-occurrence graph and clustering.

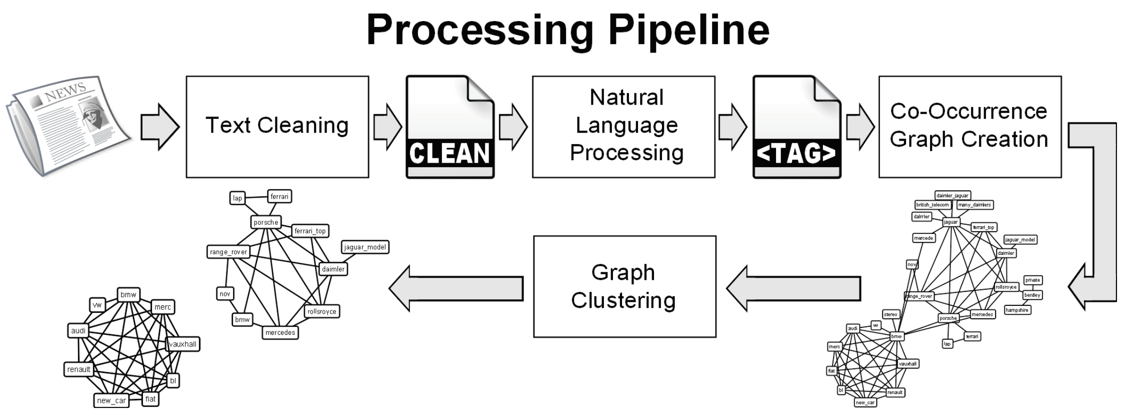


Fig. 29: Overview oft he word sense discrimination processing pipeline

The first step in the pipeline is Text Pre-Processing. The dataset used in our experiments is an archive of The Times[8]. The first step extracts the text content from the XML data provided.

8 We would like to thank Times Newspapers Limited for providing the archive of The Times for our research.

Natural Language Processing is the second step. Here, nouns and noun phrases are extracted. A linguistic processor employing part-of-speech tagging is used to identify nouns; additionally, terms are lemmatized if possible. A dictionary is created by these lemmata. The result is feed to a second linguistic processor for extracting noun phrases. These noun phrases, and additionally the remaining non-lemmatized nouns are subsequently added to the dictionary.

Now, the Co-occurrence graph is created. Using the dictionary created in the previous step, all documents are visited again to extract dictionary terms connected by an 'and', an 'or', or a comma. As example, in the sequence *… cars like BMW, Audi and Fiat …* the terms *BMW*, *Audi* and *Fiat* are therefore seen as co-occurring. After extraction, only co-occurrences above a given frequency will be retained in order to reduce noise.

Following the graph construction, the graph is clustered using the aforementioned curvature clustering. This algorithm calculates the clustering coefficient using Strogatz's method [78], which counts the number of triangles each node is involved in. This number is normalized by the maximum number of possible triangles, which is consequently named curvature value. Fig. 30 gives an illustration of the aforementioned example with the respective curvature values.

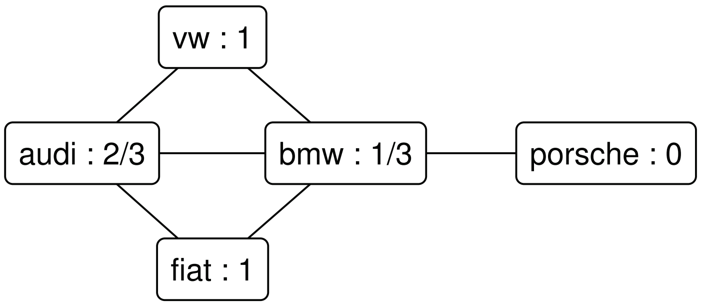


Fig. 30: Curvature values visualized by nodes showing name:curvature value

6.4 User-Interface and implementation

In order to make the results of the language evolution process end-user accessible, we devised two mobile applications. Both applications allow the user to explore the evolution of a term, however one is mainly intended for tablets while the other is intended for smart-phones or small tablets. Both applications were developed for an always-on scenario where a server stores the background knowledge and transfers upon request the needed information to the application in a compressed format.

6.4.1 Professional User-Interface

The first interface that we devised is aimed at the professional user. For a given query term, it visualizes all word sense clusters found for the term over the entire collection. The clusters are shown on a timeline and allow the user to scroll back and forward in time to search for word senses. For each cluster, a cluster representative is shown on the timeline. These are chosen from the terms in the cluster that have the highest curvature value. In Fig. 31, we see cluster representatives like *petersburg* and *ekaterinoslaff*. Choosing to click on one of the cluster representatives will show the full cluster with all its member terms and their connections like the graph shown in Fig. 30. This representation enables a deeper understanding of the cluster terms and their relations.

