KEYWORD-BASED QUERYING FOR THE SOCIAL SEMANTIC WEB – THE KWQL LANGUAGE: CONCEPT, ALGORITHM AND SYSTEM

KLARA WEIAND

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der Ludwig–Maximilians–Universität, München

15. Dezember 2010

KEYWORD-BASED QUERYING FOR THE SOCIAL SEMANTIC WEB – THE KWQL LANGUAGE: CONCEPT, ALGORITHM AND SYSTEM

KLARA WEIAND

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der Ludwig–Maximilians–Universität, München

15. Dezember 2010

Erster Berichterstatter:	Prof. Dr. François Bry
	Ludwig–Maximilians–Universität München
Zweiter Berichterstatter:	Prof. Dr. Letizia Tanca
	Politecnico di Milano
Datum des Rigorosums:	8. Februar 2011

Breathe deeply and no one can put you in a cage.

— Daniel Odier

ABSTRACT

Enabling non-experts to publish data on the web is an important achievement of the social web and one of the primary goals of the social semantic web. Making the data easily accessible in turn has received only little attention, which is problematic from the point of view of incentives: users are likely to be less motivated to participate in the creation of content if the use of this content is mostly reserved to experts.

Querying in semantic wikis, for example, is typically realized in terms of full text search over the textual content and a web query language such as SPARQL for the annotations. This approach has two shortcomings that limit the extent to which data can be leveraged by users: combined queries over content and annotations are not possible, and users either are restricted to expressing their query intent using simple but vague keyword queries or have to learn a complex web query language.

The work presented in this dissertation investigates a more suitable form of querying for semantic wikis that consolidates two seemingly conflicting characteristics of query languages, ease of use and expressiveness. This work was carried out in the context of the semantic wiki KiWi, but the underlying ideas apply more generally to the social semantic and social web.

We begin by defining a simple modular conceptual model for the KiWi wiki that enables rich and expressive knowledge representation. A component of this model are *structured tags*, an annotation formalism that is simple yet flexible and expressive, and aims at bridging the gap between atomic tags and RDF. The viability of the approach is confirmed by a user study, which finds that structured tags are suitable for quickly annotating evolving knowledge and are perceived well by the users.

The main contribution of this dissertation is the design and implementation of KWQL, a query language for semantic wikis. KWQL combines keyword search and web querying to enable querying that scales with user experience and information need: basic queries are easy to express; as the search criteria become more complex, more expertise is needed to formulate the corresponding query. A novel aspect of KWQL is that it combines both paradigms in a bottom-up fashion. It treats neither of the two as an extension to the other, but instead integrates both in one framework. The language allows for rich combined queries of full text, metadata, document structure, and informal to formal semantic annotations. KWilt, the KWQL query engine, provides the

full expressive power of first-order queries, but at the same time can evaluate basic queries at almost the speed of the underlying search engine. KWQL is accompanied by the visual query language visKWQL, and an editor that displays both the textual and visual form of the current query and reflects changes to either representation in the other. A user study shows that participants quickly learn to construct KWQL and visKWQL queries, even when given only a short introduction.

KWQL allows users to sift the wealth of structure and annotations in an information system for relevant data. If relevant data constitutes a substantial fraction of all data, ranking becomes important. To this end, we propose PEST, a novel ranking method that propagates relevance among structurally related or similarly annotated data. Extensive experiments, including a user study on a real life wiki, show that PEST improves the quality of the ranking over a range of existing ranking approaches.

ZUSAMMENFASSUNG

Eine wichtige Errungenschaft des Social Web und gleichzeitig eines der Hauptziele des Social Semantic Webs ist es, Laien die Veröffentlichung von Daten im Web zu ermöglichen. Der Frage, wie wiederum ein einfacher Zugang zu diesen Daten ermöglicht werden kann, wurde hingegen bisher nur wenig Aufmerksamkeit gewidmet. Dies ist problematisch, da mit einer geringeren Motivation der Benutzer bei der Erstellung von Inhalten zu rechnen ist, wenn die Verwendung dieser Inhalte Experten vorbehalten ist.

Anfragen in semantischen Wikis sind etwa in der Regel durch Volltext-Suche über die textuellen Inhalte und eine Webanfragesprache wie SPARQL für Annotationen realisiert. Dieser Ansatz hat zwei Nachteile, die Benutzer in der Fähigkeit einschränken, Daten zu ihrem Vorteil zu nutzen: kombinierte Anfragen über Volltext und Annotationen sind nicht möglich, und Benutzer müssen ihre Anfrage entweder durch einfache und damit vage Stichworte ausdrücken, oder aber eine komplexe Webanfragesprache lernen.

Die in dieser Dissertation vorgestellte Forschung untersucht eine geeignetere Form der Anfrage von Daten in semantischen Wikis, die zwei scheinbar gegensätzliche Eigenschaften von Anfragesprachen zusammenführt: Benutzerfreundlichkeit und Ausdruckskraft. Diese Forschung wurde im Rahmen des semantischen Wikis KiWi durchgeführt, die zugrundeliegenden Ideen sind jedoch auf das Social Semantic und Social Web im Allgemeinen anwendbar.

Wir beginnen mit der Definition eines einfachen und modularen konzeptuellen Modells für das KiWi Wiki, das eine mächtige und ausdrucksstarke Repräsentation von Wissen ermöglicht. Ein Bestandteil dieses Modells sind strukturierte Tags, ein einfacher und dennoch flexibler und ausdrucksstarker Annotationsformalismus, der darauf abzielt, die Kluft zwischen atomaren Tags und RDF zu überbrücken. Die Tragfähigkeit dieses Ansatzes wird durch eine Nutzerstudie bestätigt, die zeigt, dass strukturierte Tags von den Benutzern positiv aufgenommen werden und zur schnellen Annotierung sich kontinuierlich entwickelnden Wissens geeignet sind.

Der Hauptbeitrag dieser Dissertation sind der Entwurf und die Implementierung von KWQL, einer Anfragesprache für semantische Wikis. KWQL kombiniert Stichwortsuche und Web-Anfragen und kann sich so der Erfahrung und dem Informationsbedarf

des Nutzers anpassen: Elementare Anfragen können einfach ausgedrückt werden. Mit der Komplexität der Suchkriterien wächst auch das Fachwissen, das benötigt wird, um die entsprechende Anfrage zu formulieren. Neu ist dabei, dass KWQL die beiden Paradigmen von Grund auf verbindet. Es behandelt keines der beiden als Erweiterung des anderen, sondern integriert beide in einem gemeinsamen System. Die Sprache ermöglicht mächtige kombinierte Anfragen über Volltext, Metadaten, Dokumentstruktur, und informale und formale semantische Annotationen. KWilt, die Anfrage-Engine von KWQL, bietet die volle Ausdruckskraft von Logik erster Stufe, gleichzeitig können einfache Anfragen nahezu mit der Geschwindigkeit der zugrundeliegenden Suchmaschine ausgewertet werden. KWQL wird ergänzt durch die visuelle Anfragesprache visKWQL sowie durch einen Editor, der sowohl die textuelle als auch die visuelle Form der aktuellen Anfrage anzeigt und Änderungen in einer Darstellung in der jeweils anderen wiedergibt. Eine Nutzerstudie zeigt, dass die Teilnehmer schnell lernen, Anfragen in KWQL und visKWQL zu erstellen, selbst wenn nur eine kurze Einführung gegeben wurde.

KWQL ermöglicht es, die reichhaltige Struktur und Annotationen eines Informationssystems nach relevanten Daten zu durchsuchen. Wenn die relevanten Daten einen erheblichen Anteil aller Daten ausmachen, gewinnt deren Ranking an Bedeutung. Zu diesem Zweck stellen wir PEST vor, eine neue Methode zur Berechnung von Rankings, die Relevanz zwischen strukturell verwandten oder ähnlich annotierten Daten propagiert. Umfangreiche Experimente, darunter eine Nutzerstudie in einem realen Wiki, zeigen, dass PEST die Qualität des Rankings verglichen mit eine Reihe bestehender Ansätze verbessert.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

A. Hartl, K. Weiand, and F. Bry. visKWQL, Visual Keyword Queries for Semantic Data. In *Third Future Internet Symposium (FIS)*, 2010.

K. Weiand, S. Hausmann, T. Furche, and F. Bry. KWilt: A Semantic Patchwork for Flexible Access to Heterogeneous Knowledge. In *Fourth International Conference on Web Reasoning and Rule Systems (RR)*, 2010.

K. Weiand, F. Kneißl, T. Furche, and F. Bry. PEST: Term-Propagation over Wiki Structures as Eigenvector Computation. In *Fifth Workshop on Semantic Wikis (SemWiki)*, 2010.

A. Hartl, K. Weiand, and F. Bry. visKQWL, a Visual Renderer for a Semantic Web Query Language. In *International World Wide Web Conference (WWW)*, 2010.

F. Bry and K. Weiand. Flavors of KWQL, a Keyword Query Language for a Semantic Wiki. In *Theory and Practice of Computer Science (SOFSEM)*, 2010.

F. Bry, T. Furche, and K. Weiand. Web Queries: From a Web of Data to a Semantic Web. In *Web Information Systems Engineering (WISE)*, 2009.

F. Bry and K. Weiand. KWQL, Querying for Social Semantic Software. In *Reasoning Web, Fifth International Summer School*, 2009.

F. Bry, M. Eckert, J. Kotowski, and K. Weiand. What the User Interacts with: Reflections on Conceptual Models for Semantic Wikis. In *Fourth Workshop on Semantic Wikis (SemWiki)*, 2009.

F. Bry, J. Kotowski, and K. Weiand. Querying and Reasoning for Social Semantic Software. In *European Semantic Web Conference (ESWC)*, 2009.

K. Weiand, T. Furche, and F. Bry. Quo Vadis, Web Queries? In *International Workshop on Semantic Web Technology (Web4Web)*, 2008.

ACKNOWLEDGMENTS

I am grateful for all the support I have received during my time at LMU Munich. First and foremost, I would like to thank my advisor Prof. Dr. François Bry. He was always there to provide encouragement, support, and advice, and taught me a great deal about research and work in academia in general. I am also indebted to Letizia Tanca for agreeing to act as external reviewer for this thesis.

Working with the faculty, staff, and students at the Programming and Modeling Languages group has been a great pleasure, and I thank all of you: My fellow researchers Simon Brodt, Michael Eckert, Norbert Eisinger, Tim Furche, Steffen Hausmann, Fabian Kneißl, Alex Kohn, Jakub Kotowski, Stephan Leutenmayr, Benedikt Linse, Hans Jürgen Ohlbach, Alexander Pohl, Olga Poppe, Edgar-Philipp Stoffel, Christoph Wieser, and Harald Zauner for creating an inspirational and supportive work environment. Martin Josko and Ingeborg von Troschke for always being there to make things run smoothly. My co-authors for their expertise, insights, and many interesting and stimulating discussions. The students whose diploma theses I supervised for their devotion, hard work, and endurance. The participants in the user studies on KWQL and visKWQL, structured tags and RDF, and PEST for making this research possible.

I would also like to thank the members of the KiWi project at Aalborg University, Brno University of Technology, Logica, Oracle (formerly Sun Microsystems), Salzburg Research, and the Semantic Web Company for their collaboration, their interest in my work, and the many discussions we had.

The research described in this work was partly funded by the European Commission within the 7th Framework Programme project "KiWi - Knowledge in a Wiki." This funding allowed me to pursue my research, travel to conferences, and get in touch with many fellow researchers.

Finally, I would like to thank my family for their love and support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Most queries submitted to web search engines consist of a small number of simple keywords that approximate the user's query intent and serve as filtering criteria for the retrieval of documents. Often, no operators or other syntactic constructs are provided, and conjunction between the individual terms is assumed, meaning that a matching document must contain all terms. Consequently, queries are simple bags of terms, and selection criteria that make reference to the structure of the data cannot be expressed.

An important reason why no search engine offers this capability, apart from (substantial) concerns about increased storage and computation costs, is that the utility of augmenting content-based queries with structural selection criteria is limited in light of the heterogeneity of HTML documents: HTML is a lightweight markup language that is used by a large number of authors with varying expertise and intentions to create a vast number of web pages. There often exist many different, equally valid ways to express a given piece of information. Inversely, the structure of a piece of HTML code is not necessarily related to the meaning of the underlying text. Often, HTML is used not to structure information in a meaningful way, but simply to format it to the author's liking.

While structure as a selection criterion may often not be necessary or useful in general web search, the case is different for the *web 2.0* or *social web*, a category of web applications that has gained much popularity over the past five years. Here, the structure of individual pages is to some extent standardized, either by design or due to social conventions. Consequently, structural selection criteria can have a meaningful and consistent interpretation with respect to the semantics of the data.

Social web applications enable users to easily publish content, collaborate and interact. Up until the middle of the previous decade, the creation of web content required at least basic knowledge of web technologies. A small number of content creators therefore faced many consumers, for whom the web was a read-only medium. The Web 2.0 is more democratic in the sense that publishing content does not require technological expertise and that content creation is often an inclusive, iterative, and interactive process. Examples of social web applications include blogs, social networking sites, as well as many specialized applications, for

example for saving and sharing bookmarks,[1] publishing photos[2], and aggregating users' listening habits.[3]

Wikis are social web applications for collecting and sharing knowledge. They allow users to easily create and edit documents, so-called wiki pages, using a web browser. The pages in a wiki are often heavily interlinked, which makes it easy to find related information and browse the content. A common characteristic of all wikis is that the content is version-controlled, meaning that older versions of a wiki page can be restored at any time.

In many respects, wikis are a prototypical social web application, and their success is tightly connected to the proliferation of the social web. In particular, wikis are conceptually simple, easy to use, and support users in the content creation process.

In many social web applications, individual pages are all formatted in the same manner. The fixed layout can be seen as a template which is filled with users' contributions. This has several advantages: users do not need to know HTML, CSS, and related technologies, they can easily publish content without having to specify the layout of a page, and the usability of the website is improved by a consistent look and structure. In wikis, the formatting and structure of the wiki content itself often follows additional conventions that have developed over time through a collaborative social process. Since it is relatively consistent, the structure of these web pages could be leveraged for the targeted selection of data items, for example to retrieve all wiki pages with "Munich" in their title or all pages of users who mention programming as one of their interests.

While HTML offers the possibility to provide metadata, this possibility is often not used at all, or to give incorrect information in an attempt to influence search engine rankings. On the social web, data items are typically augmented with metadata regarding the author and time of creation. This information could be used, for example, to search a snapshot of a wiki at a certain point in the past, or to retrieve those items a specific user has commented on. So far, however, most social websites provide only simple keyword search, usually augmented with methods for result navigation, such as snippets, relevance rankings, and facets. They do not offer query languages that would allow users to exploit the structure of the data.

Social semantic web applications are social websites in which knowledge is expressed not only in the form of text and multimedia, content structure, and metadata, but also through informal to formal annotations that describe, reflect, and enhance the content.

1 See, e.g., http://www.delicious.com/.
2 See, e.g., http://www.flickr.com/.
3 See, e.g., http://www.last.fm/.

In traditional wikis, knowledge is given in the form of text in natural language, and is not directly amenable to automated semantic processing. It can therefore only be located through full text keyword search or via simple, mostly user-generated, structures like tables of content and links between pages. More sophisticated functionalities such as querying, reasoning, and semantic browsing are not available. The goal underlying semantic wikis is to provide at least some of these enhancements by relying on semantic technologies, that is, knowledge representation formalisms and automated reasoning methods. Semantic wikis extend conventional wikis by—more or less sophisticated—formal languages for expressing knowledge as machine processable annotations to wiki pages containing text or multimedia. These annotations typically take the shape of RDF graphs backed by ontologies, though less formal annotations for expressing knowledge such as free-form tags or tags from a controlled vocabulary may also be available. Semantic wikis have their foundation in semantic web research and aim at combining semantic web technologies with the collaborative nature and user-friendliness of the social web.

Data retrieval in semantic wikis is typically realized through keyword search and web query languages. Keyword search is the prevalent paradigm for search on the web. Its strength, and presumably the main reason for its success, is that it is very accessible: there is no syntax that has to be learned before queries can be issued, and relevant information can be found without any knowledge of the structure of the underlying data. On the downside, keyword search is inherently imprecise and inexpressive. It does not allow for the specification of structure-based selection criteria, and often not even for logical operations. As a matter of fact, queries remain vague. Even when users know precisely which data they are interested in, they may not be able to express the corresponding selection criteria. Finally, web search does not allow for the automation of tasks.

While structure as a selection criterion may be of arguable utility in general web search, the rich structure of social and social semantic web data could be utilized to enable the precise and targeted selection of data, thereby allowing users to fully leverage the data that they contribute to social semantic web applications.

Web query languages are in many respects the exact opposite of keyword search: similar to queries on relational databases, web queries are highly specific and select individual data items which can then be processed further to re-format the data or deduce and display new knowledge. Once defined, these tasks can be performed automatically and without human intervention. Web query languages are powerful tools that enable the selection and

construction of data items. They are comparable to programming languages both in their expressive power and their complexity.

While keyword search does not allow the structure of the data to be used as a selection criterion, web querying does not easily accommodate queries that do not use it. Web query languages only support vague queries in a very weak sense, for example in the form of wildcards and regular expressions over structure and values. Therefore, users must be aware of the respective data schema to be able to formulate queries. This is just one example for the high cognitive investment that is required before a web query language can be used to retrieve data from a given dataset: in addition to the schema, the user has to know and understand the data type, such as XML or RDF, and its characteristics and properties, and finally the query language itself. Especially for casual or beginning web users, acquiring this knowledge can be a hard and laborious process, and many may lack the time, dedication, motivation, or confidence to tackle it.

Web query languages usually do not support the gradual refinement of basic exploratory queries, fuzzy matching, or the ranking and clustering of results. This means that that the user is not supported in developing and concretizing her information need and in navigating the results. Unlike keyword search, web querying typically lacks a notion of gradual relevance of a result to a query: either a data item is a suitable answer to a query, or it is not.

Despite recent research into versatile query languages, many web query languages can only be used to query data of one specific type, such as XML or RDF. In social semantic web applications containing data of various different types, this means that any given web query language can only be used to query part of the data. Querying all of the data requires the integration of several differing query languages or the conversion of data to other data types.

In summary, web query languages are well suited for querying structured data, while keyword search is generally more appropriate for search over weakly structured or unstructured text. In a social semantic web application one typically finds both types of data. It is not enough, however, to simply provide both keyword search and a web query language, because the two could only be used to query different parts of the data separately, for example textual content with keyword search and annotations with SPARQL.

To be able to leverage the knowledge contained in these rich data repositories, a query language for social semantic web applications should be expressive enough to allow for precise selections using complex criteria and to enable the aggregation and combination of data, and thus the derivation of new data through

a simple form of reasoning. Automation in the form of embedded queries—queries that are contained in a piece of content and are evaluated when this content is retrieved—and continuous queries—queries that are evaluated repeatedly at set intervals or when the data changes—further requires query evaluation to operate without the need for human intervention.

At the same time, a query language for social semantic web applications should be accessible even to casual users and without much training: the success of a social semantic web application crucially depends on the active participation and the contributions of users, most of which cannot and should not be expected to have much experience with query languages. Making it easy for non-experts to publish data on the web is an important achievement of the social web and a primary goal of the social semantic web. The goal of making the data thus produced easily accessible in turn has received relatively little attention. This is problematic because users are likely to be less motivated to participate in the creation of content if they cannot leverage the data that they and others have contributed and the exploitation of the data is reserved to expert users.

The methods currently available, keyword search and web querying, fail to provide the desirable characteristics outlined above. Keyword search has a very basic syntax, which is an advantage in terms of usability but means that it cannot express complex queries. Web query languages, on the other hand, are powerful but do not support vague queries. They further require knowledge of the data type and schema and of the query language itself, thereby excluding many users from exploiting the collaboratively created data.

This dissertation describes the design and implementation of KWQL, a query language for the semantic wiki KiWi. KWQL combines keyword search and web querying to enable a form of querying that adapts to the user's information need and knowledge and accommodates simple search and complex selections alike. A novel aspect of KWQL is that it combines both paradigms, keyword search and web queries, in a bottom-up fashion. It treats neither of the two as an extension to the other, but instead integrates both in one framework and bridges the gap between them. Depending on the user's knowledge and query intent, the language can behave more like keyword search or more like web querying. KWQL allows for rich combined queries over textual content, metadata, document structure, and informal to formal semantic annotations.

While querying the semantic wiki KiWi is the main focus of this dissertation, the underlying ideas apply more generally to querying and search on the social and social semantic web.

1.1 CONTRIBUTIONS

The contributions of this dissertation are as follows:

- **A survey of keyword query languages for semi-structured data:** Web search and web queries have mostly been treated separately in the past, but this has begun to change over the past few years. A particular effort towards combining the two are keyword-based web query languages for XML and RDF documents. We give an overview, to the best of our knowledge the first of its kind, over the most important issues, aspects, and approaches in this area of research, and discuss achievements, limitations, and open problems.
- **A conceptual model for the KiWi wiki:** Several semantic wikis have been put to practical use, but so far there has been little explicit theoretical exploration on the possible choices for conceptual models and their consequences. We show that the design of a concept model for a semantic wiki is a non-trivial task and discuss possible design choices and their advantages and disadvantages. Based on this, we suggest a conceptual model for the KiWi wiki.
- **Structured tags, a mechanism for semi-formal annotation:** To help overcome the limitations of simple, free-form tags and to enable a transition between informal and formal annotation, we introduce an annotation formalism that is easy to use but more expressive than common tags. We then report on the findings of an extensive user study we performed. The results indicate that structured tags are well-suited as a user-friendly, flexible alternative in situations where casual users annotate evolving knowledge. We also describe how structured tags could be implemented in the KiWi wiki in a simple, intuitive way that does not require an extension to the wiki's conceptual model.
- **Accessible and flexible querying:** The central topic of this work is the development of the semantic wiki query language KWQL. We argue that a query language for social semantic data should be accessible and flexible and describe how current approaches fall short of these criteria. We then describe how the two characteristics could be realized in a query language.
- **KWQL, a semantic wiki query language:** Based on the developed requirements, and on the underlying idea of querying that adapts its expressivity—and thereby its simplicity—to the user's information need and knowledge, we describe the syntax of KWQL in an example-driven manner and provide a relational semantics for the language.
- **visKWQL, a visual rendering of KWQL:** Editors for visual languages support the creation of valid queries by providing user guidance and preventing editing operations that would result in incorrect queries. We introduce visKWQL, a visual interface for

KWQL, and describe its functionality and features for supporting users in the query creation process.

- **An evaluation algorithm for KWQL queries:** We describe KWilt, an evaluation procedure for KWQL bodies that combines information retrieval, structure matching, and constraint evaluation tools in a patchwork fashion. We show that it is possible to efficiently recognize KWQL queries that can be evaluated using only information retrieval or information retrieval and structure matching. This makes it possible to evaluate basic queries at almost the speed of the underlying search engine, but provides the power of full first-order queries where needed. Using a prototypical implementation, we compare the evaluation times for various types of queries and, based on the findings, describe how the evaluation algorithm could be improved.
- **An experimental evaluation of KWQL and visKWQL:** We describe the setup and results of a study performed to determine how KWQL and visKWQL are perceived by users and how easy it is for them to learn to write and understand KWQL and visKWQL queries. We compare the findings between participants that have some experience with other query languages and those who do not, as well as between participants who used KWQL and those who used visKWQL. We found that KWQL and visKWQL were well perceived by the participants, who specifically thought that the languages are expressive and easy to use, at least given some time and practice. Even after a very short introduction and a small amount of time to solve the assignments, participants could on average write correct queries for more than half of the tasks they were assigned and understand more than eighty percent of the queries they were presented with.
- **PEST, fuzzy matching and ranking over structured data:** We present PEST, an approach to approximate querying of graph-structured data that exploits the structure to propagate term weights among related data items. We focus on data where meaningful answers are given through the application semantics. This includes pages in wikis, but also persons in social networks or papers in a research network. The PEST matrix generalizes the PageRank matrix with a term-weight dependent leap and allows different levels of (semantic) closeness for different relations in the data. The eigenvectors for all terms together form a (vector space) index that takes the structure of the data into account and can be used with standard document retrieval techniques. In extensive experiments including a user study on a real-world wiki we show how pest improves the quality of the ranking over a range of existing ranking approaches.

In addition, two practical contributions are available in the form of software:

- A prototype implementation of KWilt, the KWQL query evaluation engine described in Chapter 10, and a lightweight implementation of the visKWQL editor that runs on the client side and is based on standard technologies are part of the latest release of the KiWi wiki. Queries can be posed via a textual query interface, as embedded inline queries, or from the combined visual and textual query editor. A showcase installation of the KiWi wiki is available at `http://pms.ifi.lmu.de/kwql/` and the KiWi wiki software can be downloaded at `http://www.kiwi-community.eu/display/SRCE/Get+KiWi/`.
- A prototype implementation of PEST, together with the dataset used for the evaluation described in Section 11.5, is available for download at `http://www.pms.ifi.lmu.de/pest/`.

1.2 STRUCTURE OF THIS THESIS

The body of this dissertation is structured into four chapters.

The first part lays the foundation for the rest of the text by giving an overview of relevant technologies and developments in the areas of the semantic web, semantic wikis, and web querying. Chapter 2 discusses the idea, formalisms, and technologies underlying the semantic web, as well as two recent directions that semantic web research has taken: Linked Data and the social semantic web. Chapter 3 reviews existing semantic wikis and the facilities they provide for searching and querying. Chapter 4 provides a brief introduction into XML and RDF and an overview of web query languages. The second part of the chapter is devoted to a survey of keyword query languages for semi-structured data and a discussion of the advantages and limitations of the approach.

The second part is concerned with the KiWi wiki and its conceptual model. In Chapter 5, we propose a conceptual model for the KiWi wiki and discuss the design choices involved. We also suggest structured tags, a formalism for semi-formal annotations. Chapter 6 reports on a user study that compares how RDF and structured tags are used and perceived in a text annotation task.

Part three introduces the KiWi query language KWQL. Chapter 7 provides an overview of KWQL, discusses the underlying principles and ideas, and presents its syntax and semantics. Chapter 8 briefly summarizes previous work in the area of visual query languages and then describes visKWQL, a visual rendering of KWQL, and an editor for visual and textual queries. Chapter 9 then describes the setup and results of a user study performed to evaluate the suitability of KWQL and visKWQL for querying in the KiWi wiki. Chapter 10 describes an evaluation procedure for KWQL bodies using a patchwork approach and reports on the

findings of an performance evaluation of a prototype implementation of the system.

The fourth and final part describes several extensions to KWQL. PEST, discussed in Chapter 11, is a scheme for exploiting connections between data items to extend the set of query answers by results that do not strictly match the query but that are still relevant, and for re-ranking results based on their structural relationships. Chapter 12 describes how structured tags could be implemented in the KiWi wiki and how KWilt could be modified to support the evaluation of queries over structured tags. Chapter 13 discusses how KWQL could be extended to RDF querying.

Chapter 14 finally provides an outlook and a conclusion.

Part I

PRELIMINARIES

2 THE SEMANTIC WEB

In this chapter, we aim at giving an overview over the idea, formalisms, and technologies of the semantic web. We then summarize criticism that has been directed at the semantic web and introduce two recent directions that semantic web research has taken: Linked Data and the social semantic web.

2.1 VISION, ACHIEVEMENTS, AND CHALLENGES

> The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.
>
> (Berners-Lee, Hendler, and Lassila [55])

The world wide web, as it has been used by an increasing number of users over the past twenty years, consists of interlinked documents containing text and multimedia. Despite advances in the fields of natural language processing and artificial intelligence, the meaning of the content of these documents can only be processed and understood by humans. The idea behind the *semantic web*, first prominently presented by Berners-Lee et al. [55] in 2001, is to change this, by shifting the focus from documents to data and making the meaning of data on the world wide web accessible to computers and thereby amenable to automatic semantic processing.

Giving information on the web a well-defined meaning could enable the development of computer agents that behave intelligently without the help of large-scale artificial intelligence systems. These agents could then autonomously perform complex tasks like scheduling on behalf of humans by querying, reasoning, and the exchange and combination of data from trusted sources.

This functionality is to be realized through a semantic web that is an extension of the human-readable web, and in which information is represented as structured data via the *Resource Description Framework* (RDF, see Section 4.1.2). The entities used in these descriptions are represented as and semantically grounded in *Uniform Resource Identifiers* (URIs) serving as unique identifiers. The underlying vocabulary or schema of a set of RDF data is defined by an *ontology*, a description of the concepts that can exist and the relations between them.

When RDF data is exchanged or combined, ontologies enable the detection of differences between the conceptualizations of

the data. *Ontology mapping* or *ontology alignment*, the combination of different ontologies into one, are used to integrate the data, thereby ensuring the interoperability between different systems and data sources. Using RDF data, ontologies, and mechanisms for querying, reasoning, and data integration, computer agents are then envisioned to solve sophisticated tasks by aggregating and exchanging information, and deducing new information as needed.

Artificial intelligence never succeeded in creating intelligent agents that display human-like "common sense" [183], despite much initial confidence within the research community and the success of AI systems that are domain-specific and narrow in scope. By contrast, the semantic web is seen as a simpler and more generic approach which, once realized, could enable computer agents to perform a wide range of tasks that require intelligent behavior.

In terms of the technologies involved, the semantic web is based on formalisms for expressing information, ontologies and mechanisms for ontology mapping and information exchange, as well as methods for querying, reasoning and determining the provenance and trustworthiness of information.

In the nine years since the original article about the semantic web vision was published, a large body of research has been dedicated to realizing the semantic web vision. Available semantic web technologies include RDF Schema (RDFS) [70] and the Web Ontology Language (OWL) [171], various XML serialization formats for RDF [42, 34], the query language SPARQL (see Section 4.2.2.1), Rule Interchange Format (RIF) [64], Resource Description Framework in attributes (RDFa) [11], RDF triple stores [366, 72] and tools for ontology development [284, 129].

Semantic web research is of course not just limited to the development of languages and tools; other topics that have been investigated include ontology evolution [283] and alignment [139], scalability and performance [6], trust and provenance [26], reasoning [82], and the role of logic and logic languages on the semantic web [83, 141].

Despite all the efforts aimed at realizing the semantic web vision, however, and despite the fact that RDFa, a simpler version of RDF, is employed by a number of large websites including Facebook and Google, the semantic web vision has not yet become a reality, and semantic web technologies so far have not been adopted by a significant, non-technical user base. Possible reasons for this have been widely discussed [263, 192, 183, 159, 336, 9, 47]. They can be grouped into three interdependent categories, which will be summarized in the following.

THE SEMANTIC WEB VISION HAS BEEN MISUNDERSTOOD

> Considering that the very word "semantic" is all about meaning, it's ironic that the term "Semantic Web" is so ill defined. (Nova Spivack [271])

According to Berners-Lee et al. [55], the ultimate goal of the semantic web is a scenario where automated computer agents assist humans in everyday tasks. However, this view is not shared by all advocates of the semantic web. The understanding of what constitutes the semantic web and what can actually be achieved has evolved during a decade of semantic web research, and the semantic web vision can also be taken as an aspirational motivating scenario for what currently is a more humble realization of the semantic web. For these reasons, there are a number of different views on the nature and goals of the semantic web, which give rise to different expectations.

Marshall and Shipman [263] discuss different views of the semantic web. According to them, the semantic web is portrayed in three ways—as a way to bring order to the web and enable targeted search, access and data collection ("taming the web"), and as a distributed knowledge base that automated agents can use to solve a wide variety of tasks and in which the knowledge base itself, technology, and application scenarios are either generic ("knowledge navigator") or targeted at a specific, limited domain ("federated data/knowledge base").

This categorization likely still holds today, although the number of proponents of each view may well have changed over time. A view similar to the "taming the web" scenario is widely accepted [191, 311, 271] (see also Section 2.2.1), but may be seen only as an intermediate step on the way to a more powerful semantic web.

MANY SEMANTIC WEB TECHNOLOGIES ARE BASED ON UNREALISTIC ASSUMPTIONS

> [A]ll sufficiently broad-based reasoning about the natural world must eventually reach conclusions that are incorrect, independent of the reasoning process used and independent of the representation employed. Sound reasoning cannot save us: If the world model is somehow wrong (and it must be), some conclusions will be incorrect, no matter how carefully drawn. A better representation cannot save us: All representations are imperfect, and any imperfection can be a source of error.
>
> (Davis, Shrobe, and Szolovits [127])

The semantic web approach has been met with some skepticism regarding its feasibility and practicality in real-life knowledge

management scenarios, especially when it is applied in a broad domain as intended in the original semantic web vision [263, 192, 183, 159, 336, 9]. In particular, it has been criticized for failing to solve problems that have been encountered before in the context of philosophy, artificial intelligence, and knowledge management such as the symbol grounding problem [186, 159, 118] and the frame problem [268, 183].

A large part of this criticism concerns ontologies, which represent the world view behind a dataset and therefore form the basis for system interoperability and reasoning. Ontologies constitute a priori agreements on the concepts in a domain and their relations. They are finite and static, represent only one point in time, and do not easily support inconsistency, vagueness, disagreements, changes, context-sensitivity, and evolving understanding of a situation. As such, an ontology can only be a limited, simplified model of the real world, which in turn has repercussions on the accuracy and the validity of reasoning and the applicability of the model to the world, meaning that computer agents can generally not be trusted to solve tasks correctly.

Even disregarding more sophisticated semantic web functionalities and general purpose scenarios, the fact that ontologies do not represent crucial characteristics inherent to human cognition, knowledge management, and interaction makes them less practical and relevant to real-life knowledge management scenarios.

A practical point concerning the development of ontologies was raised by Hepp [192]: depending on the domain and the level of abstraction, maintaining an ontology and updating it to always reflect the current state of knowledge could be hard or even impossible, or it might be so resource-intensive that the initial cost is ultimately not offset by the benefits.

Since all the information in an ontology is expressed via symbolic structures that are not grounded in the real world, and since ontologies by their very nature can only offer a biased view on the world, it is difficult to exactly match meaning when combining two ontologies, let alone to do so automatically [159, 336, 118].

SEMANTIC WEB TECHNOLOGIES FOCUS ON RICH FUNCTIONALITY, NOT ON USABILITY

> Semantic Web is one of the enabling technologies, a means to an end, and not the end itself. Every time I look critically at the current use of (information) technology, I cannot help but wonder how it is possible to actually get away with the approach taken today (where substantial burden is placed on the users).
>
> (Lassila [237])

Semantic web research for the longest time was concerned with creating technology and tools that are complex, powerful, and ex-

pressive, but neglected the issue of usability and did not provide users with an incentive to use semantic web technologies. This focus on rich functionality over simplicity and user-friendliness likely is another reason why semantic web technologies do not yet enjoy a wide user base. RDF and OWL may not be overly complex as far as formal languages go, but they are far beyond the experience of most web users, who would have to exert considerable effort to learn them.

At the same time, there is little incentive for users to provide RDF annotations. This is problematic insofar as the semantic web ultimately cannot rely on expert content creators and automatic extraction alone to generate semantic annotations. Users, on the other hand, are rarely willing to adapt their behavior to the needs of technology without an immediate benefit.

Not only can RDF be hard to learn for the layman, expressing knowledge via rigid formal structures is also very different from human cognition and thus not accommodating to users. Specifically, tacit and evolving knowledge, premature structure, and situation-specific knowledge are known to cause problems when users attempt to express their knowledge in formal representations [335].

Even if users could be assumed to spend some time learning RDF and were willing to provide annotations, several problems would remain:

- Semantic web tools in general are not very forgiving with respect to errors, inconsistencies, or disagreements that can easily arise when several users with imperfect knowledge collaborate.
- The original semantic web vision and semantic web research are concerned only with the trustability of individual sources. Such a coarse-grained concept of trust may be insufficient since a person might be very trustworthy with respect to one area, but not with respect to another one [263].
- When users annotate data with RDF vocabulary from a specific ontology, they commit to the expressed world view [192] even though they might not agree with it. However, understanding an ontology in full detail is hard and requires considerable effort.

2.2 NEW DIRECTIONS IN SEMANTIC WEB RESEARCH

Over the past few years, two emerging areas in semantic web research have received considerable attention: linked data and the social semantic web. They share a practice-driven, bottom-up approach to the semantic web and give up on the goal of "ontological purity." We give an overview over the two areas in this section.

2.2.1 *Linked Data*

The term "Linked Data" refers to structured data, typically using RDF, that is published on the web and obeys the following four principles:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs, so that they can discover more things.

(Berners-Lee [53])

Linked Data that is freely accessible is referred to as *Linked Open Data*. The idea behind Linked Data is to create a "web of data," a global database that is structured and interlinked. Both ontology and instance data can be queried, combined, and used, for example in the form of mash-ups or domain-specific applications [60, 27].

While the realization of the semantic web vision remains one of the goals, Linked Data puts the focus on structured datasets, creating additional semantics and value by interlinking them, and using the data for practical applications. In other words, Linked Data is concerned with exploiting current possibilities using comparatively simple and established tools, but it is also considered a means to realize the semantic web vision [60].

Linked Data still faces some of the same problems discussed in the previous section, for example concerning schema mapping and trust and privacy [60]. On the other hand, its pragmatic, bottom-up approach and the fact that it does not currently strive for or rely on the automatization of sophisticated knowledge tasks allows it to avoid many of the problems mentioned.

Over the past three years, Linked Data has received much attention and the Linked Open Data cloud has constantly grown. As of October 2010, it consists of more than 25 billion triples from 203 datasets. A snapshot is shown in Figure 1. Most of the data was extracted from other data sources concerning a broad range of different areas such as music, geography, law, genetics, and scientific publications.[1]

2.2.2 *The Social Semantic Web*

Up to the middle of the previous decade, the world wide web was mostly used to retrieve information, while the means for

1 Details can be found at http://www4.wiwiss.fu-berlin.de/lodcloud/.

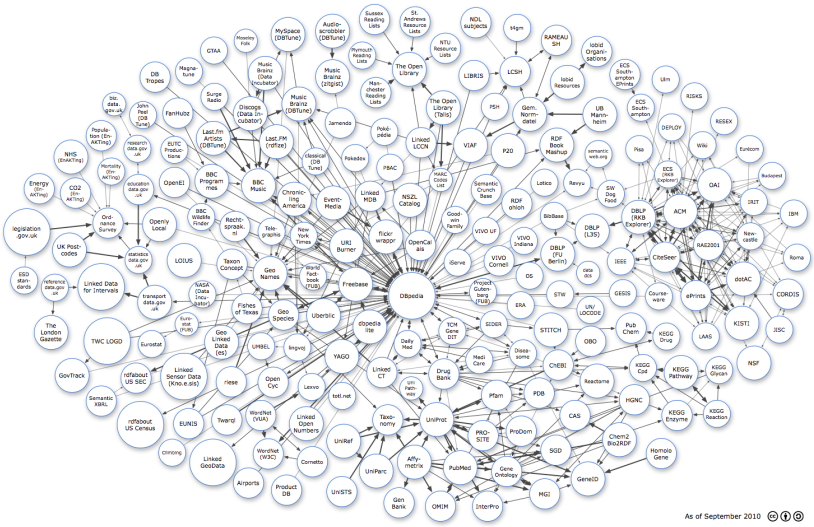


Figure 1: Snapshot of the Linked Data cloud, maintained by Richard Cyganiak and Anja Jentzsch at http://lod-cloud.net/

publishing data on the web and interacting with other users were limited. Thanks to the availability of free webspace and WYSIWYG editors, even laypeople could create and publish their own website, but the possibilities for publishing small amounts of data or for contributing to the content of another website were limited. Websites or even the web in general were "read-only."

This changed with the emergence of the *social web*, also referred to as the "web 2.0" or "read-write web." Social websites are based on user interaction, user-created content, collaboration, and information sharing. A multitude of different types of social websites have become popular over the past few years, for example blogging and microblogging platforms, wikis, and websites for bookmarking, cataloging, and networking.

While individual social websites differ greatly in their goals, application domains, and the nature and extent of the data contributed by their users, they share a low barrier for participation, and the fact that they derive their value from user-generated content and facilitate interaction and in some cases collaboration among users.

In addition to textual and multimedia content, many social websites support *tags*, simple semantic annotations in the form of freely chosen keywords that can be assigned to data items like books, songs, blog posts, photographs, or bookmarks. Unlike RDF data, which may or may not be attached to a specific document, tags by themselves express little information, but only become useful through the association with a data item and, since tag assignments are often subjective, a user. Tagging systems are thus commonly viewed as tripartite graphs [370, 184, 274]. Tags are used to categorize, describe, and group data items, thereby

facilitating search, navigation, or visualization in the form of a tag cloud.

The fact that tagging enables the ordering and retrieval of content provides an incentive for assigning tags. In many applications, tags are also pooled across users, meaning that a user can see which tags other users have assigned to a given resource and which items have been assigned a given tag. The aggregated collection of tags used in a system is called a *folksonomy*, a portmanteau of "folk" and "taxonomy." Unlike a taxonomy, a folksonomy does not natively possess a hierarchical structure or, for that matter, any structure at all.

In a tagging system, no meaning is assigned to strings a priori, and no specific string is designated to express a concept. Instead, the meaning of a tag emerges as individual users assign it to different data items. Users thus do not have to commit to a predefined view on categorization. Conversely, the meaning of a data item is given by the tags assigned to it.

In a study of tags used on the social bookmarking web site delicious[2], Golder and Huberman [161] found that the frequency distribution over the different tags assigned to a bookmark stabilizes after about 100 tag assignments. Possible reasons for this convergence are imitation and shared knowledge. In a followup study which again used data from delicious, Halpin et al. [184] showed that tagging distributions over time converge to power law distributions where a small number of tags are used consistently and a large number of tags are assigned rarely.

A consequence of the large degree of flexibility that accommodates different points of view is that the same concepts are often expressed via different tags, e.g., synonyms, tags in different languages, or inflections or different spellings of the same term. Similarly, tags of the same form may have different intended meanings due to polysemy and homography [162]. Yet another problem is *basic level variation*, i.e., varying degrees of specificity in tagging depending on the level of expertise in a given area [162]. Since in a folksonomy no explicit relations exist between tags, the relationship between tags referring to similar or identical concepts at different degrees of specificity is not recognized.

All these factors have a negative influence on the consistency of the emergent semantics and on users' understanding of the tagging conventions. The practical and social aspects of this are interrelated and mutually reinforce each other, since users who are unsure about tagging conventions are more likely to break them.

The success of tagging has prompted extensive research in the area, investigating the properties of tag sets and tagging

2 http://www.delicious.com

behavior [190, 161, 48, 261], the suitability of tags for improving search results [59], and motivations for tagging [382, 21].

Unlike the semantic web, the social web was not driven so much by research or technological innovation (although some technological innovations such as Ajax facilitated the realization of interactive websites), but rather by practical applications and changes in how the web is seen and used. As such, the social web is concerned with social factors and practicality rather than technological or scientific innovation.

The semantic web and social web have emerged mostly independently of each other, but they share some of their goals and characteristics: they both aim at collecting and organizing information that is, at least in part, generated by the users. However, while the semantic web is based on the formal representation of palpable facts, user-generated data in social web applications consists mainly of text or multimedia and often expresses differing points of view, uncertainty, and, especially in the case of wikis, work in progress.

In recent years, scientific efforts have been made to extend social web applications with semantic web technologies, in order to create systems that are user-friendly, error-tolerant, and take a bottom-up approach to knowledge management, but that also allow for the formalization of concepts and use semantic web technologies where appropriate. In the following, social web applications that employ user-provided semantic annotations will be called "social semantic web applications." Together they form the *social semantic web*.

It is worth noting that this definition includes not only applications which make use of RDF or other semantic web technologies, but also those that only use informal annotations in the form of tags: while tags alone are less powerful than formal annotations backed by ontologies, they do describe the meaning of data items and can form the basis for a (manual or automatic) transition to more formal annotations and the addition of advanced functionalities. Consequently, what makes a web application semantic in our view is not the use of specific technologies or expressive formalisms, but rather the presence of some form of semantic annotation that, if required, can serve as a starting point for a further formalization of knowledge.

A number of recent articles have pointed out the benefits of combining the social and the semantic web to achieve collective intelligence [172]. Mikroyannidis [276] points out that "[w]ith their tendency to form stable structures, folksonomies can potentially bridge the gap between the Social and the Semantic Web" [276, p. 114] and that "[i]n addition, ontologies derived from folksonomies would represent online communities' collective intelligence rather than the perception of a limited group of

experts." Ankolekar et al. [25] emphasize that the two fields need to draw from each other's strengths.

Several approaches for combining the social and the semantic web and enhancing social web applications with semantic web technologies have been investigated over the past few years. Topics in this area include the development of ontologies for tagging [173], the extraction of ontologies from social network graphs [274] and folksonomies [124], the inference of global semantic models or tag hierarchies from folksonomies [370, 193, 90], collaborative ontology evolution [251], tag similarity measures [89], reasoning over tags [81], and the effect of tag suggestions on folksonomy size [229].

3 SEMANTIC WIKIS

In this chapter we introduce wikis and semantic wikis and give an overview the search and querying functionalities provided by current semantic wiki engines.

3.1 WIKIS: COLLABORATIVE CONTENT CREATION

Wikis[1] are web applications for collecting and sharing knowledge. They allow users to easily create and edit documents, so-called *wiki pages*, using a web browser. The pages in a wiki are often heavily interlinked, which makes it easy to find related information and browse the content. While a wiki may or may not use access control to decide who may view and edit the content, a characteristic common to all wikis is that the content is version-controlled, meaning that older versions of a wiki page can be restored at any time. This feature does not only counteract "vandalism," i.e., intentional acts aiming to destroy or falsify the content of a wiki page, but can also be used to correct unintentional mistakes and to display the progress or state of knowledge at a certain point in time.

The first wiki software, WikiWikiWeb, was developed by Ward Cunningham and released in 1995.[2] Since then, wikis have been widely adopted for a large number of applications, and their usage ranges from personal to collaborative knowledge management and from hobby purposes to corporate intranets. The best-known wiki, Wikipedia, is an online encyclopedia launched in 2001.[3] It now consists of 16 million articles in 272 languages and is one of the most-visited sites on the web. While for some people Wikipedia is synonymous with the concept of wikis in general, it is by no means the only widely known and publicly accessible wiki; other examples include wikiHow,[4] a collection of how-to guides, Wikiquote,[5] covering notable quotations, LyricWiki,[6] a database of song lyrics, and The TV IV,[7] a wiki covering TV shows.

Apart from the original WikiWikiWeb wiki engine, there exists a large number of wiki engines differing in their features,

1 "Wiki" is the Hawaiian word for "fast."
2 http://c2.com/cgi/wiki/
3 http://www.wikipedia.org/
4 www.wikihow.com/
5 http://www.wikiquote.org/
6 http://lyrics.wikia.com/
7 http://tviv.org/

implementation, and application area, for example MediaWiki,[8] Atlassian Confluence,[9] and PhpWiki.[10]

In many respects, wikis are a prototypical social web application, and their success is tightly connected to the proliferation of the social web. In particular, wikis are conceptually simple, easy to use, and support users in the content creation process.

The basic elements of the conceptual model of a wiki are wiki pages and links between them. Creating or editing a wiki page is no harder than using a word-processing application, and content can be formatted using WYSIWYG editors or wiki markup. While not all features of a markup language like that of MediaWiki are easy to use, their use is optional and users can edit wiki pages without knowing anything about the syntax of the markup language. Wikis are particularly well-suited for the collaborative, gradual creation of content, and they live from user participation: a wiki page may start out as a short outline and grow and evolve as more people participate or more details become known. Discussion pages allow wiki users to exchange and, if necessary, align their views on how a wiki page should cover a given topic. A typical wiki pages is edited and enhanced repeatedly, meaning that a final, definite version does not necessarily exist, but that each wiki page is a perpetual work in progress.

At the same, wikis as knowledge management applications could profit from improved methods for structuring knowledge, making it more accessible and amenable to automatic processing. As mentioned above, wiki pages are often heavily interlinked, meaning that related concepts are often connected. In terms of structuring knowledge, this is a valuable contribution and, after all, one of the core principles of Linked Data (see Section 2.2.1). Individual wiki pages, on the other hand, are often weakly structured and only express knowledge as free text or multimedia.

Wikipedia allows wiki pages to be assigned one or more categories. In addition, many articles contain so-called *infoboxes*, fixed format tables that summarize an article by listing the values of certain attributes that depend on the category of the concept described. The English language Wikipedia article about Munich,[11] for example, has been assigned the categories "Cities in Bavaria," "1158 establishments" and "Host cities of the Summer Olympic Games," among others. Its infobox provides values for some attributes of a city, for example the country and state the city is located in, its population size, and the time zone.

To exploit these structuring mechanisms, a scheme for extracting taxonomies from Wikipedia category assignments has been

8 http://www.mediawiki.org/
9 http://www.atlassian.com/software/confluence/
10 http://phpwiki.sourceforge.net/
11 http://en.wikipedia.org/wiki/Munich

suggested [341]. Relatedly, DBpedia Knowledge Base [27] is a large set of Linked Data that consists of RDF triples extracted from Wikipedia's infoboxes, categories, internal links, etc.

While extracted RDF data can be queried using SPARQL or other RDF query languages, the structure of the data cannot be used for querying Wikipedia directly. Search in Wikipedia is limited to full text search, with links, categories and table of contents supporting the location, browsing, and navigation of information. Even though Wikipedia provides more mechanisms for the structuring of data than many other wikis, their use is limited, since only a small part and very specific aspect of the content of a page is expressed and since users cannot assign arbitrary annotations.

In summary, the content of traditional wikis consists of natural language text and possibly multimedia files and is not directly accessible to automated semantic processing. Therefore, knowledge in wikis can be located only through basic user-generated structures, like tables of content and inter-page links, and through simple full text keyword search. More advanced functionalities that would be highly desirable in knowledge-intensive contexts, such as querying, reasoning, and semantic browsing, are not available.

3.2 SEMANTIC WIKIS: CONCEPTS AND SYSTEMS

Semantic wikis extend conventional wikis by combining the wiki philosophy with semantic web technologies and introducing capabilities for specifying knowledge not just in natural language but also in more formal, machine-processable ways.

The term "semantic wiki" is used to refer to two different types of systems [236, 321]: Semantic wikis of the first type ("wikitology" [221] or "wikis for semantics") use wiki technology as a means for the collaborative authoring of ontologies. The main focus here is on creating semantic web data, and human-readable wiki content is only needed to support the editing process. When used in the second sense, "semantic wiki" refers to a wiki that uses (social) semantic web technologies to enhance the functionality of the wiki and support the process of collaborative content creation ("semantics for wikis"). Here, the focus is not (only) on metadata, but text and multimedia content.

Some semantic wiki engines fall clearly into one of these two categories, while others can be used for both purposes [321]. In the following, we use "semantic wiki" in the second meaning.

Semantic wikis extend conventional wikis by providing functionalities for expressing knowledge in a structured form. This is realized mainly by adding support for annotations to data items, most frequently wiki pages and tags, but also smaller portions of

text [218]. The annotations may be freely chosen tags [142], but sometimes more formal mechanisms such as RDF backed by (imported) RDFS or OWL ontologies are offered as well. In particular, several semantic wikis support limited RDF annotations where the subject is always the URI of the annotated resource, and predicate and object are provided by the user [344, 31, 32]. Some semantic wikis also enable the editing of ontologies, meaning that not only the metadata annotations, but also their schemata evolve over time.

The annotations, whether they have been assigned manually or extracted (semi-)automatically, can be used for realizing functionalities like consistency checking, improved navigation, search, querying, personalization, context-dependent presentation, and reasoning. Annotations are often represented as RDF. They can thus be exported and integrated with data from other sources and are compatible with standard RDF technologies such as SPARQL.

The annotation of wiki content is optional, and semantic wikis do not require users to add annotations. While in particular only some of the users may actually annotate content, this can still enable all users of the semantic wiki to benefit from the functionalities that semantic wikis offer over conventional wikis, for example an automatically generated table of contents [321]. Furthermore, the semantic wiki data may be formalized in a collaborative fashion over time, with different users providing the textual content and informal and formal annotations. This holds especially when different modes of annotations are available, for example free-form tags and RDF. Semantic wikis thus maintain, at least to some extent, the ease of use of conventional wikis.

Semantic wikis are sometimes referred to as the "semantic web in the small:" they consist of pages, links, and annotations, all of which are typically created by a number of different people. However, unlike the world wide web, semantic wikis to some extent are subject to central control in the form of administrators, and the number of users and topics and the amount of data are much smaller and often more homogeneous than those of the web. This makes semantic wikis ideal testbeds for novel semantic web technologies.

Currently, most semantic wiki engines provide few advanced functionalities beyond RDF querying and, in some cases, basic reasoning [319, 132]. To better leverage the potential of the combination of the social and the semantic web, and to provide more sophisticated and user-friendly functionalities, additional research in several areas is required. Relevant directions include querying and search (see Section 3.4), personalization, user-assisted extraction of annotations, visualization, reasoning maintenance, and inconsistency-tolerant reasoning [227, 322, 297].

3.3 SEMANTIC WIKIS: TWO EXAMPLES

Now that we have given an overview over semantic wikis and their characteristics, we will discuss two semantic wikis in more detail, Semantic MediaWiki and IkeWiki. Semantic MediaWiki is one of the oldest and most popular semantic wikis, and its data model has been thoroughly described in the scientific literature. IkeWiki is a feature-rich semantic wiki that is the predecessor of the KiWi wiki, the wiki that is one of the focal points of this dissertation.

3.3.1 *Semantic MediaWiki*

Semantic MediaWiki (SMW) [231] is realized as an extension to MediaWiki. In SMW, annotations can be added inline with the textual content in the wiki editor using a markup language that is based on MediaWiki's syntax for creating links. Aiming for the realization of a semantic Wikipedia [232], annotations are seen to describe the concept represented by the wiki page rather than the wiki page itself.

Annotations conceptually resemble simple RDF triples and take the shape of property statements that characterize a binary relationship between a wiki page and some (typed) value. These statements can be used to assign types to links and to augment wiki pages with class and attribute information. Like in MediaWiki, class information is expressed through the property "Category." For example, the annotation `[[Category:City]]` expresses that a wiki page represents a city. Semantic MediaWiki extends this to arbitrary properties of individual concepts. By default, a newly introduced property is assumed to take a wiki page as a value, i.e., the annotation is interpreted as a typed link between two wiki pages. However, SWM also provides several other data types that can be assigned to properties, for example "string," "date," and "geographic coordinate."

In addition to individual concepts, all categories, properties, and types are represented by wiki pages, and namespaces are used to distinguish between the different types of wiki pages. A property can be assigned a type through an annotation on its wiki page, and it can be designated as the subtype of another property. Similarly, just like in MediaWiki, wiki pages of categories can be annotated with category information, thus creating a category hierarchy. The wiki pages of categories, properties, and types can further be used to describe their meaning in a textual form.

While the category hierarchies of SMW constitute shallow ontologies that are not defined a-priori, the wiki also allows for the use of external ontologies, either via wiki pages representing

the elements and relations of the ontology or by mapping wiki pages to elements of existing ontologies [232].

SMW annotations are formally grounded through a mapping of annotations to OWL-DL statements, which can be exported as RDF data. The generated annotations can be used for semantic browsing and querying (see Section 3.4).

In one of the few scientific articles on the formal foundation of semantic wikis, Bao et al. [38] point out that a semantics based on description logic or RDF is not ideally suited for SMW annotations. In particular, a semantics based on description logic constrains the expressivity of SMW annotations, since categories, properties, and instances must be disjoint sets. This means, for example, that a property may not be used as a category instance and that a category cannot serve as a property value, although both of these cases have useful applications. Bao et al. [38] instead provide a model-theoretic semantics as well as an entailment system. They further describe the use of entailment rules in SMW, which can be triggered selectively in wiki pages and natively introduce a form of reasoning to SWM.

3.3.2 *IkeWiki*

IkeWiki[12] [319, 230] was envisioned as a system that supports users in the gradual formalization of knowledge in a collaborative fashion. It aims for user-friendliness and offers an interactive WYSIWYG editor for text and annotations as well as an editor supporting Wikipedia syntax, thus making it easy to copy and paste content from Wikipedia.

IkeWiki relies on semantic web standards for expressing and formalizing knowledge and is aimed at supporting a process in which the knowledge base and its degree of formalization evolve over time. To this end, it supports RDF and OWL annotations to links, as well as wiki pages containing text and multimedia.

Imported ontologies are used to determine which annotations can be assigned to a given link or wiki page; it is not possible to assign annotations that are not backed by the ontology. IkeWiki not only allows for the use of ontologies, but to some extent also for their editing, and specifically for the addition of sub- and superclasses, ranges, and inverse relations.

In contrast to Semantic MediaWiki, IkeWiki stores annotations separately from the content of the wiki pages, meaning that they are not versioned. Annotations can be leveraged immediately for presentation, navigation, and search. Unlike many other semantic wikis, IkeWiki also supports OWL-RDFS and OWL-DL reasoning. For example, class relationships and relationships between pages

12 "Ike" is the Hawaiian word for "knowledge."

are used to infer additional annotations and to make sure users only choose applicable concepts when annotating a data item.

3.4 SEARCHING AND QUERYING IN SEMANTIC WIKIS

Better search and querying is one of the main ways in which semantic wikis intend to improve upon conventional wikis. The need for simple yet powerful data retrieval [297, 30] and for combined queries over content and annotations [31] have been pointed out in particular.

So far, however, all semantic wikis that we are aware of treat the querying of content and annotations separately [297, 321], while other sources of data such as content structure and system metadata cannot be queried at all.

In many cases, semantic wikis provide simple full text search for the querying of textual content or RDF literals [218, 344, 295]. In addition, a standard RDF query language such as SPARQL or RDQL can often be used for querying the annotations [319, 142, 30, 28]. The embedding of queries in wiki pages [319, 150] allows those queries to be evaluated every time a page is loaded, and always shows an up-to-date list of results.

A number of semantic wikis also come with their own language for querying annotations that can be used in addition to or instead of a conventional RDF query language.

KAON, the query language of COW [150], can make use of simple reasoning to find query answers. The following retrieves physicists born in Europe, regardless of whether or not the data explicitly represents that their place of birth is located in Europe:

```
[#Physicist] AND
(SOME(<#is-born-in>.<#located-in>=!#Europe!)
 OR SOME(<#is-born-in>=!#Europe!))
```

Rhizome [339] and its query language RxPath aim at making RDF querying easy for users who are already familiar with XML. To this end, RDF triples are mapped to a virtual, possibly infinitely recursive tree which can then be queried with XPath expressions. The query `/foaf:Document/dc:creator/*` for example selects all authors of document resources.

WikSAR [33] uses queries consisting of a series of predicate-object pairs. The answer to such a query then consists of all wiki pages whose annotations match all predicate-object conditions. Predicate and object can be connected by operators for equality, ranges, and regular expressions. The following query for example

returns the set of pages describing authors that were born in 19th century England:

```
InstanceOf=LiteraryAuthor BornIn=~England
        DateOfBirth between 1800 and 1900
```

Two different query languages have been suggested for Semantic MediaWiki. The first, referred to as "SMW-QL" by Bao et al. [38], has a syntax similar to that used to express annotations in SMW. The simple query `[[Category:City]]`, for example, retrieves all wiki pages that have this annotation. Subqueries, denoted by `<q>...</q>`, ranges of values, or wildcards can also be used in place of fixed property values. SWM-QL further supports (implicit) conjunction, disjunction, negation and comparison operators, but no variables. The following query for example returns the wiki pages of cities that are located in the EU or have more than 500,000 inhabitants:

```
[[Category:City]]
<q>[[located in::<q>[[Category:Country]]
                    [[member of::EU]]</q>]] ||
   [[population:: >500,000]] </q>
```

By default queries return wiki pages, but so-called *print requests* can be used to display specific property values in the query answers. Krötzsch and Vrandecic [231] provide a semantics for SWM-QL through a translation to DL queries.

Bao et al. [38] present a variation of the language that uses a slightly different syntax. They point out that the open world assumption underlying the DL semantics are not well-suited for a wiki and is at odds with the implementation of SML-QL in SMW. Instead, the authors provide a semantics that is based on the translation of SMW-QL queries into logic programs.

The second query language of Semantic MediaWiki [181] employs keyword search over RDF data (also discussed in Section 4.3.3). Users express their query intent using a number of keywords. These keywords are matched in the data using a fuzzy scheme that considers semantic and syntactic similarity and assigns a score to each match. An augmented schema graph that combines the keyword matches and schema information is then constructed. Query graphs, connected subgraphs containing at least one match for each keyword, are then extracted from the augmented schema graph using a top-k procedure. The query graphs are translated into SPARQL queries and are displayed to the user in a visual, table-based form. The user can then select the query that corresponds to her query intent and the matching entity tuples are displayed together with a facets menu which can be used to further refine the results.