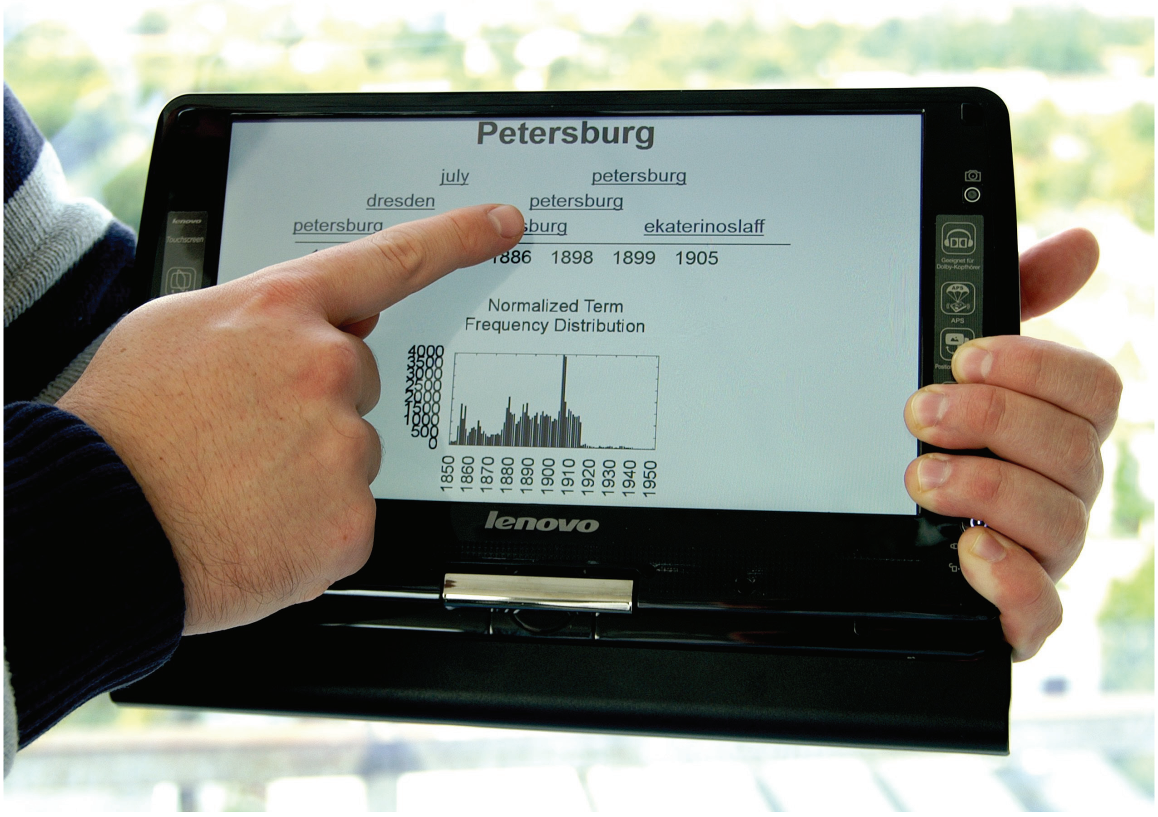
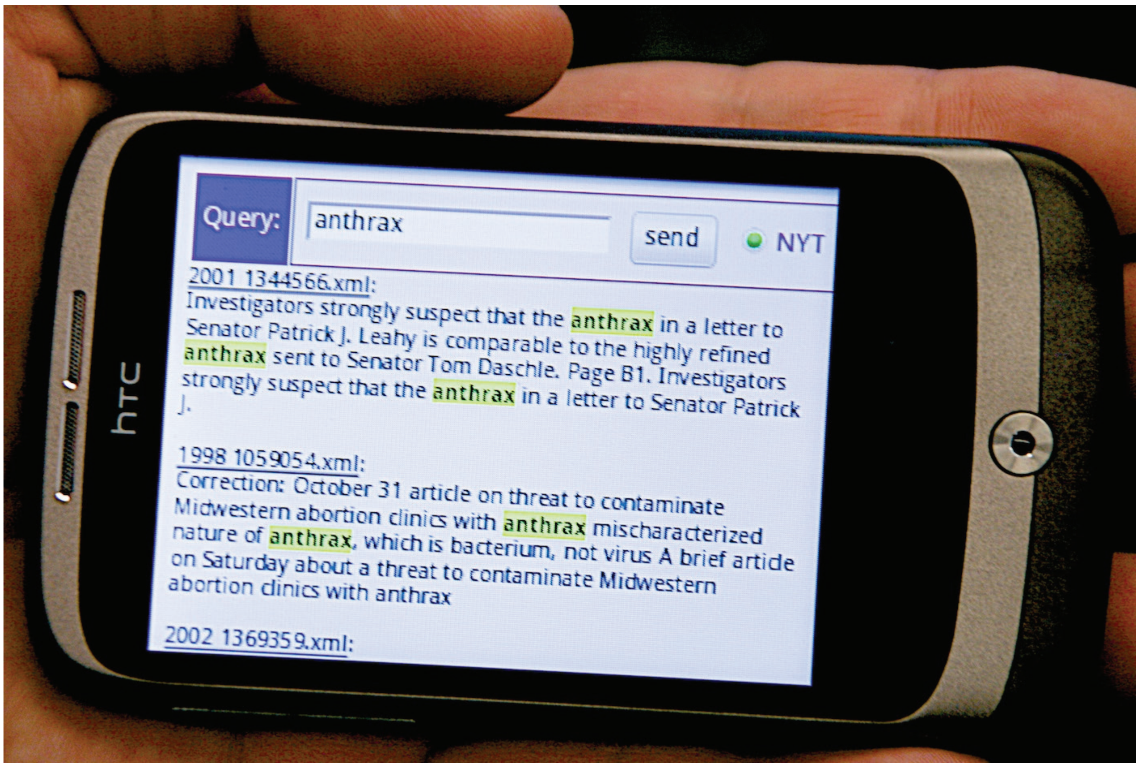


Fig. 31: Tablet Interface for Analysing Term Clusters

This professional user-interface will display, in addition to the cluster information, the normalized term frequency for the query term. This information can be very helpful for gaining a quick overview of the evolution of a term. As example, we chose the term *Petersburg* in Fig. 30. This Russian city has had many different names over the years with *Petersburg* being the predominant name over time. However, in 1914 – 1991, the city was named Petrograd as well as Leningrad. If we look at the normalized frequency distribution for *Petersburg*, we find that the frequency drops radically from 1914 to 1915. This already is a good indication that there has been some change in the meaning of *Petersburg*. If we wish to further investigate what happens with the term, we can choose to take a more in-depth look into the clusters to see if there is some indication of what happened.

We envision the use of this interface in places like museums or libraries where a deeper understanding of terms is needed and the user permits more time for the search and understanding of evolution.



2001 1344566.xml:
Investigators strongly suspect that the anthrax in a letter to Senator Patrick J. Leahy is comparable to the highly refined anthrax sent to Senator Tom Daschle. Page B1. Investigators strongly suspect that the anthrax in a letter to Senator Patrick J.

1998 1059054.xml:
Correction: October 31 article on threat to contaminate Midwestern abortion clinics with anthrax mischaracterized nature of anthrax, which is bacterium, not virus A brief article on Saturday about a threat to contaminate Midwestern abortion clinics with anthrax

2002 1369359.xml:

Fig. 32: Ambient user-interface showing example phrases for term *Anthrax*

6.4.2 Ambient User-Interface

The second interface is aimed at the casual user. While the aim of the previous interface was to provide as much detail as possible to the user, for this interface we envision users with less time and experience. Instead of giving in-depth analysis of the evolution of a term, this interface shows text examples for the most relevant clusters. This enables the user to understand the term evolution from the context of the original documents. As we can see from Fig. 32, the query term(s) are highlighted and shown in the context of one or more sentences from the original text in the archive. These sentences aid in understanding the context of the term as well as help the user decide which documents to take a closer look at.

In order to achieve this, for each query term, we retrieve corresponding clusters from the entire time period of the archive. Using these terms from the clusters, we search through a full text index to retrieve all relevant text excerpts. The cluster terms as well as the query terms are highlighted and the resulting documents are displayed in the order of their relevance.

We envision the use of this interface in a more leisure or ad-hoc manner. Here little time will be spend by the user and thus a deeper understanding of evolution is not needed. More importance is given to the ease-of-use and the speedy recognition of the meaning of a term.

6.5 Evaluation

In this section we will discuss the corpora used for our experiments, as well as a user-study to show the effectiveness of the devised ambient user-interface.

6.5.1 Corpus

To have sufficiently large background knowledge, we applied our methodology to a large dataset, the *The Times Archive, London*, as a sample of real world modern English. The corpus contains newspaper articles spanning from the year 1785 to 1985. The digitization process was started in the year 2001 when the collection was digitized from microfilm and OCR technology was applied to process the images. The resulting 201 years of data consist of 4,363 articles in the smallest dataset and 91,583 in the largest. The number of space separated tokens range from 4 million tokens in 1785 to 68 million tokens in 1928. In total we found 7.1 billion tokens that translate into an average number of 35 million tokens per year. In Fig. 33 we give an overview of these numbers on a year-by-year basis.

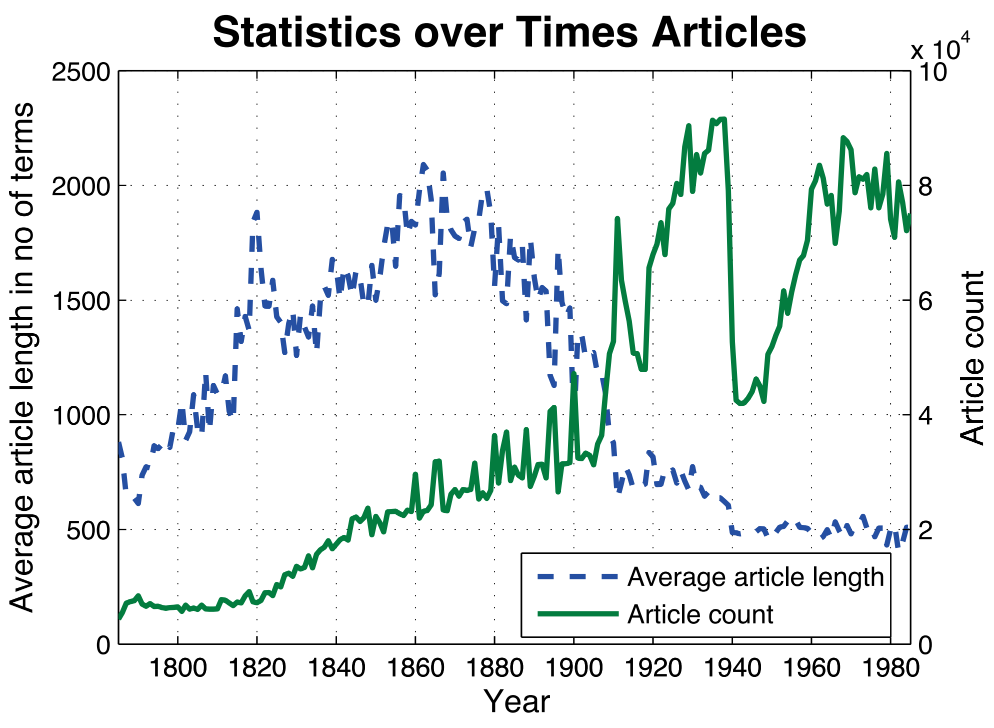


Fig. 33: Number of articles and average length of articles in The Times Archive from 1785 to 1985