As mentioned above, the languages currently employed in semantic wikis do not allow for combined queries over content, annotation, and structure. Such queries would be useful and desirable because the textual content is often not expressed completely in the annotations, and vice versa [59]. Furthermore, semantic wikis are a social medium, so users should for example be able to use the metadata to retrieve all pages edited by a specific person, without having to rely on potentially incomplete or incorrect authorship information expressed manually via an annotation.

Usability and expressiveness of the above query languages vary widely. The RDF keyword query language of Semantic MediaWiki is the only annotation query language that allows users to formulate a query right away, without having to learn the language first. On the other hand, the keyword queries are not very expressive, and for example do not allow for the expression of disjunction and negation. To select the correct query, users further need to be able to understand the visualized SPARQL queries.

AceWiki [234] differs from all the approaches presented above in that it employs a controlled natural language, Attempto Controlled English or ACE [155], to represent information in the wiki. The language is a subset of English but can be translated into a variant of first-order logic, meaning that it can be understood by humans and machines alike. Consequently, there is no distinction between content and annotations in AceWiki. The authors suggest that using ACE, queries can simply be represented as questions.

4

WEB QUERYING

When we talk about web queries, we subsume two distinct areas of research and technology: Web search as provided for example by Google or Yahoo!, and database-style queries on web data (mostly in the form of XML or RDF) as provided through languages such as XQuery or SPARQL.

Web search acts as a filter to the discovery of documents on the web that are relevant to a specified query. Given the vast amount of available data, web search has become an indispensable tool for navigating the web for casual and proficient web users alike. Web search is easy and convenient to use; query intents are approximated through queries consisting of a number of keywords, which in practice is usually small [43, 35].

While many web search engines provide additional syntax for specifying for example disjunction, classical negation and phrase searches, research has shown that these advanced features are only employed by a small number of users [202, 43].

Given a particular query, a search engine sifts through an index of (a substantial portion of) all web data and retrieves the matching documents. The assessed relevance of a document to the query is not strict or boolean, but nuanced and often approximate. For example, returned documents might not contain a query term itself, but its plural or a synonym. The answer to a query is a ranked list of documents [71, 222].

In contrast to traditional information retrieval techniques for finding and ranking relevant documents, web search exploits not only the content of each individual document, but also their relationships as expressed through hypertext links. This information must be usable without sacrificing scalability to millions or billions of documents. Web Search engines harvest structural information as well as non-local search terms (e.g., anchor text used to link to a document) at indexing time only. This allows for the independent evaluation of a search request on each document (and its associated results of the harvesting process) and enables a highly parallel—and thus scalable—evaluation of web searches.

Thus, web search allows us to filter down the huge amount of web data to what is likely related to our search request. The downside of web search is that the results to a search request are often only vaguely related to the user's search intent and, in terms of their presentation, that only a document ranking is provided. In practice, the user has to look at the individual results to gauge their true relevance to his information need.

Database-style Web queries, formulated in languages such as XQuery or SPARQL, are in many respects the exact dual of web search: we peek inside a small set of homogeneous documents to find precise data items such as the price of a book or the capital of a country. These data items can then be processed automatically, for example to place an order for a book as soon as its price is below a certain threshold. We can also deduce new knowledge such as the number of books by a specific author, or the fact that some author has published in all of the top five conferences in a given research area and year. Such queries cannot be answered by a web search engine, unless the corresponding knowledge is available in document a priori. In contrast to traditional databases, database-style web queries operate on web data formats such as XML and RDF, the presumptive foundation for the semantic web. Both XML and RDF differ from the relational data model in that they allow for more flexible schemata where repetition and recursion are common. This pushes issues that have only been treated cursorily for relational data, such as the influence of tree queries or tree data on query evaluation or efficient reachability queries in trees and graphs to the front.

The price we pay for the ability to precisely select individual data items of a certain characteristic and process them automatically is twofold: First, web query languages are significantly more complex than web search interfaces. Writing correct, let alone efficient, web queries requires significant training and is comparable to a programming task. Second, most web query languages scale no better than traditional SQL database technology, and are clearly unable to process a significant portion of all web data.

In summary, where web search allows us to operate on (nearly) all the web, database-style web queries operate only on a small fraction. Where web search is limited to *filtering* relevant documents for human consumption, web queries allow for the precise selection of data items in web documents as well as their formatting, reorganization, aggregation, and the deduction of new data. Where web search can operate on *all kinds* of web documents, web queries are usually restricted to a more homogeneous collection of documents (e.g., XHTML documents or DocBook documents). Where web search requires a *human in the loop* to ultimately judge the relevance of a search result, web queries allow automated processing, aggregation, and deduction of data. Where web search can be used by *untrained users*, web queries usually require significant training to be employed effectively.

In the context of social semantic software, both aspects of web queries play an essential role: We want to be able to precisely specify selection criteria for data items and automatically derive new information, operations that squarely fall into the domain of database-style web queries. On the other hand, the essential

premise of the social semantic web is accessibility to untrained users. In this sense a mechanism closer to web search is needed.

Web search and web queries have mostly been treated separately in the past, but recently this has started to change in more than one way:

1. Web search engines are beginning to "peek" into web documents to provide more useful answers to queries. For instance, Google integrates querying of structured data about videos, images, and items from Google Base[1] into the search result listing. Yahoo provides similar features that allow content providers to use structured data to customize their search result listings.[2] More ambitiously, Google Squared, an experimental search tool, returns tables containing facts extracted from web pages as query answers.[3]

2. Considerable effort has been put into adding information retrieval functionality and primitives to XQuery and similar XML query languages. This line of research has culminated in a candidate recommendation by the W3C that is inspired by traditional information retrieval and proposes selection and ranking operators for XQuery [15]. An overview of relevant articles and proceedings can be found in one of the more recent tutorials on XQuery and XML retrieval [116, 20].

3. The most significant effort towards combining some of the virtues of web search, viz. being accessible to untrained users and being able to cope with vastly heterogeneous data, with those of database-style web queries are *keyword-based web query languages* for XML and RDF documents. Theses languages operate in the same setting as XQuery or SPARQL, but with an interface suitable for untrained or barely trained users instead of a complex query language. The interface is often (in *label-keyword query languages*) enhanced to allow not only bag-of-word queries but some annotations to each word, most notably a context (e.g., that a term must occur as the author or title of an article). Results are excerpts of the queried documents, though the precise extent is often determined automatically rather than by the user. Thus, keyword-based query languages trade some of the precision of languages like XQuery for a more accessible interface. The yardstick for these languages becomes an easily accessible interface (or query language) that does not sacrifice the essential premise of database-style web queries, namely that selection and construction are precise enough to allow for automated processing of data.

In the following, we focus on keyword-based web query languages as the most promising direction for combining the ease of

1 http://www.google.com/base/
2 http://developer.yahoo.com/searchmonkey/
3 http://www.google.com/squared

use of web search engines with the powerful features of database-style web query languages.

To ground the discussion of keyword-based query languages, we first give a summary of what we perceive as the main contributions of research and development on web query languages in the past decade (Section 4.2). This summary is focused specifically on what sets web query languages apart from their predecessors for traditional (mostly relational) databases. It comes in two parts, one on XML (Section 4.2.1), one on RDF (Section 4.2.2). For XML, we consider three contributions: *reachability* (as expressed, e.g., in XPaths descendant axis) in trees, how the restriction to *tree queries* and tree data enables highly efficient query evaluation, and the effect of *order* as a first class concept of the data model. For RDF we consider again three contributions: *reachability* in graphs, dealing with RDF's multi-valued, *optional* properties, and how existential information (or *blank nodes*) affects querying and construction.

In both discussions we also briefly introduce the preeminent exemplars of XML, resp. RDF query languages: XQuery and SPARQL. Where illuminating or necessary for the context we also reference other query languages. However, for more extensive introductions to and an extensive comparison of the mentioned query languages (and many more) we refer to previous surveys of XML and RDF query languages [36, 157].

The main part (Section 4.3) of this chapter is dedicated to keyword-query languages: We start with a brief overview of the principles and motivation of keyword-based query languages as well as their relation to web search. The main focus of research in the area of keyword query languages for semi-structured data has been on XML. Section 4.3.2 gives an overview over the most important issues, aspects and approaches in this field, namely determining semantic entities, determining return values, the expressive power of keyword languages and ranking.

Section 4.3.3 then individually presents various keyword query languages for RDF.

We conclude this survey with a summary of how keyword-based query languages for XML and RDF aim to bring the ease of use of web search together with the capabilities of traditional web queries. Further, we discuss where the existing approaches succeed in this aim, what, in our opinion, the most glaring open issues are, and where, beyond keyword-based query languages, we see the need, the challenges, and the opportunities for combining the ease of use of web search with the virtues of web queries.

4.1 DATA ON THE SEMANTIC WEB

This section introduces XML and RDF, two of the most prevalent and well-studied semi-structured web and semantic web data formats.

4.1.1 *Extensible Markup Language (XML)*

XML [69] is the foremost data representation format for the web and for semi-structured data in general. It has been adopted in a large number of application domains from document markup (XHTML, DocBook [354]) over video annotation (MPEG 7 [264]) and music libraries (iTunes[4]) to preference files (Apple's property lists [2]), build scripts (Apache Ant[5]), and XSLT stylesheets [215].

XML is a generic markup language for describing the structure of data. In contrast to HTML, the predominant markup language on the web, neither the tag set nor the semantics of XML are fixed. XML can thus be used to derive markup languages by specifying tags and structural relationships.

Our overview of XML is based on the XML Infoset [117], which describes the information content of an XML document. The XQuery data model [148] is, for the most part, closely aligned with this view of XML documents. We deviate from the Infoset, and instead follow the XPath and XQuery data model, by viewing XML data as *tree-shaped*.[6] This is in line with most XML query languages, notable exceptions include Xcerpt [84] and Lorel [10].

XML in 500 Words

The core provision of XML is a syntax for representing hierarchical data. Data items in XML are called elements and are enclosed in start and end *tags*, both carrying the same tag name or *label*.

The following listing shows a small XML fragment that illustrates elements and element nesting.

In this example, `<author>...</author>` is an element that contains other elements or character data as *children* between the start and end tag.

```
<bib xmlns:dc="http://purl.org/dc/elements/1.1/">
  <article journal="Computer Journal" id="12">
     <dc:title>...Semantic Web...</dc:title>
     <year>2005</year>
     <authors>
       <author>
```

4 http://www.apple.com/itunes/
5 http://ant.apache.org/
6 In the Infoset, valid ID/IDREF links are resolved, resulting in a data model that is a graph rather than a tree.

```
          <first>John</first> <last>Doe</last>
              </author>
8       <author>
          <first>Mary</first> <last>Smith</last>
              </author>
10    </authors>
  </article>
12  <article journal="Web Journal">
     <dc:title>...Web...</dc:title>
14    <year>2003</year>
     <authors>
16      <author>
          <first>Peter</first> <last>Jones</last>
              </author>
18      <author>
          <first>Sue</first>   <last>Robinson</last>
              </author>
20    </authors>
  </article>
22 </bib>
```

In addition, we can observe *attributes*, i.e., name-value pairs associated with start tags. Attributes are like elements but may only contain character data and no other nested attributes or elements. Also, by definition, element order is significant while attribute order is not. For instance

```
<author><last>Doe</last><first>John</first></author>
```

represents different information than the `author` element in lines 6–9, but

```
<article id="12" journal="Computer
    Journal">...</article>
```

represents the same element information item as lines 2–15.

Figure 2 shows a graphical representation of the above XML document. When represented as a graph, an XML document without links is a labeled tree where each node in the tree corresponds to an element and its type. Edges connect elements to their children (i.e., elements nested within them), their content, and their attributes. Since the parent-child relationship can be recognized without edge labels, and since attributes are not addressed or receive no special treatment in the research presented here, edges will not be labeled in the following.

Elements, attributes, and character data are the most common information types in XML. In addition, XML documents may also contain *comments*, *processing instructions* (name-value pairs with specific semantics that can be placed anywhere an element can be placed), *document level information* (such as the XML or

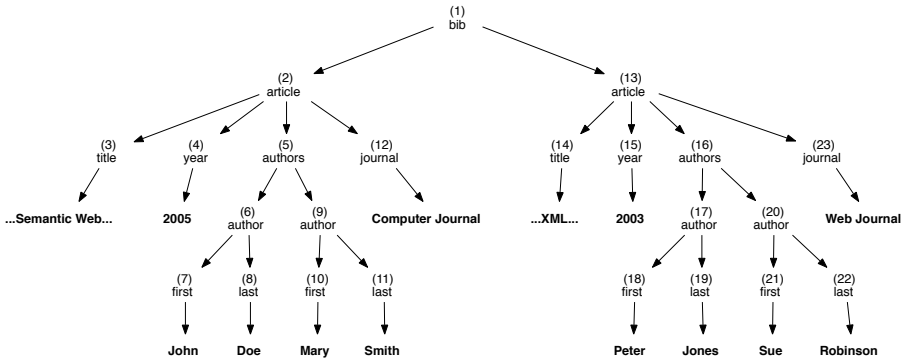


Figure 2: A graphical representation of an XML document

the document type declarations), *entities*, and *notations*, which are essentially just other kinds of information containers.

On top of these information types, two additional facilities relevant to the information content of XML documents have been introduced by subsequent specifications: Namespaces [68] and Base URIs [262]. Namespaces allow the partitioning of element labels used in a document into different containers, identified by a URI. Thus, an element is no longer labeled with a single label but with a triple consisting of the *local name*, the *namespace prefix*, and the *namespace URI*. For example, the `dc:title` element in line 3 has the local name `title`, the namespace prefix `dc`, and the namespace URI (called "name" in Cowan and Tobin [117]) `http://purl.org/dc/elements/1.1/`. The latter can be derived by looking for a *namespace declaration* for the prefix `dc`. Such a declaration is shown in line 1: `xmlns:dc="http://...` It associates the prefix `dc` with the given URI in the scope of the current element, i.e., for that element and all elements contained within, unless there is another nested declaration for `dc`, which would take precedence. Thus, we can associate with each element a set of in-scope namespaces, i.e., of pairs of a namespace prefix and a URI, that are valid in the scope of that element. Base URIs [262] are used to resolve relative URIs in an XML document. They are associated with elements using `xml:base="http://...` and, like namespaces, are inherited to contained elements unless a nested `xml:base` declaration takes precedence.

4.1.2 *Resource Description Framework (RDF)*

The *Resource Description Format (RDF)* [259, 223, 189] is emerging as the preeminent data format on the semantic web. While much less common than XML, RDF enjoys widespread use for interchanging (meta-)data together with descriptions of the schema

and, in contrast to XML, a basic description of the semantics of the data.

Not to distract from the salient points of the discussion, we omit typed literals and named graphs from the following discussion.

RDF in 500 Words

RDF graphs contain simple statements about *resources*, i.e., about elements of the domain that may partake in relations (in other contexts, these are called "entities," "objects," etc.). Statements are triples consisting of subject, predicate, and object, all of which are resources. If we want to refer to a specific resource, we use (supposedly globally unique) URIs; to refer to a resource for which we know that it exists and maybe to some of its properties, we use *blank nodes* which play the role of existential quantifiers in logic. Blank nodes may not occur as predicates. Finally, for convenience, we can directly use *literal values* as objects.

RDF may be serialized in many formats (for a recent survey see Bolzer [65]), such as RDF/XML [42], an XML dialect for representing RDF, or Turtle [34], which is also used in SPARQL. The following Turtle data represents roughly the same data as the XML document discussed in the previous section:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix bib: <http://www.edutella.org/bibtex#> .
@prefix ex: <http://example.org/libraries/#> .
ex:smith2005 a bib:Article ; dc:title "...Semantic
    Web..." ;
       dc:year "2005" ;
       ex:isPartOf [ a bib:Journal ;
           bib:number "11";  bib:name "Computer Journal"
               ] ;
       bib:author [ a rdf:Bag ;
           rdf:_1 [ a bib:Person ;
               bib:last "Smith" ; bib:first "Mary" ] ;
           rdf:_2 [ a bib:Person ;
               bib:first "John" ; bib:last "Doe" ] ] .
```

Following the definition of namespace prefixes used in the remainder of the Turtle document , each line contains one or more statements separated by semicolons, meaning that the subject of the previous statement is carried over. For example, line 6 reads as `ex:smith2005` is a (has `rdf:type`) `bib:Article` and has `dc:title` "...Semantic Web...." Lines 8–9 show a blank node. The article is part of some entity which we cannot (or do not care to) identify by a unique URI, but for which we give some properties: it is

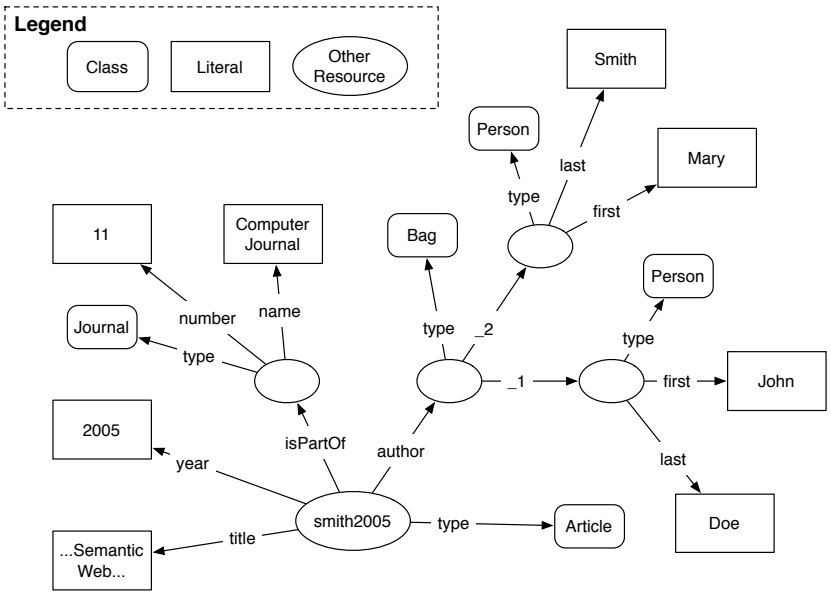


Figure 3: A sample RDF graph

a `bib:Journal`, has `bib:number` "11," and `bib:name` "Computer Journal."

Figure 3 shows a visual representation of the above RDF data. We denote classes (i.e., resources that can be used for classifying other resources) by square boxes with rounded edges, literals by square boxes, and all other resources by plain ellipses.

4.2 QUERIES AS PROGRAMS: DATABASE-STYLE QUERY LANGUAGES

Having introduced XML and RDF, the following two sections present web query languages for the two formats together with the most important research issues.

4.2.1 *Trees and Documents—XML*

As discussed in Section 4.1.1, XML differs from both the relational and previous semi-structured data models (as in Abiteboul et al. [10]) in its focus on ordered tree data. Both orderedness and tree-shape are direct consequences of XML's heritage as a simplified variant of SGML, which is primarily used for document markup. Documents in formats such DocBook [354] or (X)HTML exhibit an intrinsic hierarchical organization of the data and are strictly ordered, just like they would be in printed form. It is clearly not acceptable to reorder paragraphs even within the same section, or sections within the same chapter. While previous data models do allow the modeling of tree data and sometimes even ordered

tree data, XML is the first data format that limits itself to tree data while placing a premium on the maintenance of sibling and document order.

These novelties are reflected well in the contributions of XML query languages and will guide the following discussion. In Section 4.2.1.3 we illustrate how XML's focus on tree data pushes the issue of reachability (or descendant and ancestor) queries to the center stage and how different XML query languages address this issue. In Section 4.2.1.5 we then summarize the effect order as a first class citizen in XML has on XML query languages. Finally, we briefly recall how the limitation to tree data and consequently tree queries has lead to a number of novel evaluation strategies that are tailored to this setting and significantly outperform traditional, less focused approaches.

We start off the discussion of XML query languages with a closer look at two of the more prominent exemplars: XPath and XQuery. We focus on the essentials of these languages, and refer the reader to Bailey et al. [36] for a more in-depth comparison of more than two dozen XML query languages.

4.2.1.1 *XPath*

XPath provides an elegant and compact way of describing paths in an XML document viewed as an ordered tree. Paths are made up of "steps," each specifying a direction, or *axis*, for navigating through the document, e.g., **`child`**, **`following`**, or **`ancestor`**. An illustration of the different axes is shown in Figure 4. Along with the axis, a step contains a restriction on the type or label of the data items to be selected, called a *node test*. Node tests may be labels of element or attribute nodes, node kind wildcards such as `*` (any node with some label), **`element()`**, **`node()`**, **`text()`**, or **`comment()`**. Any step may be adorned by one or more *qualifiers*, each of which is denoted with square brackets and expresses additional restrictions on the selected nodes. The most distinctive feature of XPath, as compared to other query languages such as XQuery, SQL, or SPARQL, is the lack of explicit variables. This makes it impossible to express n-ary queries and limits XPath, for the most part, to two-variable logic (see Marx [267], Bojanczyk et al. [63] for details).

XPATH EXAMPLES. The XPath expression

```
/descendant::article/child::author
```

consists of two steps. The first step selects `article` elements that are descendants of the root (indicated by the leading slash), the second one selects `author` children of such `article` elements. More interesting queries can be expressed by exploiting XPath's

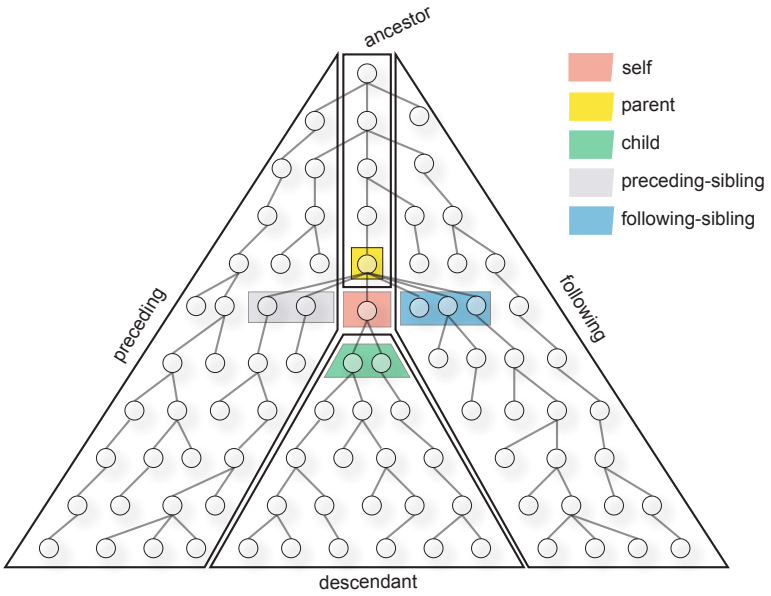


Figure 4: XPath axes, adopted from Genevès and Vion-Dury [160]

qualifiers; the following XPath expression for example selects all authors that are also PC members of a conference :

```
/child::conference/descendant::article/child::author[. =
      /child::conference/child::member]
```

In addition to the strict axis plus node test notation, XPath provides an abbreviated syntax where **child** may be omitted, **descendant** is (roughly) abbreviated by //, and the current node is referenced by .. In the following, we only use the full syntax and limit ourselves to the core feature of XPath as discussed here, thus presenting a view of XPath similar to Navigational XPath of Gottlob et al. [167] and Benedikt and Koch [46].

SYNTAX OF NAVIGATIONAL XPATH. The syntax of navigational XPath is defined as follows (again following Gottlob et al. [167] and Benedikt and Koch [46]):

For details on the semantics as well as differences to full XPath see Benedikt and Koch [46].

The theoretical properties of XPath have also been investigated in detail. *Formal semantics* for (more or less complete) fragments of XPath have been proposed in Wadler [353], Olteanu et al. [291], Gottlob and Koch [164]. Surprisingly, most popular implementations of XPath embedded within XSLT processors exhibit exponential behavior, even for fairly small data and large queries, while the *combined complexity* of XPath query evaluation has been shown to be P-complete [165, 166]. Various sublanguages of XPath (e.g., forward XPath [291], Core or Naviga-

⟨*path*⟩ ::= ⟨*step*⟩ | ⟨*step*⟩ '`/`' ⟨*path*⟩ | ⟨*path*⟩ '∪' ⟨*path*⟩ | '`/`' ⟨*path*⟩

⟨*step*⟩ ::= ⟨*axis*⟩ '`::`' ⟨*node-test*⟩ | ⟨*step*⟩'`[`'⟨*qualifier*⟩'`]`'

⟨*axis*⟩ ::= '`child`' | '`descendant`' | '`descendant-or-self`' | '`next-sibling`' | '`following-sibling`' | '`following`'

⟨*node-test*⟩ ::= ⟨*label*⟩ | '`node()`'

⟨*qualifier*⟩ ::= ⟨*path*⟩ | ⟨*path*⟩ '∧'⟨*path*⟩ | ⟨*path*⟩ '∨'⟨*path*⟩ | '¬'⟨*path*⟩ | '`lab()`' '`=`' 'λ'| ⟨*path*⟩ '`=`' ⟨*path*⟩

tional XPath [165, 44]) and extensions (e.g., CXPath [265]) have been investigated, mostly with regard to expressiveness and the complexity of query evaluation. Satisfiability of positive XPath expressions is known to be in NP and, even for expressions without boolean operators, NP-hard [194]. Containment of XPath queries (with or without additional constraints, e.g., by means of a document schema) has been investigated as well [369, 136, 275, 329]). For a recent summary of fundamental results on XPath complexity, containment, etc. see Benedikt and Koch [46]. Several methods which provide efficient implementations of XPath that rely on standard relational database systems have been published [174, 177, 292].

As part of its activity on the specification of XQuery, the W3C has recently developed a revision of XPath, XPath 2.0 [51]. An introduction to XPath 2.0 can be found in Kay [214]. The most striking additions in XPath 2.0 are a facility for defining variables (using **`for`** expressions), sequences instead of sets as answers, the move from the value typed XPath 1.0 to extensive support for XML schema types in a strongly typed language, a considerably expanded library of functions and operators [258], and a complete formal semantics [138].

4.2.1.2 *XQuery*

XQuery has achieved the status of the predominant XML query language, at least as far as database products and research are concerned (in total, XSLT [105] is probably still more widely supported and used). XQuery is essentially an extension of XPath (though some of the axes of XPath are only optional in XQuery), but most of XPath becomes syntactic sugar in XQuery. This is particularly true for XPath qualifiers, which in XPath can be reduced to **`where`** or **`if`** clauses. Indeed, the XQuery standard is accompanied by a normalization of XQuery to a core dialect of the language [138].

XQUERY PRINCIPLES. At its core, XQuery is an extension of XPath 2.0, adding features needed to capture all the use cases of Chamberlin et al. [93] in order to turn it into a "full query language" and not just a language for (mostly tree-shaped) node selection. The most notable of these features are the following:

1. *Sequences.* While in XPath 1.0 the results of path expressions are node sets, XQuery and XPath 2.0 use sequences. Sequences can be constructed or result from the evaluation of an XQuery expression. In contrast to XPath 1.0, sequences cannot only be composed of nodes but also of atomic values. For example, `(1, 2, 3)` is a proper XQuery sequence.

2. *Strong typing.* Like XPath 2.0, XQuery is a strongly typed language. In particular, most of the (simple and complex) data types of XML Schema are supported. The details of the type system are described in Draper et al. [138]. Furthermore, many XQuery implementations provide static type checking.

3. *Construction, Grouping, and Ordering.* While XPath is limited to selecting parts of the input data, XQuery provides ample support for constructing new data. Constructors for all node types as well as the simple data types from XML Schema are provided. New elements can be created either by so-called direct element constructors (that look just like XML elements) or by what is referred to as computed element constructors. The latter for example allows the name of a newly constructed element to be the result of a part of the query.

4. *Variables.* Like XPath 2.0, XQuery has variables which are defined in so-called FLWOR expressions. A FLWOR expression usually consists of one or more **`for`** clauses, an optional **`where`** clause, an optional **`order by`** clause, and a **`return`** clause. The **`for`** clause iterates over the items in the sequence returned by the path expression in its **`in`** part; **`for`** `\$book` **`in`** `//book` for example iterates over all books selected by the path expression `//book`. The **`where`** clause specifies conditions on the selected data items, the **`order by`** clause allows the items to be processed in a certain order, and the **`return`** clause specifies the result of the entire FLWOR expression (often using constructors as shown above). Additionally, FLWOR expressions may contain, after the **`for`** clauses, **`let`** clauses that also bind variables but without iterating over the individual data items in the sequence bound to the variable.

5. *User-defined functions.* XQuery allows the user to define new functions specified in XQuery . Functions may use recursion.

6. *Universal and existential quantification.* Both XPath 2.0 and XQuery 1.0 provide **`some`** and **`all`** for expressing conditions using existential or universal quantification.

7. *Schema validation.* XQuery implementations may (optionally) provide support for schema validation, both of input and of constructed data, using the **validate** expression.

8. *Full host language.* XQuery completes XPath with capabilities for setting up the context of path expressions (e.g., for declaring namespace prefixes and default namespace), importing function libraries and modules, and providing flexible means for serialization that are in fact shared with XSLT 2.0 [216].

9. *Unordered sequences.* To assist query optimization, XQuery provides the **unordered** keyword, indicating that the order of elements in sequences that are constructed or returned as result of XQuery expressions is not relevant. For example, `unordered\{for \$book in //book return \$book/name\}` indicates that the nodes selected by `//book` may be processed in any order in the **for** clause, and that the order of the resulting name nodes can also be arbitrary. Note that inside unordered query parts, the result of any expressions querying the order of elements in sequences, such as **fn:position** or **fn:last**, is non-deterministic.

There is at least one respect in which XQuery is more restrictive than XPath, namely that not all of XPath's axes are mandatory. An XQuery implementation may not support **ancestor**, **ancestor-or-self**, **following**, **following-sibling**, **preceding**, and **preceding-sibling**. This does not restrict XQuery's expressiveness, as expressions using reverse axes (such as **ancestor**) can be rewritten [291], and the "horizontal axes" such as **following** or **following-sibling**, can be replaced by FLWOR expressions using the **and** operators that compare two nodes with respect to their position in a sequence.

For a comprehensive yet easy to follow introduction to XQuery see, e.g., Katz et al. [211].

COMPOSITION-FREE XQUERY IN 1000 WORDS. In the following, we focus on a fragment of XQuery, called non-compositional XQuery [224, 45], that has a well-defined, fairly easy to understand semantics and illustrates all issues salient for this thesis. However, many of the restrictions to the syntax can be dropped without affecting expressiveness and complexity, e.g., we could integrate full navigational XPath as discussed in Section 4.2.1.1. The only real restriction of composition-free XQuery in comparison to full XQuery is that it *disallows any querying of constructed nodes*, i.e., the domain of all relations is limited to the input nodes. This limitation clearly does not hold for full XQuery (even if we do not consider user-defined functions). Its effect on expressiveness and complexity is discussed in detail in Koch [224].

(Composition-free) XQuery is built around controlled iterations over nodes of the input tree, expressed using **for** clauses. Controlled iteration is important for XQuery as it is founded on

sequences of nodes rather than sets of nodes (as XPath 1.0). In this respect it is more similar to languages such as DAPLEX [334] or OQL [88] than to XPath 1.0. **For** loops use XPath expressions for navigation and XML-look-a-likes for element construction, all of which can be, essentially, freely nested. The following query creates a list of articles containing one `author` element for each author in the input XML tree (bound here and in the following to the canonical input variable `$inp`).

```
<paperlist>
  for $a in $inp/descendant::author return
    <author> for $p in $inp/descendant::article return
      if some $x in $p/descendant::author satisfies
          deep-equal($x, $a)
      then $p
    </author>
</paperlist>
```

For each `author` element, the nested **for** loop creates a list of all its articles. The latter expression can be more elegantly expressed in full XQuery using XPath qualifiers or **where** clauses, but here it is shown in the "normalized" syntax of composition-free XQuery following Koch [224].

We use **deep-equal**, XQuery's structural equality that tests whether the sub-trees at `$x` and `$a` are isomorphic, as `authors` can be represented using **last** and **first** name elements in our context and both have to be equal for it to be the same author.

A complete definition of the syntax of composition-free XQuery is given in Table 1. In addition to the specification, the usual semantic restrictions apply, e.g., the label of the start and end tags must be the same, variables must be defined (using **for**) before use, etc. As mentioned, there is one exception to the rule for variables: the canonical input variable `$inp` is always bound to the input XML tree.

The semantics of a composition-free XQuery expression is defined in Benedikt and Koch [45].

XQUERY IN INDUSTRY AND RESEARCH. From the very start, the development of XQuery has been followed by industry and research with equal amounts of interest.Even before the development was finished, initial practical introductions to XQuery had been published, e.g., Katz et al. [211], Brundage [73]. Industry interest is also visible in the simultaneous development of standardized XQuery APIs, e.g., for Java [140], and numerous implementations, both open source (e.g., Galax [149]) and commercial (BEA [151], IPSI-XQ [146]). First results concerning the implementation of XQuery on top of standard relational databases [131, 176] indicate that this approach leads to very

⟨*query*⟩ ::= ⟨*query*⟩ ⟨*query*⟩ | ⟨*element*⟩ | ⟨*variable*⟩
| ⟨*step*⟩ | ⟨*iteration*⟩ | ⟨*conditional*⟩

⟨*element*⟩ ::= '<' ⟨*label*⟩ '>' ⟨*query*⟩ '<' ⟨*/label*⟩ '>'
| '<' 'lab(' ⟨*variable*⟩ ')>' ⟨*query*⟩ '</' 'lab(' ⟨*variable*⟩ ')>'

⟨*step*⟩ ::= ⟨*variable*⟩ '/' ⟨*axis*⟩ '::' ⟨*node-test*⟩

⟨*iteration*⟩ ::= 'for' ⟨*variable*⟩ 'in' ⟨*step*⟩ 'return' ⟨*query*⟩

⟨*conditional*⟩ ::= 'if' ⟨*condition*⟩ 'then' ⟨*query*⟩

⟨*condition*⟩ ::= ⟨*variable*⟩ '=' ⟨*variable*⟩ | ⟨*variable*⟩ '=' '<' ⟨*label*⟩ '/>' | 'true'
| 'some' ⟨*variable*⟩ 'in' ⟨*step*⟩ 'satisfies' ⟨*condition*⟩
| ⟨*condition*⟩ 'and' ⟨*condition*⟩ | ⟨*condition*⟩ 'or' ⟨*condition*⟩ | 'not' ⟨*condition*⟩

⟨*axis*⟩ ::= 'child' | 'descendant' | 'descendant-or-self'
| 'next-sibling' | 'following-sibling' | 'following'

⟨*node-test*⟩ ::= ⟨*label*⟩ | 'node()'

⟨*variable*⟩ ::= '$'⟨*identifier*⟩

Table 1: Syntax of composition-free XQuery

efficient query evaluation if a suitable relational encoding of the XML data is used.

It is intuitively clear that XQuery is Turing complete since it provides recursion and conditional expressions. A formal proof of the Turing-completeness of XQuery is given in Kepser [217]. Efficient processing and (algebraic) optimization of XQuery, although acknowledged as crucial topics, have not yet been sufficiently investigated. Moreover, techniques for efficient XPath evaluation, as discussed above, can form the basis for XQuery optimization.

Beyond querying XML data, it has also been suggested to use XQuery for data mining [355], for web service implementation [293], for querying heterogeneous relational databases [365], for access control and policy descriptions [280], for synopsis generation [114], and as the foundation of a visual XML query language (XQBE) [29], of a XML query language with full text search capabilities [16, 15], and of an update [313, 74, 95] and reactive [67] language for XML.

Recently, the W3C has proposed a revision [96] to XQuery 1.0, termed XQuery 1.1, which among minor changes adds explicit grouping (using a new **`group-by`** clause) and iteration windows (or blockwise iteration, using a new **`window`** clause with several flavors).

4.2.1.3 *Reachability in Trees*

Like XPath, most XML query languages provide some form of path expression or axis for expressing different forms of reachability in a graph, most notably direct reachability or **`child`** axis vs. **`descendant`** axis. Path expressions have been introduced already for relational databases, e.g., in GEM [375], an extension of QUEL, and for object-oriented databases, e.g., in OQL [88]. OQL expresses paths with the *extended dot notation* introduced in GEM: `SELECT b.translator.name FROM Books b` selects the names, or components, of the translators of books. Note that there must be at most one translator per book for this expression to be legal.

Generalized (or regular) path expressions [154, 104] extend this idea with operators similar to regular expressions, e.g., the Kleene closure (and thus indirect reachability) operator on (sub-)paths under the additional requirement that each component is a node label. As a consequence, and in contrast to the extended dot notation, generalized path expressions do not require explicit naming of all nodes along a path. Lorel [10] is an early exemplar of a semi-structured query language, though being based on a (graph-shaped) data model. Lorel's syntax resembles that of SQL and OQL, extending OQL's extended dot notation to generalized path expressions. To illustrate this aspect of Lorel, assume that one is interested in books having "Peter Jones" either as author or translator. Assume also that the literal giving the name of the author is either wrapped inside a **`name`** child of the **`author`** element, or directly included in the **`author`** element. The selection of such books can be expressed in Lorel by the following **`where`** clause filter on all books **`B`**: **`where`** `B.(author|translator).name? = "Peter Jones"`.

Given that these efforts predate XPath significantly, it might seem surprising that XPath chose not to offer general path expressions, but only the weaker concept of axes. Remember, XPath allows navigation in *all directions* (vertical using **`descendant`** and **`ancestor`**, horizontal using **`following`** and **`preceding`** and their respective **`-siblings`** variants), while generalized path expressions only allow vertical navigation. However, XPath only provides closure axes (i.e., a path with any number of *arbitrarily* labeled nodes), but no closure of actual expressions. Thus it is not possible to express, for example, that two elements are connected by nodes carrying a certain label.

The difference with respect to the possible directions of navigation is clearly motivated by the particular emphasis placed on order in XML. The choice to provide only closure axes, however, is less obvious. Without closure of arbitrary path expressions, XPath cannot express regular path expressions such as `a.(b.c)*.d` (meaning "select `d`'s that are reached via one `a` and then arbitrary many repetitions of one `b` followed by one `c`").

Moreover, it turns out that such a feature (sometimes called *conditional* axes) is exactly what is missing from XPath to turn it into a first-order complete language on ordered trees [266, 265].

The inclusion of reverse axes in XPath has been shown not to increase the expressive power of XPath [291]. Consequently, they are used infrequently and, with the exception of the trivial **`parent`** axis, are considered *optional* features in XQuery that do not have to be provided by a conforming implementation.

Nevertheless, the efficient realization of closure axes has proved to be one of the more fruitful issues on the road to a scalable XML query language. In the following section, we classify approaches for implementing tree queries expressed in XPath or XQuery. All of these approaches have to deal in some form or the other with the presence of closure axes.

4.2.1.4 *Tree Queries on Tree Data*

While XQuery (and even full XPath 1.0) can also express more powerful graph queries, the most significant results have been achieved on the implementation of tree queries. For tree queries, the restriction of XML to tree data can be exploited to provide highly efficient (i.e., linear time and space) evaluation of XML queries even in the absence of sophisticated indices. To keep the discussion focused we ignore index-based evaluation of XML, and refer the interested reader to the survey of Weigel [361].

Most of the remaining approaches to the evaluation of XML tree queries fall into one of the following four classes:

1. *Structural joins*, proposed by Al-Khalifa et al. [14], are most reminiscent of query evaluation for relational queries and arguably inspired by earlier research on acyclic conjunctive queries on relational databases [168]. Tree queries are decomposed into a series of (structural) joins. Each structural join enforces one of the structural properties of the given query, e.g., a **`child`** or **`descendant`** relation between nodes or a certain label. Due to its similarity with relational query evaluation it has proved to be an ideal foundation for implementing XPath and XQuery on top of relational databases [174]. It turns out, however, that the use of standard joins is often not ideal, and that structure- or tree-aware joins [66] can significantly improve XPath and XQuery evaluation.

2. *Twig (or stack) joins* [75, 101], by contrast, employ a single operator to solve an entire tree query rather than decomposing it into structural joins. Twig joins operate by keeping one stack for each step in a query, which represents partial answers for the corresponding node set. These stacks are organized hierarchically with (where possible, implicit) parent pointers connecting partial answers for upper stack entries to those of lower ones. Different approaches from this class mostly vary in how the stacks are

populated. In contrast to the other approaches, twig joins are limited to vertical, i.e., **`child`** and **`descendant`**, axes and have not been adapted for the full range of XPath axes.

3. *PDA-based* approaches, based on pushdown automata, aim to evaluate XPath queries on a single input stream similar to a SAX event stream. This is in contrast to twig joins, which assume one stream of nodes from the input document for each stack (and thus XPath step). SPEX [290, 289, 288], for example, also maintains a record of partial answers for each XPath step, but minimizes memory usage and exploits the existential nature of most XPath steps by maintaining only generic conditions rather than actual pointers to elements from the XML stream (except for candidates of the actual results set, of course). Furthermore, in contrast to the twig join approaches, it supports all XPath axes. This comes at the cost of a slightly more complex algorithm.

4. *Interval-based approaches* finally combine the tree-awareness of twig joins and SPEX with the structural join approach: The query is decomposed into a series of structural relations, but each relation is organized in such a way that all elements related to one element of its parent step lie in a single continuous interval. This allows for both efficient storage and join of intermediate answers. The first interval-based approach are the Complete Answer Aggregates (CAA) [273, 272]. Furche [156] proposed the CIQCAG algebra, which improves on the complexity of CAA and in contrast to CAA covers arbitrary tree-shaped relations. It is also shown that interval-based approaches can be extended even to a large, efficiently detectable class of graphs (so called continuous-image graphs) that is not covered by any of the other linear-time approaches discussed above.

4.2.1.5 *Supporting Order*

In the previous sections, we have focused on the tree aspect of XML and its effect on query languages and their evaluation. Another characteristic that sets XML apart from many other data formats is its emphasis of ordered data. While very appropriate in a document setting such as XHTML or DocBook [354], this presents a challenge for query languages, which traditionally prefer a set-oriented perspective under the assumption that it enables more diverse evaluation strategies and thus better automatic optimization. XML query languages have addressed this challenge in various different ways.

Most of the early proposals ignore order in XML documents entirely or support it only superficially. While XPath 1.0 allows querying the order, its results are either in document or in reverse document order, depending on the axis of the final step. For query languages like XQuery that also support construction of new XML trees, however, this would be utterly insufficient. For

example, selecting authors together with their articles from the sample data in Section 4.1.1 and then constructing one XHTML section for each author containing a list of her articles requires control over the order in which section elements (e.g., `h1`) and list elements (`ul` or `ol`) are intertwined.

This need is recognized in XQuery, and in fact there are many ways in which XQuery is designed around proper support for ordered XML. Where results of path expressions are node sets in XPath 1.0, XQuery and XPath 2.0 use sequences.

The disadvantage of XQuery's choice to make order so prevalent in the language is that implementations have to maintain this order to conform to the specification.XQuery partially acknowledges this problem by providing the **`unordered`** keyword, which allows a sub-query to be evaluated indifferent to order, as if it had a set-based semantics. Grust et al. [175] discuss how order indifference in XQuery can be exploited. Similarly, some query languages, most notably Xcerpt [320], provide both ordered and unordered queries without preference for either.

This concludes our brief overview of XML query languages. For a comparison of a larger set of XML query languages the reader is referred to the survey of Bailey et al. [36]. In all three areas discussed, XML has triggered the development of novel approaches to query evaluation that have considerably extended our understanding of hierarchical queries in general.

In the next section, we turn to RDF and try to illustrate where related questions arise for RDF querying. We will see that knowledge about RDF query evaluation is significantly less advanced, due to the fact that RDF it is less established as a data format and topic of research.

4.2.2 *Graphs and Resources—RDF*

Compared to XML query languages, the field of RDF query languages is less mature and has not received as much attention from research, just as RDF itself. Whereas XML query languages focus on trees and order, RDF query languages have to deal with the simple, but also highly flexible RDF: RDF data (see Section 4.1.2) comes in the shape of arbitrary (usually node- and edge-labeled) graphs. Surprisingly, and in sharp contrast to the XML case, many RDF query languages only provide access to direct properties but not to reachability information (see Section 4.2.2.2). In contrast to relational or object-oriented data, all properties (i.e., outgoing edges) are *optional* and *multi-valued*. For instance, an author may or may not have a last name, or even many of them. How query languages deal with this inherent optionality is discussed in Section 4.2.2.3. Resources (i.e., nodes) are in general labeled with globally unique identifiers that allow us to talk about the same

resource in different datasets. However, RDF also allows blank nodes which play the role of purely local identifiers. Blank nodes are like existential data and pose particular challenges for RDF query evaluation (see Section 4.2.2.4).

Again, we start off the discussion of RDF query languages with a closer look at one of the more prominent exemplars: SPARQL. We focus on the essentials of SPARQL, for a more in-depth comparison of more than a dozen RDF query languages see Furche et al. [157].

4.2.2.1 *SPARQL in 1000 Words*

Fundamentally, SPARQL is a fairly simple query language in the spirit of basic subsets of SQL or OQL. However, the specifics of RDF have lead to a number of unusual features that arguably make SPARQL more suited to RDF querying than previous approaches such as RDQL [278]. This comes at the price of a more involved semantics complemented by a tendency to redefine or ignore established notions from relational and XML query languages rather than build upon them [308].

Nevertheless, SPARQL is expected to become the "lingua franca" of RDF querying, and it is well worth further investigation.

Let us look at an example. The following SPARQL query selects from the graph in Section 4.1.2 all articles in the journal named "Computer Journal" and returns a new graph where the `bib:isPartOf` relation of the original graph is inverted to `bib:hasPart`.[7]

```
CONSTRUCT { ?j bib:hasPart ?a }
WHERE { ?a rdf:type bib:Article AND ?a bib:isPartOf ?j
          AND ?j bib:name 'Computer Journal' }
```

This query illustrates SPARQLs fundamental query construct: a pattern (subject, predicate, object) for RDF triples. Any RDF triple is also a triple pattern, but a triple pattern differs from a triple in that it allows variables for each of the components. To make it easier to define the syntax of the language, we use the variant syntax for SPARQL discussed in Pérez et al. [303] We consider two forms of SPARQL queries, **`SELECT`** queries that return lists of variable bindings and **`CONSTRUCT`** queries that return new RDF graphs. Triple patterns contained in a **`CONSTRUCT`** clause (or "template") are instantiated with the variable bindings provided by the evaluation of the triple pattern in the **`WHERE`** clause. We omit named graphs and assume that all queries are on the single input graph.

7 Here and in the following we use namespace prefixes to abbreviate URIs. The usual URIs are assumed for `rdf`, `rdfs`, textttdc (dublin core), **`foaf`** (friend-of-a-friend), `vcard` vocabularies. `bib` is a prefix bound to an arbitrary URI.

⟨*query*⟩	::=	'CONSTRUCT' ⟨*template*⟩ 'WHERE' ⟨*pattern*⟩
	|	'SELECT' ⟨*variable*⟩+ 'WHERE' ⟨*pattern*⟩
⟨*template*⟩	::=	⟨*triple*⟩ | ⟨*template*⟩ 'AND' ⟨*template*⟩ | '{' template '}'
⟨*triple*⟩	::=	⟨*resource*⟩',' ⟨*predicate*⟩',' ⟨*resource*⟩
⟨*resource*⟩	::=	⟨*uri*⟩ | ⟨*variable*⟩ | ⟨*literal*⟩ | ⟨*blank*⟩
⟨*predicate*⟩	::=	⟨*uri*⟩ | ⟨*variable*⟩
⟨*variable*⟩	::=	'?' ⟨*identifier*⟩
⟨*pattern*⟩	::=	⟨*triple*⟩ | '{' ⟨*pattern*⟩ '}'
	|	⟨*pattern*⟩ 'FILTER' '(' ⟨*condition*⟩ ')' |
	|	⟨*pattern*⟩ 'AND' ⟨*pattern*⟩ | ⟨*pattern*⟩ 'UNION' ⟨*pattern*⟩
	|	⟨*pattern*⟩ 'MINUS' ⟨*pattern*⟩ | ⟨*pattern*⟩ 'OPT' ⟨*pattern*⟩
⟨*condition*⟩	::=	⟨*variable*⟩ '=' ⟨*variable*⟩ | ⟨*variable*⟩ '=' (⟨*literal*⟩ | ⟨*uri*⟩)
	|	'BOUND(' ⟨*variable*⟩ ')' | 'isBLANK(' ⟨*variable*⟩ ')'
	|	'isLITERAL(' ⟨*variable*⟩ ')' | 'isIRI(' ⟨*variable*⟩ ')'
	|	⟨*negation*⟩ | ⟨*conjunction*⟩ | ⟨*disjunction*⟩
⟨*negation*⟩	::=	'¬'⟨*condition*⟩
⟨*conjunction*⟩	::=	⟨*condition*⟩ '∧' ⟨*condition*⟩
⟨*disjunction*⟩	::=	⟨*condition*⟩ '∨' ⟨*condition*⟩

Table 2: Syntax of SPARQL

The full grammar of SPARQL queries considered here, which extends that of Pérez et al. [303] by **CONSTRUCT** queries, is given in Table 2.

We impose a number of additional syntactic restrictions: *range-restrictedness* requires that all variables in the *head* (**CONSTRUCT** or **SELECT** clause) must also occur in the *body* (**WHERE** clause) of the query; *error-freeness* requires that all variables occurring in the (right-hand) condition of a **FILTER** expression must also occur in the (left-hand) pattern. Finally, we allow only *valid RDF constructions* in **CONSTRUCT** clauses, i.e., no literal may occur as a subject, variables occurring in subject position are never bound to literals, and variables occurring in predicate position are only ever bound to URIs (but not to literals or blank nodes). The first condition can be enforced statically, the others by adding appropriate **isURI** or negated **isLITERAL** filters to the query body.

Formal semantics for SPARQL were given by Pérez et al. [303] and Polleres [307].

Recently, SPARQL has been subject to a number of studies and extensions. Its complexity and formal semantics have been studied by Pérez et al. [303], who showed that full SPARQL patterns are just as expressive as relational algebra and thus PSPACE-complete with respect to their query complexity. This is somewhat disappointing, especially since many graph queries that are highly desirable for RDF query languages (including simple reachability queries) cannot even be expressed in SPARQL [23]. Extensions of SPARQL with rules have received some attention, in part because they address some of these weaknesses and can be seen as the natural next step towards a semantic web query engine [307].

Finally, embeddings of SPARQL in XQuery, or vice versa, have been studied (see, e.g., Akhtar et al. [13]).

4.2.2.2 *Reachability*

In sharp contrast to the XML case, many RDF query languages do not provide a means to access reachability information, or in fact any form of navigation in the RDF graph beyond the traversal of a fixed number of edges. Angles and Gutiérrez [23] describe a set of graph queries that are desirable for an RDF query language, but cannot be expressed by SPARQL or RQL.

If we look beyond SPARQL and RQL, however, we find that RDF query languages actually support a wide variety of path expressions:

1. *Basic path expressions* are abbreviations for triple patterns as found in SPARQL and RQL. They allow only the specification of fixed length traversals, i.e., the traversed path in the *data* has the same length as the path expression. Basic path expressions are no more expressive than triple patterns, but are nevertheless encountered in several query languages as syntactic sugar. Examples of query languages which only provide basic path expressions are GEM [375], OQL [88], SPARQL [308], and RQL [210].

2. *Unrestricted closure path expressions* augment basic path expressions by the ability to traverse arbitrary-length paths. XPath path expressions (disregarding XPath predicates for the moment) fall into this category, with closure axes such as **`descendant`**. They are strictly more expressive than basic triple patterns, and can be realized in languages that provide (at least linear) recursive views in addition to triple patterns. SQL-99 is an example of a language that provides no closure path expressions but linear recursion and can emulate (unrestricted) closure path expressions. Unrestricted closure path expressions can be found in many XML query languages, e.g., in XML-QL [135], Quilt [97] and XPath. In RDF query languages they are much less common, iTQL [3] being a notable exception. The reason for this is that RDF, in contrast to XML, has no dominant hierarchical relation but many

relations of equal importance. This makes unrestricted closure often too unrestrictive for interesting queries.

3. To address this issue, several RDF query languages provide *generalized* or *regular path expressions*. Here, full regular expression syntax is provided on top of path expressions. For example, the expression `a*.((b|c).e)+` traverses all paths of arbitrary many `a` properties followed by at least one repetition of either `b` or `c` followed by `e`. Regular path expressions are provided, e.g., by Versa's **`traverse`** operator, Xcerpt's qualified **`descendant`**, or the extension of XPath with conditional axes [266]. Marx [266] also showed that regular path expressions are even more expressive than unrestricted closure path expressions, and that a path language like XPath becomes first-order complete with the addition of regular path expressions.

To summarize, path expressions provide a convenient way to specify structural constraints in RDF queries and are therefore supported by a large number of RDF query languages. However, surprisingly many RDF query languages ignore (unrestricted or regular) closure path expressions. This is surprising insofar as these path expressions make writing queries easier and can be implemented efficiently. In particular, unrestricted closure path expressions can be implemented nearly as efficiently as basic path expressions.

EVALUATION OF REACHABILITY QUERIES ON GRAPHS. For tree data, membership in closure relations can be tested in constant or almost constant time (e.g., using interval encodings [137] or other labeling schemes such as that of Weigel et al. [363]). For graph data this is far less obvious. Fortunately, recent years have seen a considerable amount of research on reachability or closure relations and their indexing in arbitrary graph data. Obviously, we could carry out the membership test in constant time if we store the full transitive closure matrix. However, for large graphs this is infeasible. Therefore, two classes of approaches have been developed that allow membership tests in sub-linear time and significantly less space.

The first class is based on the idea of a 2-hop cover [106]. Instead of storing the full transitive closure, we allow reachable nodes to be reached via at most one other node (i.e., in two "hops"). More precisely, each node n is labeled with two connection sets, $\mathtt{in}(n)$ and $\mathtt{out}(n)$. The former contains a set of nodes that can reach n, the latter a set of nodes that are reachable from n. Both sets are assigned in such a way that a node m is reachable from n if and only of $\mathtt{out}(n) \cup \mathtt{in}(m) \neq \emptyset$. Unfortunately, computing a smallest 2-hop cover is NP-hard, and even approximating it might be too hard in practice [324].

A different approach is to use interval encoding for labeling a tree core and treating the remaining non-tree edges separately [12, 101, 356, 350]. This allows for a membership test in sublinear or even constant time (though the latter would still incur a considerable indexing cost), e.g., in Dual Labeling [356], where a full transitive closure over the non-tree edges is computed. GRIPP [350] and SSPI [101] obtain a different trade-off by attaching additional interval labels to non-tree edges. This leads to linear index size and time at the cost of increased query time.

These considerations show that reachability, at least in its basic form, does not need to have a significant negative effect on the performance of RDF query evaluation. It clearly does not affect its theoretical complexity, given that the evaluation of SPARQL **SELECT** queries is already PSPACE-complete.

4.2.2.3 *Optionality*

So far, we have focused on purely conjunctive queries. Disjunction or equivalent union constructs allow the query author to collect data items with different characteristics in one query. For example, to find colleagues of a researcher from an RDF graph containing bibliography and conference information, one might choose to select co-authors, as well as co-editors and members of the same program committees. On RDF data, disjunctive queries are far more common than on relational data, since all RDF properties are by default optional. Many queries have a core of properties that have to be defined for the data items in question, but also include additional properties (often labeling properties or properties relating the data items to further information such as web sites) that should be reported if they are defined but may also be absent. The following SPARQL query for example returns pairs of articles and editors for articles that have editors, and just articles otherwise. If one considers the results of a query as a table with **nil** values, the editor column is **nil** when an article has no `bib:editor` property.

```
SELECT   ?article, ?editor
WHERE    { ?article a bib:Article AND
           OPTIONAL { ?article bib:editor ?editor } }
```

This kind of optional selection makes life simpler for both the query author and the query processor, as compared to a disjunctive or union query which has to duplicate the non-optional part:

```
SELECT   ?article, ?editor
WHERE    { ?article a bib:Article AND
           ?article bib:editor ?editor }
         UNION
         { ?article a bib:Article }
```

In fact, the latter query is not even equivalent as it returns an additional result tuple `(X,nil)` for articles `X` that do have an editor. This raises the question of the precise semantics of an optional selection operator. The answer to this question is not the same for different RDF (or XML) query languages. The main difference between the offered semantics in languages such as SPARQL, Xcerpt, or XQuery lies in the treatment of multiple optional query parts with dependencies. For example, in the expression `A` $\wedge$ **`optional`**`(B)` $\wedge$ **`optional`**`(C)` the same variable `V` may occur in both `B` and `C`. In this case, if we just go ahead and use the `B` part to determine bindings for `V`, these bindings may be incompatible with `C`, i.e., prevent the matching of `C`.

The different ways to handle this case of multiple interdependent optionals yields the following four semantics for optional selection constructs:

1. *Independent optionals* disregard interdependencies between optional clauses by imposing an order on the evaluation of optional clauses. SPARQL for example uses the order of optional clauses in the query. The following query selects articles together with their respective editor and, if that editor is also an author, also with the author name:

```
SELECT   ?article, ?person, ?name
WHERE    { ?article a bib:Article AND
         OPTIONAL { ?article bib:editor ?person }
         OPTIONAL { ?article bib:author ?person AND
                    ?person bib:name ?name } }
```

If we changed the order of the two optional parts, the semantics of the query would also change: select all articles together with their authors and author names, if any. The second **`optional`** becomes redundant, as it only checks whether the binding of `?person` is also an editor of the same article. Whether or not the check fails does not affect the outcome of the query.

It should be obvious that this semantics for interdependent optionals is equivalent to allowing only a single optional clause per conjunction that may in turn contain other optional clauses. The above query could also be written as follows:

```
SELECT   ?article, ?person, ?name
WHERE    { ?article a bib:Article .
         OPTIONAL { ?article bib:editor ?person
```

```
            OPTIONAL { ?article bib:author ?person AND
                       ?person bib:name ?name }
        } }
```

This observation, however, only applies if the optional clauses are interdependent. If they are not interdependent, multiple optional clauses in the same conjunction differ from the case where they are nested.

2. *Maximized optionals* consider any order of optionals. In the example they would return the union of the orders, i.e., either first binding editors and then checking whether they are also authors, or first binding authors and author names and then checking whether they are also editors. This semantics, which was first used in Xcerpt [320], is more involved than the above and assigns different meanings to consecutive vs. nested optionals. The advantage of this is that it is equivalent to a rewriting of **optional** to disjunctions with negated clauses: `A` $\wedge$ **optional**`(B)` $\wedge$ **optional**`(C)` is equivalent to `(A` $\wedge$ **not**`(B)` $\wedge$ **not**`(C))` $\vee$ `(A` $\wedge$ **not**`(B)` $\wedge$ `C)` $\vee$ `(A` $\wedge$ `B` $\wedge$ **not**`(C)` $\vee$ `(A` $\wedge$ `B` $\wedge$ `C)`. This ensures that the maximal number of optionals for a certain (partial) variable assignment is used.

3. *All-or-nothing optionals* are a rare case of optional semantics where either all optional clauses are consistent with a certain variable assignment or all optional variables are left unbound. This semantics can be obtained in SPARQL and Xcerpt by using a single optional clause instead of multiple independent ones.

4.2.2.4 *Existential Information*

Recall that RDF data may contain specifically marked resources, called *blank nodes*, whose identity is limited to the RDF graph and that express only existential information. If we see an RDF graph as a logical conjunction of triples, they act as existential quantifiers over the resulting formula. Blank nodes pose a number of challenges for RDF query evaluation.

First, when blank nodes are selected by a query, should a query language return them like any other resource? Blank nodes are essentially local identifiers and thus may not carry much information outside the scope of their original graph. Furthermore, blank nodes express existential information, which may already be implied by the other data and therefore redundant. Consider for example the data of Figure 3 on page 41, and assume that this data additionally contained a statement that the article `smith2005` is part of issue 11 of some journal. That information is obviously already implied by the existing data (that `smith2005` is part of issue 11 of the journal "Computer Journal") and can thus safely be omitted. An RDF graph without such redundant information is called *lean* [189]. Ideally, we might expect an RDF query lan-

guage to return only those blank nodes that are not redundant (perhaps together with enough additional information to retrieve them again, e.g., a concise bounded description [340]). However, simply computing the lean graph for any given RDF graph is co-NP-complete [180]. Therefore, many RDF query languages choose to ignore this issue and return blank nodes just like any other resource.

Second, when constructing new RDF graphs (e.g., through SPARQL's **`CONSTRUCT`** clause), we need to be able to construct also new blank nodes to obtain an adequate RDF query language. Say we want to construct a new blank node with edges to all articles selected by this query. Then a single blank node for all articles is needed. However, we might also want to construct, for each article, a new blank node with edges to each of its authors. Now we need one "fresh" blank node for each article (otherwise all articles share all authors) but only one for each group of authors of the same article. SPARQL only allows the construction of blank nodes that are in the scope of *all* query variables and thus can express neither of the above queries. In RDFLog [78, 77] the effect of blank nodes on RDF querying is studied in detail. It is shown, in particular, that the combination of blank node support (even as in SPARQL) with (recursive) rules (as, e.g., in Schenk and Staab [323]) immediately leads to an undecidable, Turing-complete language that can be reduced, using Skolemization and a so-called un-Skolemization, to standard logic programming. It is also shown that arbitrary scoping of blank nodes is not more expensive as SPARQL-style $\forall\exists$ scopes and that, at least in presence of rules, the two are actually equivalent.

This concludes our brief summary of core issues on RDF querying and RDF query languages. The above discussion shows that RDF querying is a less mature field of research than XML querying, but that there are a number of open questions that need to be addressed for efficient and convenient access to RDF, and thus arguably the entire semantic web vision, to move forward.

4.2.3 *Outlook—Versatile Languages*

In the previous sections we have discussed XML and RDF querying separately. However, in recent years, GRDDL [115] and similar initiatives have invested effort into defining a means to conveniently access both XML and RDF data within the same application or even the same query language. This is reflected in an increasing number of approaches to integrate XML and RDF querying. Existing approaches for integrating XML and RDF access roughly fall into one of two categories: transformational and multi-language approaches. In the former, a pure XML or a pure RDF query language is used, and some encoding in the

corresponding format is used to access data in the respective other format. In the latter, a query language for one of the data formats is combined with, most often embedded into, one for the other format. Examples include XSPARQL and GRDDL. Transformational approaches have the advantage that users only need to learn a single language. However, this is offset by the need to understand the encoding of RDF in XML or vice versa and very limited support for specifics of the encoded data format that are not present in the native format.

A unique position among these approaches is held by Xcerpt and its extension Xcerpt$^{\text{RDF}}$: through a slight extensions to the pattern- and rule-based XML query language Xcerpt, convenient querying of RDF is enabled that, in contrast to languages like SPARQL, addresses also the graph nature of RDF. The vast majority of language features is shared by both the XML and the RDF version of Xcerpt, thus alleviating the problems of the above-mentioned integration approaches.

We proceed by briefly outlining the basic ideas of Xcerpt to give an impression of how a versatile semi-structured query language compares with XQuery or SPARQL as discussed in the previous sections. A more detailed description of Xcerpt is given by Schaffert and Bry [320], its RDF extension Xcerpt$^{\text{RDF}}$ is discussed by Bry et al. [79].

Xcerpt

Xcerpt [320] is a query language designed after principles given by Bry et al. [80] for querying both data on the "standard web" (e.g., XML and HTML data) and data on the semantic web (e.g., RDF and Topic Maps data, etc.). Xcerpt is *data versatile*, i.e., the same Xcerpt query can access and generate as answers data in different web formats. Xcerpt is *strongly answer-closed*, i.e., it not only allows for the construction of answers in the same data formats as the queries like, e.g., XQuery [94], but also for further processing of the data generated by this same query program. Xcerpt queries are pattern-based and allow for an incomplete specification of the data to be retrieved, by (1) not explicitly specifying all children of an element, (2) specifying descendant elements at indefinite depths (restrictions in the form of regular path expressions being possible), and (3) specifying optional query parts. The evaluation of incomplete queries is based on a novel unification algorithm called *simulation unification*. The processing of XML documents is graph-oriented, i.e., Xcerpt is aware of the reference mechanisms of XML (e.g., of ID/IDREF attributes and links).

An Xcerpt program consists of a finite set of Xcerpt *rules*. The rules of a program are used to derive new, or transform existing,

data from existing data (i.e., the data being queried). *Construct rules* are used to produce intermediate results while *goal rules* form the output of a program.

While Xcerpt works directly on XML or RDF data, it has its own data format for modeling XML documents or RDF graphs: Xcerpt *data terms*. For example, the XML snippet `<book><title>White Mughals</title></book>` corresponds to the data term `book [ title [ "White Mughals"] ]`. The data term syntax makes it easy to reference XML document structures in queries and extends XML slightly, most notably by allowing unordered data and making references first class citizens (thus moving from a tree to a proper graph data model).

The construct rule in the following query for example defines data about books and their authors, which is then queried by the goal rule. Intuitively, the rules can be read as deductive rules (like in, say, Datalog): if the body (after **FROM**) holds, then the head (following **CONSTRUCT** or **GOAL**) also holds. A rule with an empty body is interpreted as a fact, i.e., the head always holds.

```
GOAL
  authors [ var X ]
FROM
  book [[ author [ var X ] ]]
END

CONSTRUCT book [ title [ "White Mughals" ],
          author [ "William Dalrymple" ] ] END
```

Xcerpt *query terms* are used for querying data terms, and intuitively describe patterns of data terms. Query terms are used with a pattern matching technique, called *simulation unification*, to match data terms [318]. They can be configured to take incompleteness or ordering of the underlying data terms into account during matching (indicated by different types of brackets). Query terms may also contain (logic) variables. When they do, successful matching with data terms results in variable bindings, which are then used by Xcerpt rules to derive new data terms. Matching the query term `book [ title [ var X ] ]` against the XML snippet above for example results in the variable binding **`{X/"White Mughals"}`**. In addition to the query term types discussed by Schaffert and Bry [320], we also consider non-injective ordered and unordered query terms indicated by three braces or brackets, respectively.

Construct terms are essentially data terms with variables. The variable binding produced via query terms in the body of a rule can be applied to the construct term in the head of the rule in order to derive new data terms. For the above example we obtain the data term `authors [ "William Dalrymple"]` as a result.

4.3 QUERIES AS KEYWORDS: KEYWORD-BASED QUERY LANGUAGES

Two important characteristics generally desirable for social semantic web applications, accessibility and flexibility, make web querying as described in the last section a less than ideal choice as a means for data retrieval:

ACCESSIBILITY To make structured data accessible for a broad user base, a query language is needed that is easy to use, a criterion which web query languages do not fulfill. Like most programming languages [298], they are designed for usage by experienced programmers, and usability is of far lesser concern than for example scalability and expressiveness. Accordingly, the small number of usability studies carried out for web query languages focus on comparing users' performance in different query formalisms but do not assess overall usability [328, 169]. A notable exception is a study by O'Keefe and Trotman [287], which found that even researchers in the area of XML have considerable problems creating valid XPath queries, indicating that the usability of web query languages is low.

Usability studies performed on database query languages, the paradigm after which web query languages are modeled, indicate that participants generally find it hard to use these languages. Ogden and Brooks [286] give reasons for this, which are likely to apply also to web query languages: The languages require the user to have extensive knowledge of the data schema and they are highly constrained. This leads to frequent mistakes, which in turn increase the learning time and discourage the users. In addition, the languages are seen as "overly verbose and complicated" ([286], page 161).

FLEXIBILITY When many people contribute to data creation, or when data from different sources is aggregated in one data repository, strict enforcement of homogeneity of data in terms of a certain data schema is not feasible without discouraging potential users from contributing. Additionally, data of various different types, for example XML and RDF, may be present in the system, and it should be possible to query over the different data types using only one query formalism.

The heterogeneity of data necessitates a flexible query interface which does not require knowledge of the schema and offers integrated access to data in various formats. While versatile web query languages (see Section 4.2.3) are designed to address this issue, most current web query languages fail to do so.

In the domain of textual queries, keyword querying and natural language querying are the paradigms most commonly used to

facilitate easy-to-use querying. While some approaches exist for using natural language querying of XML data [248, 249], we will not consider this paradigm in the following for two reasons. First, and notwithstanding the fact that there is no unequivocal experimental evidence for the superiority of either of the two paradigms in terms of usability [360, 212, 346], keyword search has become the de facto standard for information access on the web, and practically all web users have practice creating keyword queries. Second, natural language query processors can only understand a limited subset of natural language. The limits of this understanding are hard to convey to human users and can lead to a "spiral of failure" ([360], page 114).

Keyword search is used in a wide variety of applications and domains, in web search engines such as Google,[8] Yahoo!,[9] and Bing[10] as well as in more specialized contexts and domains. Entering the query `XML Web` into Google for example yields a lists of web pages in which these terms occur; on the shopping site Amazon[11] and the auction site Ebay[12] it results in a list of products available on the site, and on the social networking site Facebook,[13] the search results for the same query contain relevant user groups, events, user profile add-ons, and users who are interested in the web and XML.

In web search, most queries are keyword-only queries. Google for instance supports a limited set of label-keyword-like constructs like **`allintitle`**`:XML`, which retrieves web sites that have the word "XML" in their `title` element. However, these are rarely used in practice [202, 43].

In general, the structure of web documents can only be queried to a very limited degree using web search. For example, it is not possible to specify an arbitrary HTML tag as a surrounding context for a keyword. In the case of web search engines, which process vast amounts of data, this can be attributed to the fact that indexing and retrieving structural information would increase the data and processing load, thereby decreasing search efficiency.

Amazon, on the other hand, which operates on much less, and much more homogenous, data, offers advanced search functionality for various categories of products like books and magazines (see Figure 5). The user can provide values for a number of attributes, for instance author and language. While the advanced search is realized in terms of a form, it is essentially equivalent to a limited label-keyword query language.

8 http://www.google.com/
9 http://www.yahoo.com/
10 http://www.bing.com/
11 http://www.amazon.com/
12 http://www.ebay.com/
13 http://www.facebook.com/

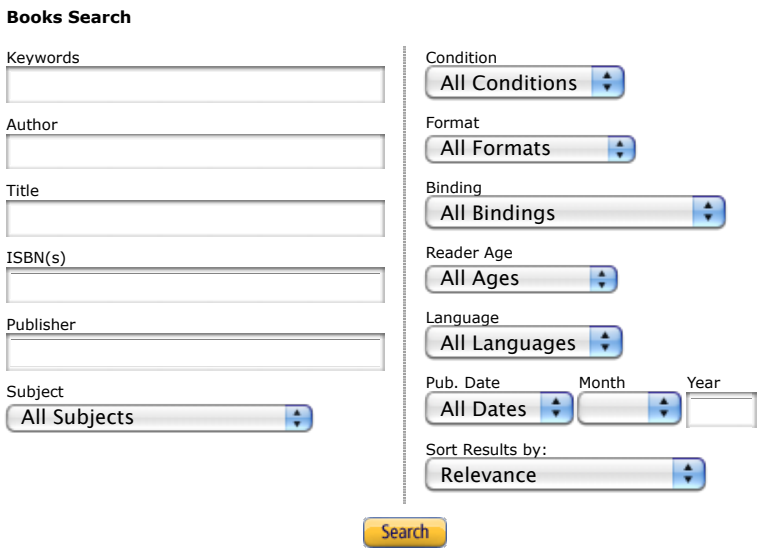


Figure 5: The Amazon advanced search interface, which can be accessed at http://www.amazon.com/gp/browse.html?node=241582011

Keyword-based querying is an established technique and has shown great effectiveness in querying the web in a variety of domains. It is increasingly applied to facilitate non-expert querying of the ever growing amount of (semi-)structured data on the web. The majority of research in the area of keyword querying for semi-structured data is concerned with XML data. The most likely reasons for this are that XML is older and more established than RDF, and that keyword querying for RDF data is harder to realize because of its graph structure, labeled edges, and blank nodes.

In the remainder of this section we will give an introduction to the topic of keyword querying on semi-structured data. Section 4.3.2 identifies the most important research issues in the area of XML keyword querying and provides an overview of the different approaches. The less numerous schemes for keyword querying of RDF are presented individually in Section 4.3.3. We assume familiarity with XML, RDF, and their respective data models (see Section 4.1). The related topic of keyword querying in relational databases (see, e.g., Bhalotia et al. [57]) will not be treated here.

4.3.1 *Classifying Keyword Query Languages*

Web search and web querying can be seen as two extremes with respect to the degree to which explicit querying of the structure of the data is supported. The former typically does not allow

querying of the structure, while in the latter the structure of the data is usually fully specified. Based on this observation, we can distinguish three types of keyword-based query languages for structured data according to the extent to which structure can be used as a selection criterion.

1. In *keyword-only query languages*, queries consist of a number of terms which are matched to the textual content of nodes in an XML or RDF document, and in some cases to node or (in the case of RDF) edge labels. Queries make no reference to the structure of the data. This category includes most keyword query languages, like XKeyword [37, 198], XRank [179], Spark [378], and XKSearch [372].

2. In *label-keyword query languages* such as XSearch [109] and XBridge [245], a query term is a label-keyword pair of the form *l:k*. The term matches data where a node with the label *l* contains, either directly or through a descendant node, text matching the associated keyword *k*. It is thus possible to indicate the context in which the keyword should occur.
Depending on the particular query language, either the label or the keyword may be optional, meaning that query terms can have the form *:k*, *l:*, or *l:k*. Applied to the example data of Figure 2, the query `title:Web` matches node 3. The query `:Web`, on the other hand, does not impose any constraints on the node label and matches nodes 3 and 23.

3. *Keyword-enhanced query languages* [250, 152, 327] extend traditional web query languages with simple keyword querying. They allow for the specification of structure to the extent to which it is known, but also include constructs for the use of keyword querying where it is not. Keyword-enhanced query languages constitute an extension of traditional query languages and therefore provide their full expressive power.
Given that (some) web query languages also offer ways to specify queries when the user lacks knowledge about the schema, for example through regular path expressions in XPath, one might wonder what distinguishes traditional query languages and keyword-enhanced query languages. As pointed out by Florescu et al. [152] and Schmidt et al. [327], regular path expressions are useful when the schema is not completely known to the user, but not when the user has no knowledge of the schema at all. The reason for this is that query evaluation in web query languages is not optimized for evaluating vague queries. Furthermore, while the schema of the data may not have to be known, knowledge of the query language itself is still necessary, making web query languages unsuitable for casual users.

A second, orthogonal characteristic of keyword query languages is the way they are implemented.

1. Most keyword query languages are implemented as standalone systems that handle all steps of the query evaluation.
2. Another group of keyword query languages translate the keyword queries into another query language and thus outsource the query evaluation. This category includes many RDF keyword query languages [349, 378, 359], but to the best of our knowledge only one XML language, XBridge [245], which translates keyword queries into XQuery. The approach of Ladwig and Tran [235] takes an exceptional position in that it tightly integrates query translation and query evaluation, and generates queries and candidate answers at the same time.
3. Keyword-enhanced query languages finally build on existing systems by combining conventional query languages like XPath or XML-QL with keyword-querying techniques.

4.3.2 *Querying XML*

This section gives an overview over the most important issues in the area of keyword queries for XML data.

4.3.2.1 *Determining Semantic Entities*

In keyword querying on the web, some structural information may be taken into account when ranking the results, for example by assigning different scores depending on whether a keyword occurs in the title or is printed in big or bold text [71]. The type of the return value, however, is fixed, and the structure of a document does not play a role when determining it. Apart from efficiency, there are two reasons for this. First, web or wiki pages typically have a comparably small size, and it is reasonable to return results at the granularity of whole pages. Secondly, in the case of domain-specific querying on a limited, homogeneous dataset like that of a shopping website, querying only serves one task, namely finding matching products. There are only few types of objects, e.g., books and DVDs, and the return types can easily be predefined. For example, keywords matching a book might yield a return entity of type `book` which by default displays the title, author, and price, while the return entity for DVDs might show the title, price and region code.

When applying the concept of keyword-search to RDF or XML documents, on the other hand, we may be dealing with a single big document that represents a bibliography or an address book and contains thousands of entries. In this case, returning the whole document would not be meaningful. To determine a useful return value, the data must first be partitioned into semantic entities.

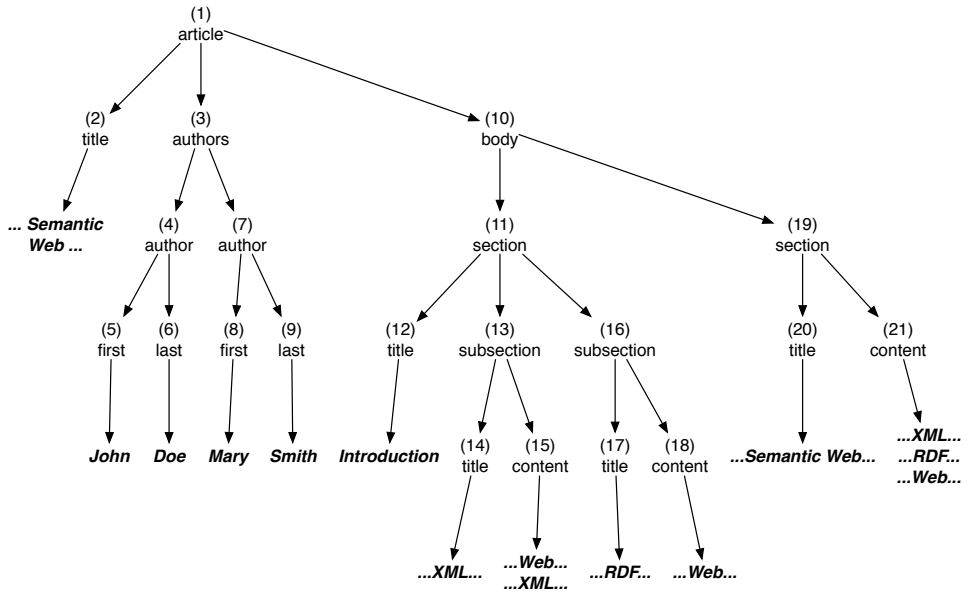


Figure 6: Document-centric XML representing an excerpt of an article

XML documents may be data-centric like in the case of a bibliography, see Figure 2 on Page 39, or also document-centric, representing a text and its structure or formatting, as shown in Figure 6. A truly versatile keyword-based query language for XML should yield useful results for both kinds of documents, and any type that might exist in between.

As an illustration of the return value problem, again consider the XML document of Figure 2, which represents a bibliography. A query $K = \{w_1, ..., w_k\}$ matched on an XML dataset T yields the result lists $L = \{L_1, ..., L_k\}$, where each list $L_i = \{v_1, v_2, ...\}$ consists of all the nodes v that contain w_i. For the data of Figure 2, the first term in the query $K = \{Smith, Web\}$ (conjunction is assumed here and in the following) has one match: the content of node 11, which is the last name of one of the authors of an article ($L_1 = \{11\}$). The second term matches the titles of both articles, and thus $L_2 = \{3, 23\}$.

Returning only the matched nodes would not provide much useful information for the user. Neither would returning the whole document, which might contain many more entries. Given the nature of the data and the query, we can assume that the user is interested in obtaining information about articles that contain the two search terms. This means that the meaningful semantic entities which should be returned (as a whole or in part) in response to the query, are subtrees governed by `article` nodes. In general, these semantic entities are determined either at the schema level or, more frequently, by connecting keyword matches in the data.

One approach to the grouping of matches is to find the most specific element that is an ancestor to at least one match instance of each keyword, and to consider it the root of their common semantic entity. The underlying idea is that the ancestor-descendant relationship indicates a strong semantic connection, particularly when the distance between ancestor and descendant is small. Correspondingly, a node which is the closest common ancestor of instances of all keywords is likely to encode the most specific concept that the keyword matches have in common. This concept is called the *Lowest Common Ancestor (LCA)* [185], and was used in an early approach to XML keyword querying by Schmidt et al. [327].

Depending on the application and the specific algorithm, an answer set $S = \{S_1, S_2, ...\}$ may contain either one ($\|S_i\| = \|K\|$) or more than one ($\|S_i\| \geqslant \|K\|$) matched node for each keyword. In the above example, and assuming the latter case, the three answer sets are $S_1 = \{11, 3\}$, $S_2 = \{11, 23\}$, and $S_3 = \{11, 3, 23\}$ with LCA nodes $LCA(S_1) = 2$, $LCA(S_2) = 1$, and $LCA(S_3) = 1$, respectively. The latter two answer sets contain nodes that belong to two different publications and thus not to a meaningful semantic entity given the context. They are considered to be *false positives*.

Several refinements of LCA have been proposed to remedy the problem of false positives, that is, the grouping of matches which do not belong to a common relevant or meaningful semantic entity. The alternative grouping semantics presented in the following reduce the set of LCA answers by eliminating matches that are considered false positives. However, this often introduces *false negatives*, meaning that not all relevant answers are returned. An extensive review of some of the connection heuristics presented here is given by Vagena et al. [351].

INTERCONNECTION SEMANTICS The assumption underlying Interconnection Semantics [109] is that two different nodes with the same label correspond to different entities of the same type, while nodes with differing labels represent concepts belonging to different types.

Accordingly, two nodes v_1 and v_2 are *interconnected* if the path from them to their LCA does not contain distinct nodes with the same labels except for v_1 and v_2 themselves. An answer set contains only one match for each keyword in the query and is interconnected if either it contains a node, the *star center*, that is interconnected with all other nodes in the set (*star relatedness*) or if all nodes are pairwise connected (*all-pairs relatedness*).

Again consider the query on the example data which yields the result lists $L_1 = \{11\}$ and $L_2 = \{3, 23\}$ and answer sets $S_1 = \{11, 23\}$ and $S_2 = \{11, 3\}$. The shortest path between nodes 11 and 3 contains every node label only once, which means that the two

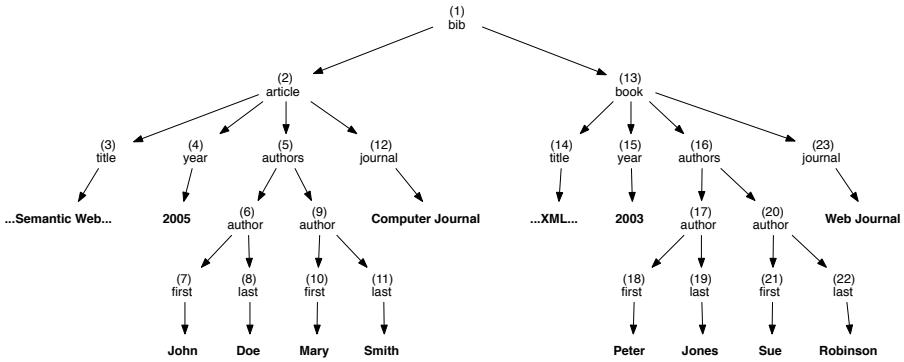


Figure 7: False positives in interconnection semantics

are interconnected. Nodes 11 and 23, on the other hand, are not interconnected since nodes 2 and 13, which both lie on the path between the respective nodes and the LCA node `bib`, have the same label, namely `article`. The only answer to the query is thus $S_1 = \{11, 3\}$. The interconnection relation in this case avoids grouping matches together that belong to different articles as simple LCA-based grouping would.

Since each set S_i in the previous example contains only two elements, the interconnected nodes are both star-related and all-pairs related. However, this is not always the case, since star-relatedness is a relaxation of all-pair relatedness in the sense that for a set of nodes to be all-pairs related, every node has to be a star center. For example, the query $K = \{Smith, Doe, 2005\}$ yields the answer set $S_1 = \{11, 8, 4\}$. Nodes 11 and 8 are not interconnected, since the path between them passes two nodes with label `author`. Node 4, however, is interconnected with both nodes 11 and 8. Consequently, $S_1 = \{11, 8, 4\}$ is not a query answer if all-pairs interconnection is used, but it is according to star related interconnection.

This example illustrates that all-pairs interconnection can lead to false negatives, since S_1 is a valid answer to the query. Both types of interconnection semantics are also sensitive to false positives when node labels differ but refer to similar concepts. Applying the query $K = \{Smith, 2003\}$ to the data of Figure 7 would wrongly return the root node as a result, because `article` and `book` are different labels but signify conceptually related entities.

EXCLUSIVE LCA XRank [179] introduces the concept of the *Exclusive LCA* (ELCA) [379], which is targeted at keyword querying on document-centric XML. The idea behind the ELCA is that more specific LCAs should be preferred, but only if this does not cause the loss of additional matches.

The ELCA is computed by first finding R_0, the set of nodes that contain at least one instance of each keyword in the query via an ancestor-descendant relationship. A query result node then is a node in R_0 which, for each keyword, contains at least one match instance that is not contained in any of its descendant nodes that are also in R_0. Formulated in terms of the LCA, the procedure yields those LCA nodes which either are not ancestors to any further LCA nodes or, if they are, are also LCA nodes when ignoring the keyword matches in the contained LCA subtree.

As an example, consider the query $K = \{XML, Web\}$ evaluated on the data in Figure 6 on page 68. The keyword match lists are $L_1 = \{14, 15, 21\}$, and $L_2 = \{2, 15, 18, 21\}$. Based on this, some exemplary answer sets are $S_1 = \{13\}$, $S_2 = \{14, 18\}$, and $S_3 = \{15, 18\}$. S_1 consists of a single node containing all keywords, meaning that the LCA of S_1 is identical with its element, node 13. Since this node is the LCA node and does not have any children, it also is query result node. Similarly, node 11, the LCA of S_2, is also a result node. Node 11 also is the LCA of S_3, and it is an ancestor of node 13, itself an LCA node. It contains an occurrence of $k_2 = Web$ which is not part of an LCA, namely in node 18. However, S_3 does not contain a match of $k_1 = XML$ in a descendant of node 11 which is not also contained in an LCA. Therefore, S_3 is not a valid grouping.

ELCA does not remove all false positives since unrelated entities are still grouped together if no better matching is possible. Additionally, false negatives may be introduced under certain conditions [351].

SMALLEST LCA The *Smallest Lowest Common Ancestor* [372], also used in XBridge [245], enhances the concept of LCA by a minimality constraint. Only LCA nodes that do not have further LCA nodes among their descendants are SLCA nodes. It should be noted that this definition is stricter than that of ELCA in that it generally forbids LCA nodes that have LCA nodes among their descendants, while ELCA only constrains the context in which an LCA node may contain another LCA node. Furthermore, SLCA only allows one match instance for each keyword in an answer set.

SLCA addresses the problem of false positives as described in Section 4.3. Evaluation of the query $K = \{Smith, Web\}$ on the data in Figure 2 leads to the LCA nodes 2 and 1. Node 2 is a node of type `article` and constitutes a meaningful result, while node 1 is the root node of the document and the keyword matches are distributed over two different articles. According to SLCA semantics, only node 2 is a suitable result node since it does not contain LCA nodes. Node 1 is an LCA node but not a return node since it is an ancestor of another LCA node, node 2.

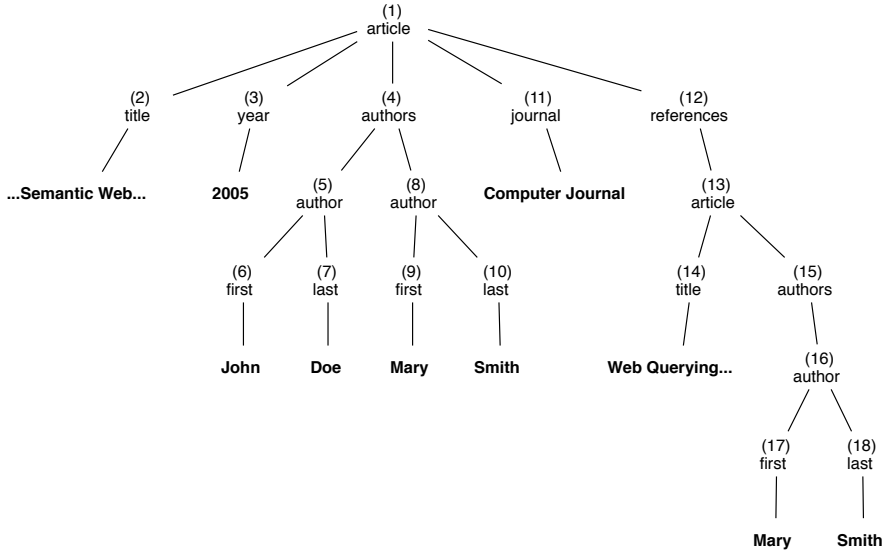


Figure 8: Sample XML data

However, as in ELCA, false positives can still occur. The query $K = \{Smith, 2003\}$ has $S_1 = \{11, 15\}$ and $LCA(S_1) = 1$ as a result according to SLCA. This answer is not meaningful since the keyword matches are distributed over two different articles, but is not filtered out since there are no further keyword matches and thus no further LCA in the data.

Additionally, disallowing nested LCAs can also lead to false negatives, for example when the same query, $K = \{Smith, Web\}$, is applied to the data in Figure 8. Among others, this produces the answer sets $S_1 = \{18, 14\}$ and $S_2 = \{10, 2\}$ and the corresponding groupings $LCA(S_1) = 13$ and $LCA(S_2) = 1$. Both LCA nodes represent articles which contain both of the query terms and thus constitute relevant results. However, since the second LCA is an ancestor of the first, SLCA filters out the latter. Consequently, only the referenced article is retrieved as a result.

MEANINGFUL LCA The *Meaningful Lowest Common Ancestor* (MLCA) [250] of a set of nodes is its LCA given that for each pair of nodes, there are no other combinations of nodes with the same label. that have an LCA node which is a descendant of their LCA node. Intuitively, this means that for all keywords, the node with the keyword label that is most closely related to the other matched nodes is found. This is based on the assumption that a lower LCA means a stronger connection. The concept of MLCA combines SLCA with Interconnection Semantics and suffers from similar problems with respect to false positives and negatives.

AMOEBA JOIN Amoeba Join [7] is another method used for grouping matched nodes. An *Amoeba* is an answer set which contains its LCA node, meaning that one of the nodes in the answer set is in an ancestor-descendant relationship with all other nodes in the set.

Applying the query $K_1 = \{Smith, Web, article\}$ to the data in Figure 2 yields the answer sets $S_1 = \{11, 23, 2\}$ and $S_2 = \{11, 3, 2\}$, among others. The former is not an Amoeba since $LCA(S_1) \notin S_1$, and is not considered a valid grouping of matches according to Amoeba join. On the other hand, S_2 is a valid grouping because it contains its LCA, node 2.

Amoeba join can be too restrictive, leading to false negatives and unintuitive results: the query $K_2 = \{Smith, Web\}$ does not yield any results in the data of Figure 2 even though it is a relaxation of K_1, and it can be expected that all query answers of K_1 are also answers to K_2. On the other hand, recursive elements can lead to false positives as discussed by [351].

VALUABLE, COMPACT, AND COMPACT VALUABLE LCA A *Valuable LCA* (VLCA) [243] is an LCA in which the keyword-matching nodes are *homogeneous*. A set of matched nodes is said to be homogeneous if no node label on the paths between them and their LCA (excluding matched nodes themselves) occurs more than once. In other words, each element in the set of the labels encountered when traversing from each matched node to the common LCA should be unique. In Figure 2, for example, nodes 7 and 22 are not homogeneous since there are two nodes with label `article` on the path between them, nodes 2 and 13. Nodes 3 and 5, on the other hand, are homogeneous. VLCA is conceptually identical to all-pairs related interconnection semantics in XSearch and has the same problems with false positives and false negatives.

Compact VLCAs (CVLCAs) were introduced to achieve faster computation of VLCA nodes. CVLCAs are compact in that they enforce maximally specific results. More precisely, a Compact LCA node is the LCA node of an answer set that *dominates* all nodes in the set. A node v_i is said to dominate another node v_j if there is no answer set involving v_j that has an LCA which is a descendant of v_i. Intuitively, an LCA is only a Compact LCA if it holds for all contained matched keywords that they could not be part of a grouping of matches that has a more specific LCA. This concept is similar to that of SLCA and suffers from the same drawbacks. A *Compact Valuable LCA* (CVLCA) finally is a CLCA that is also a VLCA.

RELAXED TIGHTEST FRAGMENT Kong et al. [226] introduce the idea of *Relaxed Tightest Fragments* (RTF), which allows for

multiple matches of a keyword in one answer set. RTF requires that, for a given answer set S_i, no subset which is also an answer set may have an LCA that is different from the LCA of S_i. Additionally, the set of keyword matches has to be the maximum set of matches for the given LCA, i.e., it should not be possible to add further keyword matches to the set without the addition resulting in a different LCA. Finally, no keyword match node in the set can be part of a keyword answer set whose LCA node is a descendant of the LCA of S_i.

RTF is a variation of CVLCA where the way of generating the answer set and the first two constraints ensure that the result subtrees are complete with respect to the keyword matches while still being as small as needed to cover at least one instance of each keyword match.

For example, the query $K = \{XML, RDF\}$ executed on the data in Figure 6 yields keyword match lists $L_1 = \{14, 15, 21\}$ and $L_2 = \{17, 21\}$ and, among others, answer sets $S_1 = \{14, 17\}$ and $S_2 = \{14, 15, 17\}$ with $LCA(S_1, S_2) = 11$. S_1 satisfies the first constraint since it does not contain an answer set as a subset. Still, S_1 is not a valid query answer, because keyword matches could be added to it without changing the LCA. S_2 has another answer set as a subset, namely S_1, but the LCA nodes of S_1 and S_2 are identical. The only keyword match that could be added to S_2 is node 21, which would change the LCA node to node 10. There are no possible LCA nodes below the LCA node of S_2. Thus, S_2 satisfies all constraints and is considered a query answer.

It should be noted that RTF can lead to false positives when keyword matches are distributed over several unrelated semantic entities (see above).

XKEYWORD Xkeyword [198] is one of the few approaches to determine semantic entities at the schema level. Here, the XML schema graph is manually grouped into possible return types, so-called *target objects*, which are then annotated with their relationships to other target objects. For example, a target object of type `article` could consist of article, author and title nodes and stand in a `contained in` relation to a target object of type `proceedings`. Queries are then processed by retrieving the objects relevant for the keywords and generating minimal cycle-free subgraphs that contain all keywords. These in turn can be mapped to subtrees of the target object graph, yielding the query results.

SCHEMA-LEVEL SLCA *Schema-level SLCA* [238] is a connection heuristic that is similar to SLCA, but further limits the number of groupings by requiring that a valid grouping may not contain any other groupings not at the instance level but at the schema level. This means that the path to a Schema-level SLCA root node