6.5.2 Cluster Quality

To evaluate the quality of the clusters we measure the correspondence between clusters and word senses and rely upon WordNet [79] as a reference for word senses. We follow the method by Pantel [73] and compare the top *k* members of a cluster to a sense S in WordNet. If the similarity is above between S and the cluster is above a given threshold, then we say that the cluster corresponds to a word sense. The quality of clusters is then given as the amount of clusters that correspond to word senses.

Applying our method to The Times Archive results in between 221–106,000 unique co-occurrences between nouns and noun phrases. Each co-occurrence corresponds to one edge in the graph as gives us a measure of the size of the graph. Each graph is clustered and the resulting clusters are evaluated. On average 69% of all clusters contain more than two WordNet terms and can thus be evaluated. On average 85%±2% of our clusters correspond to word senses. The remaining clusters are a mix between wrongly devised clusters as well as clusters containing terms that are not found in WordNet. The first category of clusters contain terms that make no sense when gathered together, while the second category of clusters can contain proper nouns like people names or locations and are therefore not recognized by WordNet as a word sense.

With an average of 85% correspondence to word senses, we found that the cluster quality was sufficiently high to continue using the clusters as a basis for our user-interface evaluations. A further evaluation of the cluster quality is indirectly done once we evaluation the efficiency of our user-interfaces.

6.6 User-Study

In order to assess the quality and applicability of our ambient user-interface, we devised an initial user-study with 5 participants. The participants were all experts in computer science and between 20 and 30 years old.

6.6.1 Participants

We first analysed their general behaviour in information retrieving tasks. Most of our participants were not keen on spending large amounts of time for an information search task. Instead, they very much plan a strategy for searching in advance. On the presentation layer, they strongly prefer good accessibility of documents as much as attractive presentation. In general, well-known, reliable sources of information are preferred. Therefore, The Times Archive provided an interesting and trustworthy corpus for our candidates, which was found to be important in search usability studies by Ingwersen [80]. Serving snippets from a newspaper corpus, as our application does, may lead to showing contradicting or incomplete information. We therefore assessed whether this might have a negative impact on the information retrieval task, but our participants generally declined this. Our participants prefer to solve retrieval tasks themselves instead of asking for professional help, and generally prefer the Internet as first source.

Even though Google and Wikipedia were significantly the preferred way of assessing information, traditional printed media was still well respected and in some cases also preferred. This was for reasons like better readability of printed-paper, and especially because traditional printed resources are more believed to be a serious and reliable source of information. On the contrary, electronic resources are believed to be faster to retrieve and more up-to-date.

6.6.2 Procedure

The participants were not paid for conducting the survey. We tested all participants in a lab setting, using Android smart phones as ambient devices. Data was recorded using paper surveys before and after each task. The participants were first briefly introduced to their task and the ambient scenario. We refrained from an in-depth description of the procedure in order not to influence the participants. Throughout the study, subjective ratings were reported on a 5-point Likert-scale, with 1 meaning always/strongly agree/very good, and 5 meaning the opposite.

The first part of the survey contained general questions as discussed in the previous paragraph. Thereafter, participants were given 5 minutes time per query of Table 4. For each query, we asked the participants to rate how well they were able to assess the meaning of the query with the presented information, how suitable the amount of presented information was, and how usable this would be in an ambient situation.

We concluded each session with three general questions on the usability of the tool.

6.6.3 Results

Evaluating comprehensive tasks like this is difficult, as there are no correct answers and the goal is not necessarily to minimize time used. We assessed the helpfulness and the ease of use of our application, as can be seen in Fig. 34. For each query, we asked the participants to rate the answers given

Term	
1. car	2. zephyr
3. flight	4. yeoman
5. aeroplane	6. camera
7. Zermatt	8. iran
9. Jet	10. anthrax
11. train	12. mussolini

Table 4: Query terms used in user-study

regarding (1) how well the answers made the meaning of the query understandable (2) how sufficient the amount of the presented information was and (3) how well the result fits in an ambient scenario.

Although we only had a limited number of participants, leading to a high standard deviation around 1, our application showed generally good-to-average results. The spikes we see for queries like Query 2 are usually because the used *The Times Corpus* has OCR issues especially with older text, leading sometimes to non-understandable sentences.