```
for $a in mlcas doc("bib.xml")//author
      $b in mlcas doc("bib.xml")//title,
      $c in mlcas doc("bib.xml")//year
where $a/text() = "Mary"
return <result> {$b, $c} </result>
```

Figure 9: Schema-Free XQuery

may not be a prefix of the path to any other Schema-level SLCA root nodes. For example, the query $K = \{web, journal\}$ evaluated on the data of Figure 6 yields the answer sets $S_1 = \{12, 3\}$ and $S_2 = \{23\}$ with $LCA(S_1) = 2$ and $LCA(S_2) = 23$.

The path to $LCA(S_1)$, however, i.e., the path `bib/article`, is a prefix of the path to $LCA(S_2)$, `/bib/article/journal`. Thus, S_1 is not a valid grouping according to Schema-level SLCA semantics. As the example demonstrates, Schema-level SLCA introduces additional false negatives compared to SLCA. Furthermore, Schema-level SLCA does not solve the false positive problem that occurs when keyword matches span several articles and their LCA does not contain further keyword matches.

4.3.2.2 *Determining Return Values*

Once the relevant semantic entities have been identified and the domain of the answer is established, the return values can be computed. Ideally, the query answer should contain all the information that is relevant to the query without being too verbose or including irrelevant information. In many systems, the return value is either the whole semantic entity, i.e., a subtree [372, 243, 179], or a summary of it, such as the paths from the keyword matching nodes to the root node [243, 226, 197]. However, several other approaches exist as well.

XSEARCH XSearch [109] returns the matched nodes together with their content. As long as the query only consists of keywords or label-keyword pairs, this leads to relatively uninformative answers. However, XSearch also allows for query terms of the form l :, thereby enabling the targeted selection of entity properties. For example, executing the query $K = \{last{:}Doe, title{:}\}$ on the data of Figure 2 returns nodes 3 and 8, thereby providing the title of the publication authored by John Doe.

SCHEMA-FREE XQUERY Schema-Free XQuery [250] is an extension of XQuery by the MLCAS (Meaningful Lowest Common Ancestor Structure) operator for keyword querying.

An example of a query in Schema-Free XQuery is shown in Figure 9. The result of this query are the years and titles of works by author "Mary." Since the **MLCAS** keyword is present, the variables `$a`, `$b` and `$c` are respectively bound to nodes with labels `author`, `title` and `year` upon evaluation. The MLCAS then is the structure consisting of those nodes among which the MLCA relationship holds.

Having determined the MLCAS, the variables are bound to the content of the children of the respective nodes in the MLCAS. The keyword aspect of Schema Free XQuery thus pertains to node labels and not, as in many other keyword query languages, to content.

XSEEK XSeek [255, 254, 253] infers return structures, automatically grouping the terms in a query into those that express search predicates and those that specify return information. If a keyword w_i matches a node label, and no other keyword in the query matches the node content of a descendant of w_i, then w_i is considered to be a *return node*. All nodes that are not found to be return nodes are *predicates*.

If no return nodes can be inferred, the entities in the paths from the matched nodes to the VLCA node as well as the lowest ancestor entity of the VLCA node are considered to be the return nodes. A node is considered an *entity* if it is in a many-to-one relationship with its parent. For example, a bibliography often has several article nodes among its children, making article nodes entities. These relationships can be inferred from the relations in the data or, if present, from the schema.

The result of a query then consists of two parts, the return nodes and their associated information and the paths from the VLCA to the matched nodes.

4.3.2.3 *Expressive Power*

In their most basic and also their most common form, keyword queries consist of unordered lists of terms connected by implicit conjunction. Using such a simple syntax, most query intents can only be approximated vaguely and a targeted selection of data according to precisely specified criteria is not possible. Keyword-enhanced web query languages provide greater expressive power, but also carry a significant overhead and are less easy to use.

Apart from label:keyword syntax and user-defined return values (see Section 4.3.2.2), some languages allow for slightly increased expressive power in the form of disjunctions [342], optional terms [109], or operators for numerical comparisons [371].

Abbaci et al. [7] present a keyword-only query language that offers an advanced syntax using the operators **AND** (conjunction), **OR** (inclusive disjunction), **INC** (inclusion, meaning that one operand

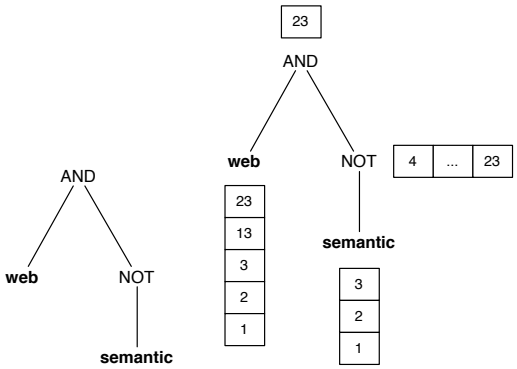


Figure 10: Query tree (left) and query evaluation (right)

must occur in a node that is a descendant of the node containing the other operand), **SIB** (sibling), and **NOT** (negation), as well as parentheses to indicate precedence.

Query evaluation is realized by transforming the query into a binary tree with leaf nodes containing the keywords and internal nodes containing the operators. Then, sets of matching elements are constructed for each leaf node, i.e., for each keyword. If a node v_i contains a keyword w_i, all ancestors of v_i are also represented in the list of matches for w_i since they contain it indirectly. The data structure recording the matches stores the ID of the node in which the keyword term occurs, the type of occurrence, and the distance from the keyword match to the respective node (where a distance of zero means that the node contains the keyword directly). The answer sets for each leaf node are then further processed by applying the operator specified in a node to the answer sets of its children. The operator **AND** corresponds to the intersection of two sets, **OR** to their union, and **NOT** to their difference. Operators **INC** and **SIB** are realized via constraints on the distance from the keyword match. The query tree is processed in a bottom-up fashion, and once the root node of the query tree has been processed only the nodes matching the full query remain. Since indirect matches via ancestors are included in the answer sets, in the case of conjunctive queries, the LCA node is among the query results.

As an illustration, consider the query `Web` **`AND NOT`** `semantic` evaluated on the data in Figure 2. The query tree of this query is displayed on the left in Figure 10. The query is evaluated by first adding information about matching nodes to the leaf nodes, as shown on the right in Figure 10. The keyword "Web" is contained directly in nodes 23 and 3 and indirectly (i.e., via a descendant), in nodes 2, 1, and 13. The keyword "semantic" is contained directly in node 3 and indirectly in all its ancestors, i.e., in nodes 2 and 1. The operator **NOT** is applied by taking the difference between

the set of all nodes in the XML data and the nodes containing "semantic," resulting in the set of all nodes that do not contain "semantic," either directly or via an ancestor. Finally, to find the nodes that satisfy both conditions, the intersection of the sets of nodes containing "Web" and those not containing "semantic" is taken. The node list after application of the **AND** operator is the final result since the root node has been reached.

XQSuggest [247] is a system that supports the user in creating more expressive queries by suggesting for each keyword a number of *semantic strings*, path-like expressions that disambiguate between the possible elements the keyword can refer to. The user can then replace the keyword by one of the semantic strings in order to state his query in more precise terms.

4.3.2.4 *Ranking*

Several metrics for ranking query answers that operate not at the document level but at the granularity of the returned structures have been suggested.

XSketch [246] and the work by Bao et al. [39] use a ranking scheme based on tf-idf [205] and structural properties that does not rank individual results but rather types of nodes (LCA nodes in the case of Li et al. [246]), thereby indicating which types of subtrees are suitable query answers. As such, the approach lies between determining return entities and result ranking.

The schemes employed for ranking query answers are typically based on the size of the answer subtree [226], the distance between the matched nodes [327], or a variant [109, 39] of the vector space model [316] and tf-idf measure. The ranking mechanism of XRank [179] employs a combination of several of these criteria with a variant of PageRank [71] adapted to XML data. We exemplarily describe it in detail in the following.

XRank uses three criteria to rank results: specificity, keyword proximity, and the connections between elements. *Specificity* refers to the distance between the matched leaf nodes and the return node, while *keyword proximity* means the distance between the keyword matches themselves. Specificity, describing vertical distance, and keyword proximity, describing horizontal distance, combine into a two-dimensional proximity metric. A variant of Google's PageRank, ElemRank, is finally used to let the links between elements factor into the ranking value of the result node.

ElemRank is an adaptation of PageRank that takes specific characteristics of XML data into account, namely the bi-directional propagation of ElemRanks through links, the aggregation semantics for reverse containment relationships, and the distinction between containment links and hyperlinks. While hyperlinks are ignored when matching the keywords, they are considered when calculating ElemRanks. Containment links, which describe

the parent-child relationship between XML elements, represent a stronger relation than hyperlinking, e.g., through IDREFs. The two are thus handled separately, with the propagation of ElemRank value between elements connected by containment edges taking place in both directions. Additionally, the ElemRank of a node is defined as the sum of the ElemRanks of its children, which means that the ranking values of the subparts of an entity in turn combine into that entity's ranking value.

The ranking value of each instance of a keyword match is then calculated as its ElemRank value, scaled by a decay factor that is inversely proportional to the distance between the result node and the keyword match. The ranking value of the result tree finally is the sum of the ranking values of the contained keyword occurrences multiplied by a measure of keyword proximity which is based on the size of the smallest text window containing all matches. If a keyword has several occurrences in the subtree governed by the result node, the value of the node with the highest ElemRank value is used.

In summary, the criterion of specificity is realized as the decay scaling factor, where decay increases as the distance between a keyword occurrence and the result node grows, meaning that the ElemRank calculated from the link connections between the elements becomes smaller. The keyword proximity criterion, on the other hand, is represented by the scaling factor of the overall ranking value of the result, with a bigger distance between the keyword occurrences corresponding to a smaller scaling factor.

4.3.3 *Querying RDF*

This section presents various approaches to keyword querying on RDF data.

4.3.3.1 *SemSearch*

SemSearch [239] is a search engine for web documents augmented with RDF annotations. As output it returns a ranked list of matching HTML documents. Only the RDF data but not the documents themselves are processed during query evaluation.

A SemSearch query consists of pairs of a subject and a keyword connected by a colon, and the operators **`and`** and **`or`** to indicate conjunction and disjunction. During query evaluation, the keywords are matched only to *semantic entities*, that is, to classes, properties, and instances, but not to relations. It is assumed that query subjects refer to RDF classes and specify the return type. If no class matches the subject, the type of the subject is determined and rules are used to infer the return type from the types of

entities of the keyword and subject. For example, in the query `Mary:John`, Mary and John are both instances, and the rule for this case says that the return type should also be an instance, e.g., an article co-authored by Mary and John.

Using the list of matching entities and their types, the user query is then translated into SeRQL via templates. Multiple queries are constructed if a keyword matches several semantic entities. Since the number of such queries can be very large when keywords in the original query have multiple matches, rules are employed to reduce this number. If there are several matches of type class, for example, only the most specific class is considered. The application of the rules can be expected to decrease the recall of the search.

Finally, the retrieved documents are ranked, and the individual results are augmented with information about the matched entities. For ranking, two factors are considered, namely the distance between each keyword and its matches and the number of keywords satisfied by a search result.

4.3.3.2 *SPARK*

SPARK [378] is a search system for RDF data that translates keyword-only queries into SPARQL and ranks the resulting queries. Keywords are mapped to resources in the knowledge base, that is, to classes, instances, properties, and literals. This is achieved by using both the form and the semantics of the keywords. The form-based mapping uses string comparison techniques like the edit distance [241] and in addition applies stemming [256]. The semantics-based mapping retrieves semantically related words like synonyms using a thesaurus. In the process, a single query term can be mapped to several resources of different types. The different translations are augmented with confidence scores based on the similarity between the keyword and the concept.

Next, the query sets are constructed. If each keyword is mapped to exactly one resource, there is only one query set, otherwise all combinations of query sets, each containing one resource for each keyword, are generated. For each query set, a query graph is constructed using the minimum spanning tree algorithm of Kruskal [233], and missing relations and concepts are introduced to obtain a connected graph, which is then translated into a SPARQL query.

Finally, ranking scores for the generated queries are computed from the similarity of the keywords and the concepts they are mapped to, the proportion of overlap in resources between the keyword query and the corresponding SPARQL query and the information content of the query.

4.3.3.3 *Q2Semantic*

Q2Semantic [359] provides a system for querying RDF data using keyword-only queries. The latter are translated into formal queries, which can in turn be mapped directly to SPARQL queries. The system aims at a higher efficiency than comparable approaches. It operates on summarized RDF graphs, so-called *RACK graphs*, instead of the original data.

Q2Semantic ranks the query results and uses Wikipedia to find concepts related to the query terms. These concepts are also used to assist the user in entering his keyword query, as the interface offers auto-completion for RDF literals and Wikipedia terms.

When displaying the query results, Q2Semantic also shows the portion of the RDF data used in the query, as well as the translated formal query and a natural language explanation.

An RDF graph is converted into a RACK graph by mapping relations, attributes, instances, and attribute values to R- and A-edges and C- and K-nodes, respectively. R- and A-edges and C-nodes are then clustered together if they have the same labels and, in the case of edges, the same connections. K-nodes are merged when they are incident to the same A-edges, and the newly merged node inherits the labels of all the original K-Nodes. Costs are calculated for edges and nodes based on the number of elements merged to obtain them.

A keyword query is first matched against an inverted index which stores the K-Node labels. To allow for a broader vocabulary in the queries, the index is augmented with related terms extracted from Wikipedia, e.g. the anchor text of articles linking to an article whose title is a K-node label. Keywords are thus only matched to RDF attribute values. If there are several matches for one term, all of them are returned and used in the next step.

Starting from the matched K-Nodes for all query terms and using the cost functions of the edges as a heuristic for guiding the search, a tree is then gradually built up in the graph in a round robin fashion. To avoid recursion, repeated exploration of the same node within one path is penalized by adding a large number to the cost. A formal query is obtained when a root that is common to at least one instance for each keyword is reached.

Since several formal queries for the same keyword query may exist, a ranking function is employed that uses the lengths of the paths in the formal query, the scores of the matched K-nodes in the formal query, and a tf-idf-like measure to calculate ranking scores.

4.3.3.4 *QuizRDF*

QuizRDF [126] is a *browse-and-query system* for web pages that combines full text search with querying of RDF annotations,

where present. The idea behind this approach is that not all web data is annotated, and that it is not possible to capture every detail of the content of a text in its annotations. Combining full text search with RDF querying can thus potentially improve recall.

QuizRDF is described as an "information-seeking system" in which information is found by an interactive, gradual process rather than a targeted one-shot search. This approach is similar to the one proposed by Schmidt et al. [327], and allows users to explore the data, refining their queries as they gain more information about the nature of the data.

Initially, a so-called *ontological index* is created from both the textual content of a web site and its RDF annotations, which are linked to the RDF Schema ontology [70]. This index can then be queried using keyword search to obtain a list of matching web sites, which are ranked using the tf-idf measure [205]. For web sites with RDF annotations, the search results can be refined by restricting matches to a certain RDF resource class and entering literal values for RDF properties. QuizRDF also provides information about the ontological structure by displaying superclasses of the currently selected class as well as relationships to other classes.

4.3.3.5 *Q2RDF*

Q2RDF [309] is a system for querying RDF data using keyword-only queries. Results are ranked in a way that is similar to Q2Semantic. Q2RDF operates on an RDF sentence graph [377], an undirected graph consisting of RDF sentences and the connections between them. An RDF sentence is the set of all RDF triples that are *b-connected*, that is, that contain the same blank node. B-connectedness is transitive, and RDF statements which do not contain blank nodes are separate sentences. The label of a node in an RDF sentence graph consists of the words contained in the subjects, predicates and objects it summarizes. Any RDF graph can be collapsed into an RDF sentence graph. Figure 11 shows an example of an RDF graph and its grouping into sentences. Due to the transitivity of the b-connectedness relation, RDF sentences are not stable and may change when a blank node is introduced in a different part of the RDF graph, see Figures 11 and 12.

In the preprocessing step, an inverted index and a path index are created. The inverted index indicates which word appears in which sentences. The path index indicates for each node which other nodes it can reach, and allows for the construction of all shortest paths between nodes. Shortest paths are calculated using Dijkstra's single source shortest path algorithm.

When a user poses a query, the keywords are first mapped to the RDF sentences in which they appear. The goal then is to find

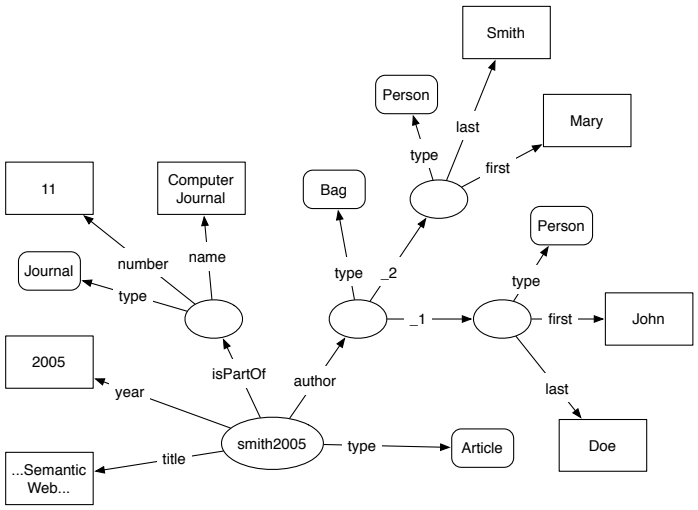


Figure 11: An RDF graph

answer trees, that is, trees that contain all keywords and in which every leaf node contains at least one keyword. This is achieved by starting from the matched nodes and gradually visiting nodes until a path connecting all matched nodes is found. The next node to visit is determined by first computing the set of keyword match nodes with the smallest cardinality and expanding the nodes contained in it first. Then, the node that is closest to the node currently being expanded is visited and added to the set of nodes to expand.

If only tree size is considered as a measure of goodness, then this method allows for the generation of the top-k answer trees without having to generate all the answer trees first, since the length of the paths and thus the size of the result trees grows as the number of visited nodes increases (the same is true for finding the top-k lowest cost answer trees in Q2Semantic, since all cost values are positive).

The algorithm can result in isomorphic answer trees, such duplicate answers are discarded. The generated answer trees are then ranked using a variant of the term frequency measure.

Q2Semantic and Q2RDF are similar in that they both summarize the initial RDF graph and then construct minimal answer trees containing all matched nodes to find the top-k results, which are then ranked using a tf-idf-like measure. The two approaches differ in the way in which they evaluate results (Q2Semantic translates queries into complex queries while Q2RDF retrieves the results directly) and reduce the RDF graph, in the element types against which keywords are matched, and in the cost function that guides the search for answer trees. Additionally, the answer trees of Q2Semantic show a lower granularity, because Q2Semantic merges edges and attributes only when they have

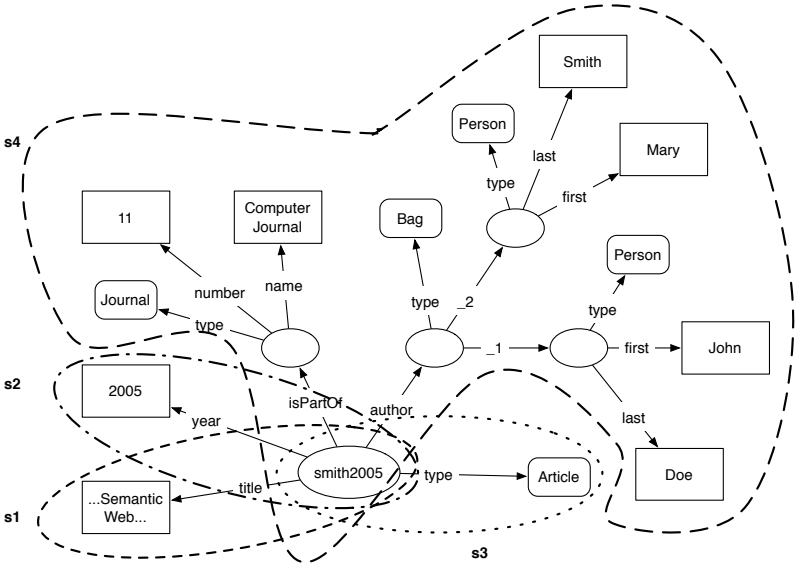


Figure 12: The data of Figure 11 represented as an RDF sentence graph

the same label, while Q2RDF collapses all elements that belong to the same sentence into a single node.

4.3.4 *Discussion*

The majority of keyword query languages for semi-structured data in the literature are concerned with keyword-only querying of XML data. Fewer proposals exist for querying RDF data, and a majority of them translate keyword queries into traditional query languages. Most XML keyword query languages, on the other hand, evaluate queries without mapping them to another query language.

At the same time, keyword query languages for XML usually limit themselves to the processing of tree-shaped data, that is, to XML without hyperlinks. Those languages that do work on graph-shaped XML, like XRank, ignore hyperlinks during the matching and grouping process and only use them for ranking. A notable exception is SAILER [242], which models XML and HTML documents as graphs. There also exists work on extending interconnection semantics to deal with XML data containing IDREF links [107], which due to its purely theoretical nature has not been discussed here. As Schmidt et al. [327] point out, one reason for the relative lack of keyword querying for graph-shaped XML is the expected increase in complexity and thus processing time, which would be very problematic in an application area dealing with large amounts of data.

Correspondingly, the lack of RDF keyword query languages that evaluate queries directly can be attributed to the fact that RDF is graph-shaped and cannot be converted into tree-shaped data as easily as XML. In addition, querying RDF poses additional challenges in the form of labeled edges and blank nodes. A possible solution to this problem is to summarize the RDF graph into a different structure [309, 359], but this comes at the cost of partially ignoring the structure of the data and thus reducing the granularity of the query result.

For XML querying, on the other hand, the grouping of matches is of great importance, and it is a central aspect of many of the approaches discussed in this chapter. Various heuristics for grouping have been proposed, a large majority of which are refinements of the established concept of LCA, e.g., SLCA [372], MLCA [250], CVLCA [243], and interconnection semantics [109]. All of these approaches add constraints to LCA in order to remedy the problem of false positives in LCA and improve the grouping of matched nodes according to their semantic entities. The approaches differ in the filter that they apply to remove undesirable results from the set of LCA nodes; each of them produces a set of results that is a subset of the results obtained by applying LCA.

The reason why determining semantic entities in structured data is so important to keyword querying is that, in contrast to traditional query languages, queries are never fully specified, and in fact often cannot be fully specified by the user. The inferred semantics are what is used to determine what constitutes a relevant result.

While most of the approaches determine the LCA or a variant thereof automatically based on keyword match instances, an alternative approach that was used in XKeyword [37, 198] but also mentioned in connection with XRank [179] and employed in keyword querying databases [57, 125] is to manually group the data into concepts and thus pre-define the possible query answer components. This method uses an extra level of processing where parts of query answers are defined a priori and therefore independent of a specific query. While this has the disadvantage of requiring manual annotation, it alleviates two fundamental problems of LCA-based methods for automatic grouping.

The first problem stems from the underlying assumption that only elements in the subtree governed by the concept root node are relevant to the query answer. As mentioned in the beginning of Section 4.3, this means that relevant information about an entity is not returned when the keywords in a query are contained in a subtree of the tree representing an entity. Given the example data of Figure 2, for example, the queries $K = \{Doe, Smith\}$ and $K = \{Semantic, 2005\}$ will produce only trivial results without any

additional information about the respective articles, like the title and year of co-authored articles in the first case and the names of the authors in the second.

There are two ways to overcome this problem: displaying the query result in conjunction with the data and enable search-and-browse behavior, or allowing matching on label nodes and enabling a more targeted specification of a return value. For example, the first keyword query above could be extended to $K = \{Doe, Smith, title, journal\}$, meaning that the concept node, i.e., the root node of the semantic entity is of type `article` and not `authors`, and that the entity subtree contains the desired information. This is possible in the query language of Cohen et al. [109] and in XSeek [255, 252] and will be discussed further below.

The second problem is closely related to the first: the different heuristics for grouping aim at being universal or at least versatile; on the other hand, they are data-driven and make assumptions about the relations between structure and semantics that may not be universal. The difference between data-centric and document-centric XML, for example, suggests different requirements concerning grouping and return values. When querying document-centric XML, multiple occurrences of the same keywords within an XML subtree indicate particular relevance. The same is not necessarily the case for data-centric XML.

Consequently, all LCA-based grouping strategies presented in this chapter are not universally applicable and under certain circumstances may lead to both false positives and false negatives. This raises the question to what extent it is possible to reliably deduce semantics from structural characteristics of data alone.

A small number of very recent approaches group keyword matches not just based on structure, but also take the distribution of keyword matches and node types in the data into account [246, 39]. Whether these methods will solve the problems associated with LCA-based grouping remains to be seen.

To summarize, manual grouping at the schema level works well and has the advantage that data containing hyperlinks does not pose a problem. An obvious disadvantage is that it requires users or administrators to invest time and effort to define the groupings. LCA and its variations, on the other hand, are computed automatically, but all algorithms require the presence of certain characteristics in the data to perform well. One way to achieve good grouping performance could thus be to simply consider the manual grouping as an additional, possibly optional, step of semantic annotation and to encourage users to actually perform the grouping.

A more promising approach is the use of modes to determine which grouping mechanism is appropriate for a given dataset or a given combination of a dataset and a query. Since the various

grouping algorithms make different assumptions about the relation between syntax and semantics in the data, the best algorithm could then be selected automatically, which hopefully would leading to an improved overall performance.

To evaluate the feasibility of this approach, several questions have to be addressed. A priori, it is not clear how many, and which, grouping algorithms should be used, whether there is a universally optimal combination of grouping mechanisms, or whether the selection should be application- and domain-dependent. A more basic question concerns the characteristics according to which the grouping mechanisms should be selected: a small number of maximally complementary algorithms could simplify mode selection, whereas a larger number of algorithms might prove more versatile and thus improve the result.

Finally, one would have to decide which features or characteristics of the data or query should trigger a change of mode, and how the optimal mode should be selected. Examples for relevant features include the amount of content relative to the amount of structural information, term frequency distributions, and structural characteristics derived either from the schema or the data itself.

Learning, either through implicit or explicit feedback, might prove useful for the automatic selection of the appropriate mode. An example for implicit feedback in this context are results that are favored and disfavored by the users, based on results that are clicked or skipped given a page of results [310]. Explicit feedback could be provided in the form of a manually annotated training set or Query-By-Example type queries [381] by which the user indicates the intended form of the result. Querying of semi-structured data using the Query-By-Example paradigm has been studied previously, resulting in the query language visXcerpt for XML data [49].

Even if automatic mode selection proves feasible, the issue of cyclic data remains problematic, since none of the existing automatic grouping mechanisms can operate on data containing hyperlinks. It is thus desirable to find a generalized universal grouping mechanism which can be applied both to XML and RDF data.

Many of the keyword languages discussed is this chapter focus on connecting keyword matches, whereas the form of the query answers has been addressed in less detail. Possible strategies are for example to return the subtree governed by the concept node, the paths from the keyword matches to the concept nodes, or just the concept node. These different return structures offer different tradeoffs between conciseness and information value.

An important characteristic of traditional query languages, namely the targeted and flexible retrieval of elements, can be

found only in two of the presented stand-alone keyword query languages, in that of Cohen et al. [109] and in XSeek [255, 252]. Both of these languages return the content of a node whose label is matched.

However, neither of them allows for the binding of specific values to variables. Query results thus cannot be used further in construction terms, which is a desirable feature in various applications, for example when embedding queries in Wiki pages. Furthermore, it is not possible in XSeek to specify explicitly that the content of a node with a specific label should be retrieved. Rather, the necessary information is inferred from the keyword query and is therefore relatively hard to control by the user, even if she knows exactly which nodes she would like to have returned.

Keyword-enhanced query languages, on the other hand, allow for a more targeted selection and enable construction to varying degrees. Schmidt et al. [327] only retrieve the label of the LCA node, the approach of Florescu et al. [152] makes the granularity of the return value dependent on the specificity of the query, and Schema-Free XQuery allows for the binding of variables to specific nodes in an entity subtree.

Another important aspect of keyword querying concerns the ranking of the results. Here, the underlying principle is that a smaller distance between matched nodes and between matched nodes and concept nodes generally means more specific and thus better results. Ranking is usually realized in terms of the vector space model and a variant of the tf-idf measure. It makes sense to rank the results before fully generating them, since this allows for retrieving only the top-k results, such that the results can be displayed faster and that processing time can be saved when the user is not interested in all results.

A different, but equally important, question is how to convey the vocabulary for queries. Keyword query languages are flexible with respect to the structure of the data being queried, and the ability to query over heterogeneous data is often highlighted as one of their advantages. In this context, heterogeneity can refer either to differences in the structural organization of the data or to differences in the vocabulary used.

Figure 13 shows the data of Figure 2 in a different structural organization, with articles grouped by their authors.[14] Due to the automatic grouping, one keyword query can be used to query both documents. However, the query results may differ since the grouping uses structural characteristics to find query answers.

Yet another reformulation of the data in Figure 2, with identical structure but different node labels, is shown in Figure 14. A query involving node labels can never successfully retrieve

14 The repetitions of the article subtrees have been omitted to increase readability.

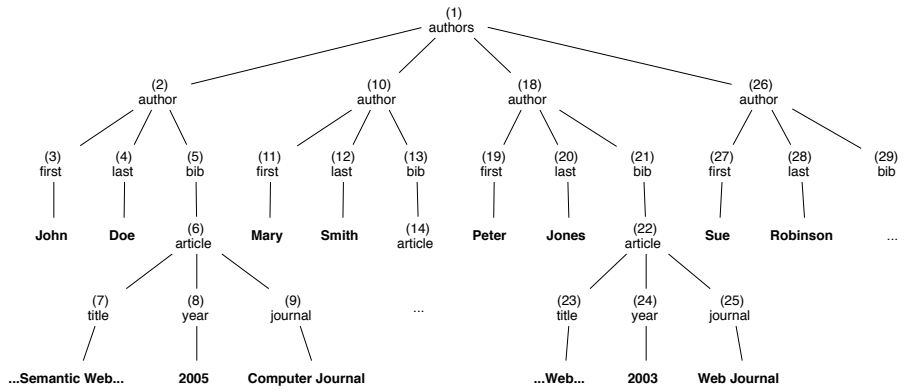


Figure 13: Alternative formalization of the data in Figure 2, articles grouped by authors

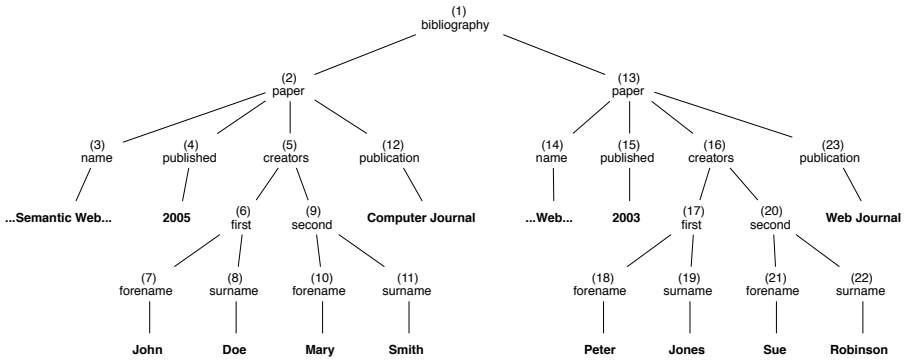


Figure 14: Alternative formalization of the data in Figure 2, different node labels

results from both the document in Figure 2 and that in Figure 14, because they use different vocabularies. For example, the label-keyword queries $K_1 = \{published{:}2005, surname{:}Smith\}$ and $K_2 = \{year{:}2005, last{:}Smith\}$ express the same informational need, but use different words. Consequently, K_1 does not produce any matches in the data of Figure 2, and the same is true for K_2 and the data of Figure 14. In general, a query term may have many synonyms, and a user may not know which words to use in her query. This problem also applies to homogeneous data, since a user may not know a priori which terms are used. It is however of particular concern when querying heterogeneous data using different vocabularies, in which case there is no standardized vocabulary that the user could learn.

In a seminal study of the *vocabulary problem*, Furnas et al. [158] found that participants used a large number of different terms to refer to the same concepts. The probability of two people

choosing the same word for a given object was found to be below 20%. At most 36% of the participants chose the "best," that is, most frequent term for an object. The proposed solution to this problem is to establish a list of synonyms or aliases for each term. For example, a system could map the term "published" to "year," thus enabling the use of both terms in queries.

More generally, *query expansion* can be used to improve the recall in information retrieval applications by finding synonyms, morphological variations, and misspellings. A variety of techniques for automatic query expansion have been proposed [123, 206, 130, 120], some of which are employed in the keyword query languages presented in this chapter. Q2Semantic uses Wikipedia to find terms similar to the keywords. Li et al. [250] identify different ways to obtain a domain-specific thesaurus to be used with the **`expand`** function of Schema-Free XQuery: deriving a list of synonyms for each term from the corpus of XML data, creating it manually, or through information retrieval techniques like bootstrapping. For cases where no domain-specific thesaurus is available, the authors suggest the use of a universal thesaurus like WordNet [277]. This is also how *semantic mapping* works in SPARK. In addition, *morphological mapping* is employed, which functions on the form (rather than the semantics) of the keywords and uses stemming and other methods and measures from natural language processing. Each term mapping is augmented with a confidence score, meaning that the list of synonyms can also serve as a controlled way to semantically relax the query.

Overall, current keyword-query languages for XML and RDF satisfy one of the two criteria laid out in the beginning of Section 4.3, simplicity. While keyword-enhanced query languages are likely not simple enough to be used by all users, a majority of keyword-query languages have a very basic syntax that is no more complex than that of regular web search. The syntactic extensions offered by some of the languages are optional and comparatively simple.

The case is less clear for the second criterion, flexibility. All languages are schema-agnostic in the sense that they can query data independent of the underlying schema, or even independent of the fact whether there is a single schema that all data adheres to. However, as grouping mechanisms are data-driven and therefore depend not only on the structure but also on naming conventions, results can vary significantly depending on the schema being used.

Flexibility with respect to the data type, i.e., the ability to query data in different formats, has received relatively little attention. XRank and Sailer can be used to query both XML and HTML documents, but do so mainly by treating HTML documents as unstructured text.

The combined querying of XML and RDF is particularly desirable in the context on the semantic web, where not all content of the (XML) data is necessarily represented in (e.g., RDF) metadata, or vice versa [59]. If both could be queried using a single query language, recall would be increased, and users would only have to familiarize themselves with one query language.

SemSearch and QuizRDF can query web documents augmented with RDF annotations. The former only evaluates the query on the RDF annotations, meaning that it is not possible to impose conditions on both the document itself and its annotations in one query. QuizRDF, on the other hand, allows for the restriction of web documents matching a given query through their RDF annotations. However, it is not possible to query the structure of web documents or to combine XML and RDF search in a single query. Moreover, QuizRDF is a search-and-browse system and returns web documents, i.e., it does not allow for the grouping of entities and consequently provides no flexible return values. Because of this, it is much more suited for interactive exploration of data than for expressive querying at a high granularity.

While to the best of our knowledge there are currently no systems for the combined keyword querying of XML and RDF data, a number of approaches to keyword querying are explicitly concerned with queries over HTML and XML data and relational databases [208, 244], thereby realizing data type flexibility to a certain extent.

Part II

THE KIWI WIKI

5

A CONCEPTUAL MODEL FOR THE KIWI WIKI

Traditional wikis excel at enabling collaborative work on emerging content and structure. Semantic wikis go further by allowing users to expose knowledge in ways suitable for machine processing using semantic web technologies. The combination of ease of use, support for work in progress, and semantic web technologies makes semantic wikis particularly interesting for knowledge-intensive areas of work such as project management and software development.

Most of the advanced technologies that semantic wikis employ were developed for use in a static environment with annotations and rules being crafted by knowledge representation experts. This is in contraposition to the ever-changing, dynamic character of wikis where content and annotations are, for the most part, created by regular users. In such an environment, inconsistencies, disagreements and ambiguities can easily arise and the system should therefore be able to cope with them to support users in their work.

While several semantic wikis have been put to practical use (see Chapter 3), each using its own conceptual model, there has been little explicit theoretical exploration on the possible choices for conceptual models and their consequences [348].

By *conceptual model*, we here understand the basic concepts or buildings blocks that a user interacts with when using the semantic wiki as well as the relations between them. In a traditional wiki, there are only few such building blocks, typically pages and links. Semantic wikis extend this model by new concepts such as typed links, tags and RDF or OWL annotations. The basic building blocks of a semantic wiki and how they relate to each other have rarely been discussed in the literature, and one can assume that many decisions in this regard have been without full consideration of the design space.

In this chapter, we seek to draw attention to this issue, suggesting a conceptual model for KiWi and showing that the design of a conceptual model for a semantic wiki is a non-trivial issue and design choices greatly influence which functionalities the system can offer and how the user sees the system. We will show that there are several possibilities for approaching various issues in a semantic wiki which have advantages and disadvantages, as well as important consequences on how other issues can be approached.

Note that the conceptual model described here is similar but not identical to the conceptual model behind the current implementation of the KiWi wiki. Specifically, KiWi does not yet support annotated links, negative and structured tags, the assignment of multiple labels to a single tag concept and the calculation of the social weight of tags.

5.1 CONTENT

This section outlines the representation of content in the KiWi wiki. "Content" here refers to text and multimedia which is used for sharing information, most frequently through the use of natural language, between the users of the wiki, and whose meaning is not directly accessible for automatic processing. Information Extraction techniques enable the computerized processing of structured data extracted from text or speech, but this introduces another level of representation which we do not consider content in this sense.

While the data in many conventional wikis is restricted to content and link structure, semantic wikis add further layers, namely annotations that can be used for human as well as automatic processing and annotations that are intended mostly for computers and not easily understandable for humans. These two other types of data, informal and formal annotations, are discussed in Section 5.2.

5.1.1 *Content Items*

Content items, the primary unit of information in the KiWi wiki, are composable, non-overlapping documents. Every content item has a unique URI and can be addressed and accessed individually. There is no inherent distinction between wiki pages and content items or rather, all content items are wiki pages.

A content item can directly contain only one type of content, for example text or video. An atomic textual content item can be thought of as being similar to a paragraph or section in a formatted text in that it contains some relatively self-contained text and can be combined with other such data items to form a content structure: Content items can be nested through transclusion [282] to enable the representation of complex composite content structure. Consequently, a content item may contain (textual or multimedia) content and any number of inclusion declarations.

Having an explicit concept of content structure in a wiki is desirable both with respect to the semantic as well as the social nature of a semantic wiki; the structural semantics of the content can be immediately used for querying and reasoning, for example

for automatically generating tables of contents through queries, as well as for facilitating collaboration and planning of content. In addition, content items constitute a natural unit for assigning annotations to content (see Section 5.2).

Allowing one content item to have several parents, that is, to be directly contained in multiple other content items, is a design decision that adds functionality but also has side-effects, some of which may be undesirable.

The multiple embedding of content items means that content items can be easily reused and shared, which is useful for example for schedules or contact data. If only a copy of the content item's content was embedded, multiple occurrences of the content item in the wiki could not be traced as naturally or easily. For example, changes to a schedule or email address would have to be made manually in all content items where the information appears. On the other hand, updating a content item that is a child of several other content items or reverting it to an earlier version can have unintuitive and unwanted side effects when the content item changes in all contexts it is embedded in without the editing user being aware of all these contexts. Therefore, upon modifying a content item that appears in several different locations, the user should be presented with a list of the embedding locations and the choice to edit the content item itself or a copy of its content.

Loops arise when a content item contains itself as a descendant through content item nesting. The resulting infinite recursion is problematic with respect to the rendering of the content item[1] as well as reasoning and querying. Since loops additionally arguably have no straightforward meaningful interpretation in the wiki context, transclusions which would cause loops are generally forbidden. However, it is possible to embed a content item several times into another content item or one of its descendant content items as long as the embedding does not lead to a loop.

Assuming that all content item nestings are resolved, the wiki content can be seen as a set of finite trees. Root nodes, that is, content items that are not contained in another content item, then have a special status in that they encompass all content that forms a cohesive unit. In this, they can be seen as being alike to a wiki page in a conventional wiki.

5.1.2 *Fragments*

Fragments are small continuous portions of text (or, potentially, multimedia) that can be annotated. While content items allow authors to create and organize their documents in a modular and structured way, the motivation behind fragments is to enable

1 At least if we assume that all of the content item is to be rendered at once.

the annotation of user-defined pieces of content independently of the canonical structure. If content items are like chapters and sections in a book, then fragments can be seen as passages that readers mark and annotate; they are linear and in that transcend the structure of the document, spanning across paragraphs or sections. Different parts of the book might be marked depending on which aspect or topics a reader is interested in, and the same is true for defining and tagging fragments.

Fragments should be maximally flexible in their placement, size and behavior to allow for different groupings of text. Towards this goal, it is generally desirable that —unlike content items— fragments can overlap. The intersection between two overlapping fragments can either be processed further or can be ignored. When two overlapping fragments f_1 and f_2 are tagged with a and b respectively, a third fragment that spans over the overlapped region and is tagged a, b could be derived automatically. Similarly, automatically taking the union of identically tagged overlapping or bordering fragments might be intuitive and expected by the user. However, this automatic treatment of fragments might not always be appropriate or wanted.

Therefore, fragments in KiWi are seen as co-existing but not interacting, meaning that the relationships between fragments are not automatically computed and no tags are added. This view has the advantage of being simple and leaving the control of the fragments and their tags to the user. It is also in tune with the philosophy that, unlike content items that always only realize one structuring, fragments are individual in that different users can group a text in many different ways and under many different aspects which are not necessarily related.

Fragments can either be restricted to be directly contained in one content item, or they can span across content items. In the latter case, a rearrangement of content items can lead to fragments that are part of multiple content items which no longer occur in successive order in the wiki and, similarly, content item nesting means that content items may contain only part of a fragment with the other part being absent (but present in some other content in which the content item is used). Fragments could be automatically deleted when the structure of content items no longer supports them, but this means that a user might find a fragment she created destroyed as a consequence of another user's rearrangement of content items.

To avoid these problems, in the KiWi wiki, fragments start and end in the same content item and cannot span over contained content items. One single content item's text then contains the whole fragment.

Two possibilities for realizing fragments are the insertion of markers in the text to flag the beginning and end of a fragment

(*intrusive*), or external referencing of certain parts of a content item, using for example XQuery (see Section 4.2.1.2), XPath (see Section 4.2.1.1), or XPointer [133] (*non-intrusive*). As the former means that fragments are less volatile and updates to the text do not affect fragments as easily, for example when text is added to the fragment, fragments in the KiWi wiki are intrusive.

5.1.3 *Links*

Links, that is simple hypertext links as in HTML, can be used for relating content items to each other and to external resources. Links have a single origin, which is a content item, an anchor in this origin, and a single target, which is a content item or external URI. Links can be annotated. Taking into account the internal links in a wiki, the content items present in the KiWi wiki form an unconnected directed graph which may contain loops.

5.2 ANNOTATIONS

Annotations are metadata that can be attached to content items, fragments and links. They convey information about the data item's meaning or properties. Annotations can be assigned manually by the users or derived automatically via rules.

Content items, fragments, links, and annotations carry system metadata such as the creation date and time and the author of a content item or tagging. These metadata are realized in the form of automatically generated annotations which cannot be modified by the user. The KiWi wiki comes with a pre-defined, application independent RDFS vocabulary expressing authorship, versions, and the like. This is not further developed in the following and the text focuses on user-generated annotations.

Two kinds of annotations are currently available in the KiWi wiki: tags and RDF triples. Tags allow to express knowledge informally without having to use a pre-defined vocabulary, while RDF triples are used for formal knowledge representation, possibly using an ontology or some other application-dependent pre-defined vocabulary. Users are by default not confronted with RDF, which is considered an advanced feature for experienced users, but the KiWi wiki aims to allow for a smooth transition between informal and formal annotation as will be described in the following.

5.2.1 *Formal Knowledge Representation—RDF*

The Resource Description Framework (RDF, see Section 4.2.2) is currently the most common format for semantic web data. RDF

data is suited for processing by machines but is often not easily human-interpretable. In practice, support for RDF is important to enable interoperability with current semantic web and linked data [53, 54] applications and a semantic wiki should therefore support at least the import and export of RDF data.

Ontologies can be specified using for example RDF Schema [70] or OWL [171]. The KiWi system uses the RDFS language to specify its ontologies because it is in many ways simpler than OWL but is sufficient for most purposes in KiWi. However, KiWi is not limited to RDFS but can also handle OWL ontologies, albeit to a limited extent and without full support for reasoning.

5.2.2 *Informal to Semi-Formal Annotations—Tags and Structured Tags*

One problem that frequently arises in the context of semantic web applications is that it is hard to motivate users to annotate content since they find the process complicated and laborious. Further, first having to learn RDF before being able to make an annotation is discouraging to users. One solution is to provide means for creating less formal annotations which are easier to use. As work progresses and users gain more knowledge, these annotations can be made increasingly more precise and can eventually be transformed into formal knowledge. Tagging (also discussed in Section 2.2.2) is one such kind of informal annotation. A tag assignment consists of the association of a term of phrase with a resource. Despite their simplicity, there are many possibilities as to how exactly tags are realized [338].

The conceptual model of the KiWi wiki employs tagging with advanced features to help overcome the downsides of uncontrolled, ad-hoc categorization and to enable a transition between informal and formal annotation, that is, tags and RDF.

Tag assignments can be explicit, that is, performed manually by a user, or implicit, inferred by a reasoner based on user-defined rules.

A tagging in KiWi is a tuple consisting of a user, a content item of the type *tag*, a *tag label*, a tagged resource, and maintenance information needed for processing. The latter includes for example information about the date when the tagging was created and a marker which allows to distinguish between explicit and derived taggings. The process of assigning a tag can then be seen as the user creating an association between a resource and a tag using a specific label.

TAGS AS CONCEPTS A hindrance in the transition from tags to more formal knowledge (e.g., RDF triples) is that tags are simple keywords that do not unequivocally map onto concepts. Often

different keywords can be used to express the same abstract concept (e.g., keywords in different languages or synonyms). Similarly, the same term might be used to express different concepts (e.g., homonyms like "bank"). A possibility that fits well in the wiki context, is to separate keywords and abstract concepts by using content items representing the abstract concepts instead of keywords for tagging.

The KiWi wiki distinguishes between tags as mere strings or labels and the abstract concepts they stand for by representing the tag concepts as content items, *tags* in our terminology. Each tag is associated with one or more labels which may overlap between tags, thus mirroring the non-unique mapping between names and concepts. Each tag concept content item may, but does not have to, contain a textual description of its meaning and a list of resources it has been assigned to, thus making it easier for users to negotiate the meaning and use of the concept.

Keywords still play an important role, as they are what is entered by the user, but the system automatically converts them to the corresponding underlying tag, possibly interactively by asking for clarifications in the case of ambiguities. When a user enters a tag label to be assigned to a resource, the system automatically resolves it to the corresponding tag concept, allowing the user to intervene if she disagrees with the result of the concept disambiguation.

Various approaches for semantic disambiguation, for example based on the co-occurrence of concepts and distances between words in WordNet [277], exist and may be employed. The URI of each tag concept's content item can be used as a tag label, meaning that each concept can be unambiguously addressed during tagging, even when all of its associated labels are ambiguous. This is especially relevant for the automatic assignment of tags where no user-intervention is possible or desired. When no resolving of ambiguous tag labels is available or desired, the system can be adjusted for this by requiring tag labels to be unique.

Using content items as tags also solves some further issues beyond synonyms and homonyms: Unlike keywords, content items have a URI and are addressable when transforming information of semi-formal tags into formal RDF models (e.g., by the use of rules). More importantly, content items also offer a place for further describing tags. This encompasses both natural-language explanations for humans on the meaning and intended use of the tag as well as machine-readable descriptions, e.g., by means of tagging a tag's content item.

Apart from the solution for resolving ambiguous mappings between concepts and labels discussed above, tags can also be matched to formal concepts in an ontology, allowing for the ontological grounding of tags.

TAG LABEL NORMALIZATION Tag normalization defines an equivalency on the set of tag labels. Tag label normalization can mean for example that the tag labels "Wifi," "WiFi," and "WIFI" are all converted to the tag label "wifi," the canonical form of this tag label equivalency class. Trailing whitespace and multiple whitespace within a tag are always removed upon saving the tag. Further, tag label normalization in KiWi is performed by converting all letters to lowercase and removing punctuation.

NEGATIVE TAGS In a collaborative context, we may be interested in tracking disagreements which requires a way to express negative information. Just as a user can tag a resource with the tag label "t," he or she may want to tag it with "not t" as a way to express disagreement or to simply state that the resource is not "t" or does not have the property "t." An example may be a medical doctor tagging a patient's card as "not lupus" to state that the patient definitely does not have "lupus." Negative tags thus explicitly express that something is not the case.

Although a tag "not t" could be seen as introducing classical negation into the system, it may in fact be only a very weak form of negation because we can allow the negation of pure tags only, not general formulae or sets of tags, and the only way to interpret this kind of negation would be by introducing a rule which says that from tag t and tag $\mathit{not}\ t$ a contradiction should be derived. Negative tags can be represented by extending the tuple representing the tagging with polarity information.

STRUCTURED TAGS Ordinary flat tags are limited in their expressiveness. To overcome this limitation, different extensions of tagging have been proposed, for example machine tags[2] and subtags [40] as used in the website RawSugar[3]. Most of the proposals employ a variation of annotations in the form of keyword-value pairs, sometimes extended to full RDF triples [373]. Note that keyword-value pairs can be seen as triples, too — the resource being annotated is the subject, the keyword is the predicate and the value is the object of the triple.

More complex schemes which involve nesting of elements might be practical in some cases, e.g. `hotel(stars(3))` could express that the tagged resource is a three-star hotel. These extensions develop the structure of the tag itself and a set of tags is interpreted as a conjunction. It is conceivable to allow users to tag resources with a disjunction of tags or even with arbitrary formulae. This may be practical for some applications but it has two drawbacks: Reasoning with disjunctive information is difficult and simplicity and intuitiveness would suffer.

2 http://tech.groups.yahoo.com/group/yws-flickr/message/2736
3 http://www.rawsugar.com/

Structured tags are used in KiWi's conceptual model to enhance the expressive power of tags and achieve an intermediate step between informal (atomic tags) and formal (RDF triples) annotation.

Two basic operations lie at the core of structured tagging: *grouping* and *characterization*.

The grouping operator, `()`, allows to relate several (complex or atomic) tags. For example, a wiki page might describe a meeting that took place in Warwick, UK on May 26, 2008, began at 8 am and involved a New York customer. Using atomic tags, this page can be tagged as `Warwick, New York, UK, May 26, 2008, 8am` leaving an observer in doubts whether "Warwick" refers to the city in Great Britain or to a town near New York. Grouping can be used in this case to make the tagging more precise: `(Warwick, UK), New York, (May 26, 2008, 8am)`. A group of tags can also be used to describe the properties of something whose name is not yet known or for which no name exists.

Characterization enables the classification or naming of a tag. The characterization operator, denoted `:`, can be used to make the tagging more precise. For example, if we wanted to tag a wiki page representing a meeting with the geo-location of the city Warwick, we could tag it as `(52.272135, -1.595764)` using the grouping operator. This, however, would not be sufficient as the group is unordered. Therefore we could use the characterization operator to specify which number refers to latitude and which to longitude, `(lat:52.272135, lon:-1.595764)`, and extend the structured tag to specify that the whole group refers to a geo-location: `geo:(lat:52.272135, lon:-1.595764)`. Similarly, `Warwick` in our example could be written as `location:(warwick)` to differentiate it from Warwick fabric or person with the last name Warwick.

Together, grouping and characterization provide a powerful tool for structuring and clarifying the meaning of tags. Structured tags are a step between informal simple tags and formal RDF annotations that allows users to assign freely chosen tags and vague annotations to content, but also gives them the opportunity to structure the annotations.

The meaning of a structured tagging is not pre-defined but rests, for the most part, on the user who specified it. Structured tags do not impose strict rules on their use or purpose, but merely provide the means to introduce structure into tag assignments. Structured tags can be seen as a wiki-like approach to annotation which enables a gradual, bottom-up refinement process during which meaning emerges as users' work and understanding develop.

Structured tags are governed by simple, minimal rules:

- Tag groupings
  - can consist of atomic and complex tags or a combination thereof,
  - are unordered,
  - cannot contain duplicate members, e.g. `(Bob, Bob, Anna)` and `((Bob, Anna), (Anna, Bob))` are not valid,
  - can contain arbitrarily but finitely many elements,
  - can be arbitrarily but finitely nested,
  - are identical in meaning to the constituent tag when they only contain one element, i.e. `(Anna)` is equivalent to `Anna`
- Tag Characterizations
  - can be used on atomic and complex tags,
  - are not commutative, i.e. `geo:x` is not the same as `x:geo`.
  - can use atomic and complex tags as a label

Using structured tags, the same information can be expressed in several different ways. The above described way of structuring tags describing a location is only one of many. Others include for example `geo:(x:y):(1,23:2,34)`, `geo:(y:x):(2,34:1,23)`, `geo:(1,23:2,34)`, and `geo:1,23:2,34`.

Different users and different communities might agree on different ways of structuring the same information and users are free to assign structured tags in a way that suits their needs and purpose the best. Of course, for structured tags to be useful in a community, users should agree on a common way of structuring information in tag assignments. We expect that such conventions for the usage of structured tags evolve over time as users collaboratively create and annotate content.

THE SEMIOTIC TRIANGLE One question to consider when designing a system that includes annotations is whether the annotations are seen to describe the data item or the concept described by it. A tag or other annotations added to a content item about a city could state a fact about the content item describing the city or about the city itself. Depending on which philosophy is adopted, different annotations can be appropriate, for example, a text but not a city can be well-written and, inversely, a city but not a text can have a certain number of inhabitants. The distinction is important because it may have consequences on how annotations are interpreted and treated. If the distinction is

not made, annotations can be ambiguous when they can apply either to the concept or to its representation in the wiki.

This problem is well known and has been addressed before in linguistics in the context of the semiotic triangle [285], Peirce's triad [302], and de Saussure's distinction between the signifier and the signified [317]. Oren [294] describes a system that lets users specify explicitly whether an annotation refers to a signifier, that is, the text, or its signified, the concept described by the text, through a syntax that supports the distinction of those two cases.

The KiWi system addresses the problem of multiple meanings of a tag label as described above. It, however, purposefully does not address the problem of what a tagging refers to —to a concept or the page that describes it. The reasons for this are that such a distinction may not be needed in practice, and might be irrelevant to most users. Above all, requiring users to be aware of the distinction and employ it during tagging reduces the user-friendliness of tagging and thus deters users from annotating content. Introducing such a distinction would therefore likely lead to an increase of the complexity of the tagging process. Where needed, structured tags can be used to specify the sense in which the annotation is intended.

TAGS VS LINKS When tag concepts are represented as content items, tag assignments to content items can be seen as links between the tagged resource and the content item representing the tag concept. Similarly, structured tags, such as keyword-value pairs, can be seen as expressing a relation (or a link), with its type given by the keyword, between the tagged resource and another resource, given by the value. In a wiki supporting semantic browsing over such tags, the question may then arise what differentiates a link from a (structured or atomic) tag. From a technical point of view, there may not be a strict differentiation and simple unstructured tags can be seen as specialized links between a taggable resource and the content item describing the tag concept, as a link is then simply a way of expressing a relation. The difference usually lies in the way links and tags are presented and used. Tags are usually represented separately from a content item, e.g. in a special area of the page, while links are represented with anchors inside the content item. Further, tags make a statement about a single content item, e.g. give it a type, whereas the purpose of links is to express an association between two content items. Finally, while links can be tagged, and tag concepts can be linked, it is not possible to link to or from a link.

TAG HIERARCHIES. Tag hierarchies constitute a step in the transition from informal to formal annotation. They are useful for example for reasoning and querying since they enable the

processing of tag relationships. Tag hierarchies could be created through assigning tags to tags, that is, tagging a tag's content item to indicate an "is-a" relationship.

Semi-formal annotations described in this section provide a means to transform knowledge from human-only content described in Section 5.1 to machine-processable information. Semi-formal annotations are an valuable feature of social software because they provide a low-barrier entry point for user participation in enrichment of content with machine processable annotations. Users can make use of gradually more expressive and formal methods of annotation as they become familiar with the system. First, they may only create and edit content and assign flat tags to content. When users become more familiar with the system and its content and want to make more expressive annotations, they may begin to use structured tags. Advanced users or system administrators can further enhance the annotations by specifying reasoning rules for semi-formal annotations [79] or translating structured tags into RDF.

5.3 SOCIAL CONTENT MANAGEMENT

To facilitate social collaboration and leverage the social aspects of a semantic wiki, several options and aspects may be considered.

USER GROUPS User groups can be leveraged, among other things, for the personalization of wiki content, for querying and reasoning and to attribute wiki data to a group of wiki users. Tags are a simple way to group things, making it an obvious idea to form user groups by tagging users' content items. Generally, every set of users that have been assigned the same tag could be considered a group. However, when there are many tags used in the system and special mechanisms are to be implemented for groups such as discussion boards, it may be more efficient and practical to use a mechanism to distinguish between user groups and all collections of users and documents tagged with the same tag. To distinguish between regular tag assignments and those that assign a user a group, groups could be created by assigning at structured tag with the label `group` to a content item, fragment or link, e.g. `group:(java)`.

ACCESS RIGHTS Users, user groups and rules for reasoning could be used to handle access rights in the wiki. Questions that arise include who owns the rules, what the access rights on rules are, and who can assign the tags that restrict the access. Static rules would not be suited for rights managements in all environments. For them to function well, the organization and

roles in the wiki have to be relatively stable, which may be the case in professional applications. In other areas, such as the development of open source software, such rules may not be desired or the social organization might not be static enough for such rules to be adequate.

THE SOCIAL WEIGHT OF TAGS When several users tag one item with the same tag, it may be useful to aggregate these tag assignments to give a clearer view of all the tags assigned and derive information about the popularity of the tag assignment. Tag assignments then can be seen to have weights depending on how often they were assigned. On the other hand, other users might not agree with the assignment of a certain tag to a content item, and add a negative component to the tags' weight to express this. The overall social weight of a tag could be calculated by assigning a value to both a tag assignment and disagreement with it and calculating the total score.

The social weight of a tag then summarizes the users' views on the appropriateness of a specific tag assignment and thus provides a valuable measure that can be used in reasoning and querying. Note that agreeing with a tag assignment is identical to assigning it but disagreement is not identical to adding the negative tag since not being content with the assignment of tag does not necessarily imply that one thinks its negation should be assigned. Negative tagging and disapproving with a tag assignment are not to be confused, the former explicitly expresses negative information, while the latter is merely a negative assessment of a tag assignment.

Further, the tag weight gives an overview over users' opinions and could help form a consensus on tag usage. Thom-Santelli et al. [347] show evidence that users have a desire for consistency with other users' tag assignments and are often willing to adopt a tag that they know has been used on the same resource by other users. Weighting tags could further foster this process.

Reinforcing a tag assignment or disagreeing with it further constitutes a low-barrier activity in the wiki, which might encourage beginning users to participate and express their opinion.

6

EXPERIMENTAL EVALUATION: STRUCTURED TAGS AND RDF

This chapter describes the design, execution, and evaluation of a user study that we performed to compare structured tags and RDF in practice.

Structured tags, as described in the previous chapter, are simple free-form tags extended by two mechanisms, grouping and characterization. Grouping is used to divide a set of tags into smaller units to indicate which tags belong together, while characterization is used to assign descriptive labels to a tag or a group of tags.

Structured tags are intended to serve as an annotation formalism that is more expressive than simple tags, but that at the same time is also more flexible and easier to use than RDF. As such, structured tags constitute a bottom-up approach to expressing structured annotations that supports and accommodates the evolution of knowledge: Structured tags make it easy for beginning users to introduce some structure into their annotations, thereby making them more expressive. At the same time, structured tags are well-suited to express information that is vague and still evolving.

The annotations created in these scenarios can then serve as the basis for further formalization, for example in the form of more complex structured tags, or RDF. This might happen for example when more information becomes available or when a user revises her annotations made previously.

Structured tags are flexible in their use, allowing users to apply grouping and characterization to simple and structured tags. The resulting annotation do not have a strict, pre-defined semantics, but, like simple free-form tags, their meaning arises from use. Additionally, structured tags can vary wildly in their complexity: In the most basic case, every single atomic tag can be seen as a structured tag consisting of a grouping with one element, but when grouping and characterizing are repeatedly applied, an intricate structure can arise.

Since the characteristics of structured tags are strongly determined by their usage, an experimental evaluation is warranted to provide insight into the respective roles of structured tags and RDF. As user-friendliness and user acceptance are crucial to social semantic web applications, the present study focuses on the user experience and aggregates participants' opinions on structured tags and their usability.

In addition, we present first findings on how structured tags are used in practice, how complex the structured tags are that users create and how suitable the formalism is for expressing evolving knowledge.

We consider structured tags to be not an alternative to RDF, but rather as a complement that helps bridge the gap between simple tags and RDF. For this reason, we evaluate RDF in the same manner as structured tags and compare the results.

The evolution of annotations from structured tags to RDF and the conversion between the two is another interesting topic worthy of investigation that is not treated in this first study of structured tags.

6.1 EXPERIMENTAL SETUP AND EXECUTION

Nineteen participants were recruited through announcements in several computer science lectures at LMU. They each were rewarded with 100 Euros for their participation in the study.

Participants' ages ranged from 21 to 26 with the average age being 23 years. All of the participants were students. Most studied computer science or media computer science at LMU, but some were students of related disciplines like mathematics, physics and computational linguistics. On average, participants had been students for 5.2 semesters with the individual durations ranging from two to ten semesters.

The study was performed in a single session that lasted about 8 hours and included a half-hour break at noon. Participants were split up into two groups that were balanced in terms of the fields of study and study progress of the group members. All materials that the participants were given were written in English, but comments could be given in English or German. At the start of the session, participants were asked to rate their experience with RDF and of tags on a scale from 1 ("never used/no knowledge") to 3 ("frequent use").

The participants were provided with a short introduction into the experiment scenario, reproduced here:

> You work in a software development company that runs several projects. When a project starts, information about it is added in the company wiki. The text there describes the project, its goals and projects and the people involved in it. Since several people collaborate to write the text and since more information becomes available as the project progresses, the text is changing and is also becoming more detailed. However, the company's wiki does not only contain text but also annotations that describe the content of the

> text, that is, it is a semantic wiki. These annotations are useful for example for retrieving wiki content and for automated reasoning, the deduction of new facts.

The first part of the experiment consisted of participants annotating different revisions of a text on project management in a software development scenario using RDF. Since structured tags are not yet implemented and available in practice, participants wrote down their annotations to the text on paper. The texts were written specifically for use in the study and represented the content of a wiki page describing a fictional software project. Two different texts of equal length with six revisions each were written, we will refer to them as "text A" and "text B" in the following. All revisions of both texts as they were presented to the participants are respectively given in Sections A.3 and A.4.

To provide participants with an introduction into the annotation formalisms, two texts of similar length describing structured tags and RDF were prepared. They are respectively given in Sections A.1 and A.2. To ensure that the texts were comparable in quality and intelligibility, they were examined by two computer science researchers at LMU who knew both formalisms and were not involved with the study. The description of RDF was simplified in that URIs and namespaces were omitted and capitalization and quotation marks were used to distinguish between classes, instances and literals. The introduction to structured tags did not mention negative tag assignments as described in Section 5.2.2.

Participants were provided with the introductory text on RDF and instructions for the experiment. Participants could refer to the introduction at any point, and were asked not to modify their annotations once they had received the following revision of the text. Changes in the text between revisions were highlighted. Participants were told to add as many annotations as they felt appropriate.The instructions further encouraged participants to refer to annotations made to previous revisions and to describe new annotations in terms of modifications to these annotations.

The annotation process was self-paced and the next revision of the text was only handed out once a participant had finished annotating the previous revision.

In this first part of the experiment, one group annotated text A, while the other group was given text B to annotate. Participants were asked to write down the times when they started and stopped annotating each of the revisions.

After they had completed annotating the final revision, participants were provided with questionnaires where they had to indicate their agreement with ten statements about the annotation formalism and their impressions of the annotation process. For each statement, participants were asked to tick one of five boxes corresponding to a Likert scale representing different extents of

agreement. The categories were "strongly disagree," "disagree," "undecided," "agree," "strongly agree."

In addition, participants could optionally provide written comments.

The second part of the experiment was executed in the same manner using structured tags instead of RDF as an annotation formalism. Additionally, the group that annotated text A with RDF was now given text B and vice versa.

When participants had completed both parts of the study, they were asked to fill out a final questionnaire describing which formalism they preferred and for which reasons.

6.2 RESULTS

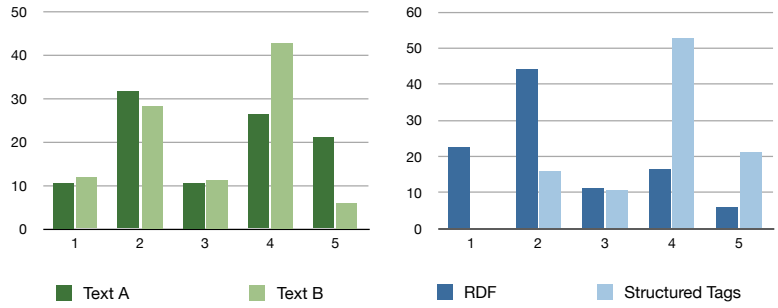
The analysis of participants' previous experience with RDF and tagging shows that out of the nineteen participants, only five had any knowledge of RDF before the experiment. All five rated their amount of experience with RDF as "a little" and no participants indicated that they used RDF frequently or knew it well.

The situation is similar for tags; all but six participants stated that they had never used tags. Out of these six participants, five indicated that they occasionally assigned tags to web content, and one participant declared that he often uses tags.

6.2.1 *User Judgments*

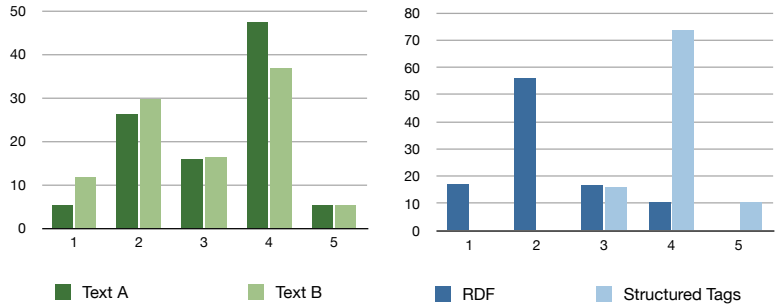
This section describes participants' level of agreement with ten statements about the annotation formalisms. For each statement, two figures are given, one showing the distribution of participants' answers ranging from 1 ("strongly disagree") to 5 ("strongly agree") grouped by the text type, and the second showing the same data grouped by the annotation formalism used. The former allows to determine for each statement whether there is a connection between the text, A or B, and the answer given. The latter shows how answers differ with the annotation formalism used. In addition, we performed Wilcoxon-Mann-Whitney tests to determine whether the differences are significant.

STATEMENT 1: *"After reading the introductory text, I felt I had understood how to use the annotation formalism"* This statement refers only to the introductory text and is independent of the annotated text. Figure 15a confirms that the answers are similarly distributed in both texts. The difference across texts was not significant (W=178, p=0.40), but a significant difference in reactions depending on the annotation formalism used was found (W=275, p=0.001).



(a) (b)

Figure 15: Percentages of participants' levels of agreement (1 to 5) with statement 1, grouped by (a) text and (b) annotation formalism



(a) (b)

Figure 16: Percentages of participants' levels of agreement (1 to 5) with statement 2, grouped by (a) text and (b) annotation formalism

Participants perceived the introduction on structured tags as being more helpful than that on RDF. Over seventy percent of participants agreed or agreed strongly with the statement with respect to structured tags, but about 65 percent of participants disagreed or disagreed strongly with the statement after they had read the introduction to RDF. In both cases, only about ten percent of participants were unsure whether they agreed with the statement.

Regardless of the annotation formalism, most participants' written comments state that more examples should have been included.

STATEMENT 2: *"The annotation formalism allowed to annotate the text in an intuitive way"* As before, the answers are similarly distributed across the texts (see Figure 16a), indicating that participants' opinions were independent of the text they had annotated. Indeed, the difference in answers between annotation formalisms ($W=317.5$, $p<0.001$) but not that between texts ($W=188$, $p=0.59$) was found to be significant.

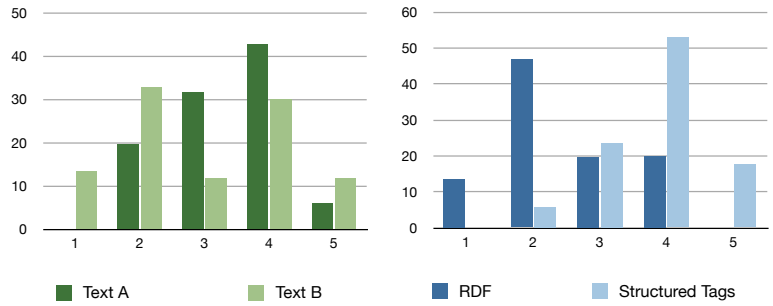


Figure 17: Percentages of participants' levels of agreement (1 to 5) with statement 3, grouped by (a) text and (b) annotation formalism

More than eighty percent of the participants found structured tags to be intuitive ("agree" or "strong agree," see Figure 16b), and a small number was undecided. No participant found structured tags to be unintuitive. The situation is reversed for RDF which almost three quarters of participants found unintuitive. Only about ten percent of participants agreed with the statement with respect to RDF.

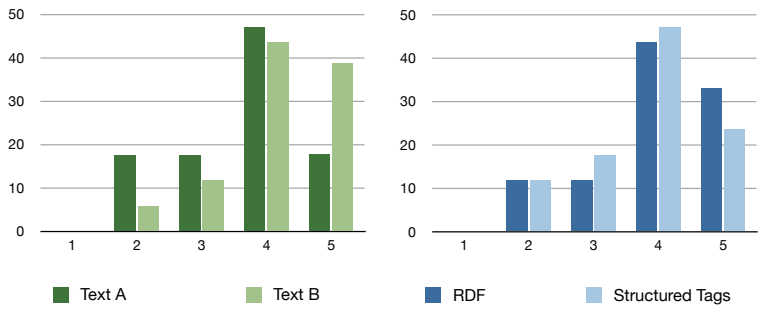
Many participants in their comments criticized what they considered to be limitations of RDF, for example that annotations always take the shape of triples, that a clear idea of the domain is needed to formalize the concepts and relations, and that the content of long and complex sentences is hard to express as RDF.

Comments about the intuitiveness of structured tags were mostly positive, although here, too, one participant found it hard to express long sentences as structured tags.

STATEMENT 3: *"The annotation formalism allowed to annotate the text in a convenient way"* The reactions to the third statement differ slightly more between texts than those to the two statements before, but overall are comparable (see Figure 17a). Again, the difference between the answers given by participants using different annotation formalisms ($W=43.5$, $p<0.001$) but not that between participants annotating different texts ($W=158.5$, $p=0.39$) is significant.

Again, with respect to structured tags, most participants agree with the statement, while the majority of participants using RDF disagrees with it (see Figure 17b). While the difference between the two formalisms is considerable, the reactions are less divided across formalisms than those to the previous statement.

Only few participants added further comments about this statement. One participant found writing RDF triples repetitive, while another remarked that structured tags often need to be rearranged when new information is provided.



(a) (b)

Figure 18: Percentages of participants' levels of agreement (1 to 5) with statement 4, grouped by (a) text and (b) annotation formalism

STATEMENT 4: *"I feel that way to represent negative facts is missing from the formalism"* The reactions to this statements are similar across texts, although participants annotating text B slightly more frequently agreed with the statement than those annotating text A (see Figure 18a). However, the difference between texts is not significant (W=136.5, p=1), while that between annotation formalisms is (W=85, p=0.05).

There is little difference in the reactions across formalisms (see Figure 18b). Overall, more participants feel that RDF is missing a way to represent negative facts, but only by a small margin. For both formalisms, more than 75% of participants agree or agree strongly with the statement and no participants disagrees strongly with it.

In the comments, one participant remarked that he considered a way to express conditions and consequences to be of greater importance than support for negation.

STATEMENT 5: *"The annotation formalism was expressive enough to let me annotate the text the way I wanted"* Here, reactions differ across texts, but less strongly than the reactions across formalisms (see Figures 19a and 19b) and, again, the difference between annotation formalisms (W=99.5, p=0.26) but not that between texts (W=48.5, p=0.001) is significant.

When using structured tags, over eighty percent of participants found structured tags expressive and agreed or strongly agreed with the statement. The levels of agreement are more varied when participants use RDF with roughly equal proportions answering "disagree," "undecided" and "agree."

STATEMENT 6: *"I feel confident about the formal correctness of the annotations I made"* Participants' answers are very similar across texts (see Figure 20a). As before, the difference in answers be-

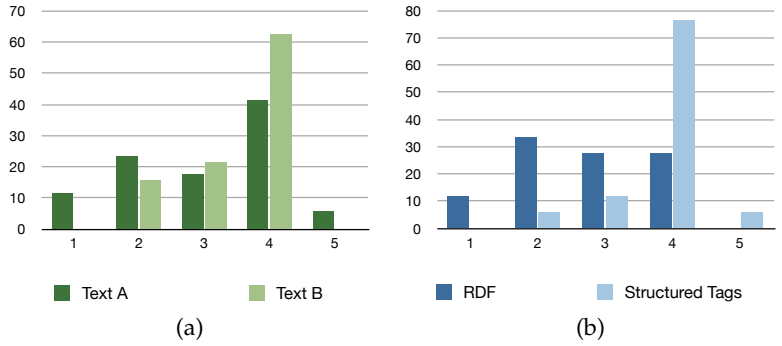


Figure 19: Percentages of participants' levels of agreement (1 to 5) with statement 5, grouped by (a) text and (b) annotation formalism

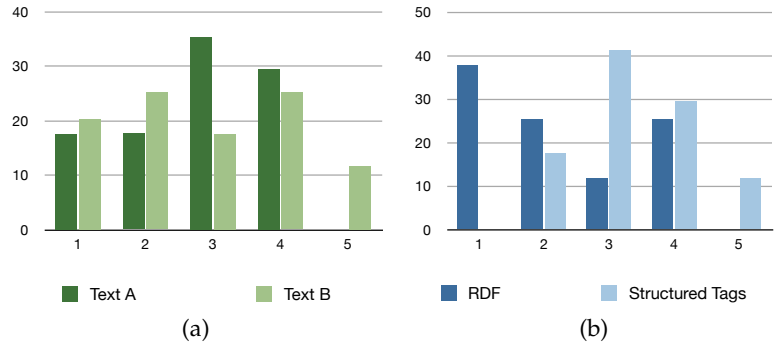
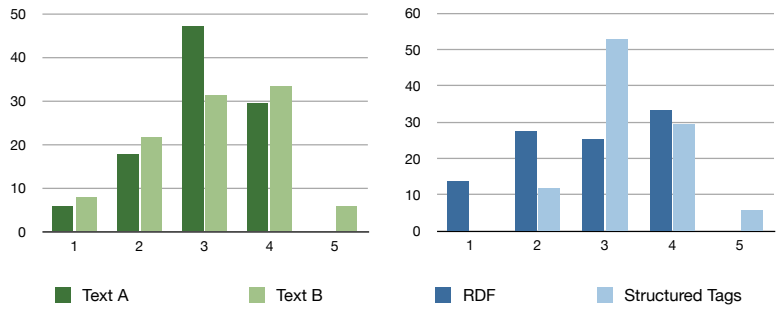


Figure 20: Percentages of participants' levels of agreement (1 to 5) with statement 6, grouped by (a) text and (b) annotation formalism

tween texts is not significant (W=130.5, p=0.85) but that between annotation formalisms is (W=69, p=0.014).

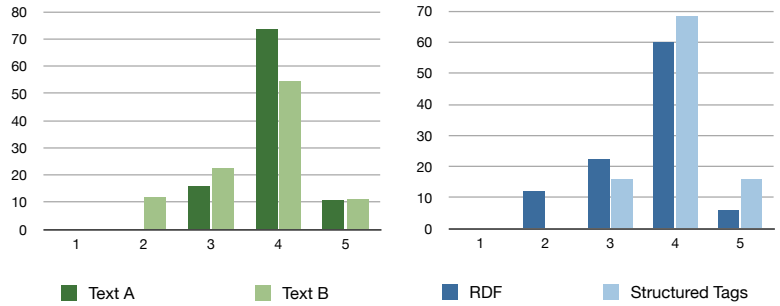
More than sixty percent of participants are not confident about the correctness of their RDF annotations ("disagree" or "strongly disagree," see Figure 20b), only few are undecided and about 25 percent agree with the statement. The situation is different for structured tags where more than forty percent of participants are undecided, that is, are not sure whether their annotations were formally correct. Another forty percent of participants agrees or strongly agrees with the statement and only about a fifth disagrees with it.

STATEMENT 7: *"I feel confident about the appropriateness of the annotations I made"* Again, the distributions of reactions across texts, while not identical, are highly similar and the difference between texts is smaller than that between formalisms (see Figures 21a and 21b). Neither the difference between texts (W=117.5, p=0.70) nor annotation formalisms (W=97.5, p=0.24) is significant.



(a) (b)

Figure 21: Percentages of participants' levels of agreement (1 to 5) with statement 7, grouped by (a) text and (b) annotation formalism



(a) (b)

Figure 22: Percentages of participants' levels of agreement (1 to 5) with statement 8, grouped by (a) text and (b) annotation formalism

The reactions of participants using RDF are distributed in comparable proportions over "disagree," "undecided," and "agree." A smaller number of participants strongly disagreed with the statement. With respect to structured tags, the reactions are very different with more than half being undecided, a small number disagreeing and about a third agreeing or agreeing strongly.

STATEMENT 8: *"I feel that the annotations I made convey the important aspects of the text"* For this statement, again, the distributions of the different levels of agreement do not greatly differ across texts or annotation formalisms (see Figures 22a and 22b), and the answer distributions do not differ significantly between texts (W=200, p=0.30) or annotation formalisms (W=128, p=0.13).

The majority of participants agrees or agrees strongly, while a minority of about twenty percent is uncertain. In the case of RDF, about ten percent of participants disagree with the statement.

Several participants in both groups wrote in their comments that they felt they might have annotated too much and expressed too many unimportant details in their annotations.

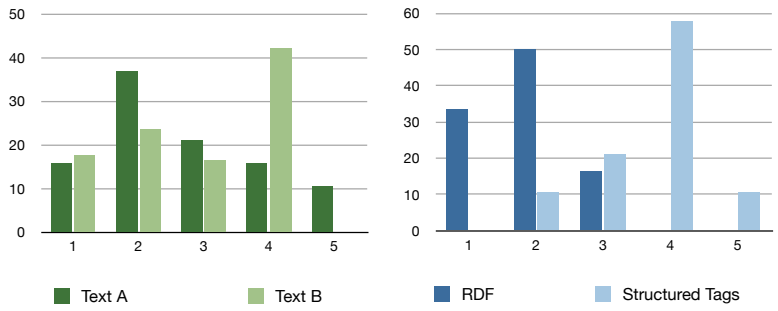


Figure 23: Percentages of participants' levels of agreement (1 to 5) with statement 9, grouped by (a) text and (b) annotation formalism

STATEMENT 9: *"I enjoyed using the formalism"* Here, a small difference in reactions between the texts can be observed (see Figure 23a). The difference in reactions between texts is not significant ($W=148$, $p=0.48$), but that between annotation formalisms is highly significant ($W=28.5$, $p<0.001$).

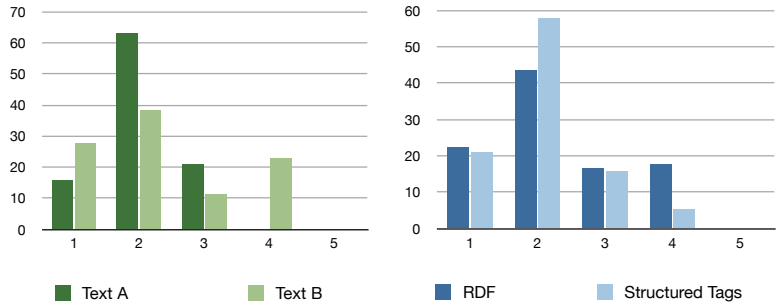
About 24 percent of participants annotating text A and 37 percent of the participants annotating text B disagree with the statement. Inversely, 16 percent of the participants annotating text A and 42 percent of participants annotating text B agree with the statement. However, ten percent of the participants annotating text A but no participants annotating text B agree strongly and the difference between texts is smaller than that between formalisms.

The difference in the distributions of levels of agreement between formalisms is very high for this statement (see Figure 23b). Over eighty percent of participants do not enjoy using RDF ("disagree" and "disagree strongly"), and the rest are undecided. No participant indicated that he enjoyed using RDF. The case is reversed for structured tags with over two thirds stating that they enjoyed using structured tags. About twenty percent are undecided and ten percent disagree with the statement.

Several participants commented that they did not enjoy using RDF since they found it too difficult, did not feel that they fully understood the formalism or because they found the process too laborious.

STATEMENT 10: *"Given more time, I would have liked to add more annotations"* Grouped by text, the levels of agreement show a difference in that the reactions of participants annotating text A overall are more negative than those of participants annotating text B (see Figure 24a).

A similar difference can be found for the reactions grouped by formalism where more participants using RDF would have liked to add more annotations (see Figure 24b). Overall, the differences



(a) (b)

Figure 24: Percentages of participants' levels of agreement (1 to 5) with statement 10, grouped by (a) text and (b) annotation formalism

found for this statement are minor compared to the differences in reactions to the other statements and neither difference is significant (text:W=173.5, p=0.95; annotation formalism: W=107, p=0.61).

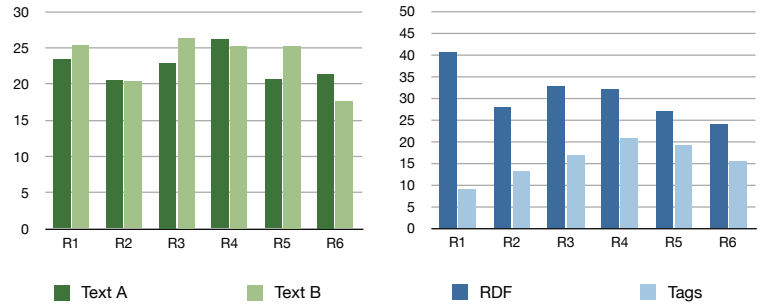
In the final questionnaire, eighteen of the nineteen participants stated that they preferred structured tags to RDF. Frequently given reasons given for this choice are that structured tags are more flexible and intuitive, easier to understand and use and quicker to write.

Several participants stated that they thought that structured tags were expressive and could be updated easily, making it possible to start annotating without having a clear idea of all concepts in the domain. The latter point was a major point of criticism with respect to RDF where many participants expressed opinions similar to the following: "[W]ithout prior knowledge of the subjects to be annotated, it is quite hard to define a structure that is compact, yet flexible enough to be used in future edits."

One participant suspected that users would be more willing to write annotations in the form of structured tags than RDF, while another remarked that, when the people annotating the content are paid, RDF is better because of it can be used for reasoning, but that structured tags are better for encouraging casual users to add annotations.

While most participants were more in favor of structured tags, several participants commented on the advantages of RDF. A number of participants found that when the domain and its concepts are known, RDF should be used since it allows for a cleaner formalization that is better suited for, for example, reasoning.

One participant remarked that when several people work on one annotation, RDF is preferable to structured tags due to its



(a) (b)

Figure 25: Average time taken to annotate the different revisions of the text (in minutes), grouped by (a) text and (b) annotation formalism

more rigid, pre-defined semantics. Another participant wrote that "RDF" delivers a more "clean" feeling, whereas [structured tags] feel somewhat "quick & dirty."

Some participants criticized both formalisms because they felt that uncertainty, negation, temporal information and complex relationships could not easily be expressed.

6.2.2 *Time Requirements for annotations*

The average times needed to annotate each revision, grouped by text and formalism, are shown in Figures 25a and 25b.

On average, participants spent 22.5 minutes annotating each revision of text A and 23.4 minutes each revision of text B. The differences in the time spent annotating per revision between texts are minor and the distributions of time spent over the revisions of each text resemble each other.

Annotating one revision of a text with RDF on average took participants 30.7 minutes, almost double the average time needed to annotate a revision with structured tags, 15.7 minutes.

Not only the average amount of time, but also the differences in time spent annotating across revisions differ.

Participants assigning RDF annotations spent 40.6 minutes on the first revision, more than on any other revision. At 28 minutes, the second revision was annotated comparatively quickly. The time spent annotating revisions three and four is higher and almost identical at 32.7 and 32 minutes respectively, and declines for the next and final two revisions (26.9 and 24 minutes).

When structured tags were used, the first revision took the shortest amount of time to annotate, 9 minutes. From there, the time taken gradually increases; revision 4 on average was annotated within 20.8 minutes, the highest amount of time across

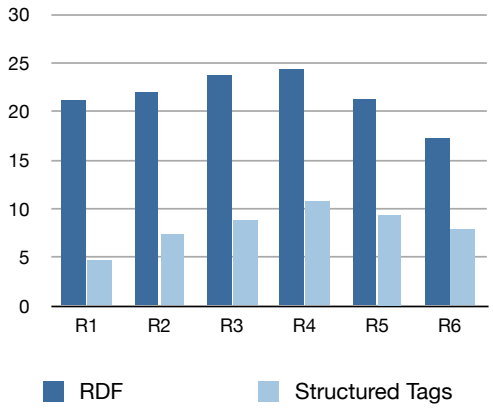


Figure 26: Average number of annotations per text revision (1 to 6), grouped by annotation formalism

a revisions. The average time then declines to 19.1 minutes for the final revision.

6.2.3 *Analysis of the Annotations*

On average, each participant wrote 178.53 annotations, 14.88 for each individual revision. 27.36 percent of those annotation are structured tags and 72.64 percent are RDF triples. 34.05 percent of the structured tags and 10.19 percent of the RDF triples were marked by participants as changes to earlier annotations.

Figure 26 shows the average number of annotations added per revision for both formalisms. In both cases, the number of annotations increases from the first to the fourth revision and then decreases for the last two revisions.

The initial increase in the number of annotations added is proportionally lower for the RDF annotations. Overall, considerably more RDF triples than structured tags were assigned; the number of RDF triples assigned is between two and four times higher than that of structured tags.

While RDF triples have a fixed number of elements, structured tags can be made up of an arbitrary number of simple tags. This means that comparing the overall number of annotations is not a sufficient metric to compare how much information is expressed through the annotations in each formalism. Figure 27 displays the average number of elements, that is, subjects, predicates, and objects, and simple tags, that were added to each revision of the text. For the first three revisions, the number of RDF elements is higher by a factor of 2 to 1.5, but for revisions 4 to 6, the number of RDF elements assigned is only about 10 percent higher than that of structured tags elements.

Note that this statistic does not fully capture the differences in the amount of information contained in the annotations either:

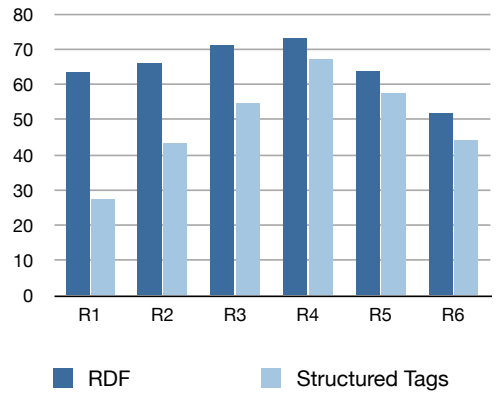


Figure 27: Average number of elements in annotations added per text revision (1 to 6), grouped by annotation formalism

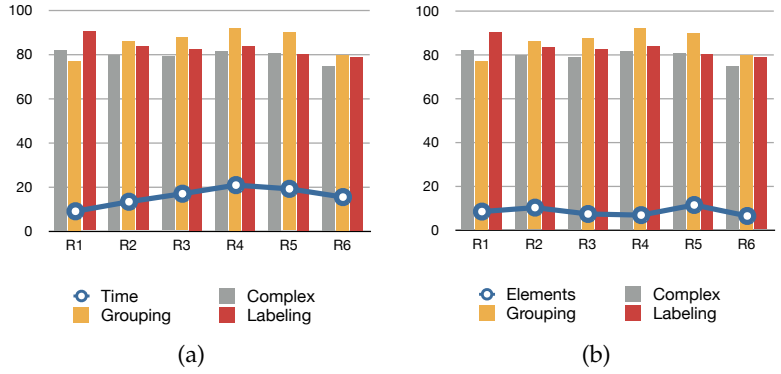


Figure 28: Relationship between tag complexity and time spent annotating ((a)) and number of total elements in the added structured tags ((b)), per revision

While an RDF triple describes the relationship between a subject and a predicate, structured tags use labels and the grouping operator to describe the relationships between tags. Since structured tags can be characterized and grouping can be applied to an arbitrary number of simple or structured tags, the number of elements alone is not sufficient to determine how much information is expressed in a structured text in terms of the relations between elements. However, since the nature of relations in RDF and structured tags differ greatly, this factor cannot be easily quantified and we use the number of elements as an approximation of the information content of an annotation.

The average number of constituent simple tags in each structured tag is 8.35 across all revisions.

Grouping and characterization are used to comparable extents and often in combination; structured tags use grouping at least once in 85.39%, and characterization in 83.20% of all annotations.

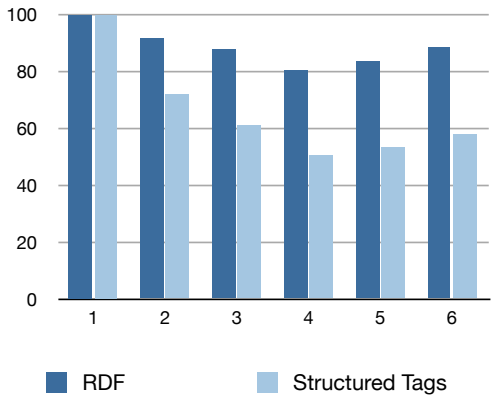


Figure 29: Average percentage of annotations not based on previous annotations, per revision

The number of complex structured tags, that is, structured tags that contain at least one occurrence of labeling or characterizing a tag that itself uses one the operations, is high, 79.62%. In most cases, it is the elements of a grouping or the tag being characterized that are complex, and 96.13 percent of labels are simple tags.

As figure 28 shows, the percentages of structured tags that use grouping, characterization or that are complex are not clearly related to the average time spent annotating or the average number of elements per structured tag.

When using structured tags, participants more frequently indicated that a newly added annotation was based on an existing annotation that it replaces (see Figure 29). Since participants wrote their annotations on paper, they had to manually specify that a new annotation was to be understood as a change to a previous annotation. It is likely that participants did not remember to add this note in all cases and consequently, the numbers shown in Figure 29 should be understood as a low estimate.

6.3 DISCUSSION

The evaluation of the results shows that participants find structured tags convenient and comfortable to use but at the same time expressive.

The short introduction given was enough for most participants to understand how to use structured tags and a majority of participants considers structured tags to be intuitive, convenient, and flexible. Participants were also more confident in the correctness of their annotations when using structured tags and found using structured tags more enjoyable than using RDF. Further, participants represented similar amounts of information in structured

tags and RDF, but needed substantially less time when using structured tags.

Due to the experimental setup, we can not fully exclude the possibility that the order in which the formalisms where used had an effect on the outcome of the experiment. For example, it is possible that the smaller amount of time needed for annotating the texts with structured tags is partially due to participants' experiences in creating the RDF annotations. If this practice would significantly decrease the time needed to write annotations, it could be expected that the later revisions of the text annotated with RDF are annotated quicker than the earlier ones. However, this is not what we observed and with the exception of the very first text that was annotated, the relative amounts of time spent on each revisions are proportional between formalisms. This might indicate that either some learning took place at the very beginning of the experiment, or that RDF is initially harder to understand or use than structured tags.

More structured tags than RDF triples were identified as being based on annotations to prior revisions. Additionally, several participants remarked that, unlike structured tags, RDF requires to have a clear idea of the conceptual domain before beginning to annotate. Together, these findings indicate that structured tags are seen as more suited for expressing evolving knowledge than RDF is.

At the same time, several users find that, when it is used in an appropriate environment where the domain concepts are known and where the annotations are assigned by experienced users, RDF has advantages over structured tags.

Overall, the experimental evaluation thus indicates that structured tags succeed at realizing a way to assign semi-formal annotations that are more expressive than simple tags but easier to use, especially for expressing evolving knowledge, than RDF.

A high number of the structured tags that participants created are complex and across revisions, no evolution from basic to more strongly structured tags could be observed. Similarly, the proportions to which the two operations, grouping and characterization, were used were roughly equal and did not change across revisions.

One reason for this could be that the conditions of the experiment were not realistic in that the experiment lasted several hours which were explicitly dedicated to creating annotations. Another way in which the annotations assigned likely differ from those used in practice is that they merely describe but do not enhance the content given in the text, for example with additional information, or with subjective opinions and assessments.

While this experimental evaluation is concerned mainly with the participants' impressions and acceptance of structured tags,

a study that focuses on the evolution of annotations, both from simple to structured tags and from structured tags to RDF, should consider this aspect in its experimental design.

Part III

KWQL

7

KWQL: DESIGN AND MODEL

KWQL, pronounced "quickel," is a rule-based query language that combines the characteristics of keyword search with those of web querying in order to enable versatile querying in the KiWi wiki. The language allows for rich combined queries of textual content, metadata, document structure, and informal to formal semantic annotations.

KWQL queries range from elementary and relatively unspecific to complex and fully specified (meta-)data selections. In keeping with the *wiki way* [240], KWQL has a low entry barrier, allowing casual users to easily locate and retrieve relevant data, while letting advanced users take advantage of its full power.

KWQL is flexible with respect to the amount of experience of the user and the specificity of the queries: semantic wikis, like social semantic software in general, live from user participation. However, the assumption that users are familiar with web query languages severely limits the potential user base of such an application. Therefore, a query mode that is accessible and immediately useful for beginning users should be provided.

Despite efforts to automatically infer users' intents from keyword queries (see Section 4.3), such queries alone do not suffice to convey complex selection criteria. Consequently, and at the inevitable cost of a less accessible query formalism, experienced users should be able to formulate their queries in a language that allows for highly specific and expressive queries.

The two requirements, simple and accessible querying for beginning users and complex selections for experts who value expressiveness over simplicity, could be realized by providing two different query languages, for example full text keyword search and XQuery. No current semantic wiki provides a web query language that can query over content, annotation as well as structure or a keyword query language that is not purely matched either on content or annotations (see Section 3.4). However, even if we ignore this limitation and assume that we integrate data-versatile keyword and web query languages in our wiki, this approach has one major disadvantage: it leaves no room for a middle ground, meaning that beginning users either have to learn XQuery or be confined to keyword search. Intermediate users may want to formulate queries that are slightly more advanced than bag-of-words keyword queries. When two languages are provided that differ greatly in the knowledge required to use them, these users cannot formulate their query without first

learning a good portion of the more expressive language, even though their query intent may only slightly exceed the expressive power of the simple one. Providing keyword search and a web query language may accommodate both novices and expert users, but not the group of users in between, which, as the system is used over time, is likely to constitute a significant portion of the user base.

One of the key principles of KWQL is that the complexity of queries increases with their expressiveness, enabling a gradual learning of the language where required. Beginning users can immediately profit from using KWQL by posing basic keyword queries. As users learn more about the system and the data contained in it, their information needs might begin to become more complex. KWQL allows users to learn the advanced features of the language bit by bit as required to realize their query intents.

KWQL does not require a specific amount of learning from the user—it is likely that some users will never venture past basic keyword queries, while others may only learn to use some slightly advanced constructs but not the full language. A third group may invest more time, study the full syntax, and use it to write complex rules. The goal for KWQL is to equally accommodate all of these users, letting them use as much or as little of the language as suits their needs.

KWQL queries may be vague and amount to simple full text search, or take the shape of selections of individual data items using precise constraints. The language is designed to support both types of queries, one similar in functionality to web search, the other similar to web querying, as well as the range of queries in between.

Since queries imposing more complex selection constraints also require more knowledge of the language, the issue of the precision of a query is related to that of the user's knowledge of the language. Beginning users who are not yet familiar with the KiWi wiki and the data contained in it will likely have relatively unspecific information needs, such as finding all content items related to a specific topic. Queries of this type can be realized easily without knowledge of KWQL's syntax. As the users learn more about the types of data that exist in the wiki and how these are arranged, they might want to use targeted queries to satisfy their more specific information needs. For this, they are required to learn some of KWQL's syntax and semantics, an investment which pays off immediately since they can now formulate a query which yields the desired results. Learning to use KWQL therefore is a gradual process driven by users' information needs. Formulating vaguer queries which impose less specific constraints generally requires less effort and knowledge of the language than composing more specific queries.

To put it shortly and succinctly, KWQL provides a means to formulate queries that range from easy and vague to complex and expressive, allowing users to employ the language according to their knowledge and information needs.

7.1 A HIGH LEVEL LOOK AT KWQL

In this section we give an overview over KWQL and its features. Our goal is to convey the design and characteristics of KWQL at an abstract level, without distracting the reader by delving into the syntax and giving concrete examples or tying the principles of the language too closely to their realization in KWQL.

While KWQL was developed and implemented in the context of the semantic wiki KiWi, the ideas and goals behind its design address issues that are universal in querying social semantic web applications. As such, the concepts of KWQL could be transferred to derive similar languages targeting other social semantic applications, and we consider KWQL to be exemplary of a novel family of query languages.

Full KWQL *rules* consist of a query *body* which specifies the data to be selected, and an optional *head* indicating how this data should be processed further. This strict separation between data selection and the construction of new data, not commonly found in web query languages [318], aims at providing conceptual clarity and reflects the fact that selecting data and then processing the selected data are two separate, consecutive tasks.

Query bodies can express selections of varying levels of complexity using any combination of data sources in the wiki. For example, a content item selection can refer not only to the textual content of the content item to be selected, but also to the structuring of its contained content items, to the links from or to the content item, and to its annotations. In short, KWQL is fully aware of the underlying conceptual model.

To improve the user experience, and simplify the mental transfer of the query intent into a query, query bodies take the shape of abstracted descriptions of the data to be matched. This query-by-example-like syntactic style is further substantiated by the fact that KWQL query terms are *injective*, meaning that no two query terms may match the same data item. For example, when a query body describes a content item with two tags, one with name "wiki" and one created by the user Mary, the query will retrieve only content items where the two conditions hold for two distinct tags, but not those where a single tag satisfies both criteria but no other tag meets either of them. Apart from enhancing the expressive power of KWQL, injectivity also more tightly couples the user experience—what the user sees and perceives when he uses the wiki—to the way in which queries are expressed in KWQL.

Each query body evaluates to a set of content items, namely those that are compatible with the given description. *Compatibility* here means that the content item has all the properties specified in the query body, for example that it contains the term "KiWi" but not the term "search" in its text, has a tag whose name contains "wiki" and in addition links to a content item with "KWQL" in the title. Query bodies are taken to be partial specifications, meaning that a content item that satisfies the selection criteria may in addition have any number of other properties that are not in contradiction with the selection criteria.

The key property of KWQL, scaling with user experience and the specificity of the query intent, is realized through far-reaching and comprehensive support for the under-specification of queries. The simplest—and at the same time most vague—description of content items to be matched consists of one or several keywords that the content items must contain. When the context in which the keywords may occur is not restricted further, all content items that contain the given keywords in their text, title, fragments, links, tags, or associated metadata—but not in linked or nested content items—are compatible with the query and returned as results. Basic keyword queries in KWQL therefore constitute a true full text search over all parts of the individual content items.

To make queries more selective and precise, the *structural context* in which the keywords should occur can be specified fully or in part. In addition to conjunction, which is implicitly assumed when no operator is given, operators for disjunction and negation may be used. KWQL bodies thus amount to descriptions of the data to be retrieved that, depending on the users' knowledge and information need, can be more or less specific.

This approach lends itself particularly well to stepwise querying, the gradual refinement of queries: starting with explorative queries using a small set of keywords, users can go through several iterations of evaluating a query, examining the results, and then further substantiating the query until the desired information is found.

KWQL queries are monotonic, that is, they become more selective as conjunctive selection conditions are added. In the above example, the answers returned by a query for content items with "KiWi" in the text are a superset of the answers returned by a query for content items that contain "KiWi" but not "search." The same is not true for the addition of disjunctions and negations: the negation of a query returns the complement of the original set of answers, while the addition of a disjunctive selection criterion means that the union of the matches of each operand query term is returned.

Query bodies may also contain variables. In the query evaluation process, these are bound to specific values of the matching

content items, for example their authors or the titles of the content items that they link to.

KWQL is not restricted to data *selection* but also offers *construction*, the reshaping of the selected data into new data, database-like views. Such views constitute a simple yet remarkably powerful form of reasoning. KWQL does not aim to be a full reasoning language, but instead limits itself to this simple form of content-based reasoning. New content items are created in the rule head from the variable bindings obtained by evaluating the body. Construction enables users to create new data through regrouping, reformatting, or aggregation of the selected data, or through the inclusion of data in a different context. The syntax used to specify the content items to be constructed resembles that used to describe which content items should be selected, with the difference that query heads must always specify the full context of a value.

Construction can for example be used to create a table of contents of the wiki, by first binding the titles and authors to variables and then specifying a new content item which lists all titles together with their authors, or vice versa. Using aggregation, the table of contents could then be extended to show the number of authors per content item, or the number of content items each author has edited.

The data queried and the constructed query results adhere to the same data model [320], meaning that query answers are amenable to further queries. This is desirable for two reasons. First, no additional concepts are required to represent query answers, which maintains the consistency and simplicity of the conceptual model. Second, *answer-closedness* means that rule chaining can be realized naturally without transformations between different data formats.

Unlike the body, the head of a KWQL rule is optional. When no query head is given, the set of content items matching the body is returned and presented to the user as is. By default, content items created through construction are automatically persisted in the KiWi database, while the lists of matching content items returned when no head is specified are not saved. However, the KWQL interface allows for the easy deletion of the newly created content items or the storage of a result list.

KWQL supports two types of queries: regular queries, evaluated only once, and embedded queries. Embedded queries are part of a content item and are evaluated every time the content is loaded. They enable pre-defined views that always display the latest information without need for manual updating.

Content items created through construction in regular queries can be further edited by the user, while the content items displayed as the result of the evaluation of an embedded query are virtual and thus cannot be changed manually.

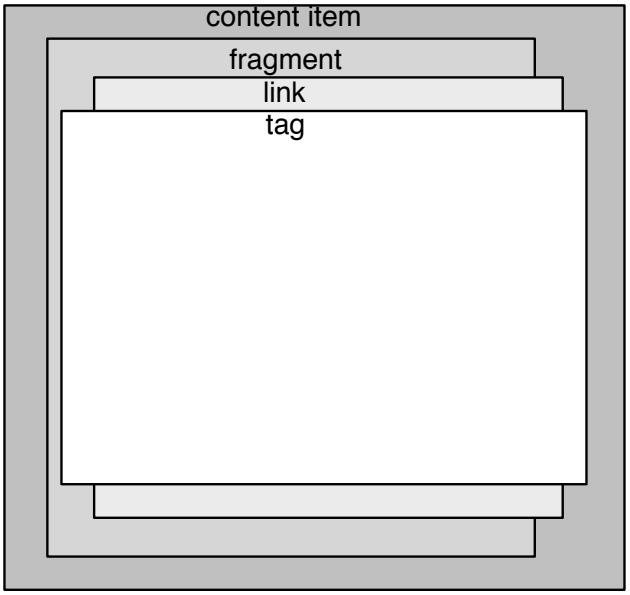


Figure 30: Resources and allowed sub-resources

7.2 KWQL SYNTAX

Having reviewed the general principles behind KWQL, we now introduce the syntax used to express KWQL selections and constructions and give various examples of KWQL queries.

7.2.1 *Data Model*

KWQL queries take an object-oriented view and describe the data to be selected or constructed in terms of the four elements of the conceptual model (see Chapter 5), their properties, and the relations between them. The elements of the conceptual model—content items, fragments, links, and annotations—will in the following be referred to as *resources*.

When we speak of resources of type "tag," we mean not tag labels or concepts, but instead individual tag assignments. KWQL as described here supports only the querying of (structured and unstructured) tags, but not of RDF or annotations of other types. Three different approaches to extending KWQL to the querying of RDF annotations are described and discussed in Chapter 13.

Resources are complex data items. They have a number of pre-defined properties depending on their type and can stand in a containment relationship to resources of other types. For example, a content item has a title and at least one author and may contain a fragment, which in turn contains (that is, has been assigned) a tag.

Resources contained in resources of other types will in the following be called *sub-resources*. The allowed containment relationships in the KiWi wiki are illustrated in Figure 30. Wherever two rectangles in the figure share a border, the resource represented by the darker rectangle may contain the resource represented by the lighter rectangle. The containment relationship is transitive. Content items can be assigned tags or contain links or fragments in their text. Fragments can contain links and tags, links can contain tags, and tags cannot contain any sub-resources. A resource may contain several sub-resources of the same type: a content item, fragment or link can be assigned several tags, several fragments can be defined in one and the same content item, and the text of a content item or fragment can contain several links.

The different types of properties of a resource are called *qualifiers*. Each qualifier is associated with a *qualifier value*, together they form a *qualifier term*. Similar in form to label-keyword terms, qualifier terms represent the content and metadata of a resource, but also express linking and nesting relationships. Depending on the type of the qualifier, qualifier terms have different arities indicating how often a qualifier term of this type may occur within a particular resource. For example, every content item has exactly one title and one URI, at least one author, and any number of nested content items.

Qualifier values, that is, the content associated with qualifiers, are of different types depending on the type of the qualifier. Qualifiers referring to data and metadata are associated with data in the form of dates, integers, URIs or text. Structural qualifiers on the other hand describe nesting and linking relationships among pairs of content items or fragments. When used in a query, they take as a value a subquery describing the linked or nested resource.

Some qualifiers can occur within resources of several different types. This for example is the case for the **`author`** qualifier, since both content items and tags have authors. Qualifier terms of a specific type typically maintain their arity and the data type of their value regardless of the type of resource they appear in. The only exception to this rule are the qualifiers **`child`** and **`descendant`**, which refer to a fragment when used within a fragment and a content item when used within a content item. Table 3 lists all qualifier types together with the resources in which they can appear, the data type of their value, and the arity of the qualifier term. $*$ and + here are used as in regular expressions, indicating that the qualifier can appear any number of times ($*$), or arbitrarily often but at least once (+).

Values, qualifiers, and resources specify the content items to be selected or created.

Qualifier	Resource Type(s)	Value Type	Arity
	data		
title	content item	string	1
text	content item fragment	string	1
anchorText	link	string	1
name	tag	string	1
	metadata		
URI	content item fragment tag	URI	1
author	content item fragment tag	string	+
created	content item fragment tag	date	1
lastEdited	content item	date	1
numberEdits	content item	integer	1
	structure		
child	content item	content item	*
child	fragment	fragment	*
descendant	content item	content item	*
descendant	fragment	fragment	*
target	link	content item	1

Table 3: KWQL qualifier types

Non-structural qualifiers and sub-resources describe the *intra-content item* structure. Structural qualifiers impose constraints on the *inter-content item* structure, and the *inter-fragment* structure in the case of **`child`** and **`descendant`**.

7.2.2 *KWQL Terms*

With the data model firmly in place, we can now direct our attention to the building blocks of KWQL queries.

OPERATORS The Boolean operators **`NOT`**, **`AND`**, and **`OR`** are used to represent classical negation, conjunction, and inclusive disjunction. When query terms are concatenated but no operator is given, as a default, conjunction is assumed. The precedence of negation is higher than that of conjunction and disjunction, which are evaluated from left to right. Moreover, bracketing may be used to indicate precedence.

VALUE TERMS An atomic *value term* consists of a single keyword or several terms enclosed in quotation marks. Atomic and non-atomic value terms can be combined using the Boolean operators. KWQL does not currently support wildcards or regular expressions as values, however, support for both could easily be added. The following are examples of value terms:

```
KiWi
```

```
NOT(KiWi AND KWQL)
```

```
NOT(KiWi) AND KWQL
```

```
NOT(KiWi) KWQL
```

```
NOT(KiWi) AND (KWQL OR search)
```

QUALIFIER TERMS An atomic *qualifier term* consists of a qualifier and a value term. Atomic and non-atomic qualifier terms can again be combined using the Boolean operators. The following are examples of qualifier terms:

```
text:KiWi
```

```
created:2010
```

```
text:NOT(KiWi AND KWQL) OR title:search
```

RESOURCE TERMS A *resource term* consists of a resource of type r_1 together with its value. The value of a resource term must in turn consist of at least one qualifier term, or of another resource term where the resource type r2 of the inner resource is a valid sub-resource of r_1. The value can also be a combination of qualifier or sub-resource terms using the Boolean operators. Resource terms of type content item can be negated and be used as operands in disjunctions, but not in conjunctions. The following are examples of resource terms. The resource type content item is abbreviated as "ci" in KWQL.

```
tag(name:KiWi)
```

```
fragment(tag(name:KiWi))
```

```
ci(text:NOT(KiWi AND KWQL) OR title:search)
```

CONTEXT By the *context* of a value term we understand its enclosing resource(s) and associated qualifier, that is, its location in the described content item.

7.2.3 *KWQL Bodies*

FULLY SPECIFIED QUERY BODIES A fully specified KWQL body consists of a content item term, that is, a resource term with a resource of type content item, together with its contained qualifier and sub-resource terms. In such a query body, all value terms occur in the context of a qualifier term which in turn is associated with a resource term. Furthermore, all resource terms whose resource is not of type content item are contained, directly or indirectly through an ancestor-descendant relationship, within a content item resource term.

Consider for example the query described in natural language in Section 7.1, which selects content items that contain the term "KiWi" but not the term "search" in their text, have a tag whose name contains "wiki," and link to a content item containing "KWQL" in the title. In KWQL this query is represented as follows:

```
ci(text:(KiWi AND NOT search) AND tag(name:wiki) AND
    link(target:ci(title:KWQL)))
```

Note that *fully specified* here does not mean that the query exhaustively describes all properties of the content items, but rather that the context in which each keyword occurs is unambiguously specified.

UNDERSPECIFIED QUERY BODIES In an underspecified query body, the context of at least one value term is not fully described. This means that there is at least one keyword that occurs outside of a qualifier or resource term, a qualifier term that occurs outside of a resource term, or a resource term that is not a content item term and is not contained directly or indirectly within a content item resource term. Underspecified KWQL bodies allow users to easily specify selection criteria that are vaguer than a fully specified query by not (fully) specifying the context in which the values must occur. When a qualifier or resource is not given, the query is extended to cover all possible qualifier or resource types that can occur in the context.

Consider for example the underspecified query `tag(name:KiWi)`. Since content items, fragments, and links can all be tagged, it can be extended to the following fully specified queries:

```
ci(tag(name:KiWi))
```

```
ci(link(tag(name:KiWi)))
```

```
ci(fragment(tag(name:KiWi)))
```

```
ci(fragment(link(tag(name:KiWi))))
```

The underspecified query thus returns content items which either have been tagged with "KiWi" themselves or contain fragments or links that have been assigned such a tag.

Another example is the underspecified query `ci(KWQL)`. Since "KWQL" is a string, only queries where the qualifier takes a string or URI are valid, and we obtain the fully specified queries `ci(title:KWQL)`, `ci(text:KWQL)`, `ci(author:KWQL)`, and `ci(URI:KWQL)`.

The only part of KWQL bodies that is not optional are value terms, so the most simple query consists of a single keyword. Since in that case the internal structure of the content item is not specified at all, such queries return content items where the term occurs anywhere within the content item or its sub-resources, but

not in its linked or contained content items. Similarly, when a query consists of several keywords, the content items that contain all keywords are returned, regardless of the keyword contexts. Thus, all of the examples given above for value, qualifier, and resource terms also constitute valid KWQL queries.

Since underspecified queries do not consider linked or contained content items for matching, the qualifiers labels of value terms that are queries have to be specified explicitly.

Depending on the query intent and knowledge of the user, the selection criteria may be more or less underspecified. For example, a user might not want to restrict the occurrence of "KWQL" to the title of the linked content item, but instead extended the query to the other qualifiers of the content item (but not its sub-resources). This can be achieved by changing the above query as follows:

```
ci(text:(KiWi AND NOT search) AND tag(name:wiki) AND
    link(target:ci(KWQL)))
```

The fact that the term "KWQL" should occur anywhere in the content item or its contained resources can be expressed by the following query:

```
ci(text:(KiWi AND NOT search) AND tag(name:wiki) AND
    link(target:KWQL))
```

On the other hand, the user might simply want the given terms to occur anywhere in the respective (linking and linked) content items. This is achieved by the following query:

```
KiWi AND NOT search AND wiki AND link(target:KWQL))
```

The optional conjunctive operators can also be left out:

```
KiWi NOT search wiki link(target:KWQL))
```

QUERY CONTAINMENT A KWQL query Q_1 is said to be *contained* in another query Q_2, here denoted $Q_1 \sqsubseteq Q_2$, if for any KiWi dataset D, $Q_1(D) \subseteq Q_2(D)$, where $Q_i(D)$ is the set of query answers obtained from evaluating Q_i on D.

QUERY QUALIFICATION We say that a KWQL query Q_1 *qualifies* an underspecified KWQL query Q_2, or is a *qualification* of Q_2, if Q_2 can be transformed into Q_1 by adding qualifiers or resource types—but not operators or resource, qualifier or value terms—to Q_2.

QUERY REFINEMENT We call a KWQL query Q_1 a *refinement* of another query Q_2 if Q_1 is a qualification of Q_2 or if Q_2 can be transformed into Q_1 by adding qualifiers or resource types or arbitrary resource, qualifier, or value terms, as long as none of the new terms are added as disjuncts.

When a query Q_1 refines another query Q_2, and in particular when Q_1 qualifies Q_2, then Q_2 is contained in Q_1: the addition of qualifiers or resource types to an underspecified query constrains the context in which the respective value term may occur to a subset of the contexts which are allowed without the additional qualification. More generally, adding a new selection criterion to a query by means of conjunction means that the answers to the resulting query will be the intersection of the content items that match the query before refinement and those that match the new selection criterion.

QUERY EQUIVALENCE A KWQL query Q_1 is called *equivalent* to another query Q_2 if they mutually contain each other.

Among the last four example queries we have considered, the first one is a qualification of the second one. The latter qualifies the third and fourth, which in turn are equivalent. Denoting by R_i the set of content items returned by the ith example, it holds that $R_1 \sqsubseteq R_2 \sqsubseteq R_3 = R_4$.

As we have already seen, KWQL distinguishes between queries that give a resource but no qualifier or sub-resource terms, like `ci(KWQL)`, and queries that do not define any structural context, like `KWQL`. In the former case, only those content items are returned in which KWQL occurs within the content item itself. The latter query returns all content items that contain the term directly or within a contained sub-resource. More generally, when a resource is given in the query term, the description is assumed to be complete, that is, sub-resources not stated in the query are not considered for matching. For example, the query `ci(author:Mary)` returns content items with the given author, but not content items where the given author is only the author of a tag assignment (the latter could be obtained with the query `ci(tag(author:Mary))`).

In the first and second of the four example queries above, the value term `KiWi AND NOT search` inside the text qualifier term is given in brackets. This is necessary to express that both conditions apply only to that qualifier term, because the colon, the operator binding a value term to a qualifier, takes precedence over disjunction and conjunction (it does, however, not take precedence over negation). The query term `text:KiWi AND NOT search` would thus be interpreted to mean that "KiWi" must occur in the text, while no qualifier term (including **`text`**) may contain "search."

There are two reasons underlying this design choice. First, it is consistent with the principle used in what is likely the most widely used form of label-keyword queries, Google's extended syntax.[1] Secondly, a user who is not aware of the higher precedence of qualifier-value association will pose a query that is more general than intended, and will obtain a superset of the answers to intended query. The mistake will thus be visible in the form of results that are not answers to the intended query, and can be traced back to the query. On the other hand, if qualifier-value term association had lower precedence than conjunction and disjunction, a mistake on the part of the user would produce a set of results that are a subset of the results of the intended query, making it harder for the user to discover the mistake.

It should be noted that these observations only apply when the second value (or, more generally, the kth value for $k \geqslant 2$) in the value term is not negated. Again consider the query `text:(KiWi AND NOT search)`. Removing the brackets actually makes the query more selective: `text:KiWi AND NOT search` expresses that search may not occur anywhere in the enclosing resource, while the bracketed version of the query only specifies that "search" may not occur in the `text` qualifier.

Underspecification in KWQL queries serves to express vagueness. As we have already explained above, underspecified queries can be transformed into one or more fully specified queries that together cover all possible valid qualifications of the underspecified query. When these are combined through disjunction, the set of query answers of the resulting query are identical to those of the original, underspecified query. We thus have the following result:

Proposition 1. *For every underspecified KWQL query, there is at least one equivalent fully specified KWQL query.*

For example, consider once more the underspecified query `ci(KWQL)`. A fully specified query equivalent to this query is the following:

```
ci(title:KWQL) OR ci(text:KWQL) OR ci(author:KWQL) OR
    ci(URI:KWQL)
```

This concludes our discussion of the basic elements and properties of KWQL bodies. The rest of this section illustrates the range of possible KWQL bodies and introduces some advanced features. A (somewhat simplified) grammar for KWQL query bodies is given in Figure 31.

1 An exception to this are labels that explicitly indicate that they apply to all following terms and that cannot occur in a query with keywords to which the query does not apply, namely "allinanchor," "allintext," "allintitle," "allinurl."

⟨*kwql-query*⟩ ::= ⟨*resource-term*⟩

⟨*resource-term*⟩ ::= ⟨*value-term*⟩ | ⟨*qualifier-term*⟩
| ⟨*structure-term*⟩
| ⟨*resource-term*⟩ (‘OR’ | ‘AND’)? ⟨*resource-term*⟩
| ‘(’ ⟨*resource-term*⟩ ‘)’
| ‘NOT’ ⟨*resource-term*⟩
| ⟨*resource*⟩ ‘(’ ⟨*resource-term*⟩ ‘)’

⟨*resource*⟩ ::= ‘link’ | ‘ci’ | ‘fragment’ | ‘tag’

⟨*structure-term*⟩ ::= (‘child’ | ‘descendant’ | ‘target’) ‘:’
⟨*resource-term*⟩

⟨*value-term*⟩ ::= ⟨*STRING*⟩

⟨*qualifier-term*⟩ ::= ⟨*qualifier*⟩ ‘:’ (⟨*value-term*⟩ | ⟨*variable*⟩)

⟨*qualifier*⟩ ::= ‘text’ | ‘title’ | ‘name’ | ‘URI’ | ‘agree’
| ‘disagree’ | ‘lastEdited’ | ‘numberEd’
| ‘author’ | ‘created’ | ‘anchorText’

⟨*variable*⟩ ::= ‘$’⟨*IDENTIFIER*⟩

Figure 31: KWQL Syntax

VARIABLES In a qualifier term with specified qualifier, a variable may be used instead of a value term. Variables can occur in underspecified queries, as long as the qualifier of the variable itself is given. Variable names consist of a dollar sign followed by an alphanumeric string. Upon query evaluation, all *answer substitutions* are generated, that is, all valid tuples of values that the variables in the query body may take, and each variable is bound to its possible values. In KWQL bodies, variables can serve three purposes:

- To bind values for further use in the construction part of a rule. Consider the following query:

```
ci(tag(name:KiWi) author:$a title:$t)
```

This query retrieves all content items tagged with “KiWi,” and for each of them binds the names of its authors to the variable $a and the title to $t. Only qualifier values, but not qualifiers or resources themselves, can be bound to variables. The main reason for this, apart from conceptual simplicity, is the strict separation of selection and construction—binding complex entities to variables would necessitate a way to selectively access individual values, and thereby introduce a form of selection into the process of construction.

- As a wildcard or existential quantifier. The following query retrieves all content items that have been assigned at least one

tag, regardless of the name, author or other properties of said tag assignment:

```
ci(tag(name:$t))
```

Together with negation and injectivity (see below), variables can be used to realize counting constraints, for example to retrieve all content items that have exactly one tagging:

```
ci(tag(name:$t) NOT(tag(name:$t) tag(name:$u)))
```

This query selects content items that have at least one tagging but at the same time excludes content items that have two or more taggings. As a result, only content items with exactly one assigned tag are returned.
By adding an asterisk operator (`*`) to express wildcards, as is common in many applications, this functionality could be made more accessible in a way that is more convenient and does not require users to understand how variables are used in KWQL. This could easily be implemented in terms of a pre-processing phase that replaces each occurrence of a wildcard with a new variable, i.e., one that is not used elsewhere in a query. The two previous examples could then be expressed in the following way:

```
ci(tag(name:*))
```

```
ci(tag(name:*) NOT(tag(name:*) tag(name:*)))
```

- To enforce that two qualifiers have identical values or—in the case of **`text`**, **`title`**, and **`anchorText`**—share at least one token. In KWQL, all occurrences of a variable in a query body must have the same value; using the same variable several times therefore amounts to imposing equality constraints on the values of the respective qualifiers. The following query retrieves content items that contain a link to another content item. Both content items must have been assigned at least one tag and contain a fragment which in turn has also been tagged at least once. The name of the tags of the content items and the name of the fragment taggings must respectively be identical.

```
ci(tag(name:$c) fragment(tag(name:$f))
    link(target:ci(tag(name:$c) fragment(tag(name:$f)))))
```

Similarly, the following query returns content items which have also been tagged by at least two of their authors:

```
ci(author:$a author:$b tag(author:$a) tag(author:$b))
```

The values of the qualifiers **text**, **title** and **anchorText** are treated as lists of tokens. The following query thus retrieves content items that have at least one token in common with the text of one of the content items that they link to, and the variable $a is bound to the terms that occur in both texts:

```
ci(text:$a link(target:ci(text:$a)))
```

In the first usage described above, variables serve the function of **SELECT** statements, while regular value-based KWQL terms can be likened to **WHERE** clauses in, for example, SQL.

The two can be combined using the operator **->**, which allows the user to both specify a selection constraint and bind the value of the qualifier to a variable. This functionality is particularly useful in the context of partial matchings, for example to retrieve the full text of all content items that contain the term "KWQL":

```
ci(text:KWQL -> $a)
```

Another operator, **OPTIONAL** (optionality in the context of RDF web query languages is discussed in Section 4.2.2.3), can be used in connection with qualifier terms or sub-resources that contain variables. It is similar to conjunction but matches content items regardless of whether the corresponding selection criteria or variable bindings can be met or not. Consider the following query:

```
ci(text:Java author:$a OPTIONAL tag(name:$t))
```

When this query is executed, the authors of content items that have "Java" in their text are bound to the variable $a. Moreover, if the content item has been assigned at least one tag, the tag names are bound to the variable $t. The same functionality could be achieved by executing two queries, with and without the optional query terms, and fusing the results. However, this would be less convenient for the user and more costly to evaluate.

KEYWORDS FOR STRUCTURE KWQL allows for the selection of data based on the structure of content items and fragments through the **child** and **descendant** qualifiers. To keep the language simple, navigational queries are avoided and no qualifiers are offered for parents and ancestors. Olteanu et al. [291] have shown that adding these *backward axes* does not increase the expressiveness of a query language.

KWQL's structural qualifiers give rise to recursive data retrieval through a wiki page structure. These qualifiers take subqueries as a value, that is, arbitrary KWQL queries specifying selection

constraints on a linked or nested content item or fragment. For example, the following query selects content items that are tagged with "Java" and have a child content item tagged with "XML."

```
ci(tag(name:Java) child:ci(tag(name:XML)))
```

Structure qualifiers can thus be seen as edges to other content items or fragments, and recursive querying as traversal of the resulting graph.

Link traversal can be expressed similarly. The following query for example matches content with a tag "Java" and a link that points to a content item with "XML" in the title:

```
ci(tag(name:Java) link(target:ci(title:XML)))
```

It should be noted that, despite the fact that structural queries and link traversals can be nested, no infinite loops can occur. This is due to the fact that queries are always finite and KWQL does not support Kleene closure.

INJECTIVITY The same type of resource can occur several times within another resource. For example, a content items can contain several links, can be tagged with multiple tags, or link to several other content items. Similarly, a qualifier term of the same type may occur several times, for example in a content item with more than one author.

In such cases, the selection constraints given in the qualifier or resource terms have to match on different qualifier or resource term instances. When constraints on two or more terms are to be expressed, this is realized by repeating the respective term.

```
ci(author:Mary)
```

This query could match twice on a content item which has Mary Smith and Mary Miller among its authors. To explicitly match only content items which have two authors with the first name Mary, the query is expressed as

```
ci(author:Mary author:Mary)
```

To specify criteria that should be satisfied by two or more distinct instances of a resource, the resource and its respective selection criteria have to be given the appropriate number of times. For example, to match a content item tagged with two distinct tags, "Java" and "XML," the following query can be used:

```
ci(tag(name:Java) tag(name:XML))
```

QUERYING STRUCTURED TAGS So far, we have only considered the querying of simple tag assignments using the resource **tag** and the qualifier **name** to select content items, fragments, or links whose tag label contains a certain value.

Underspecification, the mechanism underlying KWQL queries, can be applied to structured tags by specifying either the full tag or only part of it. Consider the following two queries:

```
ci(tag(name:'(Warwick, UK), New York, (May 26, 2008,
    8am)'))
```

```
ci(tag(name:New York))
```

The first query selects content items that have been tagged with the structured tag `(Warwick, UK), New York, (May 26, 2008, 8am)`, while the second one selects content items that have a tag, atomic or structured, in which "New" and "York" occur.

In addition, however, it should be possible to specify selections that neither express the structured tag fully nor merely list tokens that must occur in the tag label. In order to allow for more expressive querying of structured tags without requiring the user to learn a full tree query language (or, rather, a language for extended trees since structured tags do not strictly have to be tree-shaped), KWQL uses subsumption to match structured tags. This means that a structured tag assignment is matched when it satisfies all criteria specified in the tag selection, regardless of whether the tag also contains elements not mentioned in the query.

For example, the query

```
ci(tag(name:'Warwick, UK'))
```

matches all content items that have been assigned a tag with a label where "Warwick" and "UK" occur at the same nesting level, independently of whether there are further elements in the structured tag. The query

```
ci(tag(name:'geo:'))
```

selects all resources that have been tagged with a structured tag which contains "geo" as a categorization.

7.2.4 *KWQL Heads*

While a KWQL body describes content items to be selected, the optional head specifies the content items that should be created. Construction in KWQL combines ideas borrowed from

Xcerpt [320] (also discussed in Section 4.2.3) construct terms with a syntax that is largely identical to that of KWQL selections. However, the distinction between selection and construction gives rise to a number of syntactic differences.

- The head of a KWQL rule can specify content, annotations, and linking and nesting structure of the content items to be created. URIs and metadata like authoring information are added automatically upon creation of the content item and cannot be specified explicitly. The qualifiers **`author`**, **`created`**, **`URI`**, **`lastEdited`** and **`numberEdits`** may therefore not be used in KWQL heads.
- In KiWi, each content item must have a title, each fragment must have a text, each link must have an anchor text and a target, and each tag must have at least one label. When the construction does not assign a value to a mandatory qualifier upon creation of a resource, a randomly generated value is assigned.
- Unlike selection, construction fully describes the content item or content items to return. Apart from the automatically generated metadata and, where necessary, values for mandatory qualifiers, a content item does not have any content or properties except those defined in the rule head.
- The description of the content item to be created must be unambiguous. Therefore, all heads must be fully specified, qualifiers and resources may not be omitted, and negation or disjunction may not be used. The **`descendant`** qualifier is not available in rule heads because it is ambiguous with respect to the content item or fragment nesting structure.
- Links, fragments, and children have to be specified directly in the text of the content item or fragment to define their position.
- Additional constructs for grouping and aggregation can be used to process and reformat the selected data.

GROUPING CONSTRUCTS Assume that the query below is evaluated on a dataset about the KiWi project and yields the variable bindings given in Table 4.

```
ci((title:KiWi->$t OR (tag(name:KiWi) title:$t))
    author:$a)
```

Each row of this table represents an answer substitution, that is, a tuple satisfying the condition given in the query. The first element of each tuple is the title, and the second element the name of an author of a content item that contains KiWi in the title.

To access, collect, and arrange variable bindings in content items to be constructed, KWQL provides the grouping constructs **`ALL`** and **`SOME`**. **`ALL`** collects all possible bindings for a variable, while **`SOME`** collects a given number of variable bindings to be chosen at random. In both cases, at least one binding must exist.

$t	$a
KiWi Partners	Mary
KiWi Partners	John
KiWi Partners	Lisa
KiWi System	John
KiWi System	Michael

Table 4: Example variable bindings

Given the variable bindings in Table 4, the following head creates a content item with title "Contributors" that lists, for each author, the titles of the content items the author has contributed to:

```
ci(title:Contributors text:ALL($a - ALL($t,", "),\n))
```

The text of the resulting content item then looks as follows:

```
Mary - KiWi Partners
John - KiWi Partners, KiWi System
Lisa - KiWi Partners
Michael - KiWi System
```

A *grouping term* consists of a *grouping construct* together with a *template* and optional *separator*. The grouping construct **SOME** additionally requires an integer as an argument that specifies the number of bindings to be retrieved.

A template consists of at least one variable and possibly further grouping terms, strings, and KWQL construction terms. The above rule head uses two grouping terms: `$a - ALL($t,","),\n` and `$t,", "`. The former consists of the variable $a, the string "-" and the grouping term `$t`, which in turn only consists of the variable $t. The outer grouping term uses the newline character as the separator, while the nested grouping term uses a comma.

When a grouping term is evaluated, all variables contained directly in the template but not in one of the nested grouping terms are replaced with all (or some) of their possible bindings. The template is repeated for every binding or valid combination of bindings, delimited by the string given as the separator. A combination of variable bindings is *valid* when the bindings occur together in an answer substitution. The strings in the template are reproduced unchanged in every repetition, while contained grouping terms are evaluated in the same manner.

To instead list each content item together with all its authors, one could simply swap the variables names to obtain the following construction term:

```
ci(text:ALL($t - ALL($a,", "),\n))
```

In this query and the ones that follow, no title is specified for the return content item. Since all content items are required to have a title, they are automatically assigned a randomly generated title upon execution.

The content of Table 4 can be recreated with the following rule head:

```
ci(text:ALL($t - $a,\n))
```

If instead the following construction term was used, only one, randomly chosen, title-author pair would be included in the text:

```
ci(text:SOME(1,$t - $a,\n))
```

Assuming that the URIs of the content items are bound to a variable $u, a linked table of contents of all content items matching the query body can be created as follows:

```
ci(text:ALL(link(anchorText:$t target:$u),\n))
```

KWQL heads can also be used to create several independent content items. Given the query body above, the following construction term creates a project member page for every wiki user who has contributed to an article about KiWi, i.e., an article that contains KiWi in the title or the label of an assigned tag:

```
ALL(ci(title:$a tag(name:KiWi project member)))
```

AGGREGATION FUNCTIONS In addition to grouping, existing data can also be aggregated to form new data. Table 5 lists the aggregation functions available in KWQL. As arguments, these functions take either a grouping term or a comma-separated list of values. Whether a specific aggregation functions can be applied depends on the type of the arguments—numeric or textual.

Using these aggregation functions, the list of authors who have written about KiWi could be modified such that authors are listed alphabetically, and only the number of content items is given for each author rather than the titles of all content items. This is achieved by the following rule head:

```
ci(title:Contributors text:SORT(ALL($a - COUNT($t),\n)))
```

For the example data in Table 4, the text of the resulting content item would looks as follows:

name	type	return value
COUNT	any	number of arguments
SORT	any	arguments sorted in ascending order
SUM	numeric	sum of all arguments
AVG	numeric	average of all arguments
MIN	numeric	minimum of all arguments
MAX	numeric	maximum of all arguments

Table 5: Aggregation functions

```
John - 2
Lisa - 1
Mary - 1
Michael - 1
```

THE RENDER FUNCTION KWQL also provides way to embed existing content items as children in return content items. This is achieved by a function **RENDER** that takes a URI as an argument. Consider for example the following query body, which binds the variable $u to the URIs of content items that are linked to from content items containing "KiWi" and share at least one tag name with them:

```
ci(KiWi tag(name:$t) link(target:ci(URI:$u tag(name:$t)))
```

The following head can then be used to aggregate all content items satisfying this condition in a single content item:

```
ci(text:ALL(child:RENDER($u)))
```

OPTIONAL As described in the Section 7.2.3, **OPTIONAL** can be used in query bodies to indicate that a content item should match regardless of whether or not the variables contained in the operand term can be bound. Assume for example that the wiki user John wants to create a summary of all content items linking to his user page. To get an idea of the nature of these content items, he would like to list not just the titles, but also the names of the tags assigned to each such content item. Since some of the content items may not have been assigned any tags, it makes sense to make the term that binds a variable to tag names optional. We thus obtain the following query:

```
ci(link(target:ci(title:John tag(name:user))) title:$t
    OPTIONAL tag(name:$n))
```

The tuples of variable bindings produced by this selection may not all have the same number of elements, depending on whether or not the matching content items have a been assigned a tag. The variable $n can still be used with grouping terms when setting a default in the construction term that is inserted when the variable has no binding. The following rule head uses the text "none" for content items that have not been assigned a tag:

```
ci(text:ALL(Title: $t - Tags: OPTIONAL ALL($n,",")
    DEFAULT none,\n))
```

7.2.5 *KWQL Rules*

A KWQL rules consists of a head and a body, separated by the character "@":

```
ci(text:ALL(Title: $t - Tags: OPTIONAL ALL($n,",")
    DEFAULT No Tags,\n))@ci(link(target:ci(title:John
    tag(name:user))) title:$t OPTIONAL tag(name:$n))
```