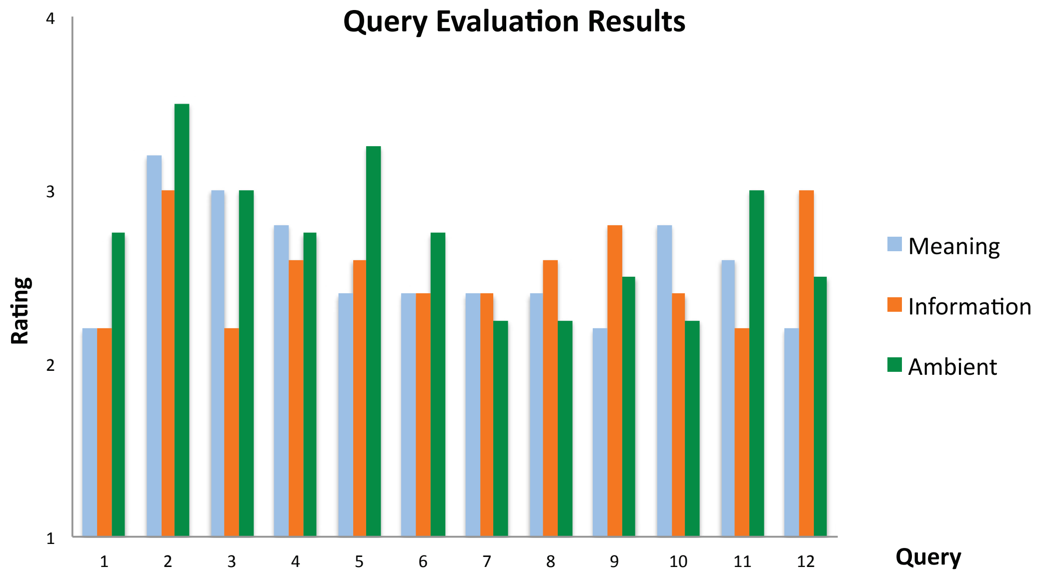


Fig. 34: Evaluation results of Queries in Table 4 rating 1-5 from very good to bad on how clear the meaning of a term became, information quantity, and ambient usability

As the general questions yielded, the information need satisfaction was rated good (2.8±1.3). The application seemed to be helpful for the information retrieval task (2.6±0.54) and comparably easy to use (2.4±1.67).

6.7 Discussion

In this chapter, we presented a solution for providing language evolution "on the go". As a basis we used *The Times Archive*, a large real-world corpus, allowing us to identify significant evolutions in language. We devised two applications tailored for mobile devices, one for professional, one for contemporary users, which allow for easy access to the corpus. We conducted an initial user-study on the contemporary, ambient user-interface, which establishes the good behaviour and usability of our interface.

We conclude that word sense discrimination is a helpful device for automatically generating domain knowledge for an information retrieval system, and carefully designed user interfaces are of great help for helping users satisfying their information need. In the future, it would be helpful to extend such approaches to structured data, and devise a system that allows combining the power of generated domain knowledge with the flexibility of query construction on structured data, as has been studied in the previous chapters.

7. Conclusion

In this thesis we aimed to improve on current methods for automated information retrieval, with a user-centric view in mind. We first introduced in Chapter 1 a general psychological model to understand the needs and requirements of users trying to satisfy their information needs. Then, in Chapter 2, we presented an approach for keyword search based ranking in relational database systems and RDF stores, called SUITS. This approach allows for easy access to commonly searched topics, and using construction options allows guiding the ranking algorithm to the semantically intended direction. In Chapter 4, we discussed QUICK, which fully supports all possibly user intents with a keyword query on RDF stores. As the underlying SetCover problem is NP-complete, we devised an incremental greedy algorithm with polynomial run time. As both of these approaches still require substantial domain knowledge, we devised in Chapter 5 a ranking based approach for facetted search on relational database systems, which makes use of external domain knowledge by a modified Castanet approach. While this is already greatly helpful to users, the need for pre-defined knowledge bases specifically covering the domain of the underlying data is restrictive. Therefore, we discuss in Chapter 6 a system that automatically generates the needed domain knowledge using word sense discrimination.

With the ever-growing amount of information, accessing this information in a fast and practical way is important, and we believe it to become even more prevalent in the time to come. We believe that current approaches often either tend neglect the importance of user interaction, or do not focus enough on the complexity and scale of the underlying data. With a strong focus on users' requirements of the whole search process, we presented several approaches to tackle these issues. We contributed several algorithms and new approaches addressing specifically the complexity of the structured data often employed in the hidden web. As every complex technical system requires a deep understanding of the access mechanisms as well as the underlying data, we additionally dived into the need of domain knowledge for the ordinary user. We showed the use of pre defined domain knowledge, as well as of generated domain knowledge in an extensive user study.

As always, these contributions can only be singular steps in fulfilling the need of users. The approach for query construction allows reaching all possible queries, but for this it needs to compute the full search space. This space grows exponentially with the size of the schema, which makes it infeasible to use on larger schemas. To solve this issue, researching progressive, approximate, and heuristic algorithms would be a fruitful endeavour, in order to still support the full search space, but reduce computational requirements. It also seems feasible to interpret the search process as cooperative game, in which the search system tries to predict next moves of the user. This allows applying game tree search algorithms like alpha-beta search. Furthermore, the discussed user interfaces are still rather restrictive; with QUICK, a user cannot control directly the direction within the search space the query construction takes. The facetted interface allows for more flexibility, but also this becomes more and more complex if the underlying schema grows. Employing aggregation and hierarchical structuring will help to make the amount of options manageable. This could be done by using concept hierarchies and exploiting attribute affiliations, which allows to group semantically similar entities. As result, the search space complexity would be reduced, and the schema would be easier to understand.

The presented interfaces also do not directly allow representing the join paths through the schema. With more complex queries, it therefore becomes difficult to understand the structure of the data. Here, using associative search approaches as presented by Feldspar [40] will be helpful. Currently, these approaches only allow visualizing linear join paths; this concept needs to be extended to support non-cyclic graphs. To support this, a measure for the cognitive effort needed to understand such graphs and respectively applied aggregation mechanisms would help in ensuring not to overstrain the user. Integrating such approaches with automatically generated domain knowledge will allow applying keyword search also to huge and complex web data sources, where manually generated domain knowledge would not be feasible. Also with this, aggregation and hierarchical structuring will allow to make deeply complex and big web data sources understandable, and queryable in an easy manner. Lastly, presenting example results for the currently constructed query is known to be of great help for users. Here, existing result sampling technologies could be applied for the case of structured web data, and be displayed during the whole query process.

8. References

[1] M. Bergman, “The deep web: Surfacing hidden value,” *Bright Planet*. 2001.

[2] S. Raghavan and H. Garcia-Molina, “Crawling the hidden web,” *27th International Conference on Very Large Data Bases*, 2001.

[3] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, B. Croft, D. DeWitt, M. Franklin, H. G. Molina, D. Gawlick, J. Gray, L. Haas, A. Halevy, J. Hellerstein, Y. Ioannidis, M. Kersten, M. Pazzani, M. Lesk, D. Maier, J. Naughton, H. Schek, T. Sellis, A. Silberschatz, M. Stonebraker, R. Snodgrass, J. Ullman, G. Weikum, J. Widom, and S. Zdonik, “The Lowell Database Research Self-Assessment,” *Communications of the ACM*, vol. 48, no. 5, pp. 111–118, 2005.

[4] G. Zenz, X. Zhou, E. Minack, W. Siberski, and W. Nejdl, “From keywords to semantic queries—Incremental query construction on the semantic web,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 3, pp. 166–176, Sep. 2009.

[5] A. Sutcliffe and M. Ennis, “Towards a cognitive theory of information retrieval,” *Interacting with Computers*, 1998.

[6] B. Shneiderman, D. Byrd, and W. B. Croft, “Sorting Out Searching A User-Interface Framework for Text Searches,” *Communications of the ACM*, vol. 41, no. 4, pp. 95–98, 1998.

[7] H. He, H. Wang, J. Yang, and P. S. Yu, “BLINKS: ranked keyword searches on graphs,” *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 305–316, 2007.

[8] G. Li, B. Ooi, J. Feng, J. Wang, and L. Zhou, “EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured

data," *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[9] F. Liu, C. Yu, W. Meng, and A. Chowdhury, "Effective keyword search in relational databases," *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, p. 563, 2006.

[10] S. Chaudhuri, R. Ramakrishnan, and G. Weikum, "Integrating DB and IR Technologies : What is the Sound of One Hand Clapping ?," in *Second Biennial Conference on Innovative Data Systems Research CIDR*, 2005.

[11] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: a system for keyword-based search over relational databases," *Proceedings 18th International Conference on Data Engineering*, pp. 5–16, 2002.

[12] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," *Proceedings of the 28th VLDB Conference*, 2002.

[13] D. Maier, J. D. Ullman, and M. Y. Vardi, "On the foundations of the universal relation model," *ACM Transactions on Database Systems*, vol. 9, no. 2, pp. 283–308, May 1984.

[14] V. Hristidis, "Efficient IR-style keyword search over relational databases," *Proceedings of the 29th VLDB Conference*, 2003.

[15] E. Kandogan, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu, "Avatar semantic search: a database approach to information retrieval," *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 790–792, 2006.

[16] Q. Zhou, C. Wang, M. Xiong, H. Wang, and Y. Yu, "SPARK : Adapting Keyword Query to Semantic," *6th International Semantic Web Conference*, pp. 694–707, 2007.

[17] B. Aditya, G. Bhalotia, S. Chakrabati, A. Hulgeri, C. Nakhe, Parag, and S. Sudarshan, “Banks: Browsing and keyword searching in relational databases,” *Proceedings of the 28th VLDB Conference*, 2002.

[18] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, “Finding Top-k Min-Cost Connected Trees in Databases,” *23rd International Conference on Data Engineering*, pp. 836–845, 2007.

[19] B. Kimelfeld and Y. Sagiv, “Efficient engines for keyword proximity search,” *Eighth International Workshop on theWeb and Databases (WebDB 2005)*, 2005.

[20] X. Dong and A. Halevy, “Indexing dataspaces,” *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD ’07*, p. 43, 2007.

[21] L. Zhang, Q. Liu, J. Zhang, H. Wang, Y. Pan, and Y. Yu, “Semplore : An IR Approach to Scalable Hybrid,” *6th International Semantic Web Conference*, no. 1, pp. 652–665, 2007.

[22] M. Jayapandian and H. V. Jagadish, “Expressive query specification through form customization,” *Proceedings of the 11th international conference on Extending database technology Advances in database technology - EDBT ’08*, p. 416, 2008.