```
ci(title:Contributors text:ALL($a - ALL($t,",
    "),\n))@ci((title:KiWi->$t OR (tag(name:KiWi)
    title:$t)) author:$a)
```

A rule can be read as an if-then statement where the query body specifies a set of conditions, and query evaluation identifies the objects—content items or variables—that satisfy the conditions. The query head describes which action should be taken, or more specifically how the return content items should look; it is executed if the evaluation of the rule body yields at least one result. In their functionality, KWQL rules also resemble views in query languages for relational databases.

7.3 A FORMAL SEMANTICS FOR KWQL

For defining a formal semantics of KWQL, we introduce an abstraction of the data model of KWQL, called KWQL graphs:

Definition 1 (KWQL Graph). *Let $\mathcal{Q} = \{text, title, \ldots\}$ be the set of all n KWQL qualifiers and $\mathcal{V}$ the set of all qualifier values. Then, a* KWQL graph *is a $(n+6)$-tuple $G = (\mathcal{C}, \mathcal{F}, \mathcal{L}, \mathcal{T}, S, C, Q_{\lambda_1}, \ldots, Q_{\lambda_n})$ such that 1. $\mathcal{C}$ is the set of all content items (wiki pages), 2. $\mathcal{F}$ is the set of all fragments, 3. $\mathcal{L}$ is the set of all links, 4. $\mathcal{T}$ is the set of all tags, 5. $\mathcal{R} := \mathcal{C} \uplus \mathcal{L} \uplus \mathcal{T} \uplus \mathcal{F}$ is the set of all resources, 6. $S \subset (\mathcal{C} \times \mathcal{C}_<) \cup (\mathcal{F} \cup \mathcal{F}_<) \cup (\mathcal{L} \cup \mathcal{L}_<)$ is the association relation between resources where $\mathcal{C}_< = \mathcal{F} \cup \mathcal{L} \cup \mathcal{T}, \mathcal{F}_< = \mathcal{L} \cup \mathcal{T}, \mathcal{L}_< = \mathcal{T} \cup \mathcal{C} \cup \mathcal{F}$.*

7. $C \subset \mathcal{C} \times (\mathcal{C} \cup \mathcal{F})$ *is the containment relation between wiki pages and fragments and* $C^+ = \bigcup_{n \geqslant 1} C^n$ *is the transitive closure of C. 8. for each qualifier* $\lambda \in \mathcal{Q}$, $Q_\lambda \subset \mathcal{R} \times \mathcal{V}$ *associates the values for* λ *to a KWQL resource.*

The KWQL semantics is defined based on KWQL graphs and given in Table 6 in terms of three functions, $[\![]\!]_{ci}$, $[\![]\!]$, and $[\![]\!]_{dir}$. A KWQL query is constrained by $[\![]\!]_{ci}$ to return only content items (i.e., elements of $\mathcal{C}$). Most expressions can occur in two contexts, represented by the semantic functions $[\![]\!]$ and $[\![]\!]_{dir}$: In the first context a query such as `Java` returns all resources that contain "Java" directly in any of their qualifiers or indirectly in the qualifiers of any of their fragments, tags, and links. In the second context only resources that contain "Java" directly are returned. The exception to this rule are keyword queries which are always interpreted in the first manner.

The semantics in Table 6 handles variables, but omits the injectivity constraints for readability reasons. To handle variables, we introduce the set $\mathcal{I}$ of KWQL variables and the set $\mathcal{B} = 2^{\mathcal{I} \times \mathcal{V}}$ of possible variable assignments (pairs of variables and value). We further extend the set operators $\cup$ and $\cap$ to pairs of resources and variable assignments as follows: Let $A, B \in 2^{\mathcal{R} \times \mathcal{B}}$. Then $A \sqcap B = \{(r, \beta) \in \mathcal{R} \times \mathcal{B} : (r, \beta') \in A \wedge (r, \beta'') \in B \wedge \beta = \beta' \cap \beta'' \wedge \beta \neq \emptyset\}$, $A \sqcup B = \{(r, \beta' \cup \beta'') \in \mathcal{R} \times \mathcal{B} : (r, \beta') \in A \wedge (r, \beta'') \in B\} \cup \{(r, \beta') \in \mathcal{R} \times \mathcal{B} : (r, \beta') \in A \wedge \nexists \beta'' : (r, \beta'') \in B\} \cup \{(r, \beta') \in \mathcal{R} \times \mathcal{B} : (r, \beta') \in B \wedge \nexists \beta'' : (r, \beta'') \in A\}$.

$$
\begin{aligned}
&[\![\langle \textit{kwql-query}\rangle]\!]_{\text{ci}} && = \pi_1([\![\langle \textit{kwql-query}\rangle]\!](\emptyset)) \cap \mathcal{C}\\
&[\![\langle \textit{STR}\rangle]\!]_{\text{dir}}(\beta) = [\![\langle \textit{STR}\rangle]\!](\beta) && = \big\{(r,\beta) \in \mathcal{R}\times\mathcal{B} : \exists \lambda, \nu : Q_\lambda(r,\nu) \wedge \texttt{contains}(\nu, \langle \textit{STR}\rangle)\big\} \cup\\
& && \quad \big\{(r,\beta) : \exists r' \in \mathcal{R} : S(r,r') \wedge (r',\beta) \in [\![\langle \textit{STR}\rangle]\!](\beta)\big\}\\
&[\![\langle \textit{qualifier}\rangle\texttt{':'}\langle \textit{STRING}\rangle]\!]_{\text{dir}}(\beta) && = \big\{(r,\beta) \in \mathcal{R}\times\mathcal{B} : Q_{\langle \textit{qualifier}\rangle}(r,\nu) \wedge \texttt{contains}(\nu, \langle \textit{STRING}\rangle)\big\}\\
&[\![\langle \textit{qualifier}\rangle\texttt{':'}\langle \textit{STRING}\rangle]\!](\beta) && = [\![\langle \textit{qualifier}\rangle\texttt{':'}\langle \textit{STRING}\rangle]\!]_{\text{dir}}(\beta) \cup\\
& && \quad \big\{(r,\beta) : \exists r' \in \mathcal{R} : S(r,r') \wedge (r',\beta) \in [\![\langle \textit{qualifier}\rangle\texttt{':'}\langle \textit{STRING}\rangle]\!](\beta)\big\}\\
&[\![\langle \textit{qualifier}\rangle\texttt{':''\$'}\langle \textit{IDENT}\rangle]\!]_{\text{dir}}(\beta) && = \big\{(r,\beta \cup \{(\langle \textit{IDENT}\rangle,\nu)\}) \in \mathcal{R}\times\mathcal{B} : Q_{\langle \textit{qualifier}\rangle}(r,\nu) \wedge\\
& && \quad (\nexists \nu' : (\langle \textit{IDENT}\rangle,\nu') \in \beta \vee (\langle \textit{IDENT}\rangle,\nu) \in \beta)\big\}\\
&[\![\langle \textit{qualifier}\rangle\texttt{':''\$'}\langle \textit{IDENT}\rangle]\!](\beta) && = [\![\langle \textit{qualifier}\rangle\texttt{':'}\langle \textit{STRING}\rangle]\!]_{\text{dir}}(\beta) \cup\\
& && \quad \big\{(r,\beta) : \exists r' \in \mathcal{R} : S(r,r') \wedge (r',\beta) \in [\![\langle \textit{qualifier}\rangle\texttt{':''\$'}\langle \textit{IDENT}\rangle]\!](\beta)\big\}\\
&[\![\langle \textit{resource}\rangle\texttt{':'}\langle \textit{res-term}\rangle]\!]_{\text{dir}}(\beta) && = \big\{(r,\beta') \in \mathcal{R}\times\mathcal{B} : \texttt{type}(r,\langle \textit{resource}\rangle) \wedge (r,\beta') \in [\![\langle \textit{res-term}\rangle]\!]_{\text{dir}}\big\}\\
&[\![\langle \textit{resource}\rangle\texttt{':'}\langle \textit{res-term}\rangle]\!](\beta) && = [\![\langle \textit{resource}\rangle\texttt{':'}\langle \textit{res-term}\rangle]\!]_{\text{dir}}(\beta) \cup\\
& && \quad \big\{(r,\beta) : \exists r' \in \mathcal{R} : S(r,r') \wedge (r',\beta) \in [\![\langle \textit{resource}\rangle\texttt{':'}\langle \textit{res-term}\rangle]\!](\beta)\big\}\\
&[\![\texttt{'child'':'}\langle \textit{kwql-query}\rangle]\!](\beta) && = \big\{(r,\beta') \in (\mathcal{C}\cup\mathcal{F})\times\mathcal{B} : \exists r' \in \mathcal{R} : C(r,r') \wedge (r',\beta') \in [\![\langle \textit{kwql-query}\rangle]\!]\big\}\\
&[\![\texttt{'descendant'':'}\langle \textit{kwql-query}\rangle]\!](\beta) && = \big\{(r,\beta') \in (\mathcal{C}\cup\mathcal{F})\times\mathcal{B} : \exists r' \in \mathcal{R} : C^+(r,r') \wedge (r',\beta') \in [\![\langle \textit{kwql-query}\rangle]\!]\big\}\\
&[\![\texttt{'target'':'}\langle \textit{kwql-query}\rangle]\!](\beta) && = \big\{(r,\beta') \in \mathcal{L}\times\mathcal{B} : \exists r' \in \mathcal{R} : S(r,r') \wedge (r',\beta') \in [\![\langle \textit{kaw-query}\rangle]\!]\big\}\\
&[\![\langle \textit{res-term}\rangle_1\ \langle \textit{res-term}\rangle_2]\!](\beta) && = [\![\langle \textit{res-term}\rangle_1]\!](\beta) \sqcap [\![\langle \textit{res-term}\rangle_2]\!](\beta)\\
&[\![\langle \textit{res-term}\rangle_1\texttt{'AND'}\langle \textit{res-term}\rangle_2]\!](\beta) && = [\![\langle \textit{res-term}\rangle_1]\!](\beta) \sqcap [\![\langle \textit{res-term}\rangle_2]\!](\beta)\\
&[\![\langle \textit{res-term}\rangle_1\texttt{'OR'}\langle \textit{res-term}\rangle_2]\!](\beta) && = [\![\langle \textit{res-term}\rangle_1]\!](\beta) \sqcup [\![\langle \textit{res-term}\rangle_2]\!](\beta)\\
&[\![\texttt{'('}\langle \textit{res-term}\rangle\texttt{')'}]\!](\beta) && = [\![\langle \textit{res-term}\rangle]\!](\beta)\\
&[\![\texttt{'NOT''('}\langle \textit{res-term}\rangle\texttt{')'}]\!](\beta) && = \mathcal{R} \setminus \pi_1([\![\langle \textit{res-term}\rangle]\!](\beta)) \times \{\beta\}\\
&[\![\langle \textit{res-term}\rangle_1\ \langle \textit{res-term}\rangle_2]\!]_{\text{dir}}(\beta) && = [\![\langle \textit{res-term}\rangle_1]\!]_{\text{dir}}(\beta) \sqcap [\![\langle \textit{res-term}\rangle_2]\!]_{\text{dir}}(\beta)\\
&[\![\langle \textit{res-term}\rangle_1\texttt{'AND'}\langle \textit{res-term}\rangle_2]\!]_{\text{dir}}(\beta) && = [\![\langle \textit{res-term}\rangle_1]\!]_{\text{dir}}(\beta) \sqcap [\![\langle \textit{res-term}\rangle_2]\!]_{\text{dir}}(\beta)\\
&[\![\langle \textit{res-term}\rangle_1\texttt{'OR'}\langle \textit{res-term}\rangle_2]\!]_{\text{dir}}(\beta) && = [\![\langle \textit{res-term}\rangle_1]\!]_{\text{dir}}(\beta) \sqcup [\![\langle \textit{res-term}\rangle_2]\!]_{\text{dir}}(\beta)\\
&[\![\texttt{'('}\langle \textit{res-term}\rangle\texttt{')'}]\!]_{\text{dir}}(\beta) && = [\![\langle \textit{res-term}\rangle]\!]_{\text{dir}}(\beta)\\
&[\![\texttt{'NOT''('}\langle \textit{res-term}\rangle\texttt{')'}]\!]_{\text{dir}}(\beta) && = \mathcal{R} \setminus \pi_1([\![\langle \textit{res-term}\rangle]\!]_{\text{dir}}(\beta)) \times \{\beta\}
\end{aligned}
$$

Table 6: Semantics for KWQL

8

VISKWQL

Visual languages, employing elements like shapes and colors instead of a strictly textual syntax, have two advantages over textual languages that specifically benefit beginning users [86]: first, their visual nature can make them easier to learn and understand than textual languages; second, editors for visual languages can support the creation of valid queries by providing guidance to the user and preventing editing operations that would result in incorrect queries.

In this chapter we present visKWQL, which is a *visual rendering* of KWQL rather than a separate query language. visKWQL extends the textual query language KWQL to provide two cohesive and tightly integrated querying modi for user-friendly and powerful querying in the KiWi wiki. visKWQL is the first visual interface to a keyword-based query language. Unlike many other visual query languages which add a user-friendly component to conventional web or database query languages, visKWQL faces the unique challenge of complementing and further improving the usability of a textual language that itself puts a heavy emphasis on user-friendliness.

visKWQL fully supports KWQL in the sense that every KWQL rule can be expressed as an equivalent visKWQL rule. In order to avoid introducing additional constructs and thus additional complexity, visKWQL stays as close as possible to the textual language in its visual representation.

An accompanying editor, the *KWQL Query Builder* (KQB), allows for the easy and straightforward construction of queries using drag-and-drop, and in addition supports the user during query construction by displaying tooltips, preventing syntactic errors where possible, and pointing the user to the problematic parts of a query. The goal of visKWQL is to allow even beginning users to quickly create useful queries.

The KQB further provides features like information hiding to only display parts of larger queries, or the highlighting of all occurrences of a variable when the mouse pointer is positioned over a variable in a query. A particularly important feature of KQB is round-tripping, which enables the user to edit both the textual or the visual form of a query, and see any changes made to one representation reflected in the other.

The remainder of this chapter is structured as follows: Section 8.1 provides an overview of the area of visual querying and briefly introduces a number of visual query languages together

with examples. Section 8.2 then outlines the requirements for a visual query language based on KWQL, and Section 8.3 describes how these requirements are met by visKWQL and its editor KQB. A practical guide to the usage of KQB is given in Section 8.4. Section 8.5 finally provides an overview over the implementation of visKWQL and the KQB.

8.1 VISUAL QUERY LANGUAGES

A wide variety of visual languages have been created over the years, including programming, modeling, and query languages. Those languages differ as much in their visual formalisms as in their purpose and underlying textual languages. A 1997 survey alone mentions more than 50 visual database query languages [86]. Visual languages range from table-based database query languages like QBE [381], the very first visual language, to UML [170], the modeling language that has become an integral part of software engineering, to the comic-strip based ComiKit [220], which allows children to program simple games.

Research into visual languages was mainly triggered by two things: the availability of the technological means to move from text to graphics, and a trend towards improved usability.

The first modern computers were restricted to textual input via keyboard and textual output via terminal or printer, so naturally programming and query languages were text-based. In the early 1980s, however, important advances were made in the area of computer hardware. Video cards enabled the use of both dimensions of a monitor since individual pixels could now be addressed. Further, the computer mouse now allowed pointing anywhere on the screen. Where text used to be white on black, there now was support for multiple colors, and meaning could be expressed through the spatial placement of program elements on the screen, shapes, textures, and sizes of program elements, and the nesting of elements.

A second factor that started research into visual languages was the increased trend towards usability that emerged roughly at the same time. In the early years, computers were expensive and difficult to use, and were used only by few, many of them scientists. But as hardware prices went down and the new technology was adopted more widely, the spectrum of people working with computers became broader and now also included casual users who had little or no prior knowledge of computer languages or the internal structure of a database.

Ziegler and Fahnrich [380] list several advantages of visual query formulation, or *direct manipulation techniques*:

- The distance between the user's mental model of reality and the representation displayed by the computer is decreased,

i.e., the query representation becomes less abstract and more intuitive.

- A visual formalism is usually less language-dependent.
- Basic functionalities of the language can be learned easier.
- A high efficiency rate can also be obtained by expert users, partly because of the possibility of defining new functions and features.
- The error rate in formulating queries is reduced significantly.

Of course, the validity of these arguments varies between different visual query languages, and in the worst case, a visual language may fail to reach any of these goals and may instead only introduce an additional layer of complexity on top of a textual language.

Measuring and comparing abstract concepts like ease of use is not an easy endeavor, and many visual programming languages are so different from textual languages in their expressiveness and their intended use that no valid comparison can be made. However, many visual query languages are based on textual ones and share their expressive power. In this case the corresponding queries can be compared, and a study by Catarci and Santucci [87] provides evidence that visual languages can indeed be easier to use than textual ones. In the study, users with different skill levels were taught both the textual query language SQL and the visual query language QBD* [22]. They were then asked to construct queries in both languages, and Catarci and Santucci measured the accuracy of the resulting queries and the time taken to formulate the query. Users reached 100% accuracy with QBD* but only about 90% with SQL, and query formulation in QBD* in some cases took only about 50 percent of the time required to formulate the same query in SQL, even for expert users.

In the following we discuss various examples of *form-based* and *diagram-based* visual query languages.

8.1.1 *Form-Based Approaches*

Form-based visual query languages employ a visualization technique inspired by paper forms or spreadsheets. They consist of basic elements, so-called cells, which cannot be nested. Users formulate queries by filling in cell values.

visKWQL can be considered form-based, but also employs features typical for diagram-based systems such as element nesting and the usage of colors to distinguish between element types.

TYPE	ITEM	COLOR	SIZE
	P.ROD	GREEN	

Figure 32: A simple QBE query

QBE QBE [381], short for *Query-by-Example*, was the very first visual query language. It was developed in the mid-1970s as an alternative to textual SQL. Its design goals were to provide a convenient, unified, high-level language to query, update, refine and control a relational database. The language should require only little prior knowledge from the user and consist of only a small number of concepts, making it easier to learn than a textual language.

The approach chosen to achieve these design goals is to provide the user with two-dimensional skeleton tables, which can be filled with an example of the intended solution. The user can provide constant elements, attribute values, and example elements, which act as variables. He can also use a number of special instructions, for example to sort the results, print them, or count the number of results.

Figure 32 shows a simple QBE query. When the user starts the query formulation, only the empty outline of a table is displayed. The user can enter a table name into the field in the upper left corner, in this example "Type." The system will then fill in the remaining column headings with the attributes of that table, or allow the user to enter them manually. When the table skeleton is complete, the user can proceed to fill in example values. **P.** in the "item" column stands for "print" and indicates the desired output. The underlined "rod" indicates an example value for a possible output and is used like a variable. The entry "green" finally is a constant value, a restriction or required condition on the output. Upon evaluation, the query will thus print all items that have green color.

Qualified retrieval can be accomplished in QBE by adding inequality operators to constant elements (for example, ">100" in the "Size" field of Figure 32) or by partially underlining elements to express partial string matching (for example, **P.X visual Y** returns all items that contain the word "visual"). Negation can be expressed through a special marking of cells or elements. Arithmetic expressions can be used on all attributes with numerical types. QBE also provides a condition box, which allows the user to enter various conditional expressions involving the variables defined in the query. Insertion, deletion, and update of database entries are possible by replacing the print instruction **P.** with **I.**, **D.**, or **U.** The insert and delete instructions can also be used to create, expand, or drop entire tables.

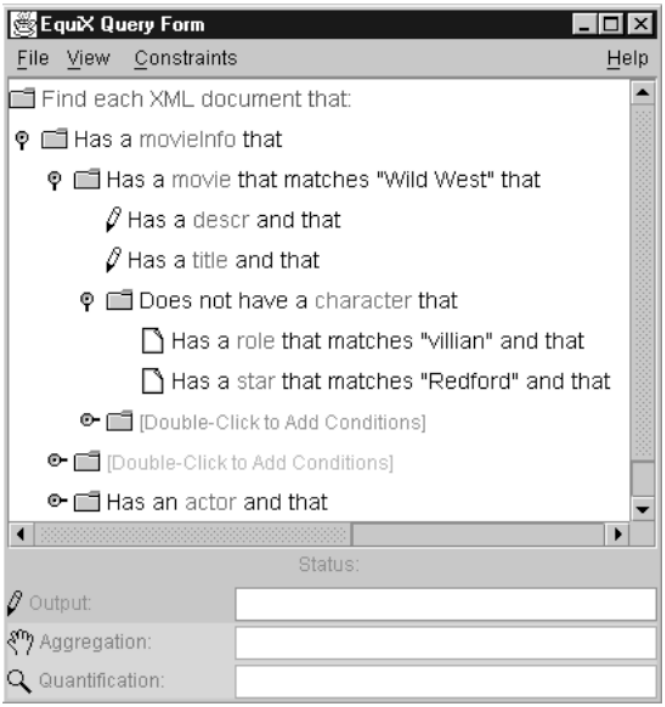


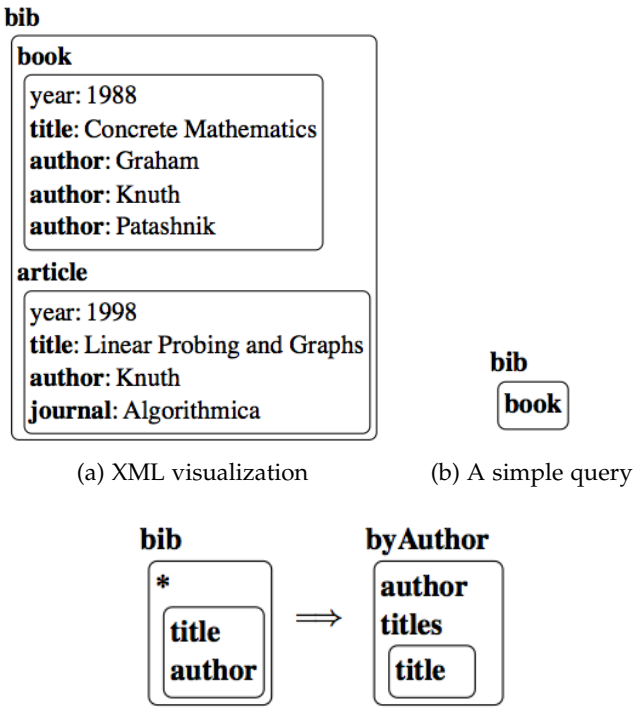
Figure 33: An EquiX query form, adopted from Cohen et al. [108]

QBE is a seminal work that has inspired many visual languages. However, apart from some software packages like Microsoft Access, it is no longer used today, most likely due to its limited expressiveness.

EQUIX EquiX [108] is a search and query language for XML data aimed at users without any previous knowledge of (textual) query languages. For that reason, Equix favors simplicity over expressiveness, and for example offers no data construction capabilities.

EquiX applies the Query-By-Example philosophy to XML data by letting users construct template-like query forms that contain an example of the desired query result. After the user has selected an XML catalog to query, the system displays the root node in the query form. Elements can be explored by clicking on them, which opens a view of their attributes and child elements. The user can then iteratively select attributes and child elements, fill text fields to impose constraints, specify quantifications, and choose which elements should appear in the output. The system then takes this example document, matches it against the documents in the selected catalog, and returns all matching documents, automatically generating a new DTD from the query form.

Figure 33 shows an EquiX query form. XML elements are indicated by folder symbols, attributes by file symbols, with



(a) XML visualization (b) A simple query

(c) A rule

Figure 34: Xing examples, adopted from Erwig [145]

those selected for output having a pen symbol. The bottom of the window contains boxes into which aggregation or quantification expressions can be entered for a selected attribute or element.

XING Xing [145], short for "XML in Graphics", takes an approach to form-based XML querying and transformation that is radically different from that of EquiX, and visually has more in common with the visual XML transformation language VXT [305] and with visKWQL in that it also employs nesting of elements.

Xing lets users draw examples of the documents they are interested in. XML elements are represented as nested boxes, with the element label printed above the box, and the element's children nested within the box together with their attributes. An example is given in Figure 34(a). Queries in Xing are based on document patterns that describe properties of the requested information and, optionally, how this information should be restructured. Patterns consist of constants, which must appear in the search result to match the pattern, and variables, which can be used to bind data for further processing.

Figure 34(b) shows a simple query that returns all `book` elements that are children of `bib` elements. When a DTD is present, the system provides the user with a list of possible child elements and attribute names. Xing allows the use of regular expressions

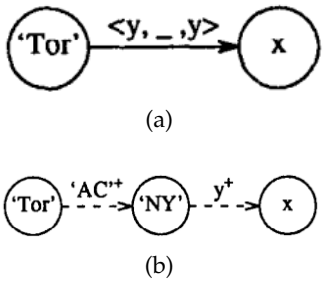


Figure 35: Queries in G, adopted from Cruz et al. [122]

to specify elements and attribute values. Asterisks can be used as wildcards, for example, all direct children of the `bib` element would be returned if "book" in Figure 34(b) was replaced by "*."

Xing also allows for the restructuring of documents with rules. Rules consists of a query and a construction part. The construction part may restructure the elements of the query part and introduce new elements.

Figure 34(c) shows a rule that retrieves all publications in a bibliography and creates a list of authors and their titles. For each author, a new element is automatically created, along with a new `titles` element containing all the titles that author has contributed to.

8.1.2 *Diagram-Based Approaches*

Diagram-based visual query languages express queries as graphs or diagrams and typically rely more on visual elements and less on text than form-based visual query languages. In diagram-based systems, information is expressed through shape, position, color, and other properties of graphical objects.

G G [122] was among the first diagram-based visual query languages, and probably the first to receive widespread recognition. Unlike other query languages at the time, including textual ones, it has complete support for recursive queries on cyclic data. What motivated G is the observation that most data consists of objects that stand in some relationship with one another, and that the graph structure formed by these relations lends itself well to visual querying.

G expresses queries as labeled, directed *multigraphs*, that is, graphs that can have multiple connections in both directions between each pair of nodes. G supports recursion through connection labels, which can be regular expressions over tuples of variables and constants. This can for example be used to express a connection of arbitrary length in a query.

Figure 35 shows two simple G queries to an airline database. Nodes represent cities with airports, edges represent flights between the cities, and edge labels indicate airlines. The query in Figure 35(a) gives Toronto as starting city and any city as end point, indicated by the variable x. The constraints on the connection between the two cities are described through a regular expression. The first and third flight may be with any airline, as long as it is the same airline in both cases, as indicated by the variable y. The second flight can be with an arbitrary airline, indicated by an underscore. The query thus searches for all cities x and all airlines y, such that x is reachable from Toronto with exactly three flights, and where the first and last flight are with the same airline.

The second example, shown in Figure 35(b), illustrates the use of connections of arbitrary length, indicated by dotted arrows and labeled with a regular expression that contain the operator +, typically used to represents an arbitrary non-zero number of repetitions. The query will return all cities x and airlines y, such that x is reachable from Toronto via New York, with an arbitrary number of Air Canada flights between Toronto and New York, and an arbitrary number of flights with airline y between New York and the destination x.

GOOD Good [299] is used to query object databases with *graph patterns*, which lend themselves well to diagrammatic representation. Apart from database querying and displaying database instances, Good also supports schema visualization.

A database schema is represented as a labeled, directed graph, with nodes representing classes of objects and edges representing relationships between these classes, or properties that can exist between objects of these classes. The types of classes are represented as different shapes: rectangles for abstract classes, which typically are usually user-defined, and ovals for system-defined basic classes like "String" or "Number." Database instances are visualized in the same manner.

Another novel feature of Good is user support during query construction. Good queries are graphs that conform to the scheme graph of the database. The system enforces this by allowing query construction only through identification, copying and duplication of nodes in the scheme graph, thus making it impossible for a user to construct an illegal pattern.

Finally, Good can act as a complete database manipulation language, supporting database browsing, querying, restructuring, updating, and meta-modeling. Good takes an approach in which a distinguished edge is added to the search pattern, to be created when the pattern is found. A query pattern can contain additional elements to represent actions, which allow the creation

and deletion of nodes and labels, abstraction, and method calls. A query pattern together with the action defined in it is referred to as an "operation", since it results not only in a query, but a graph transformation. A number of subsequent Good operations can thus be thought of as a program that gradually transforms the database.

XML-GL The object database query language G-Log [300] adopts ideas from Good, but combines them with first-order logic and introduces the notion of rules. The Language WG-Log [113] is formally based on G-Log, and applies its graph-based visualization schema and its notion of rules to query the web, more precisely HTML documents and link structures [111]. Its successor, XML-GL [91, 112], applies the principles of G-Log and WG-Log to second generation web data, i.e., XML. It was the first visual language to cover full XML.

XML-GL is motivated by the observation that the hierarchical structure of an XML document corresponds to a tree, or to a graph when references between elements are included. Querying can thus be interpreted as locating a sub-graph in a larger graph, which lends itself to an easy construction of queries by copying and pasting nodes and arcs from a schema graph. Document transformations can then easily be rendered as the construction of a new graph.

XML-GL supports a wide range of visual queries, like selecting portions of input elements based on existential conditions and comparison predicates, expressing joins, creating arbitrary new documents from selections and new elements and relationships, applying arithmetic, aggregate, and grouping functions both in the selection and the construction of elements, and computing set operations like union, difference, and Cartesian product.

The graphical data model of XML-GL consists of three building blocks, similar to those of its predecessors: objects, represented as rectangles, indicate abstract items, or aggregations of properties. Properties are represented as circles connected to the objects they refer to, which are representable values, like strings or numbers, and possess a name and a type, both represented as labels of the circle. Directed relationships, represented as arcs between objects, indicate semantic connections like containment or referencing.

Each XML-GL query consists of two graphs, each of which can in turn consist of two parts. A full query is similar to SQL's from-where-select-create view.

- The *extract part*, matching SQL's **`from`**, allows for the specification of target documents and document elements.
- The *optional match part*, corresponding to SQL's **`where`**, specifies logical conditions the target elements must satisfy.

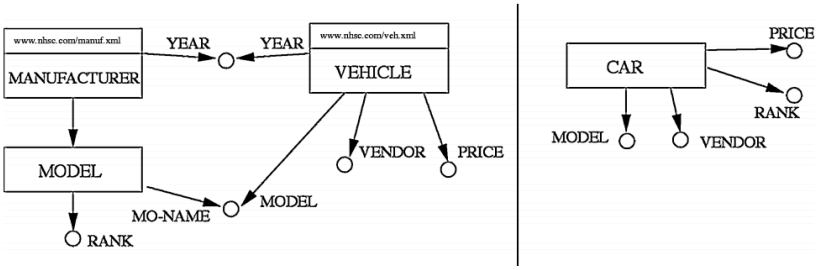


Figure 36: An XML-GL query, adopted from Comai et al. [112]

- The *clip part*, similar to **`select`** in SQL, specifies those elements from the extract part and satisfying the match part that should appear in the query result.
- The optional *construction part*, corresponding to **`create view`** in SQL, allows the creation of new elements or links and the restructuring of information local to a given element.

Figure 36 shows a query that uses all four parts. The graph on the left side consists of the extract and match parts. It specifies two XML files, the XML elements `manufacturer`, `model`, and `vehicle`, as well as a number of properties like `year` and `price`. The conditions of the match part in the example are the shared values for year and model name. The graph on the right represents the clip and construction parts. The clip part preserves the properties `price`, `rank`, `model`, and `vendor` from the extract-match part, while the construction part introduces a new element, `car`, to which the properties are assigned. The query thus retrieves all vehicles built in the same year in which a manufacturer built a car, and that have the same model name as some model the manufacturer built. If such a pattern is found, a new `car` element with the appropriate properties is created and returned as the query result.

VISXCERPT visXCerpt [50] is a visual language that is based on Xcerpt [320] and that served as an inspiration for the development of visKWQL. It is discussed here because of its query capabilities, however, like most XML transformation languages, it transcends the line between query and programming languages since it is Turing-complete.

One feature that visXcerpt shares with the KWQL Query Builder is its realization in DHTML, meaning that users can edit and run visXcerpt programs from within their web-browser and without a special editor or run-time environment.

Like visKWQL, but unlike the other visual languages presented here, visXcerpt does not translate between the visual and textual

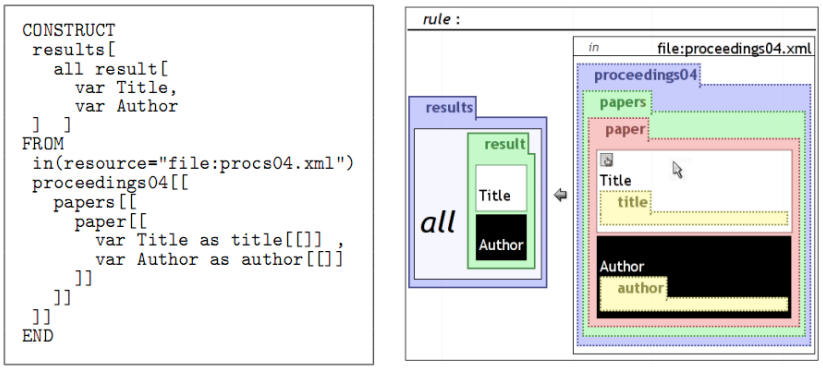


Figure 37: An Xcerpt program and its rendering in visXcerpt, adopted from Berger et al. [50]

representations of a program, but merely renders textual Xcerpt queries.

A program is edited by selecting building blocks within a template and copying them to the program. This has the advantage that the system can prevent the insertion of the templates at the position specified by the user if the operation would lead to a syntactically incorrect program.

Figure 37 shows a simple program written in Xcerpt (in its abbreviated syntax) and its rendering in visXcerpt.

The program itself consists of a rule with a selection and a construction part. In the query body, the **`FROM`** part in the textual and the right part in the visual program, a file named "proceedings04.xml" is read and then queried for an element `proceedings04` that contains an element `papers` which itself consists of `paper` elements. If such elements are found, the program adds the values of their `author` and `title` attributes to the variables `Author` and `Title`, both defined as lists. In the construction part of the program, a new XML element `results` is created. The **`all`** operator then creates child elements for it by taking each pair of a title and an author from the list variables, and constructing a new `result` element that holds the respective values.

NITELIGHT NITELIGHT [315] is a visual query system consisting of the vSPARQL language and an interactive editing environment for ontology navigation. Unlike other visual query systems, NITELIGHT emphasizes expressiveness over ease of use, and it is intended for users with some prior SPARQL experience.

vSPARQL represents SPARQL triple patterns as graphs, with nodes representing the object and subject elements. Shape and color of a node are used to express element type, URIs, literal value, or variable. Object and subject nodes are connected by a labeled edge, representing either a predicate or a query variable.

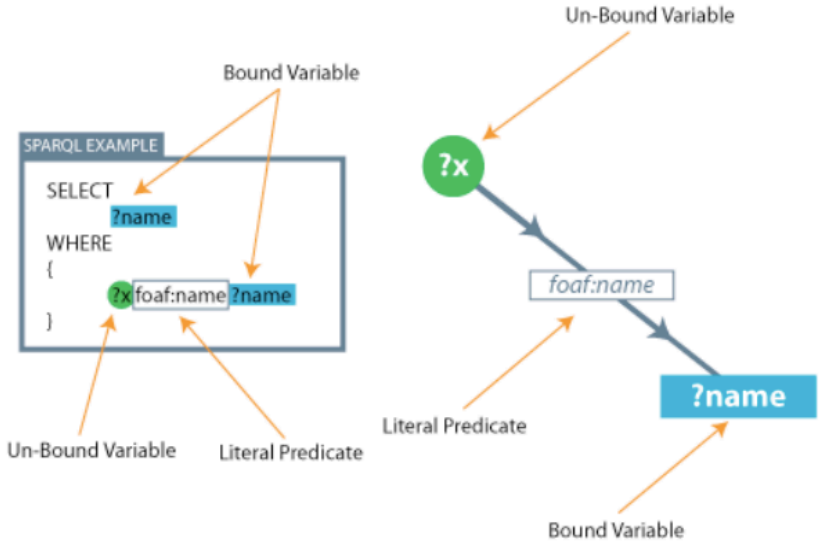


Figure 38: A triple pattern in SPARQL (left) and NITELIGHT (right), adopted from Russell and Smart [315]

Figure 38 shows the NITELIGHT representation of a SPARQL triple pattern. Multiple triple patters are represented by graphs with additional nodes and connections, connected by nodes representing shared variables or values. The ordering of variables and triples can be represented by numbers next to the node labels.

SPARQL graph patterns, collections of triple patterns matched against the entire RDF graph, are represented graphically by grouping triples within boxes. Variables are not shared, but are local to a graph pattern and could be bound to different values. Different colorings of the boxes, and connections between them, allow for the representation of optionality and unions.

The NITELIGHT prototype consists of four parts:

- the *Query Design Canvas*, a graphical query editor that allows the user to move elements freely, edit them through context menus, and zoom in and out;
- the *SPARQL Syntax Viewer*, a dynamically updated display of the textual SPARQL query corresponding to the graphical query, similar to the output view of the KWQL Query Builder;
- the *Ontology Browser*, a text-column based display of the source ontologies and their classes; and
- the *Query Results Viewer*, a table-based display of query results.

RDF-GL RDF-GL [195] is another visual query language for RDF data based on SPARQL. It employs a visualization that is, although graph-based, very different from that of NITELIGHT.

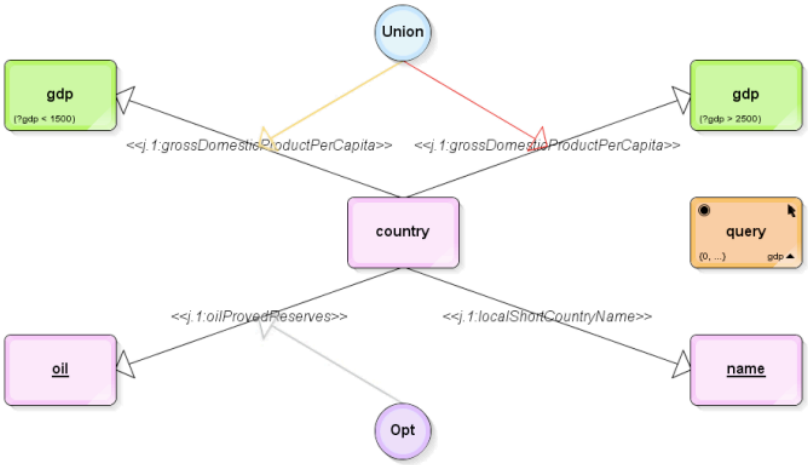


Figure 39: An RDF-GL query adopted from Hogenboom et al. [195]

Queries in RDF-GL are represented using boxes, circles, and arrows. Orange boxes, called *result boxes*, contain information about the execution of a query, like query type and result order. Pink boxes represent the subject and object elements of a SPARQL triple. Green boxes represent filtered subjects or objects. Blue circles can be connected to elements to represent union, while purple circle represent optionality. Black arrows represent SPARQL predicates, while gray arrows represent optional statements. Yellow and red arrows are used to indicate the first and second part of a union.

Figure 39 shows a simple query in RDF-GL using the different elements. The lines under "oil" and "name" indicate that these variables are to be included in the output. The example query returns the name and, if present, the oil supply, of every country whose gross domestic product per capita is smaller than 1500$ or greater than 2500$.

8.2 DESIGN GOALS

The goal of visKWQL is to provide a visual rendering of KWQL that serves as an alternative to the textual language, and is particularly suited for users who are unfamiliar with query languages.

A goal shared by all visual query languages is to make querying easier, both for users without prior experience with query languages and for intermediate or even advanced users. KWQL, however, was itself designed to combine ease of use with high expressivity. visKWQL therefore stays close to the structure of KWQL queries, to preserve the characteristics of KWQL and to make it easier for users to switch between KWQL and visKWQL. This is in contrast to many other visual query languages, espe-

cially graph-based ones, which often differ radically from their textual counterparts.

The following are desirable characteristics of a visual equivalent of KWQL:

Full expressiveness visKWQL must be able to express all syntactically valid KWQL queries and the visual rendering should not sacrifice expressive power.

Browser compatibility A visual KWQL query editor should be realized as a web application, and should not require the installation of special software or browser plug-ins. This can be achieved using DHTML, with HTML and CSS for the presentation and Java Script for the program logic and user interaction.

Round-tripping Users should not have to decide in advance which formalism, textual or visual, they use to create a query, but should be able to switch between both at any time. Round-tripping means that users are not restricted to visually create and edit a query and then execute it, but that they can edit both the visual and the textual representation of a query and see changes in one representation reflected in the other. For example, the user should be able to start with a simple textual query, add an element to it in the visual representation, and finally edit a value in the textual representation before evaluating the query.

User support Users should be supported at all times during the query construction process. This includes both the language representation and editing features, as well as context-sensitive support to create syntactically correct queries and guidance that enables even inexperienced to create useful queries right from the start.

Easy editing The mechanism for creating queries should be conceptually simple and easy to understand. The majority of computer users are familiar with drag-and-drop techniques universally employed by modern operating systems. Drag-and-drop is therefore well suited as a mechanism for query editing and should be supported by the editor.

Information hiding More complex visual queries can often become large and occupy a big portion of the screen. The editor should therefore support some form of information hiding, to temporarily hide parts of a query.

Self-explanatory system Users should not be required to learn the syntax of visKWQL in order to be able to use the query editor. The editor should therefore provide tooltips for all

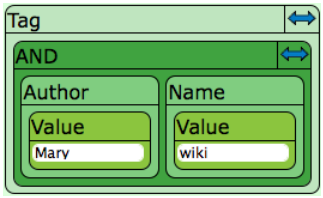


Figure 40: A visKWQL query

elements, especially the KWQL structures that make up a query, to help users build valid and useful queries.

Error prevention The system should prevent or correct syntactic errors where possible, for example by preventing the user from dropping a KWQL qualifier on a resource that does not possess this qualifier. It should also inform the users about errors, like a string containing invalid characters, or problems, like an **AND** operator with only one argument, and correct them automatically whenever possible. When the system corrects a problem, for example by removing the **AND** node, it should inform the user of this to support her in learning to use visKWQL.

8.3 LANGUAGE AND EDITOR FEATURES

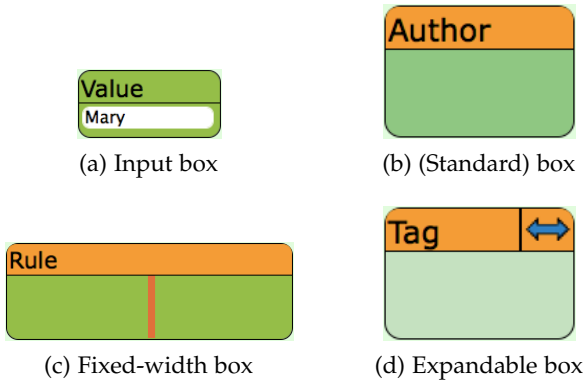
We will now give an overview of how the design goals described in the previous section are realized in the visKWQL system.

VISUAL FORMALISM visKWQL uses a form-based approach. All KWQL elements, including resources, qualifiers, and operators, are represented as boxes. Resource-value or qualifier-value associations are represented as nestings. Boxes consist of a *label*, in which the name of the represented KWQL element is included, and a *body*, which can hold one or more child boxes. This approach has several advantages: it stays close to KWQL's textual structure, keeping visKWQL simple and making it easy to translate between the two representations; it also lends itself well to rendering in HTML.

Figure 40 shows an example of a visKWQL query corresponding to the textual KWQL query `tag(author:Mary AND name:wiki)`, which retrieves content item that Mary has tagged with "wiki" or that contain a fragment or link with such a tag.

The following types of boxes, also shown in Figure 41, are used in visKWQL to represent KWQL elements:

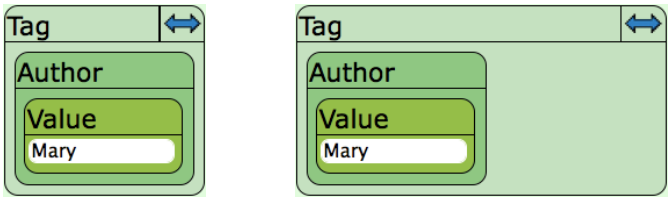
Input boxes represent keywords and variables. Their body contains a text field where the user can enter a value.



(a) Input box (b) (Standard) box

(c) Fixed-width box (d) Expandable box

Figure 41: visKWQL box types



(a) Unexpanded (b) Expanded

Figure 42: Expandable box containing a child box

(Standard) boxes consist of a label with the name of the box, and an empty body that can hold a child element. They are used to represent qualifiers and the unary operator **`NOT`**.[1]

Fixed-width boxes are used for elements that have a fixed number of children that is greater than one, like a rule or the operators **`ALL`** and **`SOME`**. For example, a rule must have two children, a head and a body.

Expandable boxes are used to represent KWQL items that can have a variable number of children. To save screen estate, such boxes normally only show their child boxes and no free space like fixed-width boxes do. They contain a resize button within their label, indicated by a two-headed arrow. When clicked, this button expands the box by a free space onto which another child box can be dropped. Figure 42 shows an expandable box before and after the resize button has been clicked.

Resources, qualifiers and operators are all represented as boxes. For resources, the nesting of a child box corresponds to the addition of an expression within the parentheses in a KWQL query,

1 The orange labels in Figure 41(b) and the following figures are not particular to the box types, but constitute a warning that a box is empty. Warnings are explained in detail below.

following the pattern `resource (child1 child2 ...)`. For qualifiers, the child box corresponds to the expression after the colon in a query, that is, a value term. For operators like **`AND`** and **`OR`**, the children include both the expressions preceding and following the operator in the textual query (as in `(child1 AND child2)`, for example), similar to a pure graph-based visualization.

INTERACTION STRATEGY The mode of interaction with the query builder to construct or edit a query should based on principles that are consistent and easy to learn and understand. This is realized by employing the familiar drag-and-drop mechanism. All actions in the editor apart from entering text into text fields consist of drag-and-drop or left-click operations. There are no context menus or other interaction modes that might confuse the user.

The available user actions based on drag-and-drop or left mouse clicks are as follows:

Element creation To create a new element, the user selects the appropriate element type from a simple drop-down menu. The element then appears on a free position within the work area.

Element deletion An element is deleted by dragging it outside the work area and dropping it.

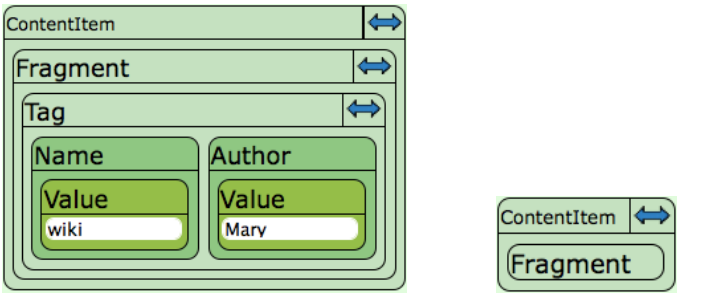
Child addition When a box is dropped on the body of another box, it will be added as a child element of this box if the box nesting is valid.

Child removal To remove it as a child, a box can be dragged out of its parent box and dropped on the work area or the body of another box.

Type switching When a box is dropped on the label of another box, and the two boxes represent syntactically equivalent elements (for example **`AND`** and **`OR`**, or two qualifiers with the same underlying data type such as **`text`** and **`title`**), the two boxes switch their types. This functionality makes it easy to change the type of a box without having to remove and re-insert all its children in the process.

Text switching When a value box is dropped on the label of another value box, their text values are switched. When a variable box is dropped on the label of another variable box, the variable names are switched.

Information hiding Clicking the label of a box collapses the box and hides its body. Clicking again expands the body. Boxes can be thus be reduced to show only their label, making



(a) No hiding (b) Fragment collapsed

Figure 43: Information hiding in visKWQL

complex queries more comprehensible and preventing them from becoming too large to be displayed on the screen. The system also saves the state of all child boxes, so when a child box is collapsed and the user clicks twice on the label of the parent to hide and restore its body, the child box will still be collapsed. Figure 43 shows a simple query before and after a click on the label of the fragment box.

ROUND-TRIPPING Round-tripping allows users to edit a query in both representations, visual and textual, at the same time. The side-by-side display of both representations offers the additional advantage of helping users to learn KWQL by creating queries in visKWQL.

The first part of round-tripping is the mapping from visKWQL to KWQL. For this, the system translates a visual query to a textual one after each action by the user, or shows an error message if the query is currently not syntactically correct. For the inverse direction, the mapping from KWQL to visKWQL, the system includes a KWQL parser. The user can edit the current textual query or enter a new one at any time. On the click of a button, the system will parse the textual query and update the visual query accordingly, or generate an error message if the textual query is invalid.

USER GUIDANCE Support throughout the query construction process is one of the key aspects of visKWQL and the query builder. For users who are new to the system in particular, the query builder provides a *hint pane*. This text area below the editor workspace displays additional information and provides hints if there is a problem with a query.

Another step towards the goal of making the system self-explanatory are tooltips that provide information about different elements on the screen. These tooltips come in two variants: The first variant gives explanations of the elements of a query. Tooltips of this kind are displayed in the tooltip pane at the bottom of the

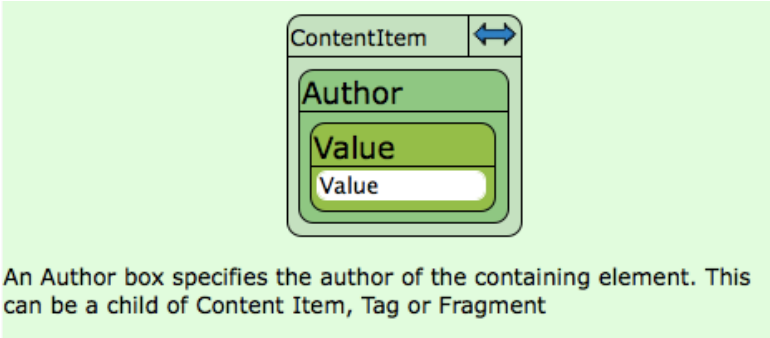


Figure 44: The KQB tooltip pane

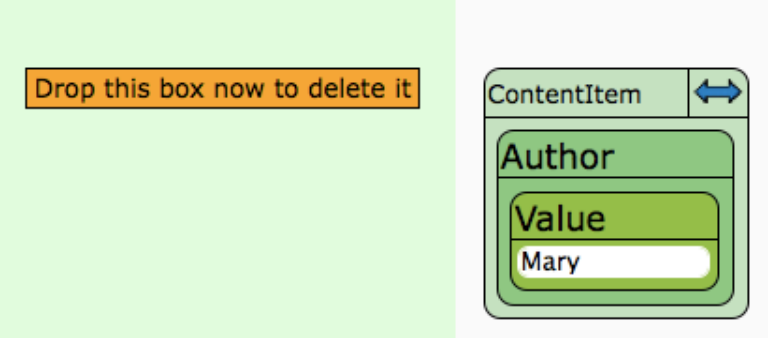


Figure 45: A KQB tooltip

editor workspace whenever the user moves the mouse over a box, as illustrated in Figure 44 for an author box.

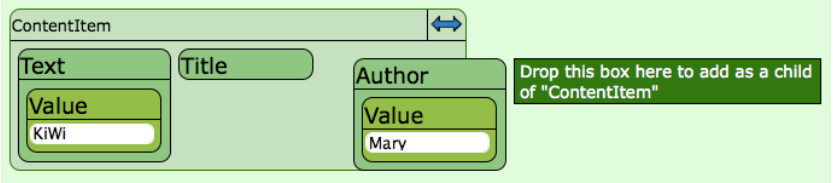
The second kind of tooltip provides information about the current state of a box, like the validity of dropping actions while a box is being dragged, or warnings or error messages concerning the box the mouse currently hovers over. Tooltips of this kind are displayed next to the box in question. Figure 45 shows a box that is being dragged outside the green work area. Dropping the box there would cause it to be deleted, as indicated by the tooltip.

Tooltips are further displayed when a user is about to select a box to be added to the workspace from a drop-down menu. In this case, the tooltip is displayed next to the mouse pointer when it hovers over a menu entry, and provides an explanation of the element corresponding to this entry.

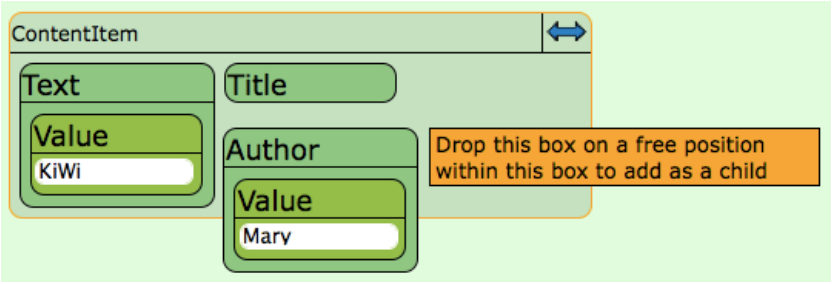
ERROR PREVENTION Another way in which users are supported during query construction concerns the prevention of syntactic errors. To this end, the KQB uses two devices to inform the user about the validity of actions and the state of query elements: colors and tooltips.

Colors are used to indicate different states: green signals a valid action, orange a warning, and red an error.

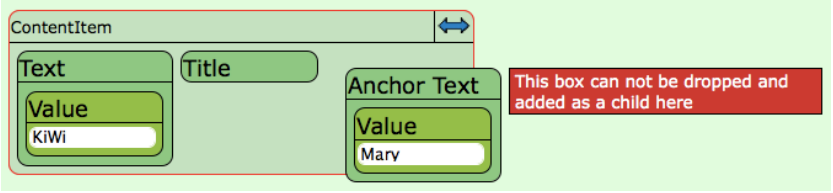
To prevent syntactic errors, for example, the Query Builder only allows the creation of valid box nestings, and prevents the



(a) Valid drop location



(b) Invalid drop location within a valid parent box



(c) Invalid drop location

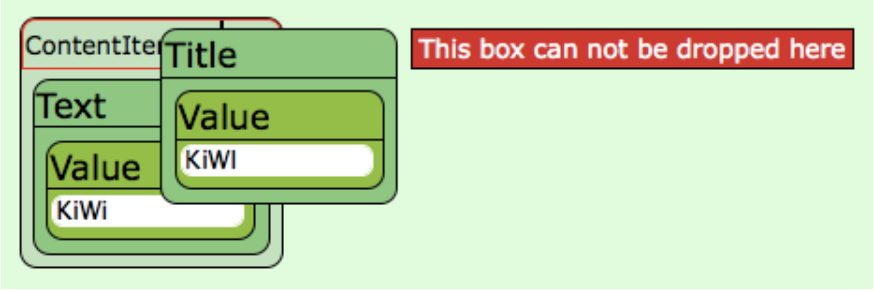
Figure 46: Dragging over a box

addition of child boxes where they are not allowed, as well as type switching between nodes of different types. When adding a child to a box, there are three possible outcomes, all of which are conveyed to the user through color and tooltip.

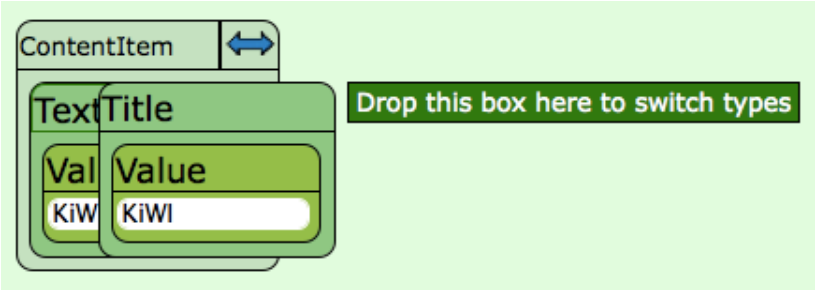
Figure 46(a) shows a valid operation: an **`author`** box is dragged over free space within a content item. **`author`** is a valid qualifier and thus valid child of a content item, and may be dropped. This is indicated by a green border around the content item, and a green tooltip.

Figure 46(b) shows a warning: the **`author`** box is dragged over the content item box, which is a valid parent. However, the **`author`** box is currently in an invalid position within the content item, which is displayed to the user through an orange border around the parent box and an orange tooltip. The position in question is occupied by a **`title`** child, and it would be unclear if **`author`** should become the second or third child of the content item. With the chosen approach, the user sees that the **`author`** box cannot be dropped in the current position, and can either drop it to the right of the **`title`** child, or first move the **`title`** child to the right and then drop **`author`** between **`text`** and **`title`**.

Figure 46(c) finally shows a situation in which a drop would lead to an invalid nesting. In this situation, the user has dragged



(a) Dropping not allowed



(b) Dropping allowed

Figure 47: Dragging over a box label

an **`AnchorText`** qualifier over a content item, where this qualifier is not allowed. This is indicated to the user by a red border around the parent box and a red tooltip. If the user drops the **`AnchorText`** box in this position, the system will not allow it to be added as a child of the content item box and instead return it to the position where it was picked up.

Unchecked type switches could also lead to syntactic errors in the form of invalid nestings, and their validity is therefore checked when the user drags a box over the label of another box. Similar to the case where a box is dragged over the body of another box, the system informs the user through a colored border and tooltip whether a drop action is allowed in the current situation. In this case, however, the colored border is put only around the label and not the whole box.

In the situation shown in Figure 47(a), for example, dropping is not allowed since **`title`** and **`ContentItem`** are a qualifier and a resource and cannot switch. In the situation of Figure 47(b), on the other hand, dropping is allowed, because **`text`** and **`title`** both are string-valued qualifiers.

ERROR REPORTING AND CORRECTION Not all syntactic errors in a query can be prevented by the editor. For example, it constitutes an error if the construction part of a rule uses a variable that was not defined in the query part. To prevent the user from making such an error, however, one would have to

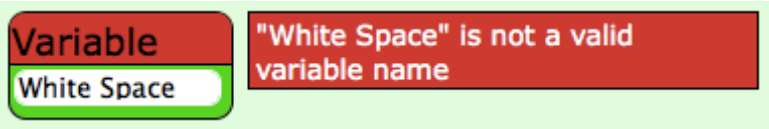


Figure 48: Box containing an error

require that the head of a rule is always specified after the body. This would limit the flexibility of the query creation process, and would stand against the design goal of high user-friendliness.

The KWQL Query Builder therefore allows the user to take actions that lead to errors of this kind. However, it detects them, and informs the user about the problem. The types of errors detected by the query builder include the following:

- Variable names or keywords containing invalid characters like white spaces
- Empty strings
- Invalid value formats
- Misplaced operators (for example, operators **`ALL`** and **`SOME`** may only occur in the head of a rule, **`OR`** only in the body)
- Undefined variables in the head

When a query contains an error, this problem is indicated to the user through a variety of means:

- Instead of the textual KWQL query, the text field shows the error message of the first error in the query.
- The labels of all boxes containing an error are colored in red.
- When the mouse is moved over a box containing an error, the error message associated with that box is displayed in a red tooltip, as shown in Figure 48.
- The hint box below the workspace displays a more detailed explanation of the first error in the query, and let the user know how to correct it. In the example of Figure 48, the hint pane would display a message that variable names may not contain white spaces.

To help the user in locating an error, especially when parts of the query are hidden, the query editor also colors the labels of all parents of a box containing an error in red, and displays a tooltip when the mouse is moved over one of the parents. This behavior is illustrated in Figure 49.

Some errors that are less severe are corrected automatically by the query builder. This category includes empty boxes, that is,

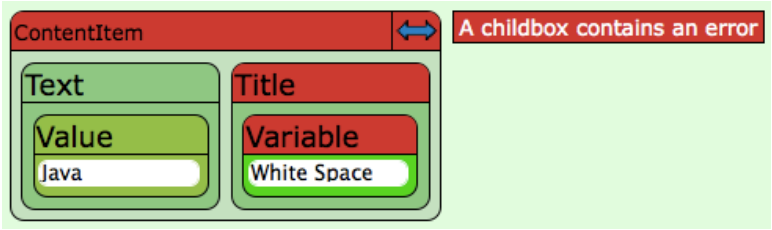
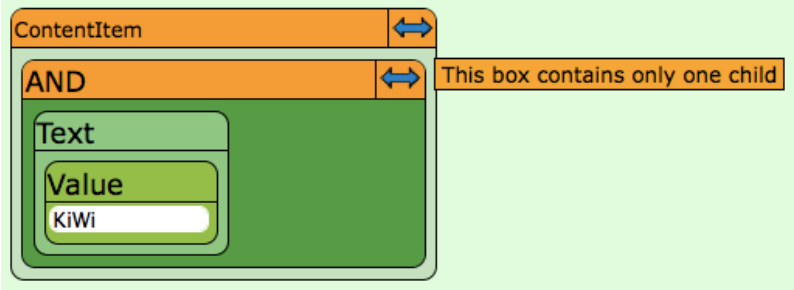
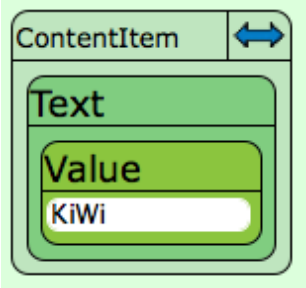


Figure 49: Child box containing an error



(a) Query containing a problem



(b) Corrected query

Figure 50: Problem correction in visKWQL

boxes which contain neither a child nor a value or variable, and **AND** and **OR** boxes with only one child. Empty boxes are removed completely. In the case of **AND** and **OR** boxes with only one child, the box itself is removed, but its child is appended to the parent box. Figure 50 shows an example of a query containing an **AND** box with only one child, as well as the corrected version of the query.

Just like for errors and warnings during drag-and-drop, errors are reported by red labels and tooltips, while orange is used for less severe problems which do not require the user to change the visual query to get a correct textual query. When a problem of the latter kind is detected, the hint pane will display an explanation of the problem and how to correct it, and also inform the user that the offending box is being ignored for the generation of the textual KWQL query.

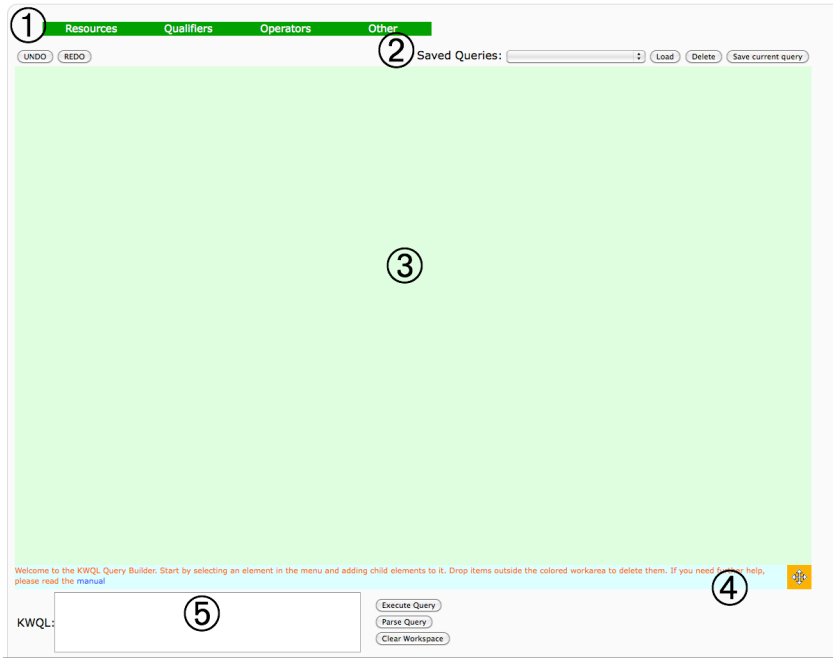


Figure 51: The KiWi Query Builder

8.4 VISKWQL QUERIES IN PRACTICE

When the query editor initially loads, the workspace (③ in Figure 51) is empty. The user can start either by writing or pasting a textual KWQL query into the text box (indicated by ⑤), by selecting an element from the menu bar (indicated by ①), or by loading a query saved earlier (using the menu indicated by ②).

To create a query that retrieves all content items where Mary is an author, the user simply clicks on "Resources" in the menu bar and selects "content item." A new content item box will appear in the workspace. As only content items authored by Mary should be matched, the qualifier "author" must next be selected from the menu and dragged into the content item element. Finally, the value "Mary" is entered using the keyboard. The text box displays the textual version of the current query, `ci(author:Mary)`.

Further children can be added to the content item element by clicking the blue double arrow in the upper right corner of the box. This will increase the size of the box to make room for another child, for example a **`text`** qualifier box with the value "XML" to limit the query answers to content items authored by Mary and containing "XML" in the text.

If the text is altered—for example to reformulate the query to `ci(title:Mary)` and find content items where "Mary" appears in the title—and "Parse query" is clicked afterwards, the visKWQL query is adjusted to reflect the change. The same result could be achieved by dragging the qualifier box out of the content item box and replacing it with a **`title`** qualifier. Type switching, by

dropping a **title** qualifier box onto the label of the text qualifier box, offers a way in which this can be achieved without the need to re-enter the value "Mary."

The query can be saved for future use by clicking on the button labeled "Save current query." A name for the query can be entered and will consequently appear as one of the choices in the drop-down menu listing the saved queries. Currently, queries are saved on a per user basis.

The query can be deleted simply by dragging it out of the workspace area colored in light green. Any change to the query, including the addition and deletion of elements, can be reverted using the "undo" button in the upper left corner of the editor.

Evaluation of a query is finally triggered by pressing the button "execute query" next to the KWQL text box.

8.5 IMPLEMENTATION

visKWQL and the KWQL Query Builder are implemented in DHTML (Dynamic Hyper Text Markup Language), using HTML for the KQB page structure, JavaScript for the dynamic and interactive parts, and CSS for the graphical presentation. As a consequence, the system runs almost completely on the client side, within the user's web browser.

Apart from query evaluation, the only parts of the KQB that need to interact with the KiWi server is the code that loads and saves visKWQL queries on the server. It is implemented in Java, as an action in the Seam Framework[2] upon which KiWi is built. The KQB interacts with this server-side Java code via AJAX (Asynchronous Java Script and XML), which enables client-server communication between Java and Java Script. The Seam action includes functions that provide the KQB with a list of all saved queries, load or delete one of the queries, or save a new query to the list. Since the KQB supports full round-tripping, only the textual KWQL query needs to be saved on the server, and a KWQL query is converted to visKWQL only when it is loaded. The fact that no visual information needs to be saved reduces the amount of space required on the server.

For query execution, the KQB simply uses the KWQL query evaluation engine (see Chapter 10) and renders the returned results below the editor work space.

Thanks to the closeness of the visual and textual representations, the translation from visKWQL to KWQL can be seen as a serialization of the visual query. In most cases, each box nesting corresponds to a value, qualifier or resource term, making the translation straightforward.

2 http://seamframework.org/

The inverse direction, however, the translation from KWQL to visKWQL, is more complex and requires a KWQL parser. To this end, visKWQL uses the KWQL grammar specified in ANTLR that is also employed by the query evaluation module (see Section 10.3).

The design of the visKWQL editor makes it easy to change its appearance and localization. All graphical data is contained in a single CSS file, all text displayed to the user in a single text file. None of the values are hard-coded and changing the color scheme or language of the visKWQL editor is as easy as editing one of the files.

9

EXPERIMENTAL EVALUATION: THE KWQL USER EXPERIENCE

This chapter describes the setup and results of a user study performed to evaluate the suitability of KWQL and visKWQL for querying tasks in the KiWi wiki. The goal of this experimental evaluation is to gain a first insight on how the query languages are perceived by users and how easy it is for them to learn to write and understand KWQL and visKWQL queries. With respect to both of these aspects, a question of particular interest is whether the results differ between (1) users with varying amounts of previous experience in the area of query languages and social semantic software and (2) between participants using textual KWQL and those using its visual rendering.

The present study does not aim to be a comprehensive evaluation of KWQL and all features and design choices involved; rather, it should be considered a first exploration into evaluating the query language which sheds light on the questions raised above, but which also raises further questions to be explored in follow-up studies. While the goal of this study is to provide answers and insights on the usability of KWQL and visKWQL, it also serves the purpose to identify particular aspects of the languages and their usage that warrant more detailed evaluation.

The evaluation discussed here was performed as a single-session experiment where participants were given a short introduction into the KiWi wiki and, depending on the group they had been assigned to, KWQL or visKWQL, and were then asked to formulate queries ranging from simple and vague to precise and expressive. In a second task, participants were confronted with KWQL or visKWQL queries which they then translated into natural language descriptions of the data selected by the queries. Throughout the process, participants were encouraged to write down their thoughts and opinions on KiWi, the query language and individual tasks. This controlled setting is well-suited for gathering impressions and comments and assessing to which extent participants can quickly learn how to write and understand simple to advanced queries. Further, it allows to examine the types of mistakes that users make and therefore to conclude which elements of the query language are particularly hard to master. The results can also easily be compared between participants with different amounts of previous knowledge and between participants using KWQL and visKWQL.

However, to limit the scope of the experiment and focus on the aspects outlined above, several factors are intentionally not treated in this first evaluation:

- Participants were only given a short amount of time to familiarize themselves with the KiWi wiki and KWQL or visKWQL. Their learning was thus not gradual and self-paced, but the speed at which they were introduced to the subject matter was pre-determined. In particular, participants learned about all of the query languages' features at once using the same amount of instruction, meaning that they were not given the choice to only learn as much about KWQL or visKWQL as suited their needs or to read further material when something was unclear to them.
- The queries that the participants had to write and understand were pre-defined. As such, they do not reflect an individual participants' information needs or degree of knowledge about the wiki. This means that the experiment can only determine how well a participant performs in translating a query intent into a query or vice versa, but not which queries he would pose when using KiWi. Additionally, when a participant fails to correctly formulate a query, the reason may not be his lack of knowledge of the query language, but may rather indicate a lack of understanding the data itself. On the other hand, in a realistic scenario, a user would likely just formulate a less specific query if he was unsure about the exact structure of the data that he wants to select.
- Since the experiment consisted only of a single session, no development and progression over time could be recorded.
- Participants could not freely decide whether they wanted to use KWQL or visKWQL, but were assigned one of the query languages. As such, participants that would have preferred to use KWQL may have been forced to use visKWQL and vice versa. Additionally, it is not possible to conclude what the number of participants in each group would have been if participants had been allowed to decide whether they wanted to use the textual or visual version of KWQL, that is, whether there is a preference for one over the other.
- Finally, to not overstrain participants with a big amount of material and to ensure that all participants spent the same amount of time learning, round-tripping was not part of the evaluation.

The following sections describe the experimental setup, and outline and discuss the results of the study.

9.1 EXPERIMENTAL SETUP AND EXECUTION

Twenty-one participants were recruited via an internet forum aimed at LMU Munich's computer science students and via announcements in several computer science lectures at the university. Participants were between the ages of twenty and thirty-three with the average age being 23.95 years. Sixteen of the participants were students of computer science or media computer science, one was a researcher in a non-related area of computer science, and four participants were students of subjects other than computer science. Participants took part in the study in individual sessions which lasted about 90 minutes; each participant was rewarded with thirty-five Euros.

Participants were asked to fill out a questionnaire about their previous experience in areas relevant to semantic wikis and KWQL. Nine different topics such as the semantic web, tagging and XML were given and users rated their experience with each on a scale ranging from 1 (no knowledge at all) to 5 (expert-level knowledge). Participants were also asked which programming and query languages they knew.

Each participant was randomly assigned to either the KWQL or the visKWQL group. The first assessment was then followed by an introductory phase were participants familiarized themselves with the KiWi wiki and KWQL or visKWQL. They were supplied with written, example-driven introductions describing both the wiki and all features of the query language. Access to an installation of KiWi pre-loaded with sample data on the KiWi project was provided so that participants could try out the system while reading the introductions. Like all information material and questionnaires in the study, these texts were written in German; they can be found in Chapter B in the appendix. Participants in the KWQL-group were not made aware of the existence of a visual version of the language, while the text on visKWQL mentioned KWQL and visKWQL's round-tripping capabilities, but did not explain them in detail.

Participants were instructed to take approximately thirty minutes to work through both introductory texts at their own pace. After each of the two introductions, they answered a questionnaire with a number of open questions about their impressions and thoughts on what they had read.

The remaining hour of the session was dedicated to the actual study and its two tasks —query creation and query understanding.

The KiWi wiki used for the experiment contained data imported from a wiki on the TV-Show "The Simpsons"[1]. The motivation for using this data for the study is that it is not only

1 The wiki can be found at http://simpsons.wikia.com/

readily available, but that it also deals with an entertaining topic that many participants are likely familiar with.

To reflect the collaborative process of content creation and annotation, several user accounts were created in the KiWi wiki. Each account was used to compose a number of content items containing text from the Simpsons wiki. These content items were then annotated with tags. The final dataset, used in this study, consisted of 653 content items.

Participants were provided with instructions on the experiments and information about the dataset used. The instructions contained a text which introduced the application scenario and the semantics behind the organization of the wiki data; it is reproduced here in its English translation:

> Seven friends, Estelle, David, Milton, Ian, Doris, Ashley, and Betsy, have decided to create a KiWi Wiki about their favorite TV show, The Simpsons. After some time has passed, they have gathered a lot of knowledge in the wiki, and several conventions for organizing the data have developed.
>
> Most content items in the wiki describe characters or locations in the Simpsons universe or episodes of the TV show. In order to be able to distinguish these different types of content items, they are tagged with "character," "location" or "episode" respectively.
>
> Content items about characters have the full name of the character as a title, content items about episodes the name of the episode.
>
> The friends have set the goal to create an additional trivia content item for each character described in the wiki. These trivia content items are nested in the content item describing the character and should be tagged with "trivia."
>
> Every user also indicates which episodes and characters he likes best by tagging the respective content items with "favorite." Additionally, all quotes from the show are represented as fragments that are tagged with "quote."

In the query creation task, users were given 45 minutes to use KWQL or visKWQL to find answers to questions (given in natural language) about the data in the wiki. In total, there were ten assignments of increasing difficulty. To solve an assignment, participants were asked to write down the number or titles of matching content items as well as the query used.

The reason for this is that while a query may have been underspecified and therefore only have returned a superset of the content items that answer the questions, participants could browse

	Select content items that...	Correct answer
1	contain "Abraham" or "Abe"	`Abe OR Abraham`
2	contain "Homer" and "Bart" but not "Lisa"	`Homer AND Bart NOT Lisa`
3	describe a location	`ci(tag(name:location))`
4	describe Betsy's favorite characters	`ci(tag(name:favorite author:Betsy)tag(name:character))`
5	link to the content item that describes Carl	`ci(link(target:ci(title:Carl tag(name:character))))`
6	contain fragments tagged with "quote"	`ci(fragment(tag(name:quote)))`
7	describe episodes with "Lisa" in the title	`ci(tag(name:episode)title:Lisa)`
8	do not contain a nested content item tagged "trivia"	`ci(NOT(child:ci(tag(name:trivia)))`
9	describe an episode which is among both Estelle's and Ian's favorites	`ci(tag(name:episode)tag(name:favorite author:Ian)tag(name:favorite author:Estelle))`
10	contain a link to a content item which has at least one tag with the same name	`ci(tag(name:$t)link(target:ci(tag( name:$t))))`

Table 7: Questions and solutions for task 1

the list of query answers to determine the true answers to the query. For example, instead of using the solution for assignment 7 given in Table 7, `ci(tag(name:episode)title:Lisa)`, a participant could also have posed the query `ci(title:Lisa)` and consequently used the answers to that query to count how many of them were tagged with "episode." In the following, unless indicated otherwise, when we speak of the *answer* to an assignment, we mean the query written by the participant.

Participants were instructed to tackle the assignments sequentially, and to only skip an assignment if they were convinced they were not going to find an answer. They were also told not to be concerned if they did not succeed in solving all ten assignments within the given amount of time. Table 7 gives a shortened English version of each of the ten assignments together with their answers.

The second task was the inverse of the first — given six KWQL or visKWQL queries of intermediate to advanced complexity, participants were asked to describe the underlying query intent, that is, the common characteristics of the content items selected by each query, in natural language. Participants were given fifteen

	Query given	Correct answer
1	`ci(tag(name:episode)France)`	Episodes whose description contains the term "France"
2	`ci(tag(name:character)NOT(tag(name:favorite)))`	Descriptions of characters who are not anyone's favorite
3	`ci(tag(name:$a)tag(name:$b)tag(name:$c))`	Content items that have at least three tags
4	`ci(link(target:ci(title:Carl)))`	Content items that link to content items which contain "Carl" in their title
5	`ci(Carl link(target:ci(title:Lenny)))`	Content items that contain "Carl" and that link to content items which contain "Lenny" in their title
6	`ci(URI:$a tag(name:character)link(target:ci(link(target:ci(URI:$a))tag(name:location))))`	Character content items that link to a location content item that links back to them

Table 8: Questions and solutions for task 2

minutes to complete the task. Table 8 shows the queries and correct descriptions for all of the questions.

All assignments, as in task 1, were accompanied by a questionnaire that assessed, among other factors, how hard participants found it to answer the question and whether there were particular aspects that they found difficult. In order not to distract participants from the tasks, answering these additional questions was optional.

Finally, after participants had spent one hour on the two tasks, they were asked to fill out one final questionnaire that asked about their opinions on the query language and assessed how

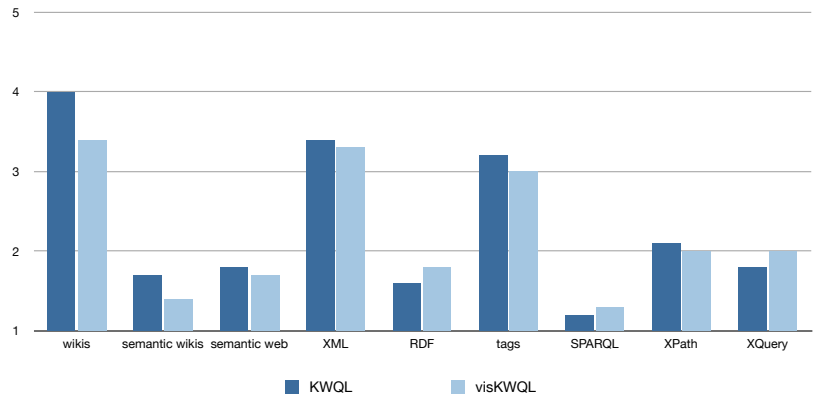


Figure 52: Previous knowledge of relevant concepts and technologies among participants in both groups

	KWQL	visKWQL	overall
novice	5	6	11
advanced	5	4	9
overall	10	10	20

Table 9: Number of participants per group

they had changed now that participants had used the query language for an hour.

9.2 RESULTS

A total of twenty-one participants completed the first task of the user study, eighteen of them also completed the second task; three participants could not take part in the query comprehension task for technical or organizational reasons.

One participant was found to perform far below the other participants in the study, an examination of this participants' written answers indicated problems with understanding the introductory texts —likely due to a language barrier. Consequently, this participant's data was excluded from further processing. Overall, the evaluation of the results is thus based on twenty sets of answers for the first task and seventeen sets of answers for the second task.

Figure 52 shows participants' self-assessed average knowledge of various areas relevant to KWQL per group. As can be seen, results overall are very similar for the two different groups with wikis, XML and tags being the only concepts that participants are somewhat familiar with. The concepts pertaining to semantic web technologies are unknown to most users with an average value of 2 or below.

Seven members of the KWQL group and four of the participants in the visKWQL group knew at least one query language, while all participants except for two members of the visKWQL group knew one or more programming languages. Across both groups, SQL and Java were the most frequently mentioned query and programming language respectively.

The findings described in the previous paragraph were used to further divide participants into two groups based on their previous knowledge of query languages, social software and semantic web technologies. All participants who indicated that they did not know any query languages or who had never heard of at least four of the nine concepts (that is, who rated their knowledge of those areas as 1 out of 5) were considered to be novice users with little relevant experience, while the other participants were considered to be advanced users. In the following, participants will

	KWQL	visKWQL	overall
novice	7.8	8.2	8.02
advanced	8.6	8.0	8.34
overall	8.20	8.12	8.16

Table 10: Average number of questions (out of 10) answered

	KWQL	visKWQL	overall
novice	4.2	3.4	3.76
advanced	6.8	7.5	7.11
overall	5.5	5.04	5.27

Table 11: Average number of questions (out of 10) answered correctly

frequently be referred to as "novice participants" and "advanced participants" based on the group they were assigned during this analysis. Note that the terms refer only to the participants' relevant previous knowledge as assessed during the experiment, but not to their amount of experience with KWQL and visKWQL; no participant had used either query language before taking part in the study.

Participants can thus be classified along two dimensions, their amount of relevant previous knowledge and the query language they were assigned in the study. The number of participants in each group is given in Table 9.

9.2.1 *Task 1: Query creation*

Table 10 shows the average number of questions (out of a total of ten) answered by the participants in each group, ignoring whether the solution was correct or not. The number is higher for advanced participants (8.34) than for novice participants (8.02) and slightly higher for KWQL users (8.20) than for visKWQL users (8.12). Further, KWQL and visKWQL show reversed effects with respect to how the amount of questions answered differs with proficiency: While advanced KWQL participants on average answered 0.8 questions more than their less experienced counterparts, advanced visKWQL users answered 0.2 questions less than visKWQL novices.

While the number of assignments completed overall did not vary greatly between groups, bigger differences can be observed with respect to the number of questions answered correctly (see Table 11). The answer to an assignment is considered to be correct if either the query formulated retrieves the intended set of content items as an answer or if the query yields a superset of the correct

	KWQL	visKWQL	overall
novice	53.33	35.66	43.70
advanced	78.17	93.33	84.91
overall	65.75	58.73	62.24

Table 12: Average percentage of given answers that are correct

content items and the number or list of content item titles is correct.

With an average of 7.5, visKWQL users in the advanced group were the most successful in finding correct answers to the assignments. However, novice participants in the visKWQL group performed worse than all other groups with only 3.5 correct answers. The results for the KWQL groups are less divided with novice users performing better than visKWQL novice users, but worse than advanced KWQL users. Overall, KWQL users on average answered 0.46 more questions correctly than those participants using visKWQL. As expected, novice users overall answered considerably fewer questions correctly (3.76) than the participants in the advanced group (7.11).

Table 12 combines the information in the previous two tables to show the average percentage of the given answers that were correct. Among all participants, almost two thirds of all answers given, 62.24%, are correct. Again, the visKWQL group is responsible both for the best and worst results with 35.66% of all answers given by novice visKWQL users but 93.33% of the answers by advanced visKWQL users being correct. This result is particularly noteworthy since, as shown in table 10, both groups answered a very similar amount of questions. However, while novice visKWQL users on average answered more questions than the participants in the novice KWQL group, fewer of those answers are correct, leading to the result that the absolute number of correct answers is higher for the KWQL group.

Among the advanced groups, the case is reversed: visKWQL users answer fewer questions but do so at a very high rate of correctness, meaning that the average absolute number of correct answers is higher for advanced visKWQL users than for the KWQL counterparts. Having explored the response data aggregated over all ten assignments, we can now move on to examine the results for individual assignments. Figure 53 displays for each assignment (the assignment numbers correspond to those given in Table 7) the percentage of participants who did not give an answer, gave a correct answer and gave an incorrect answer. As can be observed, all participants answered the first two questions which, unlike the consequent assignments, require the composition of queries that consist only of values and operators.

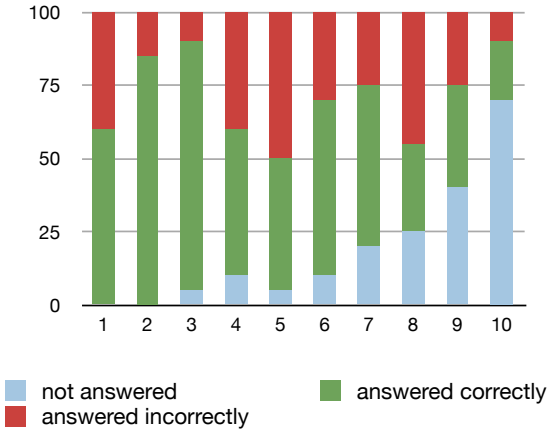


Figure 53: Average response percentages per question

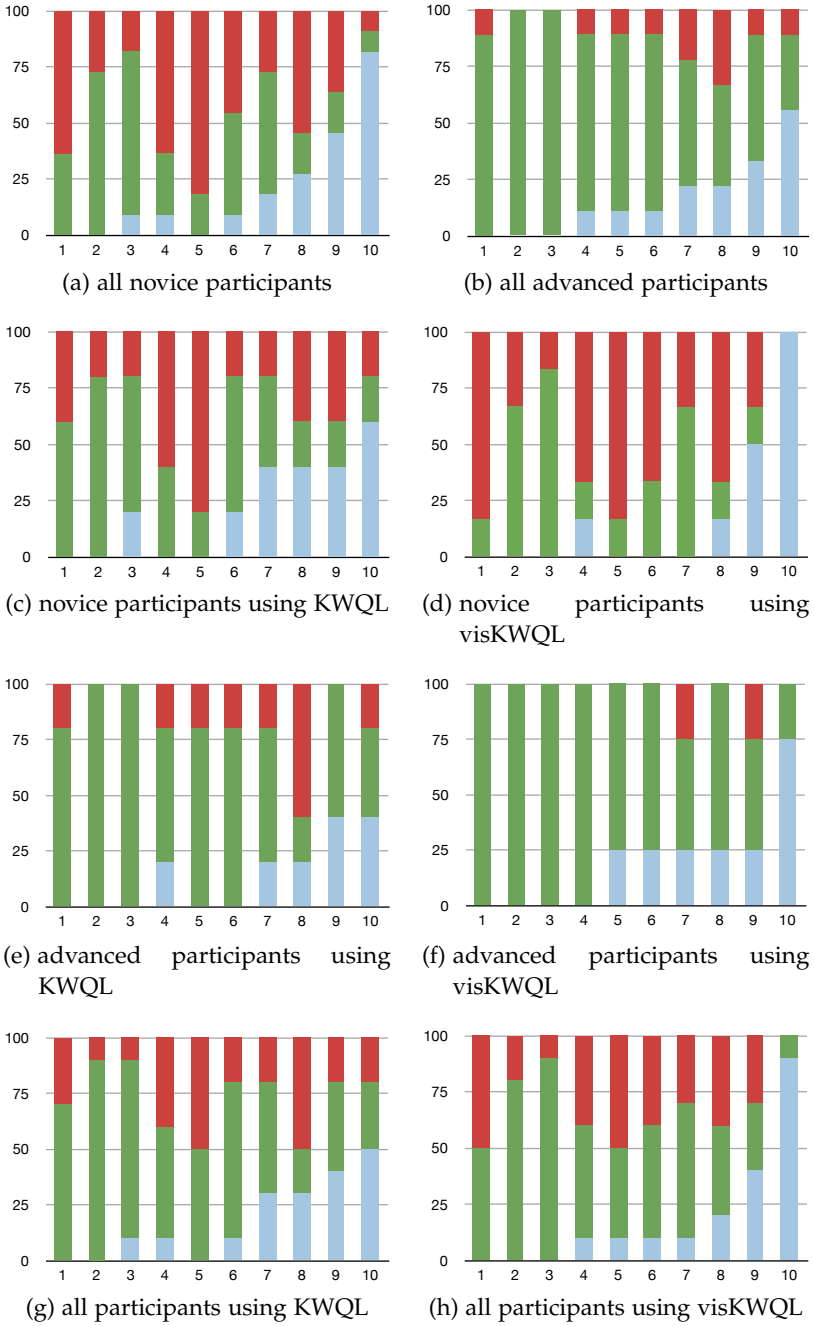
The percentage of participants who did not provide an answer consequently increases gradually with only little more than a quarter of all participants answering the tenth and final question. The only outlier is question four which was answered by fewer participants than question five.

The percentage of correct answers develops less straightforwardly over the assignments. At about 85%, questions 2 and 3 were most frequently answered correctly, considerably more often than the first question. A comparatively big proportion of participants gave incorrect answers to question 5; apart from this, the percentage of correct answers is stable across queries four to seven at 45% to 60%, but drops off considerably to about 30% starting with question 8.

Figure 54 shows the data from Figure 53 broken down according to group memberships along the two dimensions, prior experience and the query language used. In the following, results will be compared for each pair of groups that differ among one dimension but share the other.

Comparing the performance of users with little previous experience with the social semantic web and querying (Figure 54a) to that of the advanced group (Figure 54b), one obvious difference is that the advanced users made fewer mistakes, even in the first two assignments that only require the formulation of relatively simple queries. The percentage of advanced participants who answered correctly decreases across the assignments: Questions 1 to 3 were answered correctly by almost all participants, questions 4 to 6 by about three fourths of the participants and the final four questions by only between half and a third of participants.

Such a pattern cannot be observed in the novice participants' data; the questions most frequently answered correctly are the second and third (73%), seventh (54%) and sixth (45.5%). The first assignment was answered correctly by about one in three partici-



(a) all novice participants

(b) all advanced participants

(c) novice participants using KWQL

(d) novice participants using visKWQL

(e) advanced participants using KWQL

(f) advanced participants using visKWQL

(g) all participants using KWQL

(h) all participants using visKWQL

Figure 54: Average response percentages per question by group

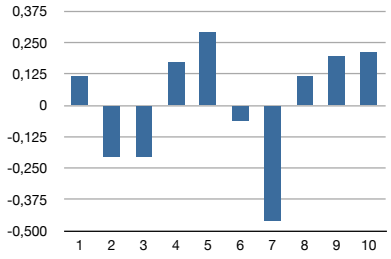
pants, while the answers provided for questions 4 and 5 and 8 to 10 are only correct in a small minority of cases. Comparing the results of the two user groups, questions 1, 4, 5 and 8 to 10 appear to have been particularly hard to solve for participants in the novice group, while identical correctness scores were obtained for question 7, indicating that this assignment was relatively easy to solve for novice participants.

Figures 54c and 54d show the performance data for participants in the novice group broken down by the query language used. Overall, those participants using KWQL had a higher proportion of correct answers than those using visKWQL. In particular questions 1, 4 and 6 were solved correctly by the KWQL group at a rate two times of that of the visKWQL group. On the other hand, assignments 3 and 7 were answered correctly considerably more often when visKWQL was used. Question 5 posed a considerable problem for both groups, while 8 and 9 were solved correctly in comparable proportions. The visKWQL novice group is the only group where no participant provided an answer to assignment 10.

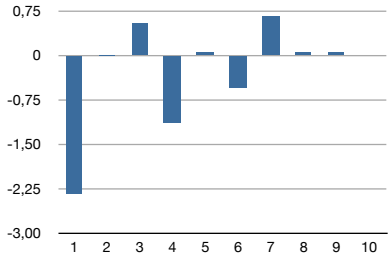
The results for advanced participants using KWQL (Figure 54e) and visKWQL (Figure 54f) show that these participants overall made only few mistakes. In particular the answers given by the visKWQL users are almost all correct; the only assignments that received any incorrect answers were questions 7 and 9. In this group, question 10 was not answered by most participants, but those who did all gave a correct answer.

The results of the advanced KWQL group show more incorrect answers; here, the only assignments that all participants answered correctly are questions 2 and 3. However, only assignments 8 and 10 were not answered correctly by a majority of participants. Comparing the results for the KWQL and visKWQL advanced groups, it appears that questions 4 and especially 8 were especially hard to solve for participants in the advanced KWQL group.

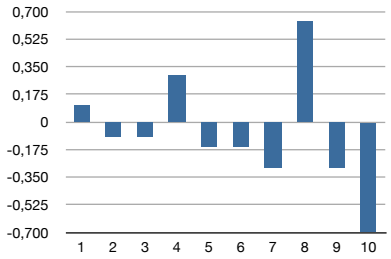
Finally, Figures 54g and 54h show the results for all participants using KWQL and visKWQL respectively. In both cases, question 2 and 3 have the highest number of correct answers. Question 1 was solved correctly less frequently, especially by participants using visKWQL where, as discussed, only few novice users were able to find the right answer. Questions 4 to 7 were answered at a correctness rate between 50 and 70 percent in the KWQL group and 40 to 60 percent in the visKWQL group. While for KWQL, the question answered correctly most frequently among these is question 6; for the visKWQL groups, it is question 7. Question 8 was not answered correctly by most participants in the KWQL group, but fared better in the visKWQL group.



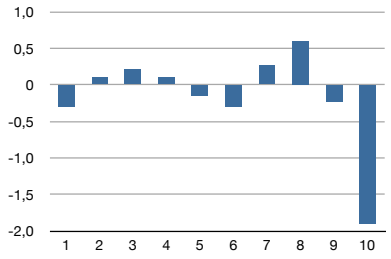
(a) novice and advanced participants



(b) novice participants using KWQL and novice participants using visKWQL



(c) advanced participants using KWQL and advanced participants using visKWQL



(d) participants using KWQL and participants using visKWQL

Figure 55: Deviation from average difference of the percentage correct between two groups

To further investigate the effect of prior knowledge and query language on the task performance and to confirm the observations made with respect to which queries were particularly hard to solve for each group, for each pair of results discussed above, the deviation from the average difference in the percentage of correct answers was calculated for each question. This method is used to identify those questions whose percentage of correct answers differed from the expected value, that is, the average difference in correctness scores between the two groups. For a given question i and two groups j and k, the deviation is calculated as

$$dev_i^{j,k} = \frac{pc_i^j}{pc_i^k} - \sum_{l=1}^{n} \frac{1}{n} \frac{pc_l^j}{pc_l^k}$$

where n is the total number of questions, here 10. A positive deviation score means that participants in group k answered question i correctly more often than expected, while a negative score indicates that group j showed a better than average performance.

Figure 55 visualizes the deviation scores for the four pairs of groups, which confirm the observations made above.

Finally, let us take a closer look at the queries given as answers that were not correct. Out of the total of 171 queries given as

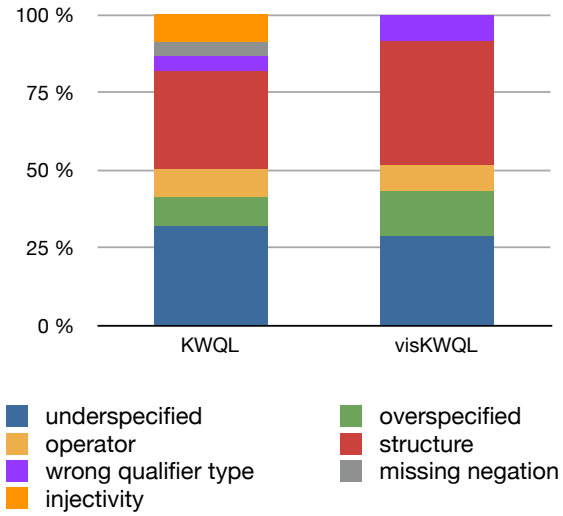


Figure 56: Frequency of different types of mistakes per group

answers to questions in this task, only 7, four KWQL queries and 3 visKWQL queries, were invalid, meaning that they either could not be parsed or violated one of the validity constraints (see Section 10.3)[2]. Consequently, 95% of all KWQL queries and 97% of all visKWQL queries given as answers were valid. The majority of incorrect answers therefore consists of queries that are generally valid, but that do not correspond to the assignment. The different reasons for this will be analyzed in the following.

Figure 56 shows the different types of mistakes that occur in valid queries given as answers by participants in the KWQL and visKWQL groups. Table 13 describes the types of mistakes made and for each shows one exemplary incorrect answer of this type that was given during the experiment. The frequency of each type of mistake was determined by examining all queries that are valid but return the wrong set of answers. Note that one such query may contain several distinct mistakes of different types and thus may add to the frequency count in several categories.

The two most frequent types of errors in both query languages are queries that are not selective enough and retrieve a superset of the intended result, and mistakes pertaining to the structural selection conditions expressed in the queries. KWQL users tended to underspecify their queries more than visKWQL users, which in turn have a slightly higher proportion of queries that are incorrect due to overspecification. Further, participants in the visKWQL group made a proportionally higher amount of mistakes with respect to the structure of the query, while KWQL users experienced problems accounting for injectivity.

2 In addition, six queries were bracketed incorrectly, but since participants had to write down their answers by hand, this is likely due to clerical errors and was ignored

Type	Description	Example
underspecified	correct query is contained in the query given	`ci(title:Lisa)` (question 7)
overspecified	query given is contained in the correct query	`ci(text:Abraham OR text:Abe)` (question 1)
operator	wrong operator used or no operator used where disjunction was intended	`Abe Abraham` (question 1)
structure	structure is left out or qualifier term is placed in the wrong resource	`ci(character NOT trivia)` (question 8)
		`ci(tag(name:character) author:Betsy)(question 4)`
wrong qualifier type	the wrong type of qualifier is used	`ci(link(anchorText:Carl))`
missing negation	a part of the query should be negated	`ci(tag(name:character descendant:ci(tag(name: trivia)))` (question 9)
injectivity	mistake related to injectivity	`ci(tag(favorite)tag(name: character)tag(author:Betsy))` (question 4)

Table 13: Types of mistakes made by participants

While the frequency of each type of mistake gives some insight into which parts of KWQL may be particularly hard to learn for users, they are also biased by various factors and should not be taken at face value. The statistics do not reflect the fact that different questions prompt different kinds of mistakes, for example, most of the questions required the formulation of queries that made use of various structural constraints, either in the form of nestings, links or intra-content item structure, but fewer questions involved injectivity. As such, it can be expected that more mistakes involving structure are made, even if in reality fewer participants can make adequate use of KWQL's injectivity feature. Further, not all questions are answered by the same amount of participants which can intensify the effect. Additionally, not all types of mistakes are independent of each other: When a query is underspecified, all structural constraints might have been left out, making it impossible to decide whether the query contains any structural mistakes.

	KWQL	visKWQL	overall
novice	4	5	9
advanced	4	4	8
overall	8	9	17

Table 14: Number of participants per group in task 2

	KWQL	visKWQL	overall
novice	5.5	5.5	5.5
advanced	6	6	6
overall	5.75	5.7	5.72

Table 15: Average number of questions (out of 6) answered in task 2

9.2.2 *Task 2: Query understanding*

The results of the second task are based on a total of 17 participants, their distribution over the different groups is shown in Table 14. All advanced participants provided answers to all six questions, while the novice participants on average answered 5.5 questions (see Table 15).

The correctness of each answer was judged based on whether the description was correct and all selection constraints were mentioned. Overall, participants answered 4.89 of the questions correctly. There was no difference in the number of correct answers between advanced participants who used KWQL and those who used visKWQL, both on average gave 5.5 correct answers. The case is different for the novice users, here, those who used KWQL on average had 4.75 correct answers, while participants in the novice visKWQL group on average only answered 4.0 questions correctly. Overall, this means that KWQL users gave more correct answers than visKWQL users by 0.53 questions while advanced users on average answered 1.16 more questions correctly than novice users did.

The overall distributions of correct, incorrect and missing answers across for each question are displayed in Figure 57. The second question was answered correctly by all participants. Question four has the second highest number of correct answers, fol-

	KWQL	visKWQL	overall
novice	4.75	4.0	4.34
advanced	5.5	5.5	5.5
overall	5.13	4.6	4.89

Table 16: Average number of questions (out of 6) answered correctly

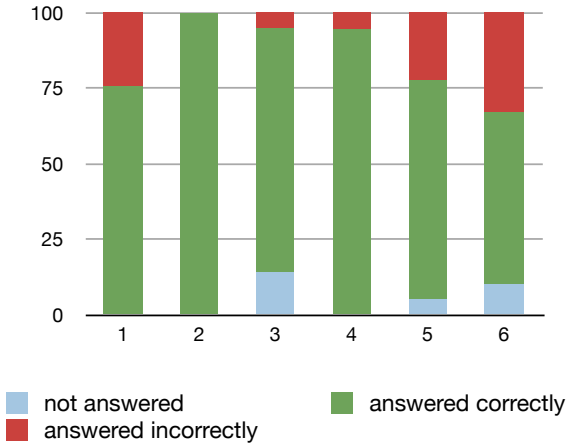


Figure 57: Average response percentages per question

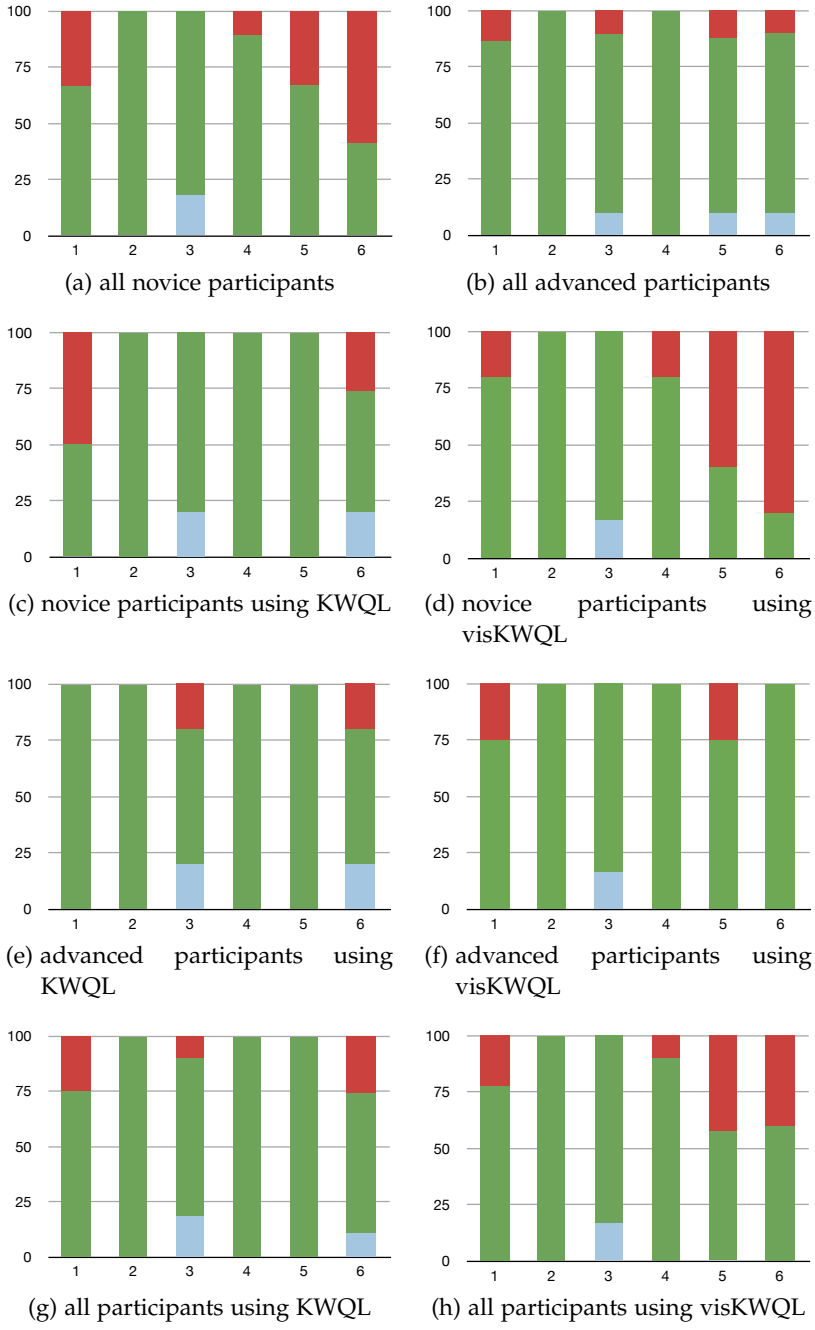
lowed by questions 3, 1 and 5. The final question was answered correctly least often, but the percentage of correct answers still is over 55 percent. Questions 1,2, and 4 were answered by all users, while only a low percentage of users did not answer questions 3, 5 and 6.

Figure 58 shows the data from Figure 57 broken down by group. Novice participants (Figure 58a) gave more wrong answers than the advanced participants (Figure 58b), especially to questions 1 and 6. While novice KWQL users (Figure 58c) had bigger problems answering question 1 correctly than did novice visKWQL users (Figure 58d), the latter overall made more mistakes, particularly when answering questions 5 and 6. Advanced users (Figures 58e and 58f), regardless of the query languages used, gave only few incorrect answers. Only two questions had any wrong answers at all, 3 and 6 in the KWQL group and 1 and 5 in the visKWQL group. Overall, both participants using KWQL (Figure 58g) and those using visKWQL (Figure 58h) had some problems answering questions 1 and 6. About a third of visKWQL participants additionally answered question 5 incorrectly.

9.2.3 *User Judgments*

Complementing the quantitative analysis of the answers given by participants, this section gives an account of the participants' comments on the questions asked as part of the study.

The KiWi wiki was perceived very positively by almost all users, several participants said they found it well-structured and clear; KiWi's layout and choice of colors were also complimented. The annotation and structuring features that set KiWi apart from conventional, non-semantic wikis were frequently mentioned as being helpful and easy to understand, even for beginning users. In particular, fragments were named numerous times as



(a) all novice participants

(b) all advanced participants

(c) novice participants using KWQL

(d) novice participants using visKWQL

(e) advanced participants using KWQL

(f) advanced participants using visKWQL

(g) all participants using KWQL

(h) all participants using visKWQL

Figure 58: Average response percentages per question by group

a welcome novel feature. While the perception of KiWi and its conceptual model largely was very positive, a small minority of participants either did not think that KiWi differed significantly from conventional wikis (in particular Wikipedia, that is, MediaWiki) or found it too complicated and confusing.

When asked about properties that a query language for KiWi should have, participants universally agreed that such a language should be easy to use and intuitive so as to accommodate beginning users. Several participants additionally stated that at the same time, the query language should also be powerful in order to allow for complex queries. These answers are interesting in that, although participants answered the question before having received any information about KWQL, they reflect the main underlying premise of KWQL and thereby also visKWQL.

Other features that were named by a number of participants were that the query language should be self-explanatory and that the user should be guided in constructing her queries. Other requirements named were that the query language should be "fast," similar to "related" query languages, should be able to query metadata and full text, should use a syntax that is in close relation to the structure of the content and should also display results that are not strict matches but that are relevant to the query.

After having read the introductory texts on, depending on the group, KWQL or visKWQL, participants were asked about their impression of the language and its expressiveness and ease of use. Again, the feedback was mostly positive with participants remarking that the language seemed well-structured and clear, though several also thought that it required practice and time to learn and was not ideally suited for laymen. Two participants in the KWQL group found the language too complicated and stated that they would prefer a form-based or visual method. Inversely, one participant in the visKWQL group suspected that visual querying was unnecessary and that a purely textual language would be easier to use. Generally, visKWQL participants, aware of the existence of both querying modi, liked the availability of visual as well as textual querying. visKWQL's context help was also perceived positively.

When asked about KWQL/visKWQL's expressive power, nearly all participants agreed that the language was expressive with one participant even suspecting that KWQL was too expressive. Further, one participant missed the feature of relational queries. Opinions on whether KWQL is easy to use were more divided, while some participants thought that it was, others were not sure, thought that the language required at least some practice to be easily usable or stated that the language was only easy to use for users with previous querying experience.

Only few participants remarked on the individual assignments in task 1 and 2. Two participants stated that they needed some time to understand how to solve the first question of task 1, but that, having solved it, question 2 was easy to answer.

A number of participants were unsure about the difference between a value being contained in a tag and being contained in text and about the difference between qualifiers and resources. Further, several participants recognized that the **`name`** qualifier could be left out in many queries. A number of comments were concerned with aspects of KWQL or visKWQL that the participants found hard to understand, these were variables, URIs, injectivity, content item nestings and links. In several cases, participants did not understand the question or were unsure about how to translate it into a query. One participant said that over the course of task 1 he had realized that he had not understood how visKWQL works.

After completing the two tasks, participants were again asked about their opinion on KWQL or visKWQL. Now, most participants, 13 out of 20, said that they felt they had understood how to use the query language. Six participants stated that they had understood the language to some extent, but had trouble with specific concepts or needed more time to understand it fully. Only one participant claimed to not have understood visKWQL at all. As before, many participants commented positively on KWQL and visKWQL, saying that they found it powerful and easy to understand. One participant said he hoped it would be used in practice, while another one particularly like that KWQL allows for underspecified queries. Another group of users overall was generally in favor of the query language, but said that it required some knowledge, practice and time to learn. Two participants stated that they initially found visKWQL confusing but after using it for a while thought it was intuitive and convenient to use. Only three participants commented negatively, saying that they found the language they used too complicated and confusing. All three were in the novice group, two of them used visKWQL and one KWQL. All participants thought that KWQL/visKWQL had high expressive power, although one said that the queries making use of the expressive power were considerably harder to understand than simple queries.

With respect to the question whether KWQL/visKWQL was easy to use, a majority of participants answered that it was, however, many qualified their response and listed particular things that they found hard or that could be done to improve KWQL's and visKWQL's usability. Specifically, participants experienced problems with variables and links. Some found the bracketing required to be confusing, while others wished for syntax highlighting and autocompletion in KWQL.

The overall hardness of the questions in the study was judged to be easy or intermediate by most participants, although some added that some questions were much harder than others, that they would have needed more time to answer all questions and that they needed many tries to find the correct answers. One participant, a novice visKWQL user, found the questions very hard.

Most participants thought that the introductory texts provided were of good quality and helped them answer the questions. Participants also remarked that the instructions should contain more examples, should make the difference between resources and qualifiers clearer and should explain variables in more detail. One participant found the introduction to be too verbose, and others wished for an index to make it easier to find specific pieces of information in the text. Several participants also commented that there was little time to read the instructions and they would have preferred for the study to consist of several sessions and allow for more practice.

Finally, participants were asked what they considered to be KWQL's and visKWQL's advantages and disadvantages. All participants were of the opinion that KWQL/visKWQL is powerful and allows for precise queries, but some also remarked that it was harder to use than web search and takes some time to learn.

9.3 DISCUSSION

All in all, the results of the experimental evaluation are very positive: KWQL and visKWQL were well perceived by the participants who also thought that the languages are expressive and easy to use, at least given some time and practice. Even given only a very short introduction and a small amount of time to solve the assignments, participants overall could provide correct answers to more than half the questions in the query writing task and over eighty percent of the questions in the query understanding task.

One remarkable result of the study is that nearly all queries given as answers in the first task were valid KWQL queries. The reasons for this are likely that visKWQL prevents many editing operations that would lead to invalid queries, and that both KWQL and visKWQL return meaningful and detailed error messages when a query is found to be invalid upon evaluation. However, at least in the case of KWQL where users receive no guidance during query creation, these factors alone are unlikely to be the only reason for the high percentage of valid queries, indicating that participants learned enough of KWQL to be able to, given feedback, write valid queries.

The percentage of queries that were not only valid but that also constituted a correct answer to the question was considerably lower at 52.7% overall. While participants with previous experience in querying and the social semantic web performed well, beginning users only could answer about 37.6% of the questions correctly. The analysis of the mistakes made as well as the user comments indicate that participants encountered problems with using variables, URIs, injectivity and data structure in general. Some participants commented that there were questions that they did not understand. This remark is revealing in that it helps understand the reason for the problems that particularly novice participants encountered: To compose a query that is not only valid but also selects the correct set of content items, participants need to understand not only KWQL, but also KiWi's conceptual model, the semantics behind the data organization in the dataset and what the given question means in the context of the conceptual model and the dataset. As such, there is a big amount of knowledge that participants have to acquire before being able to compose appropriate KWQL queries, but that is not directly related to KWQL. Given that participants spent only about half an hour to acquaint themselves with KiWi and KWQL, it is likely that novice participants could not acquire all the knowledge needed to understand how to compose correct KWQL queries.

Apart from questions 2 and 3, the questions of task 1 answered correctly most frequently by novice participants are questions 6 and 7, both of which require the formulation of queries that use structural constraints but that, unlike the answer to other advanced queries, do not make use of URIs, variables or injectivity, all features that were named as being hard to understand.

This result is not unexpected since, as described in the introduction, this study was intentionally set up to tests participants' performance after only a short learning phase, but it can help to understand the reasons behind participants' performances. It also shows that a further study where participants can acquire KWQL under more realistic conditions, that is, slowly, gradually, and based on their own information needs, would likely yield insightful results as to how KWQL is learned and used.

The learning required could also explain why visKWQL novices performed worse than the participants in the novice KWQL group: Aside from having to learn all the new concepts, they also had to acquaint themselves with visual querying, which likely was unfamiliar to them. The novice KWQL users, on the other hand, had to write textual queries, which, given that all participants can be assumed to have used web search engines before, was more familiar to them.

Another contributing factor to visKWQL novices' comparatively bad performance could be that visKWQL is not ide-

ally suited for creating vastly underspecified queries. visKWQL makes it easy to understand the structure of queries and to create structured queries, but offers no advantage when then queries involved are very simple. For example, to compose the query that is the correct answer to the first solution, `Abraham OR Abe`, the user must create three boxes, nest them, and enter two values, which arguably is more complicated than typing two keywords and an operator. The resulting visKWQL query is weakly structured and thus likely not easier to understand than its textual version. Indeed, novice visKWQL participants performed particularly badly on question 1 of task 1 compared to novice KWQL participants, and the difference in the percentage of correct answers is smaller for all consequent questions, which are all answered by queries that contain more structure.

This result indicates that it might be better to introduce beginning users whose queries exclusively consist of keywords to textual KWQL, and to only add visKWQL once the queries become more complex. On the other hand, given visKWQL's round-tripping capabilities, it is possible that users could achieve equivalent or better results when textual and visual query editing are introduced simultaneously; a follow-up study could investigate which of the three methods yields the best results.

Not only novice visKWQL participants, but also novice KWQL participants had considerable problems answering the first question of task 1 correctly, meaning that visKWQL's characteristics alone could not have caused this result. One possible explanation for the finding is that participants, after reading the introduction which presents all the query language's features, were not expecting the correct solution to be so simple. Another possibility is that the effect is simply due to participants not having any practice writing KWQL or visKWQL queries. This second assumption is supported by participants commenting that, having found an answer to question 1, question 2 became easy to solve, as well as several participants who stated that their understanding of KWQL/visKWQL improved as they were solving the assignments.

Advanced participants achieved good results regardless of the query language used, on average, they could answer 71% of the questions in task 1 and over 90% of the questions in task 2 correctly. Their results also showed that visKWQL can help improve the performance; advanced visKWQL participants answered fewer questions overall than advanced KWQL participants but nearly all answers given were correct, meaning that overall, the advanced visKWQL group had the highest average number of questions answered correctly. In particular questions 4 and 8 which required answers using structure combined with injectivity and negation were answered correctly more frequently

when participants used visKWQL. These findings indicate that participants who are familiar with querying and structured data and to whom the information in the introductions was less novel, can make effective use of visKWQL and the advantages it offers over textual KWQL. This result gives further weight to the explanation that the comparatively bad performance of novice visKWQL participants is due to them being confronted with an overwhelming amount of new information that makes it hard for them to additionally absorb the concepts of visual querying and visKWQL.

One interesting finding in the study is that novice visKWQL participants in task 1 answered more questions than the KWQL novice group, while the situation is reversed for advanced users where participants in the KWQL group answer more questions. A possible explanation for this effect is that it is easier to construct valid queries using visKWQL than using KWQL due to the guidance that visKWQL provides. visKWQL novices can thus quickly create queries that, while they may not correctly answer the question, are valid and return a number of answers. Participants in the KWQL group on the other hand, if they do not want to restrict themselves to keyword queries, have to use the introductory text to learn how to use KWQL and thus have to invest more work to create valid queries. This difference could also be the reason why KWQL novices did better in task 1 than visKWQL novice users: visKWQL makes it easy to create valid queries without much knowledge about the query language, while KWQL requires more learning and a deeper understanding of the language.

Advanced KWQL participants then simply might answer more questions than participants in the visKWQL group since, when the syntax is known, writing a textual query is faster than creating and nesting boxes and inserting values.

Across all groups, participants had more success understanding queries than writing them themselves. In the query understanding task, like in task 1, novice KWQL users outperformed novice visKWQL users, both of which had a lower percentage of correct answers than either advanced group. Both advanced groups on average answered more than 90 percent of the questions correctly, indicating that this task overall was very easy for them. The fact that participants perform better at understanding queries than at writing them indicates that users could benefit from the addition of query templates that users can modify according to their needs.

In conclusion, the results of this experimental evaluation overall are positive: Even given minimal introduction, participants succeeded, to different degrees, in writing and understanding KWQL and visKWQL queries. The principles behind the languages conform to participants' expectations to a query language

for KiWi and the languages overall received positive comments both with respect to expressive power and ease of use. Many participants remarked that, the advanced features of KWQL and visKWQL require time and practice to learn, which is in accordance with how we intend the languages to be learned and used in practice.

10 ARCHITECTURE AND IMPLEMENTATION

To accommodate all users, modern knowledge management systems such as the KiWi wiki and other social semantic web applications must deal with both unstructured (textual or multi-modal) information as well as structured data carrying varying degrees of semantics: hierarchical data for document and simple classification structures, social classifications in form of tag networks, formal ontologies in RDF or OWL. Expert users in such systems can define semantically rich, automated analysis or derivation tasks. However, the vast number of users has little understanding of formal knowledge representation, produces unstructured information with lightweight semantic annotations such as free-form tags, and interacts with the system through simple but imprecise (keyword) queries.

In consequence, modern knowledge management needs to combine aspects from classical information retrieval, databases, social media and semantic technologies.

From these observations, we derive two properties that characterize successful modern knowledge management systems:

(1) "Interfaces must be adaptable and flexible": Interfaces should scale with user experience: For novice users, simple, but imprecise queries are useful for satisfying their information needs; for expert users precise, but necessarily fairly complex queries that enable automated action and derivation are required. Interfaces should also be able to adapt to different types of knowledge in a system, providing a consistent interface. This property of course is the driving force behind KWQL and discussed in more detail in, among others, Chapter 1 and Section 7.1.

(2) "Patchwork knowledge management": Due to the increase in data size and formats, knowledge management systems face a dual challenge: Users expect high performance for (at least basic) queries regardless of the data scale, as in Web search engines. On the other hand, knowledge management systems must be able to adapt quickly to additional knowledge sources, providing scalable yet sufficiently expressive interfaces to query and process such data.

This chapter presents an implementation of KWQL, *KWilt*, that is based on a patchwork approach. This approach to knowledge management is illustrated using KWQL, KWilt and the KiWi wiki, a setting which exemplifies the challenges outlined.

KWQL: Scaling with User Experience

To illustrate how KWQL provides a consistent interface that easily adapts to different levels of user experience, consider the following scenario: *"In a KiWi wiki describing the KiWi project, we would like to find all wiki pages that describe (knowledge management) systems that have influenced the development of KiWi."*

In a conventional knowledge management system, we would expect a formal relation (e.g., `wk:influences`) that represents the very intent of our query. Indeed, assuming `wk:KiWi` represents the KiWi system, we can query such relations using an RDF-enhanced version of KWQL (adopting the syntax described in Section 13.2):