[23] M. Jayapandian and H. V. Jagadish, “Automated creation of a forms-based database query interface,” *Proceedings of the VLDB Endowment*, pp. 695–709, 2008.

[24] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik, “EquiX—A Search and Query Language for XML,” *Journal of the American Society for Information Science and Technology*, 2002.

[25] M. Petropoulos, Y. Papakonstantinou, and V. Vassalos, “Graphical query interfaces for semistructured data: the QURSED system,” *ACM*

Transactions on Internet Technology, vol. 5, no. 2, pp. 390–438, May 2005.

[26] R. Abraham, “FoXQ - XQuery by forms,” *IEEE Symposium on Human Centric Computing Languages and Environments*, pp. 289–290, 2003.

[27] T. Catarci and M. Costabile, “Visual query systems for databases: A survey,” *Journal of Visual Languages & Computing*, 1997.

[28] E. Augurusa and D. Braga, “Design and implementation of a graphical interface to XQuery,” *Proceedings of the 2003 ACM symposium on Applied computing*, pp. 1–5, 2003.

[29] D. Braga, A. Campi, and S. Ceri, “XQBE (XQuery By Example): A visual interface to the standard XML query language,” *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 398–443, 2005.

[30] A. Russell, P. Smart, D. Braines, and N. R. Shadbolt, “NITELIGHT: A Graphical Tool for Semantic Query Construction,” *Semantic Web User Interaction Workshop (SWUI 2008)*, pp. 1–10, 2008.

[31] K.-P. Yee, K. Swearingen, K. Li, and M. Hearst, “Faceted metadata for image search and browsing,” *Proceedings of the conference on Human factors in computing systems - CHI ’03*, p. 401, 2003.

[32] M. c. Schraefel, “Building Knowledge: What’s Beyond Keyword Search?,” *IEEE Computer*, 2009.

[33] M. Wilson, A. Russell, and D. Smith, “mSpace: improving information access to multimedia domains with multimodal exploratory search,” *Communications of the ACM*, vol. 49, no. 4, pp. 47–49, 2006.

[34] M. Wilson and M. c. Schraefel, “A longitudinal study of exploratory and keyword search,” *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, pp. 52–55, 2008.

[35] R. van Zwol and B. Sigurbjornsson, “Faceted exploration of image search results,” *Proceedings of the 19th international conference on World wide web - WWW ’10*, p. 961, 2010.

[36] S. Basu Roy, H. Wang, G. Das, U. Nambiar, and M. Mohania, “Minimum-effort driven dynamic faceted search in structured databases,” *Proceeding of the 17th ACM conference on Information and knowledge mining - CIKM ’08*, p. 13, 2008.

[37] C. Li, N. Yan, S. Roy, L. Lisham, and G. Das, “Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia,” in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 651–660.

[38] V. Sinha and D. R. Karger, “Magnet: Supporting navigation in semistructured data environments,” *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005.

[39] M. Hildebrand, “/facet: A browser for heterogeneous semantic web repositories,” *5th International Semantic Web Conference*, pp. 272–285, 2006.

[40] D. Chau, B. Myers, and A. Faulring, “What to do when search fails: finding information by association,” *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, 2008.

[41] M. Schraefel, M. Wilson, and M. Karam, “Preview Cues: Enhancing Access to Multimedia Content,” *School of Electronics and Computer Science, University of Southampton*, 2004.

[42] G. Robertson, K. Cameron, M. Czerwinski, and D. Robbins, “Polyarchy visualization: visualizing multiple intersecting hierarchies,” *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2002.

[43] T. Paek, S. Dumais, and R. Logan, “WaveLens: a new view onto Internet search results,” *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004.

[44] W. Kules, M. Wilson, and B. Shneiderman, “From keyword search to exploration: How result visualization aids discovery on the web,” *Technical Report*, 2008.

[45] Y. Wang, L. Xuemin, L. Wei, and X. Zhou, “SPARK: Top-k Keyword Query in Relational Databases,” in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[46] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano, “Efficient Keyword Search Across Heterogeneous Relational Databases,” in *23rd International Conference on Data Engineering*, 2007.

[47] T. Tran, P. Cimiano, S. Rudolph, and R. Studer, “Ontology-based interpretation of keywords for semantic search,” *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, 2007.

[48] R. Fagin, “Combining Fuzzy Information from Multiple Systems,” *Journal of Computer and System Sciences*, vol. 58, no. 1, pp. 83–99, Feb. 1999.

[49] J. Widom, “Indexing Relational Database Content Offline for Efficient Keyword-Based Search,” *9th International Database Engineering and Application Symposium*, pp. 297–306, 2005.

[50] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu, “Making database systems usable,” *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD ’07*, p. 13, 2007.

[51] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, “Natural language interfaces to databases – an introduction,” *Natural Language Engineering*, vol. 1, no. 01, Sep. 1995.

[52] Y. Li, H. Yang, and H. Jagadish, “Constructing a Generic Natural Language Interface for an XML Database,” in *Advances in Database Technology-EDBT*, 2006, pp. 737–754.

[53] M. Al-Muhammed and D. Embley, “Ontology-based constraint recognition for free-form service requests,” in *IEEE 23th International Conference on Data Engineering*, 2007, pp. 366–375.

[54] H. Bast, A. Chitea, F. Suchanek, and I. Weber, “ESTER: efficient search on text, entities, and relations,” *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 671–678, 2007.

[55] A. Nandi and H. V. Jagadish, “Assisted querying using instant-response interfaces,” in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD ’07*, 2007, p. 1156.

[56] D. Carmel, Y. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer, “Searching XML documents via XML fragments,” in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in information retrieval*, 2003, p. 151.

[57] N. Fuhr, M. Lalmas, and A. Trotman, “Comparative Evaluation of XML Information Retrieval Systems,” in *5th International Workshop of the Initiative for the Evaluation of XML Retrieval*, 2006.

[58] S. Amer-Yahia, L. Lakshmanan, and S. Pandit, “FleXPath: flexible structure and full-text querying for XML,” in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD ’04*, 2004, vol. 1.

[59] Y. Li, C. Yu, and H. Jagadish, “Schema-free xquery,” *Proceedings of the Thirtieth international conference on Very large data bases*, pp. 72–83, 2004.

[60] R. Karp, “Reducibility among combinatorial problems,” *50 Years of Integer Programming 1958-2008*, 2010.

[61] J. Broekstra, “Sesame: A generic architecture for storing and querying rdf and rdf schema,” *International Semantic Web Conference*, pp. 1–16, 2002.

[62] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *The Journal of Machine Learning*, pp. 993–1022, 2003.

[63] D. Cutting, D. Karger, J. O. Pedersen, and J. W. Tukey, “Scatter/gather: A cluster-based approach to browsing large document collections,” *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, 1992.

[64] M. a. Hearst, “Clustering versus faceted categories for information exploration,” *Communications of the ACM*, vol. 49, no. 4, p. 59, Apr. 2006.

[65] W. Dakka and P. G. Ipeirotis, “Automatic extraction of useful facet hierarchies from text databases,” *IEEE 24th International Conference on Data Engineering*, pp. 466–475, 2008.

[66] E. Stoica, M. A. Hearst, and M. Richardson, “Automating Creation of Hierarchical Faceted Metadata Structures,” *Proceedings of NAACL HLT*, pp. 244–251, 2007.

[67] J. Caumanns, “A Fast and Simple Stemming Algorithm for German Words,” *Technical Report*, 1998.

[68] D. Davidov and A. Rappoport, “Efficient unsupervised discovery of word categories using symmetric patterns and high frequency words,” *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, 2006.

[69] T. Pedersen and R. Bruce, “Distinguishing Word Senses in Untagged Text,” *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, 1997.

[70] T. V. D. Cruys, “Using three way data for word sense discrimination,” *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, no. August, pp. 929–936, 2008.

[71] H. Schütze, “Automatic word sense discrimination,” *Computational linguistics*, 1998.

[72] D. Lin, “Automatic retrieval and clustering of similar words,” *COLING ’98 Proceedings of the 17th international conference on Computational linguistics*, 1998.

[73] P. Pantel and D. Lin, “Discovering word senses from text,” *Proceedings of the eighth ACM SIGKDD international conference*, vol. 41, 2002.

[74] B. Dorow, “A graph model for words and their meanings,” 2007.

[75] O. Ferret, “Discovering word senses from a network of lexical cooccurrences,” *COLING ’04 Proceedings of the 20th international conference on Computational Linguistics*, 2004.

[76] B. Dorow, D. Widdows, K. Ling, J.-P. Eckmann, D. Sergi, and E. Moses, “Using curvature and markov clustering in graphs for lexical acquisition and word sense discrimination,” *Arxiv*, 2004.

[77] N. Tahmasebi, K. Niklas, T. Theuerkauf, and T. Risse, “Using word sense discrimination on historic document collections,” *Proceedings of the 10th annual joint conference on Digital libraries - JCDL ’10*, p. 89, 2010.

[78] D. J. Watts and S. H. Strogatz, “Collective dynamics of ‘small-world’ networks.,” *Nature*, vol. 393, no. 6684, pp. 440–2, Jun. 1998.

[79] G. a. Miller, “WordNet: a lexical database for English,” *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[80] P. Borlund and P. Ingwersen, “The development of a method for the evaluation of interactive information retrieval systems,” *Journal of documentation*, vol. 53, no. 3, pp. 225 – 250, 1997.

9. Wissenschaftlicher Lebenslauf

Name	Gideon Zenz
Geboren am	03.05.1979
Geboren in	Neunkirchen / Saar

Schulabschluß

8/1989 – 6/1998	Allgemeine Hochschulreife Kardinal-Frings Gymnasium, Bonn/Beuel

Studium

9/2002 – 9/2006	Ingenieur Informatik (FH) Thema der Diplomarbeit: *Integration IT - Embedded Systems: Wissensbasierte Kommunikation in verteilten Systemen mit der Web Ontology Language (OWL)* Universität Lüneburg
10/2006 – 9/2008	Informatik Thema der Masterarbeit: *Precomputing Indexes for Efficient Keyword Search in Databases* Leibniz Universität Hannover

Promotion

1/2008 – 06/2013	Wissenschaftlicher Mitarbeiter und Doktorand Forschungszentrum L3S Leibniz Universität Hannover