```
ci(rdf(predicate:'wk:influences' object:'wk:KiWi'))
```

However, usually this relation is not present explicitly. Even if it is, users are often unable to express their intent in such a formal manner.

Accustomed to Web search interfaces, novice users might start with a query that returns all content items containing "KiWi":

```
KiWi
```

Obviously, such a query is likely too unspecific to capture the above query intent and, may omit a number of systems that are described without reference to KiWi, but that are referenced from the description of KiWi.

Thus, we might refine the query to return such referenced resources, i.e., content items that are the target of a link originating from a content item containing "KiWi"[1]:

```
$u @ ci(KiWi link(target:ci(URI:$u))
```

However, that query is not specific enough, as it returns also, for example, content items about technologies used in KiWi. We know that KiWi is a semantic wiki and might be tempted to amend that query to return only resources that are also semantic wikis:

```
$u @ ci(KiWi link(target:ci(URI:$u tag(name:"semantic
    wiki")))
```

1 Here and in the following, `$u` is used as syntactic sugar for `ci(text:ALL(child:RENDER($u)))`, that is, we assume that if the head consists only of a variable bound to one or several URIs, a content item containing the respective referenced content items will be returned.

But there might well be systems that are not semantic wikis but have a significant influence on KiWi. To capture them, we choose the query:

```
$u @ ci(KiWi tag(name:$t) link(target:ci(URI:$u
    tag(name:$t)))
```

It returns all content items that are tagged with a tag that has the same name as a tag of a content item containing "KiWi" that also links to the returned resource. This way, we likely capture resources with similar characteristics as KiWi that are also mentioned in its description.

To summarize, KWQL's main contributions over existing query languages and similar interfaces for knowledge management systems are:

1. KWQL provides a **consistent interface** for access to the wide range of knowledge present in the semantic wiki KiWi: unstructured wiki pages, the link (and containment) graph over wiki pages and tags.
2. KWQL is designed to **scale with the user experience**: Queries can take the form of "just a bag of keywords," but also be extended with increasingly more precise constraints on the content, structure and annotations of wiki pages.

KWilt: Patchwork Knowledge Management

KWilt is KWQL's implementation in KiWi. It provides an easily extensible, yet performant implementation of KWQL's features over the wide range of data available in the KiWi wiki.

Previous approaches have tried to engineer a knowledge information system for such diverse information and user needs from the start. In contrast, KWilt uses a patchwork approach, combining performant and mature technologies where available. For example, KWilt uses a scalable and well established information retrieval engine (Solr [337]) to evaluate keyword queries. In fact, KWilt tries to evaluate as large a fragment of any KWQL query in the information retrieval engine as possible.

If necessary, the results are further refined by (1) checking any structural constraints of the query and (2) finally enforcing all remaining first-order constraints, e.g., from multiple variable occurrences.

KWilt's patchwork approach has three main advantages:

1. Many queries can be evaluated at the speed of search engines, yet all the power of first-order logic is available if needed, as detailed in Section 10.6: The three steps use increasingly more expressive, but also less scalable technologies. Thus even for queries that involve full first-order constraints, we can, usually,

substantially reduce the number of candidates in the information retrieval engine by enforcing structural constraints before evaluating the first-order constraints.
This property is particularly relevant in the context of KWQL, as (novice) users that use KWQL like a search engine also expect the speed of a search engine, likely unaware of the additional expressiveness provided by KWQL.

2. Each part is implemented using proven technologies and algorithms with minimal "glue" between the employed tools (see Section 10.4).

3. The separation makes it easy to adapt each of the parts, for example to include additional data sources. If KiWi were to introduce data with different structural properties, e.g., strictly hierarchical taxonomies, only the part of KWilt that evaluates structural constraints needs to be modified. Similarly, if KWQL would introduce other content primitives other than keywords (e.g., for image retrieval), only the first (retrieval) part of KWilt would be affected. This separation also ensures that the planned extension of KWQL to include querying of structured tags (see Chapter 12) and RDF (Chapter 13) can easily be realized without fundamentally changing the query evaluation scheme.

10.1 RELATED WORK

As discussed in Chapter 4, there are two very different approaches to querying data on the web: On the one hand, keyword queries and search engines can be used to search the content of the web, often at a large scale. On the other hand, query languages like XPath and XQuery enable precise selections over the structure of individual web pages. Both approaches have been applied for querying structured data like XML and RDF at the scale of wikis (rather than a significant portion of the whole web). The main difference remains that languages like XPath and XQuery enable experts to write precise queries, whereas keyword queries are suitable also for novice users, at the price of lower precision.

In the context of XML, a number of recent approaches have been proposed that combine keyword queries on the content of documents or XML elements with XPath-style constraints on the structure of those documents. There are two different strategies for this combination: The first approach interleaves structure and content in a single data structure, whereas the second one uses different indices for content and structure.

KWilt falls into the second category, but is unique in three main aspects:

1. Through KWQL, a **single, integrated interface** that scales with the experience of the user is provided whereas other ap-

proaches either lean towards full query languages like XQuery or provide only slightly enhanced keyword query interfaces.

2. KWilt uses a **lightweight patchwork integration** approach that reuses existing technologies where possible and makes it easy to add or exchange parts of the implementation components. Furthermore, this allows KWilt to execute basic (keyword) queries at the speed of the underlying search engine with only little overhead.

3. KWQL and KWilt are carefully adapted to the needs of a semantic wiki, namely KiWi. Particularly, they allow querying of graph-structured data in the form of links between web pages and, in the future, RDF graphs. In contrast, most of the prior approaches are limited to XML tree data.

HYBRID INTERLEAVING APPROACHES Interleaving structure and content for query evaluation has the big advantage that unselective keywords or structural elements of a query do not cause the index to retrieve large parts of the database which are then filtered out by the subsequent join. Instead, the content can be used during the structural matching to prune irrelevant parts of the search space. The price is that entirely new index structures and evaluation algorithms have to be developed.

Content-Aware DataGuides [362] (CADGs) are an extension of DataGuides [163] that interleave structural and textual elements in a single index. In contrast to solutions that match structure and content separately and combine the results by a join, the content-aware DataGuide can use keyword information during path matching. This is realized by enhancing the DataGuide with a materialized (conservative) approximation of the content/structure join (which keywords may occur in the sub-tree of the current node) and adequate algorithms that use this information for pruning irrelevant paths during the matching of the structure.

Unfortunately, path indices such as DataGuides cannot easily be extended to graph data as used in KiWi. Furthermore, the performance of CADGs relies on a novel matching algorithm that cannot be easily implemented at the scale of a wiki.

ViST [358] unifies structure and content in a single index by transforming XML into *structure-encoded sequences*. These sequences are a pre-order representation of the XML document with additional information which preserves the structure of the tree. Thus, querying the document is equivalent to searching for sub-sequences in the structure-encoded sequence of the XML document which eliminates any need for a join, even for branching path queries. To be able to carry out sub-sequence matching efficiently, the authors propose a *virtual suffix tree* (ViST) which is can be stored in a B^+-tree and supports dynamic insertion and

deletion of nodes. Like CADGs, ViST cannot easily be extended to graph data and does not support phrase queries.

HYBRID JOIN APPROACHES So-called *join approaches* are the more common approach to combining content and structure queries: These approaches employ two different indices, one for the content and one for the structure. Both indices are queried independently and the intermediate results are combined through a join.

As an example, Kaushik et al. [213] propose a framework that, similar to the 1-index [279], uses an index based on equivalence classes to answer the structural part of a query. An inverted list is adapted for XML and augmented with the node identifiers from the structural index to cover the keyword parts of the query. Since both indices refer to nodes in the XML tree by their identifiers in the structural index, the gathered answers can be easily joined to retrieve the answers to the complete query. The benefit of this framework is that it does not propose its own data structures, but instead relies on structural indices, inverted lists and join algorithms which need only slight adjustments to be used in the framework. In this respect it is similar to KWilt. However, KWilt extends this approach to graph data and provides a consistent interface in the form of KWQL.

Apart from this framework, several other systems were proposed that focus on specific implementations of index structures [269, 102, 333, 383]. Also, a number of query algebras have been proposed for such systems [270, 18] to enable the logical optimization of queries. The XFT algebra [18] is only suitable for querying content of XML, but the authors state that it can be easily combined with algebras for structural queries which would enable the optimization of a complete query for structure and content.

For RDF, Wang et al. [357] suggests to store the structure and content of an RDF graph in a single inverted list, which is also capable of storing the position of the occurrence of a keyword. For each triple, an entry for the subject is generated in the index structure, and a field corresponding to the predicate is created. The object is then stored as the position of the corresponding predicate field. During evaluation, a query is split into several subqueries which consist only of single triple queries (called *triple patterns* in SPARQL). These triple patterns are then processed stepwise, joining the intermediate results, before the next subquery is evaluated. Since this technique does not need to modify standard implementations of inverted lists, highly efficient implementations can be used to carry out combined structure and content queries.

Though the use of a single inverted list leads to an elegant approach, the evaluation of basic keyword queries is significantly affected by the ability of the system to also evaluate more complex structural queries.

10.2 A FEW WORDS ON INJECTIVITY

One of KWQL's most challenging properties for evaluation is injectivity, which ensures that if a resource or qualifier is stated twice in a query, it is matched at least twice in an answer as well. The evaluation on injectivity constraints on unordered tree data, which the data model of KiWi is an extension of, has been shown to be NP-complete [219]. To see why the validation of injectivity constraints is so demanding, consider the following example:

A content item has been tagged three times with tags with the tag name "quicksort." While the tag names are identical, each tag has a different URI and not all taggings have the same author:

```
tag(name:quicksort author:admin URI:kiwi#tag1)
tag(name:quicksort author:admin URI:kiwi#tag2)
tag(name:quicksort author:anonymous URI:kiwi#tag3)
```

Consider the following query which is evaluated on the content item:

```
ci(tag(name:quicksort)
   tag(author:admin)
   tag(URI:kiwi#tag2))
```

A naive algorithm that takes the ordering of the tags into account, matches the first tag of the query with the first tag of the content item and the second tag of the query with the second tag of the content item. But then, no matching partner remains for the last tag of the query, since the URI of the third tag of the content item has a different value.

However, if the first tag of the query is matched with the third tag of the content item and the second with the first, then the third tag of the query can be successfully matched with the second tag of the content item. And therefore the content item is a valid answer to the query.

As this example indicates, the order of resources stated in the query not necessarily needs to match the order in which the resources actually appear during evaluation. Therefore, a proper algorithm must regard any permutation of the resources to decide reliably whether a query containing injectivity constraints matches a content item or not.

Consequently, during the validation of the sub-resources the algorithm has to keep track of the permutations already investigated. Otherwise, it cannot be guaranteed that all possible

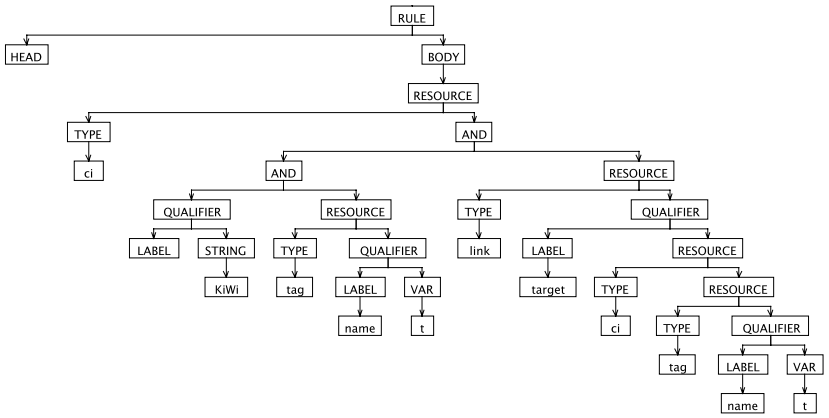


Figure 59: AST representation of the query `ci(KiWi tag(name:$t)link(target:ci(tag(name:$t))))`

permutations are taken into account and query answers might be dropped.

10.3 QUERY PREPROCESSING

Before a KWQL query can be evaluated in KWilt, the query is syntactically analyzed, that is parsed, and verified.

First, the query is parsed using an ANTLR-generated parser. ANTLR (ANother Tool for Language Recognition) [301] is a software tool that converts ENBF-like [368] language descriptions into LL(*) parsers. ANTLR can produce source code in a number of different programming languages, making it possible for visKWQL, written in JavaScript, and KWilt, written in Java, to employ equivalent grammars generated from the same language description.

The parser does not only create a parse tree for the query, but also generates an Abstract Syntax Tree (AST), a normalized representation of the query's syntactic structure. Figure 59 shows the AST for the last query mentioned in the introduction of this chapter.

Each query AST is a binary tree consisting of a `HEAD` and a `BODY` subtree which together form a `RULE` tree. Each value node in the tree is the descendant of at least one `QUALIFIER` and one `RESOURCE` node. Even when the context of a keyword is not fully specified, all qualifiers and resources in the context are represented as nodes in the AST, but their type or label may be empty. For example, the AST in Figure 59 shows that the qualifier label for the keyword "KiWi" is not given in the query. The figure also illustrates that conjunctions, that is, `AND` nodes, are introduced where no operator is given in the query.

Once the query has been converted into an AST, a number of criteria are used to ensure the validity of the query. Specifically, the following must hold:

- Regardless of the dataset used, all variables used in the rule head are bound in the rule body
- All sub-resource nestings are valid according to the principles described in Section 7.2 and shown in Figure 30
- All resources only occur with the qualifiers allowed according to table 3 in section 7.2
- In accordance with the same table, no qualifiers occur more often than allowed

10.4 KWILT: ARCHITECTURE AND EVALUATION PHASES

Evaluating KWQL queries is a challenging task that cannot be accomplished by existing query engines for (semantic) wikis. For instance, the aforementioned query

```
ci(KiWi tag(name:$t) link(target:ci(tag(name:$t))))
```

combines content and structural elements with variables to retrieve content items that contain "KiWi" and that link to content items with which they have at least one tag name in common.

Despite the unique combination of features found in KWQL, KWilt does not try to "reinvent the wheel." In particular, we chose not to build a new index structure capable of combining all these aspects in a single index access, as this approach has several drawbacks. First and foremost, the rich data model would require a fairly complex index structure that can support content and structure queries, fast access to hierarchical data and link graphs as well as navigation over containment and link relations. Moreover, it is likely that the data model evolves over time with new kinds of data or different representation formats introduced, in particular in the field of social semantic media and RDF data. Using a complex index structure which is adapted for a certain data model makes it hard or even infeasible to react to these potential changes.

Instead, we use a *patchwork*, or integration, approach to combine off-the-shelf state-of-the-art tools in a single framework. To this end, the evaluation is split into three consecutive evaluation phases, each dedicated to a particular aspect of the query (see Figure 60). Throughout these phases, we keep a *candidate set* of those content items that potentially match the query, that is, for which no evidence to the contrary has been found so far. Initially, the candidate set consists of all content items in the wiki.

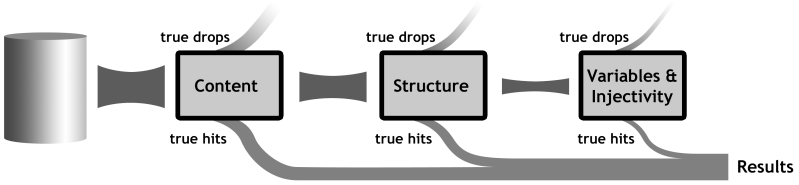


Figure 60: The KWilt evaluation pipeline

Each step then makes use of a tool which is particularly suitable for evaluating the query constraints covered by that aspect of the evaluation, e.g., for keyword queries we use a traditional search engine. Every evaluation step may remove content items from the candidate set by determining either that they fully match the query (*true hit*) or that they do not fulfill the selection criteria that the step evaluates and therefore do not match the query (*true drop*). If the candidate set is not empty after the evaluation step, that is, when there are content items in the candidate set which cannot be determined to be either a true hit or a true drop, the consequent evaluation step is called with the candidate set as an input.

Thus, efficient and mature algorithms form the basis of our framework while the framework itself remains flexible with lightweight "glue" to combine the evaluation phases.

10.4.1 *Keyword Queries*

Many KWQL queries, in particular those formulated by novice users, mostly or exclusively address the content of the pages. Therefore, the first evaluation phase treats the keywords in the query. Since this step uses technology that can operate quickly on big amounts of data, it is well-suited as a first evaluation phase. Not only can the two subsequent phases be skipped if all constraints of the query can be validated in this phase, but the candidate set can also be efficiently reduced using constraints on the content as a filter. This means that fewer potential matches will have to be considered in the consequent, less performant evaluation steps.

During this first phase, the content and metadata of each content item, the values of its non-structural qualifiers, and to some extent also their tags, links and fragments, are searched using the information retrieval engine Solr [337], an extension of Lucene [187].

Solr provides a highly optimized inverted list index structure to carry out keyword queries on a set of documents. Each document consists of an arbitrary number of named fields which are used to store the document text and metadata. In order to benefit from

Solr, the content of the wiki needs to be stored in this index, i.e., all resources including their dependencies must be translated into Solr documents.

The main principle of our translation is to materialize joins between content items and their directly associated resources, that is, their contained tags, fragments and links. These materialized joins are then stored in the fields of the document representing the content item. Thus, not only queries over the content and metadata of a content item can be answered directly by Solr, but also queries regarding the data and metadata of tags, fragments and links which belong to the content item. However, the transformation of the resources connected to a content item to fields in the Solr index is lossy since the values of multiple qualifier instances are stored in a single field. Thus, if multiple properties of a resource are queried, it cannot be guaranteed that hits in the index belong to the same resource. Therefore, for certain kinds of queries, it is necessary to validate the result set returned by Solr.

To keep the index small, only dependencies within content items are materialized, omitting in particular nested and linked content items. Under certain circumstances (see Section 10.5.1), queries that only access content items with their content, metadata and directly contained resources can be evaluated entirely in Solr. As soon as nesting and linking of content items comes into play however, we use Solr only to narrow down the set of candidates to those which match those parts of the query for which all necessary information is stored in the Solr index, that is, all content items for which it cannot be determined that they violate a selection condition.

With respect to underspecified queries, there are two different approaches to storing KiWi data in the index and translating the query: Either the query is translated into its corresponding fully specified form (see Section 7.2), or the data are indexed redundantly to cover situations where the context of a keyword is underspecified.

The first case means that the space occupied by the index is smaller since every piece of data is only stored once. However, particularly for vastly underspecified queries, the fully specified form of the query may be very long since all possible combinations of contexts must be covered. This means that more processing time is needed, both for constructing the query and for evaluating it.

In the second case, the translation process does not treat underspecification and the resulting Solr query is as vague as its corresponding KWQL query (or portion of a KWQL query). Instead, the index reduplicates the data in different fields depending on the extent to which the context is specified. Consequently, this second approach is less costly in terms of computation time since

the query translation is simpler and the resulting Solr queries are shorter. However, space requirements here are higher because every piece of data is saved multiple times. For example, a term t that occurs in the title of a content item is stored in the index three times: Once in a field for titles of content items (corresponding to the query `ci(title:t)`), once in a field for all content item qualifiers (`ci(t)`) and once in a catch-all field for the case that no context is given at all (`t`).

Choosing between the two approaches means to either optimize for space or processing time, either of which might be more appropriate in a specific scenario.

Note that this situation is specific to Solr and its current feature list, and the addition of support for wildcards for field names[2] would allow for an implementation of KWQL keyword search that alleviates the need for both calculating the fully specified form of queries and redundant indexing.

As mentioned, not all parts of KWQL can be covered by materialized joins. For instance, the query `ci(title:$t text:$t)` retrieves content items whose title also occurs in their text. It is possible to precompute whether this circumstance is fulfilled and store it in the index, but in combination with partial matching and arbitrary deep nesting of content items, a vast number combinations would have to be saved.

Nevertheless, it is possible to transform the parts of a query which contain variables to an existentially quantified Solr expression to find as many true drops as possible. In the case of mandatory qualifiers such as metadata, such a transformation is useless since this data is present for every resource. But in the case of queries for sub-resources, the transformation can filter out invalid answers. Consider the following two KWQL queries together with their translations in the Solr query language:

```
ci(title:$t text:$t)
```

is translated to

```
type:'kiwi:ContentItem'
```

and

```
ci(tag(name:$t))
```

is translated to

```
type:'kiwi:ContentItem' AND tag_names:[* TO *]
```

2 http://wiki.apache.org/solr/FieldAliasesAndGlobsInParams

In the first case, for the reasons explained in the last paragraph, the two variables cannot be utilized in the query translation and are therefore disregarded. Here, Solr cannot filter out any content items and therefore returns every content item in its index, meaning that the candidate set remains unchanged. The second query, on the other hand, uses a range query to retrieve all content items that have a tag. In this case, the answers of Solr are exactly the answers to the KWQL query and no further processing is needed, since there are no further constraints on the value of the variable that are not captured by the Solr query.

10.4.2 *Structural Constraints*

The second phase evaluates the parts of a query that impose constraints on the data structure. In this step, all resources are represented as common objects in the KiWi system and their dependencies are modeled by references between the interrelated objects. The objects are persisted using a relational database in combination with an object-relational mapping.

In the current prototype, we validate the structural properties of a query individually for each candidate content item. That means, constraints on related resources (contained tags, fragments and links, and linked and nested content items) which are specified in the query are evaluated by traversing the object currently being investigated.

The structure of the Wiki is represented in the database through pointers between objects. For instance, each content item has a list containing pointers to nested content items, and to tags that have been assigned to the content item. To verify the structure of a candidate, these pointers are traversed recursively and the properties of each visited object are examined.

We chose this approach since structural constraints are often validated fairly quickly and far less selective than the keyword portions of KWQL queries. However, for future work we envision an extension of KWilt that improves the current implementation in two aspects: (a) It estimates whether the structural part is selective enough to warrant its execution without considering the candidates from the previous phase, followed by a join between the candidate sets from the two phases. (b) If structural constraints become more complex, specialized evaluation engines for hierarchical (XML-style) data, e.g, a high-performance XPath engine, for link data and various graph reachability indices for RDF data might be advantageous. We are also considering the use of a multi-format, multi-language database such as MonetDB [66] that supports efficient hierarchical data access through XPath and XQuery engines as well as efficient RDF navigation.

In the second evaluation phase of our example query

```
ci(KiWi tag(name:$t) link(target:ci(tag(name:$t))))
```

every content item still in the candidate set is loaded into memory. The qualifiers of the candidate are then examined until one of them matches the keyword "KiWi." In this specific example, this step is not strictly necessary since "KiWi" may occur in any qualifier and Solr queries can fully capture this condition, but other queries do require it. For example, if we change the query to

```
ci(tag(name:KiWi author:Mary) tag(name:$t)
    link(target:ci(tag(name:$t))))
```

every match returned by Solr must be verified to ensure that one and the same tagging has both "KiWi" in its name and "Mary" as its author, since Solr cannot distinguish between different instances of the same sub-resource.

Consequently, the links of the content item are successively examined by traversing the references of type "link" of the content item. This yields one content item, the target of the link, for each traversed reference, which is in turn examined by traversing the references pointing to tags.

Finally, content and structural selection conditions on the linked content item are verified, here by merely checking whether the content item in question has been tagged at least once.

USING TERNARY LOGIC TO TREAT VARIABLES The second evaluation phase treats constraints pertaining to the structure of resources, but does not consider injectivity. But to decide whether a possible answer generated by Solr should be further regarded or can be safely dropped, the concept of a ternary logic is needed.

In order to illustrate the necessity of this logic, consider the following two examples:

```
ci(title:$t tag(name:$t))
```

```
ci(title:$t NOT(tag(name:$t)))
```

In a first approach, any occurrence of a variable can be interpreted as a valid match. This seems reasonable, as variables can only be stated before qualifiers and each qualifier of every resource always has a (possibly empty) value. This strategy works well for queries of the first kind. Given that the currently analyzed content item has a tag, the recursive traversal of the content item will evaluate to "true."

However, considering the second query, this naive approach does not show the desired behavior. Again, if the analyzed content item has a tag, `tag(name:$t)` will be regarded as a match, but as this part of the query is negated, the recursive traversal will always evaluate to "false." Therefore, every content item in the candidate set that has a tag will be filtered out regardless of whether the tag label matches its title or not.

As this example indicates, it depends on the examined resource and the query, if a variable needs to be evaluated to "true" or "false." To deal with this problem, ternary logic, first introduced by Łukasiewicz [257], is used. Ternary logic is an extension of the classical boolean logic by a third value which can be used to indicate uncertainty. Table 17 shows the truth tables for the operators **`and`**, **`or`** and **`not`** in ternary logic.

a \ b	false	unknown	true
false	false	false	false
unknown	false	unknown	unknown
true	false	unknown	true

(a) $a \wedge b$

a \ b	false	unknown	true
false	false	unknown	true
unknown	unknown	unknown	true
true	true	true	true

(b) $a \vee b$

a	false	unknown	true
$\neg a$	true	unknown	false

(c) $\neg a$

Table 17: Truth tables for the operators **`and`** (17a), **`or`** (17b), and **`not`** (17c)

Ternary logic is employed to adjust the recursive traversal used to refine the result set. In addition to the boolean values "true" and "false" which indicate whether a resource matches a query or not, the value "unknown" can be used to represent variable bindings. This is needed, since variables can, depending on their binding, either match a query or not. But as this part of the KWQL evaluation does not treat variable bindings, it cannot know whether a certain variable should be regarded a match or not. Moreover, having this third value avoids the problem of needing to determine whether a variable matches the value of a qualifier or not, which was shown to depend on the individual query.

Given that the recursive traversal evaluates to "false," the analyzed result can be safely dropped from the result set as it, independently of the variable bindings, cannot fulfill the constraints specified by the query. However, if it evaluates to "true," it would be premature to conclude that the content item is a valid match since there are additional constraints stemming from the injectivity property of KWQL which are not checked in this step either.

But if we can discern between matches that also need to satisfy additional injectivity constraints and those that can be completely verified in the given step, it becomes possible to distinguish between true hits and content items that must remain in the candidate set. The three way distinction in ternary logic then corresponds to the distinction between true hits ("true"), true drops ("false") and elements of the candidate set ("unknown") and we can say that the KWilt evaluation scheme in general is based on ternary logic.

10.4.3 *First-Order Constraints*

In the final evaluation phase, first-order constraints over wiki resources, induced by the KWQL variables and injectivity, are evaluated using choco [207, 228], a state-of-the-art open source constraint solver. It offers efficient implementations of popular constraint types and supports the use of arbitrary relations. These characteristics make choco a good fit for the evaluation of variables and the evaluation of injectivity constraints.

Following constraint programming notation, we consider a first-order constraint a formula over logical relations on several variables. In order to use these constraints to express a KWQL query, every expression of a query that is involved in constraints not yet fully validated is represented by some variables. These variables are then connected using relations which reflect the structural constraints between the expressions from the query.

These relations are collected during a recursive traversal of the elements in the candidate set. Here, values of qualifiers and the structural dependencies of their resources are stored in a relation structure. More precisely, for each required qualifier, a tuple containing an identifier for the respective resource and the value of the qualifier is inserted into the appropriate relation. When processing our example query, the names of the tags and the information which tag is contained in which content item is stored in a relation.

For instance, to express that a content item has a certain title, the relation R_{title} is used: $(C, \mathit{KiWi}) \in R_{\text{title}}$. This constraint causes the variable C to be bound only to identifiers of content items

with the title *"KiWi"*. Likewise, the relation R_{tag} is used to specify that a content item has a tag: $(C, T) \in R_{tag}$.

For each KWQL variable in the query, a new first-order variable is generated that can be used in the structural constraints. For instance, the query

```
ci(title:$t tag(name:$t))
```

can be represented as: $(C, \$t) \in R_{title} \wedge (C, T) \in R_{tag} \wedge (T, \$t) \in R_{name}$.

The formalization of this query uses the relation R_{title} to gather content item ids and titles and stores them in the variables C and $t: $((C, \$t) \in R_{title})$. These variables are then used to express that the corresponding content item should be tagged $((C, T) \in R_{tag})$ and that the tag should share the content item's title $((T, \$t) \in R_{name})$.

```
ci(tag(name:wiki) tag(name:wiki))
```

Consider the preceding query, which matches content items that have been tagged with "wiki" at least twice. In addition to the constraints which are generated as described, a global "all distinct" constraint ensures that all tags of the answer have a different id, hence that there are two different "wiki" tags:

$$\begin{aligned}&\text{alldistinct}(\$tag_1, \$tag_2) \wedge \\ &\Big((\$ci, \$tag_1) \in R_{tag} \wedge (\$tag_1, \text{wiki}) \in R_{name} \\ &\qquad\qquad \wedge\ (\$ci, \$tag_2) \in R_{tag} \wedge (\$tag_2, \text{wiki}) \in R_{name}\Big)\end{aligned}$$

Note that the injectivity property is only local to a resource and sub-resources have their own injectivity properties. Therefore each resource of a query needs its own "all distinct" constraint as well, since different sub-resources of a query may reference the same resource without violating the injectivity.

However, there are cases which cannot be handled that easily and require further adoptions of the presented approach.

A DISJUNCTIVE NORMAL FORM FOR INJECTIVITY Consider the following query:

```
ci(tag(name:"knowledge management") OR (tag(name:wiki)
    tag(name:wiki)))
```

The third step of the evaluation of this query may lead to a problematic situation where the variables of the second disjunct of the query remain unbound. More precisely, in such a situation the variables $\$tag_2$ and $\$tag_3$ are set to -1 and the constraint

alldistinct($\$tag_1$, $\$tag_2$, $\$tag_3$) will always fail, as the values of $\$tag_2$ and $\$tag_3$ are both -1 and therefore equal.

One simple solution to this problem is to use different (negative) ids to mark the variables as unbound, but this is unnatural since different values suggest different resources bound to them whereas in reality they are completely unbound. Additionally, this would make it necessary to keep track of the values which have so far been used during the conversion of the query.

For a more natural solution, the disjunction of the query must be handled differently. While the current approach assumes that all tags which occur in the same resource of a query must be distinct, this is no longer valid if disjunctions are taken into consideration. Through disjunction, constraints which would have applied for the whole resource in the conjunctive case, are broken into two parts which are independent of each other. In the example at hand, this means that instead of one "all distinct" constraint two constraints are necessary, namely alldistinct($\$tag_1$) and alldistinct($\$tag_2$, $\$tag_3$). The first constraint is trivially always satisfied and therefore can be disregarded.

Since conjunctions and disjunctions can be arbitrarily nested, it may be hard to determine which parts of the query belong together. Consider the following abstracted version of a KWQL query, where tags are only represented by a short form, without any of their qualifiers.

```
ci(tag1 AND (tag2 OR (tag3 AND tag4)))
```

As demonstrated by the previous example, the all distinct constraint cannot be used in combination with disjunctions which therefore need to be eliminated. This can be achieved by transforming the expression into disjunctive normal form and processing all obtained disjuncts independently. This procedure is admissible because disjunctions represent alternatives in the query and during the evaluation all these alternatives are tested one after another. Thus only the particular injectivity constraint which is responsible for the currently evaluated alternative needs to be regarded. This can easily be achieved by testing whether all variables used by the constraint are bound.

$$(\$tag_1 \neq -1 \wedge \$tag_2 \neq -1) \Rightarrow \text{alldistinct}(\$tag_1, \$tag_2)$$

$$(\$tag_1 \neq -1 \wedge \$tag_3 \neq -1 \wedge \$tag_4 \neq -1) \Rightarrow \text{alldistinct}(\$tag_1, \$tag_3, \$tag_4)$$

In order to generate the disjunctive normal form of a query, the operators $\cup$ and $\times$ with the following semantic are used. Suppose A and B are families of sets, then

$$A \cup B := \{X \mid X \in A \vee X \in B\}$$
$$A \times B := \{a \cup b \mid a \in A \wedge b \in B\}$$

By substituting **AND** by $\times$ and **OR** by $\cup$ in the preceding query, these two operators can be used to generate a family of sets over KWQL variables. This family has the property that each contained set contains exactly those variables which occur in one of the disjuncts of the disjunctive normal form.

$$\begin{aligned}
&\{\{tag_1\}\} \times \Big(\{\{tag_2\}\} \cup \Big(\{\{tag_3\}\} \times \{\{tag_4\}\}\Big)\Big) \\
&= \{\{tag_1\}\} \times \Big(\{\{tag_2\}\} \cup \{\{tag_3, tag_4\}\}\Big) \\
&= \{\{tag_1\}\} \times \{\{tag_2\}, \{tag_3, tag_4\}\} \\
&= \{\{tag_1, tag_2\}, \{tag_1, tag_3, tag_4\}\}
\end{aligned}$$

Using this formalism, constraints establishing injectivity can be generated which are only applied to the necessary variables and therefore can be used together with disjunctions without dropping valid responses.

Generating the disjunctive normal form of a query is not only useful in the context of injectivity constraints and variables, but can be used as a simple way to implement true drops, which are not yet realized in the current KWilt prototype. As an illustration, consider the query

```
ci(java OR link(target:java))
```

This query returns all content items that contain “java” or that link to a content item containing “java.” When this query is evaluated, the first step does not remove any content items from the candidate set, since the second disjunct cannot be converted to a Solr query. As a consequence, the second query step must sift through all content items in the wiki to retrieve the results.

However, when the query is converted into its disjunctive normal form, `ci(java)OR ci(link(target:java))`, the two disjuncts can be evaluated separately. All content items that match the first disjunct, i.e. that contain “java” are then true hits that can be removed from the candidate set and need not be considered when evaluating the second disjunct query. As a consequence, fewer content items have to be treated in this step.

One limitation of this approach is that treating multiple query disjuncts is unnecessary when both disjuncts require the same amount of evaluation steps, and that the disjunct queries should be ordered in increasing complexity. Before beginning the query evaluation, the number of steps necessary to evaluate each disjunct query must therefore be determined.

10.5 KWQL SUBLANGUAGES

As described in the previous section, KWQL queries in KWilt are evaluated in three phases. However, not all evaluation phases are required for every KWQL query. In the following, we give a characterization of KWQL queries that can be evaluated using only the first phase (and skipping the remaining ones), or only the first and second phases.

10.5.1 *Keyword KWQL*

Keyword KWQL or KWQL_K is the restriction of KWQL to flat queries.

Since tags and fragments can not be nested more than one level, we can materialize all sub-resources for each content item. However, in contrast to (string-valued) qualifiers, a content item can have multiple tags or fragments. To allow evaluation with an information retrieval engine such as Solr, we have to ensure that multiple tag or fragment expressions always match with different tags or fragments of the surrounding content item. This avoids that we have to enforce the injectivity in a later evaluation phase.

To ensure this, we allow fragment, link, and tag queries but disallow

- Two keywords or qualifier terms as siblings expressions in a sub-resource term.
- Two sub-resources terms as sibling expressions.
- Variables not used in the function of wildcards (see Section 7.2.3). That means that the query head, if given, may not contain any variables and no variable may occur more than once in the query body.
- Structure qualifiers, that is, **`child`**, **`descendant`**, and **`target`**.

KWQL_K expressions can be evaluated entirely by the information retrieval engine, here Solr. This is obvious for keywords. String-valued properties are materialized in Solr together with their resources (as specific fields) and thus can be queried through Solr as well.

KWilt determines whether a query can be fully expressed in KWQL_K during the pre-processing step of the query. If this is the case, only the first step of the evaluation is performed.

10.5.2 *Tree-shaped KWQL*

Tree-shaped KWQL or KWQL_T allows only queries corresponding to tree-shaped constraints. Thus, no multiple occurrences of the

same variable, and no potentially overlapping expression siblings are allowed.

Intuitively, two expressions are called *overlapping* if there is a KWQL node in any document that is matched (i.e., returned by their semantics according to Chapter 7.3) by both expressions. For example, `ci(tag(name:Java))` and `Java` are overlapping. Unfortunately, this definition of *overlapping* does not lead to an efficient syntactic condition, as it is easy to see that containment of KWQL queries (see Section 7.2.3) is a special case of overlapping. Further, containment of KWQL queries is NP-hard by reduction from containment of conjunctive queries.

Therefore, we define an equivalence relation on expressions, called *potential overlap*, as a conservative approximation of overlapping. It holds between two expressions if they have the same return type in the KWQL semantics or if the return type of one is a subset of that of the other one. E.g., `descendant:ci(Lucene)` and `child:ci(Java)` potentially overlap, but `target:ci(Java)` does not overlap with either. This is only an approximation. For instance, `child:ci(URI:a)` and `child:ci(URI:b)` potentially overlap, though each content-item has a unique URI and thus the two expressions never actually overlap.

KWQL_T expressions can be evaluated by using only Solr and checking the remaining structural conditions in the second evaluation phase. Full first-order constraints are not needed and the third (choco) phase can be skipped.

Given an arbitrary KWQL query, we can decide in linear time and space in the size of the query if that query is a KWQL_K query and in quadratic time if it is a KWQL_T query.

Proof. From the definitions of KWQL_K and KWQL_T it is easy to see that testing membership of a general KWQL expression can be done by a single traversal of the expression tree. In the case of KWQL_T we also have to test each (of the potentially quadratic) pairs of siblings for overlap and storing already visited variables. □

The test whether a query is a KWQL_T query can actually be achieved as a side effect of transforming the KWQL query into first-order constraints in the third evaluation phase: Only if certain first-order constraints are generated during this transformation we need to execute the constraint solver at all. In practice, this is often cheaper than a separate test, as the generation of first-order constraints is fairly cheap and polynomial, except for queries with many potentially overlapping expression siblings.

10.6 EVALUATING A KWQL QUERY IN KWILT

In order to illustrate the different evaluation phases of our framework, the evaluation of the final query from the introduction is

described here in detail, showing for each phase which of the KWQL constraints are solved.

```
$u @ ci(KiWi tag(name:$t) link(target:ci(URI:$u
    tag(name:$t)))
```

First, the query is partially translated to the Solr query language. In the Solr index, the metadata of a content item is stored together with the metadata of the fragments, links, and tags which are directly connected to the content item. Thus, not only the keyword "KiWi" is included in the query but also the query for the tag, but since Solr does not support variables, the query for the tag is just an existence constraint (indicated by **`[* TO *]`** as the value of the **`tags`** qualifier). Nevertheless, its inclusion is reasonable as there might be content items without any tags.

```
type:ci AND (title:KiWi OR text:KiWi OR ...)
        AND tags:[* TO *]
```

The result of evaluating this query using Solr is the set of content items which have "KiWi" among their qualifier values and that have at least one tag. Any other constraints of the KWQL query have to be validated in the subsequent phases.

In the second evaluation phase, the structural properties of the content items are validated against the query constraints. In order to gain full access to all properties of the content item, not just the simplified version that is stored in the Solr index, the full representation of the resources is retrieved from the KiWi database.

For instance, consider a content item with the identifier C_1 that describes the KiWi project and is titled "KiWi — Knowledge in a Wiki." In the text of the content item is a link to another content item (C_2) on IkeWiki, the predecessor of the KiWi wiki. Furthermore, both content items are tagged with "wiki" and the content item describing KiWi is additionally tagged "knowledge management." Only C_1 is contained in the candidate set of the Solr query, since it both contains "KiWi" and has been assigned a tag. Therefore it is loaded into the main memory for further investigation of the actual structure of the content item.

A candidate content item is dropped from the candidate set if it does not have a link, or has a link to a content item that has no tags.

In the third evaluation phase, the necessary relations are generated.

In the case of the example query, all tags of the candidate content item C_1 are considered and the corresponding tuples (T_1, wiki) and $(T_2, \text{knowledge management})$ are stored in the relation R_{name} where T_i is the identifier for the i-th tag. In addition

to that, the structural information about the tags is stored in the relation R_{tag} by adding (C_1, T_1) and (C_1, T_2). Finally, the link reference is resolved to C_2, the content item describing IkeWiki, and the name of the tag of this content item is stored in the appropriate relation as well. Overall, the following relations are generated:

$$\begin{aligned} R_{name} &= \{(T_2, \text{wiki}), (T_1, \text{wiki}), \\ &\qquad (T_1, \text{knowledge management})\} \\ R_{tag} &= \{(C_1, T_1), (C_1, T_2), (C_2, T_3)\} \\ R_{link} &= \{(C_1, L_1)\} \\ R_{target} &= \{(L_1, C_2)\} \\ R_{URI} &= \{(C_1, C_2)\} \end{aligned}$$

Next, the valid variable bindings are determined. Therefore, the query, or rather the still unverified parts of the query, are expressed in a way suitable for the constraint solver choco. To this end, the relations containing the information about the resources are connected by variables.

$$\begin{aligned} &(C_1, T_1) \in R_{tag} \wedge (T_1, \$t) \in R_{name} \wedge (C_1, L) \in R_{link} \\ \wedge\ &(L, C_2) \in R_{target} \wedge (C_2, \$u) \in R_{URI} \wedge (C_2, T_2) \in R_{tag} \\ \wedge\ &(T_2, \$t) \in R_{name} \end{aligned}$$

Note that the title does not occur in this formal representation of the query, since the last evaluation phase guarantees that all content items of the candidate set have *KiWi* occurring in one of their qualifiers. The constraint solver then tries to determine bindings for all variables which satisfy the given constraint. The variable `$t` ensures that both tags of the query have the same name, whereas `$u` is used to obtain the URI of the content item that is pointed at, which will be returned as an answer to the query.

If the constraint solver does not succeed in finding a valid binding for the variables, the content item is dropped from the candidate set, since it does not fulfill the constraints and therefore does not match the query.

10.7 PERFORMANCE EVALUATION

To analyze the performance of the KWilt prototype, the evaluation times of various queries of all three types were measured. For the experiment, the KiWi system was executed on a virtual server with a dual core 2.5 GHz processor and 4 GB of RAM running Ubuntu Linux.

Since KWilt is part of the KiWi system and the two cannot be run separately, other tasks running in KiWi such as reasoning,

query						
phase 1		phase 2		phase 3		total time
time [ms]	results	time [ms]	results	time [ms]	results	[ms]
KiWi						
31	14	–	–	–	–	31
ci(title:"KWQL Examples")						
5	1	–	–	–	–	5
ci(text:KWQL title:KiWi)						
6	1	–	–	–	–	6
KiWi tag(name:$t)						
42	9	–	–	–	–	42
NOT(tag(name:$r))						
463	330	–	–	–	–	463
ci(tag(name:KWQL)child:ci(tag(Example)))						
10	5	44	1	–	–	54
ci(Munich link(target:ci(KiWi)))						
33	4	52	4	–	–	85
ci(KiWi link(target:ci(KiWi)))						
60	10	206	4	–	–	266
ci(link(target:ci(KiWi)))						
520	339	9963	4	–	–	10483
ci(tag(name:$r)text:$r)						
149	9	53	9	103	9	305
ci(title:$r text:$r)						
570	254	5	254	81	59	656
ci(tag(author:admin)tag(name:KWQL))						
9	5	55	5	67	4	131
ci(KiWi tag(name:KiWi) link(target:ci(URI:$u tag(name:KiWi))))						
11	4	164	2	188	2	363
ci(KiWi tag(name:$t)link(target:ci(URI:$u tag(name:$t))))						
44	9	181	4	194	3	419
ci(tag(name:$t)link(target:ci(URI:$u tag(name:$t))))						
165	9	268	4	178	3	611
ci(tag(name:$a)tag(name:$b)tag(name:$t))						
163	9	225	9	177	7	565

Table 18: Evaluation times in the KiWi dataset

the calculation of recommendations and the automatic importing of data and other maintenance tasks influence and distort the measurements taken. To minimize the amount of background activity in the system, the reasoning and information retrieval modules of KiWi were deactivated. Further, every query was evaluated fifty times and the average was taken. While these measures may alleviate the influence of varying background activities in the system on query evaluation times, the results presented in the following still should be taken as approximate measurements. In particular, small differences of few milliseconds should not be considered meaningful or significant.

In a first experiment, a number of queries, among them our example query from the introduction, were evaluated on a dataset consisting of 339 content items on the KiWi project. For all queries, preprocessing, that is, parsing, verification and determining whether the query can be fully processed using only phase 1, was found to take between 27 and 42 milliseconds. Table 18 further shows the processing times and number of results per query and processing phase. The first part of the table gives the numbers for queries that are covered by KWQL_K. As the results show, these queries can overall be evaluated fairly quickly and the evaluation time is roughly proportional to the number of results. The last KWQL_K query, `NOT(tag(name:$r))`, has an evaluation time of nearly half a second, which, apart from the fact that many documents match the criterion, is likely caused by the combination of negation and a wildcard, both of which require an evaluation that is more computationally intensive than simple term matching.

The second group of queries displayed in the table are those that can be evaluated using KWQL_T. As the table illustrates, those queries can be evaluated quickly, but only if the first evaluation step has sufficiently reduced the candidate set. One underlying assumption behind KWilt is that most queries exclusively or predominantly use value-based selection criteria, that is, selection criteria that can be covered by the information retrieval engine in the first phase of the evaluation. When this assumption does not hold, the candidate set still contains a considerable amount of content items after the first evaluation phase. As the second evaluation phase is considerably slower than the first, evaluation times in such a situation can become very high. For example, the fourth KWQL_T query in table 18 consists of a single structural selection criterion. This means that the first evaluation phase retrieves all content items in the system, each of which is then traversed in the second evaluation phase. This effect is further amplified by the fact that, as observed, Solr queries are evaluated the quicker the fewer results they yield. Correspondingly, the evaluation times

query						
phase 1		phase 2		phase 3		total time
time [ms]	results	time [ms]	results	time [ms]	results	[ms]
`tag(author:Mary)`						
48	35	–	–	–	–	48
`semantic web`						
51	22	–	–	–	–	51
`tag(name:web author:Peter)`						
99	34	1769	34	–	–	1868
`ci(link(target:ci(semantic)))`						
644	2049	43123	0	–	–	43767
`ci(example title:$r text:$r)`						
105	38	21	38	15	1	141
`ci(title:$r text:$r)`						
679	505	15	505	75	58	769
`ci(tag(name:web)tag(author:Peter))`						
92	34	863	34	3246	34	4201

Table 19: Evaluation times in the RSS dataset

for all four KWQL_T queries is roughly inversely proportional to the size of the candidate set after the first evaluation phase.

Finally, the lower seven queries in the table make use of the full power of KWQL and require all three evaluation phases. The time taken for the third evaluation phase here is not directly related to the size of the candidate set and `ci(title:$r text:$r)`, the query with the biggest candidate set after the second evaluation phase takes the shortest amount of time for evaluation phase three.

Overall, the results of this first experiment show that KWQL_T queries can be evaluated using Solr with only little overhead for preprocessing the query. However, Solr queries involving wildcards are evaluated comparatively slowly. More critically, the second evaluation phase constitutes a bottleneck in the query evaluation process, particularly when the first phase does not sufficiently decrease the size of the candidate set.

To investigate the effect of data size on query evaluation times, another experiment was conducted using a dataset consisting of 2049 content items. The dataset was created by importing a number of RSS feeds of technology blogs into the KiWi wiki. The resulting content items were tagged but contained no fragments, nestings or links to other content items.

As in the previous experiment, every query was evaluated fifty times and the average was taken. Query preprocessing took between 36 and 53 milliseconds, slightly more than it did using the smaller KiWi dataset. While the difference is small and thus potentially insignificant, it may indicate that the KiWi wiki, itself a research prototype, overall shows slower performance as the size of the dataset increases.

The results of this second experimental evaluation are shown in Table 19. As in the experiment using the smaller dataset, KWQL_K queries are evaluated very quickly, and the nearly sevenfold increase in the size of the dataset has little effect on the evaluation times.

Processing times for KWQL_T queries, the second group of queries, on the other hand show that this processing phase becomes very slow as the amount of data increases. Given that in this evaluation phase all candidate content items are traversed one by one, the time taken increases linearly with the amount of candidate content items. The second phase already performed slowly when using a smaller dataset, and the increase in dataset size only increases this problem, bringing the evaluation time of the query `ci(link(target:ci(semantic)))` to prohibitively long 43.8 seconds.

The processing times in phase three change in a way that is less easy to interpret: Some full KWQL queries such as `ci(title:$r text:$r)` stay roughly constant in their per-content-item processing time, while the same measure increases more than sevenfold when the phase three processing time of the following two queries is compared:

```
ci(tag(name:KWQL) tag(author:admin))
```

```
ci(tag(name:web) tag(author:Peter))
```

In summary, the evaluation of KWilt shows that the approach overall is viable and delivers good results as long as the underlying assumption holds true, namely that most selection criteria used in queries are value-based. As long as this is true, KWilt can employ Solr which quickly evaluates the query, either in total or by reducing the candidate set to a size that is manageable for the following evaluation phases.

These, in particular the second evaluation phase, constitute the weak point of KWilt as it is currently implemented: The simple traversal of all candidate content items that constitutes the second phase in the current implementation performs very slowly. When a query does not use mainly value-based selection criteria or when the dataset is big, the size of the candidate set is

not sufficiently decreased in the first evaluation phase and the second evaluation phase can take several seconds or longer.

The third evaluation phase using choco fares better with respect to performance and scaling and overall delivers good results with processing times generally not exceeding 100 milliseconds. In one case, discussed above, the third evaluation phase scales comparatively bad at an execution time of about 3.2 seconds.

Overall, the system delivers good results, but changes to the system are required to improve the performance of the second evaluation phase. The following section discusses possible steps that could be taken.

10.8 OUTLOOK

KWilt as described in this chapter is integrated in the KiWi system and part of the current release. However, there is still room for further improvements:

Not all features for KWQL have been fully implemented and some constructs that are described in the language definition (see Chapter 7.2) are missing from the current implementation. Specifically, the features that have not been implemented so far are:

- the **`OPTIONAL`** operator
- the binding of partially matched values to a variable using the **`->`** operator
- the construction of new content items
- the querying of structured tags (discussed in Chapter 12)

Fortunately, all these features are either syntactic sugar for already supported constructs, or can be easily integrated into the current implementation. For instance, a **`OPTIONAL`** b is equivalent to $(a \wedge b) \vee (a \wedge \neg b)$, which can already be expressed in the current implementation. However, using this equivalence to solve the problem is inefficient, since a is evaluated twice and a more efficient evaluation is desirable.

Since variables are already supported, the implementation of the **`->`** operator requires only minor adaptions to validate if a value matches the given criteria before actually binding it to the variable.

The implementation of content item construction contains is straightforward, especially since the API of the KiWi system provides various functions for the generation of new content items.

The implementation and querying of structured tags, still missing from KiWi and KWQL, are described in Chapter 12.

Additionally to these features not yet implemented, due to the way that links are currently represented in the KiWi wiki, it is not yet possible to query the anchor text and tags of links.

Apart from features not yet implemented, several improvements could be made to the current implementation:

As the performance evaluation has shown, the integration of a more efficient evaluation approach for structural constraints as discussed in Section 10.4.2 would be a major improvement to KWilt.

Despite the two possibilities discussed there, namely an evaluation strategy more closely tailored to the individual queries and their keyword and structure constraints and a reimplementation of the second evaluation phase using web querying technology, two further changes could be employed to improve query performance:

- Some structural properties could be represented in the index, thereby making it possible for Solr to partially take structural constraints into account. While saving all information about structurally connected content items in the index representation of a content item is clearly not practicable, some basic structural information could be represented. For example, the index could indicate whether a content item has any children or links to any other content items. Depending on how frequently nesting and linking relations are used in the wiki, this information could then help narrow down the candidate set, meaning that fewer content items have to be processed in the second evaluation phase.

- In a more comprehensive realization of the second suggestion in Section 10.4.2, queries that cannot be fully evaluated using Solr could be handled through a translation into SQL that treats both the second and third evaluation phases. The relational semantics given in Section 7.3 can serve as a basis for such a translation of KWQL into SQL. The resulting alternative implementation of KWQL would not be based on the principle of gradually refining the query results like KWilt, but rather on choosing the best-suited tool before query evaluation begins. An evaluation of the resulting system could also show whether the use of Solr is justified, or whether translating fully translating KWQL into SQL is preferable.

Finally, the current prototype still evaluates some constraints in more than one phase and it may be possible to avoid re-evaluating certain structural constraints again in the constraint solver, for example by providing only a summary view to the constraint solver, rather than the detailed structural constraints.

Part IV

EXTENSIONS

11

PEST: APPROXIMATE QUERYING OF GRAPH-STRUCTURED DATA

Mary wants to get an overview of software projects in her company that are written in Java and that make use of the Lucene library for full text search. According to the conventions of her company's KiWi wiki, a brief introduction to each software project is provided by a content item tagged with *"introduction"*.

Thus, Mary enters a KWQL query that retrieves content items containing "java" and "lucene" that are also tagged with "introduction": `ci(java lucene tag(introduction)) .`

However, the results fall short of Mary's expectations for two reasons that are also apparent in the sample wiki of Figure 61:

(1) Some projects may not follow the wiki's conventions (or the convention might have changed over time) to use the tag "introduction" for identifying project briefs. This may be the case for Document 5 in Figure 61. Mary could loosen her query to retrieve all pages containing "introduction" regardless of whether the term appears as a tag or not. However, in this case pages that follow the convention are not necessarily ranked higher than other matching pages.

(2) Some projects use the rich annotation and structuring mechanisms of the KiWi wiki to split a wiki page into sub-sections, as in the case of the description of KiWi in Documents 1 and 2 from Figure 61, and to link to related projects or technologies (rather than discuss them inline), as in the case of Document 4 and 5 in Figure 61. Such projects are not included in the results of the original query at all. Again, Mary could try to change her query to allow keywords to occur in nested or in linked content items, but such queries quickly become rather complex (even in a flexible query language such as KWQL) or impossible with the limited search facilities most wikis provide. Furthermore, this solution suffers from the same problem mentioned above: Documents following the wiki's conventions are not necessarily ranked higher than those only matched due to the relaxation of the query.

Though we introduce these challenges in the context of the KiWi wiki and KWQL, they appear in a wide range of applications involving (keyword) search on structured data, for example in social networks, in ontologies, or in richly structured publication repositories. The common characteristic of these applications is that relevant answers (e.g., a wiki page or a person in a social network) are not fully self-contained documents as in the case

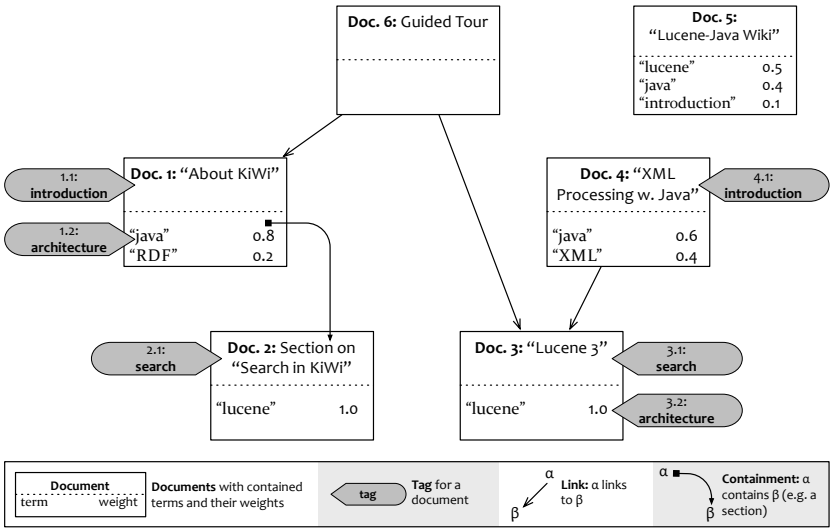


Figure 61: Link and containment graph for a sample wiki

of standard web search, but obtain a big part of their relevance by virtue of their structural relations to other pages, persons, etc. At the same time, they are sufficiently self-contained to serve as reasonable answers to a search query, in contrast to, e.g., elements of an XML document.

Since data items such as wiki pages are often less self-contained than common web documents, PageRank and similar approaches that use the structure of the data merely for ranking of a set of answers do not suffice: As Figure 61 illustrates, also pages that do not contain the relevant keyword can be highly relevant answers due to their relations with, e.g., tags that prominently feature the keyword.

To address this challenge, not only the ranking, but also the selection of answers needs to be influenced by the structure of the data.

In this chapter, we generalize the idea of term propagation over structured data: PEST, short for short for term-**p**ropagation using **e**igenvector computation over **st**ructural data, is a novel approach to approximate matching over structured data. PEST is based on a unique technique for propagating term weights (as obtained from a standard vector-space representation of the documents) over the structure of the data using eigenvector computation. It generalizes the principles of Google's PageRank [71] to data where the content of a data item is not sufficient to establish the relevant terms for that item, but where rich structural relations are present that allow us to use the content of related data items to improve the set of keywords describing a data item.

In contrast to many other fuzzy matching approaches (see Section 11.1), PEST relies solely on modifying term weights in the document index and requires no runtime query expansion,

but can use existing document retrieval technologies such as Lucene. Furthermore, the modified term weights represent how well a data item is connected to others in the structured data and therefore one can omit a separate adjustment of the answer ranking as in PageRank.

PEST's computation can be performed entirely at index time. Yet, PEST is able to address all the above described problems in the context of structured data with meaningful answers such as a wiki, a social network, etc. To illustrate how PEST propagates term weights, consider again Figure 61. As in PageRank, the "magic" of PEST lies in its matrix, called the PEST *propagation matrix*, or PEST matrix for short. The PEST matrix is computed in two steps:

(1) Weighted propagation graph: First, we extend and weight the graph of data items (here, wiki pages and tags).

(a) For each wiki page without tags, a *dummy tag* without term weights is created. (b) For each link (containment relation) between two wiki pages A and B, a link (containment relation) between each pair of tags from A and B is added.

These insertions are used to enable direct propagation between tags. Therefore, we can configure how terms propagate between tags of related pages independently from term propagation between the pages.

(c) Each edge in the resulting graph is assigned a pair of *weights*, one for traversing it from source to sink and one from sink to source.

In general, if we have multiple types of relations and data items, this extension allows us to have strong connections between related properties of related items, e.g., between the classes of two highly related instances in an ontology.

The resulting graph for the sample wiki is shown in Figure 62. We have added the tags 5.1 and 6.1 and containment edges from tag 1.1 and 1.2 to tag 2.1, as well as link edges, e.g., from the tag 6.1 to tag 1.1, 2.1, 3.1 and 3.2. In the following, we assign edge weights based solely on the type of the edge (link, containment, tagging), though PEST could also allow edge-specific weights.

(2) "Informed Leap": The weighted propagation graph does not encode any information about the differences in *term distribution* in the original nodes, but only information about the structure of the wiki graph. To encode that information in the PEST matrix, we use an *informed leap*: First, we transpose the weighted adjacency matrix of the weighted propagation graph and normalize it. To preserve differences in absolute edge weights between data items, we choose a normalization that is independent of the actual edge weights and the result is a sub-stochastic rather than stochastic matrix. Second, these remaining probabilities together with a fixed leap parameter α (e.g., 0.3) is used for an informed leap to an arbitrary node. The probability to leap to a node A in

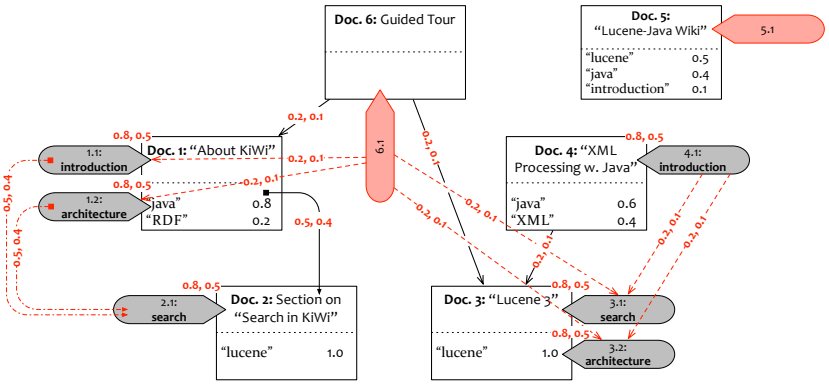


Figure 62: Edge weights and virtual nodes and edges for the graph in Figure 61

such an informed leap is not equal for all nodes (as in the case of PageRank), but depends on the original term weight distribution: A page with high term weight for term τ is more likely to be the target of a leap than one with low term weight for τ.

The resulting matrix is called the PEST matrix $\mathbf{P}_\tau$ for term τ. It must be computed individually for each term, but does not depend on the query. Finally, the eigenvectors of the PEST matrix for each term τ are combined to form the vector space representation (i.e., the document-term matrix) for the data items (here, content items and their tags). Keyword queries can be evaluated on this representation with any of the existing IR engines using a vector space model (e.g., Lucene). Queries mixing keywords and structure require an engine capable of combining keyword matches with structural constraints such as the KWQL engine KWilt (see Chapter 10).

It is worth emphasizing that only the second step is term-dependent and, as shown in the experimental evaluation in Section 11.5, the time needed for the calculation of the term-dependent part of the evaluation is in the low seconds on a single core for each term, even without sophisticated optimizations, and can be computed independently for each term. Thus, PEST's index computation scales well even for document collections containing hundreds of thousands of relevant terms (a significant portion of the English vocabulary).

Contributions

To summarize, PEST improves on approximate search approaches for structured data (summarized in Section 11.1) in the following aspects:

(1) It is based on a simple, but flexible model for structured content that captures a wide range of knowledge management systems and (tree- or graph-shaped) structured data applications.

We introduce the model in Section 11.2 and discuss how it can represent the core concepts of the KiWi wiki.

(2) The main contribution of PEST is the PEST matrix for propagating term weights over structured data. The computation of that matrix for a given graph of structured content is formalized in Section 11.3.
The PEST matrix allows the propagation of term weights at index time and yields a modified vector space representation that can be used by any IR engine based on the vector space model (e.g., Lucene).

(3) We prove in Section 11.3.3 that any PEST matrix has 1 as dominant eigenvalue and that the power method converges with the corresponding eigenvector if applied to a PEST matrix.

(4) Though the PEST matrix is inspired by the Google Matrix used in PageRank, PEST deviates from and generalizes PageRank significantly:

- PEST uses the structure of the data not only for ranking but for *answer selection* as well: PEST also finds relevant answers that do not directly contain the considered keyword at all. Thus, it generalizes the limited form of term propagation in PageRank (from anchor texts to linked pages) and allows flexible propagation of terms between data items, based, e.g., on their type, the type of their relation etc.
- To achieve this term propagation, PEST uses an informed leap based on term weight distribution rather than a random leap. The employed technique is similar to that of personalized PageRank, where the leap probability is based, e.g., on a user's preferences, however, with different goal and outcome. Where personalized PageRank only computes a ranking, PEST computes a new vector-space representation of the data items that integrates propagated term weights with relevance based on the relations of a data item.
- Since the term weight distribution and thus PEST's informed leap is in general different for each term, PEST needs to consider term propagation for each term separately. Though this increases index time compared to PageRank, the term-dependent computation is only a small part of the overall indexing time and can be computed independently for each term .

(5) In an **extensive experimental evaluation** of PEST on an entire real-life wiki (Section 11.5), we compare PEST with three existing keyword search approaches: a simple TF-based ranking, the ranking used by Wikipedia, and the ranking returned by Google.
The experimental evaluation demonstrates

- that PEST significantly improves the ranking for each of the compared approaches over a range of keyword queries.

- that users generally prefer the ranking returned by PEST,
- that PEST achieves that improvement at the cost of an increase in index time that is notable, but also easily offset, as it is linear in the number of relevant terms with constants in the low seconds and can be well parallelized.

Though Section 11.5 clearly shows that PEST as discussed here can significantly improve existing keyword search in wikis, there are a number of open issues and further challenges to PEST summarized in Section 11.6.

11.1 RELATED WORK

PEST differs from the majority of approximate matching approaches including those reviewed in the following in several important aspects:

(1) PEST does not realize fuzzy matching by defining a structural distance function and ranking results according to how close they are to a strict match. Instead, it uses the structure of the data to determine which terms are relevant to a document, regardless of whether or not they explicitly occur in it. As a consequence, not only are new matches introduced, but strict matches may also be re-ranked depending on the structural connections.

(2) PEST is designed for *graph-shaped data* rather than purely hierarchical data like the XML-based approaches discussed in the following.

(3) PEST can be used with any information retrieval engine based on the vector space model. The only modification to the evaluation process is the computation of the actual term weights. Otherwise existing technology (such as Lucene or similar search engines) can be utilized. In particular, the PEST matrix is query independent and can be computed at *index time*. No additional computations such as query transformations are needed during query evaluation.

Before we consider specific approaches, it is worth recalling that *approximate matching* and *ranking* are closely related. Though they do not have to be used in conjunction, this is often the case, in particular to allow an approximate matching engine to differentiate looser results from results that adhere more strictly to the query. The full power of ranking and approximate matching is unleashed only when they are combined—approximate matching extends the set of results, ranking brings the results into an order for easier consumption by the user.

While approximate matching is widely used in web search and other IR applications, conventional query languages for (semi-)structured data such as XQuery, SQL or SPARQL do not usually employ approximate matching or rank results. These languages

have been applied to probabilistic data, but this area is distinct from approximate querying: In probabilistic data management, the data itself introduces uncertainty, in approximate matching uncertainty is introduced under the assumption that the user is also interested in matches that do not quite match her query.

As the amount of structured web data increases and the semantic web continues to emerge, the need for solutions that allow for layman querying of structured data arises. As described in this dissertation, research has been dedicated to combining web querying and web search and introducing IR methods to querying. KWQL is one such research effort, other approaches include extensions to conventional query languages, visual tools for exploratory search, extension of web keyword search to include (some) structure and keyword search over structured data. With the arrival of these techniques, the need for approximate querying that does not apply solely to individual terms or phrases but that takes the data structure into account arises.

Approximate matching on data structure has been researched mainly in the context of XML [345]. The majority of work in this area can be divided into two main classes of approaches:

Tree edit distance: Tree edit distance [343, 100, 58] is a popular and well-researched approach for assessing the similarity of tree-shaped data. It is based on the concept of the edit distance for strings [241]. Given a pair of XML trees, a set of edit operations (typically at least node deletion and insertion) and a cost function, the tree edit distance is the cost of the cheapest sequence of operations that transforms one tree into the other. For a given XML tree, its best-matching XML trees are those that have the lowest tree edit distance. Tree edit distance thus quantifies the similarity between the documents through the number of steps and types of operations needed to eliminate the differences between them. XML search can be straightforwardly formulated as finding XML documents that have a sufficiently low tree edit distance to a query represented as a labeled tree [325].

As finding the best-matching trees through the exhaustive computation of tree distances, itself costly, is computationally expensive, research has been focused on developing efficient distance algorithms [178, 374].

Amer-Yahia et al. [17, 19] present a conceptually related approach where queries in the form of tree patterns are gradually generalized by applying different types of pattern relaxation operations such as turning a parent-child node into an ancestor-descendant node. To limit the amount of necessary calculations, a procedure for the efficient generation of queries whose answers lie above a certain threshold is introduced.

Shasha et al. [331] present an approach where the distance between a query, represented as an unordered labeled tree, and

an XML document is quantified by counting the number of root-to-leaf paths in the query that do not occur in the document.

In contrast to PEST, the described approaches are hard to generalize to graph data. Both pattern relaxation and tree edit distance require expensive calculations at query time, either a loop to relax the query and the evaluation of a (often quite considerable) number of relaxed queries, or, a high number of distance calculations. PEST's computation on the other hand can be performed entirely at index time. Further, it is not obvious how different edge types, as easily treated by PEST, affect tree edit distance and pattern relaxation. Additionally, both procedures make the assumption that the query can be represented as a single tree. It is unclear how vagueness and advanced query language features like disjunction and negation can be efficiently integrated into the approaches. This is in contrast to PEST which is independent of the query formalism being used.

Adapting the vector space model: Another class of approaches aims, like PEST, to adapt the vector space model, a well-established IR technique, to the application on XML data. In the vector space model, documents and queries are represented as vectors consisting of a weight for each term. Their similarity is then computed using for example the cosine angle between the two vectors. Different schemes can be used for calculating the term weights, the most popular one being tf-idf.

Pokorný [306] represent individual XML documents in a matrix instead of a vector, indicating the term weight for each combination of term and path. A query, also expressed as an XML tree, is transformed into a matrix of the same form. The score of a query with respect to a possible result is then calculated as the correlation between the two matrices. In an extension, the matrix is adapted to also reflect the relationship between paths.

In Carmel et al.'s work [85], document vectors are modified such that their elements are not weights for terms but rather weights for term and context pairs—the contexts of a term are the paths within which it occurs. The vector then consists of a weight for each combination of term and context. Queries, represented as XML fragments, are transformed into query vectors of the same form. Further, the similarity measure is modified by computing context similarities between term occurrences in the query and data. These are then integrated in the vector similarity measure.

Schlieder and Meuss [326] calculate adapted term frequency and inverse document frequency scores replacing textual terms by so-called *structured terms*, that is, all possible XML subtrees in a dataset. Each structured term can then be described by a vector of tf-idf scores. In contrast, query vector weights are defined by the user. The vector space model is applied to compare subtree and query vectors.

Activation propagation is used in Anh and Moffat [24] for approximate matching over XML structure. Here, a modified version of the vector space model is used to calculate similarity scores between query terms and textual nodes in the data. The calculation of term weights takes into account the structural context of a term as well as its frequency. In a second step, these scores are propagated up in the tree. Finally, the nodes with the highest activation are selected, filtering out some results which are considered to be unsuitable such as the descendants of results that have already been selected. This approach resembles ours in that activation propagation and the vector space are used to realize approximate matching over structure. However, here, propagation happens upon query evaluation and is unidirectional. Like the other approaches in this class, it is also limited to tree-shaped data.

PEST **and PageRank:** Where the above approaches for approximate (keyword) search and querying on XML data are similar in aim, PageRank is closely related to PEST in technique, but differs considerably in aim and scope. The original PageRank article [71] suggests to exploit anchor-tags for web search. The anchor text of a link to a web page is treated as if it is part of the text of that web page. This suggestion can be seen as a special case of the approach suggested in this paper where the only kind of propagation is that from anchor tags to linked pages and where link weights are ignored. The application of this approach is limited to anchor tags and does not apply to general graphs or generalize to different link types. However, there are a number of extensions [52] of the original PageRank that share more of the characteristics of PEST.

PageRank is based on the intuition that a link from one webpage to another can be seen as an endorsement of the linked page. A page then is important if many important pages link to it, even more so if the number of pages linked to by these important pages is low. This idea is implemented by transforming the link graph into a transition matrix which is augmented by a random leap component that ensures that the probability to transition from any state to any other state is nonzero. The contribution of the random leap factor relative to the transition values is determined by the factor α. As a consequence of introducing the random leap and setting α to a non-zero value, the normalized matrix is stochastic and strictly positive and the principal eigenvector (called PageRank vector) for eigenvalue 1 exists and is unique. In standard PageRank, the random leap factor operates using a uniform distribution, that is, the likelihood to transition to a state is identical for all states. For a set of N pages, the corresponding leap or teleportation vector can be seen to consist of N entries with value $\frac{1}{N}$. Brin and Page [71] point out the possibility to real-

ize a personalized version of PageRank by using a non-uniform leap vector.

Several schemes for improving the scalability of personalized PageRank have been presented in recent years [203, 209, 188, 312]. They are discussed in this section as well as in Section 11.5.4.

Topic-sensitive PageRank [188] builds on the idea of a personalized teleportation vector by introducing a number of topic-specific leap vectors, each assigning a uniform value to all pages relevant to the respective topic and 0 otherwise. The topic-dependent importance scores for each page are calculated offline. At query time, a weighted classification of the query into topics is computed. A query-specific PageRank can then be calculated as a mixture of topic-specific scores. The motivation behind topic-sensitive PageRank is to avoid generally important pages getting highly ranked despite not containing information relevant to the query.

Query-dependent PageRank [312] is another extension of PageRank which is based on the idea that webpages matching a query that are connected to other matching webpages should be ranked higher. The PageRank algorithm is adapted in such a way that the probability of a transition from one webpage to another is determined by how relevant the target webpage is to the query. Towards this end, both the distribution underlying the leap vector as well as the mode of calculating transition probabilities are adjusted. In both cases, the probability to transition to a webpage is given as the proportion between that webpage's relevance score and the sum of all matching pages' relevance scores. When a webpage has no non-zero out-links, the leap vector is used to jump to another webpage. The transition matrix is not strictly positive and nodes which do not contain the relevant term are ignored. The PageRank vector is calculated for each term at index time, the scores for each term in the query are combined upon query evaluation.

PEST differs from the approaches described in various ways. In contrast to PageRank (but similar to topic-sensitive and query-dependent PageRank), several matrices and eigenvectors are calculated, namely one per term, each using a term-dependent leap vector. In contrast to topic-sensitive PageRank as well as PageRank, the leap vectors do not use a uniform distribution.

Most importantly, PEST differs from all three approaches described in the following ways:

- None of the approaches implement approximate matching over structured data or generally add additional relevant results; they are purely approaches to *ranking* sets of webpages.

- The assignment of edge weights in PEST is more flexible in that they can be set explicitly and individually or different weights can be chosen depending on the type of the edge. In contrast, in PageRank and topic-sensitive PageRank edge weights are uniform and in query-dependent PageRank, edge weights are derived from keyword matches.
- While PEST can be used on webpages with linking as the only relation between pages, its versatile and extensible data model allows for the application to many other types of graph-shaped data such as a fine-grained modeling of structured web data.
- In PEST, the probability of a leap is variable depending on the number and weight of outgoing edges of a node, thus encoding the intuition that a user jumps to a new page when he cannot find what he is looking for by following links. The leap distribution is the combination of a uniform distribution (as in PageRank and topic-sensitive PageRank) but takes term distributions into account (as in query-dependent PageRank).

PEST can be seen as a generalization of PageRank and topic-sensitive PageRank: The results of classic PageRank can be reproduced by choosing edge weights in such a way that they are all identical after normalization and using only the random leap component of the leap vector.

The behavior of topic-sensitive PageRank can be achieved by clustering all words belonging to a topic together, setting the edge weights as described above, and using only the informed leap component of the leap vector.

Query-dependent PageRank cannot directly be emulated by PEST as PEST relies on a strictly positive matrix and does not ignore nodes that do not (yet) contain the respective term. Further, different edge types corresponding to the different possible edge weights would have to be introduced.

ObjectRank [200] is a system for *authority-based ranking* for keyword search in databases that, like PEST, uses PageRank to exploit the connections between data items for propagating authority with respect to a keyword across a data graph. Given a database modeled as a labeled graph and a schema graph that assigns bidirectional authority transfer rates to the different types of edges, an *Authority Transfer Data Graph* is derived. The weight of a node with respect to a given keyword is then established by a modified version of PageRank where a random surfer walks across the graph either traversing edges or jumping to any of the nodes that literally contain the keyword. The probability for following any outgoing edge or jumping to one of the keyword nodes is steered by the damping factor d: The probability to follow an edge is

given as the product of d and the authority transfer rate of the edge, while the probability to randomly jump to one of the nodes containing the keyword is $(1 - d)$.

While ObjectRank shares some characteristics with PEST, it differs in several significant ways and has a number of drawbacks:

- ObjectRank uses a binary measure to represent literal keyword containment and randomly decides which of the nodes containing the keyword to jump to. Differences in the frequency of the keyword between documents are ignored and do not factor into the ranking. However, term frequencies are an important factor for ranking, particularly in text-heavy areas of application where it is of high relevance whether a term only occurs once or is frequently used.

- There are no jumps to nodes that do not contain the keyword and those nodes can only be reached through an edge traversal. Therefore, ObjectRank cannot be applied to graphs that are not strongly connected. While this may be of less concern in the area of databases, this constraint severely limits the possibility of applying ObjectRank to other types of graph-shaped data such as web or wiki pages which are frequently not strongly connected.

- Unlike PEST, ObjectRank is not a simple modification of term frequency distributions. As such, it cannot be directly used with standard information retrieval engines and easily used in conjunction with the vector space model.

- Queries are limited to simple keywords combined using disjunction and conjunction and it remains unclear whether ObjectRank could be combined with more powerful query languages.

- The probability to make a leap is steered only by the damping factor d and thus remains constant regardless of whether there is a promising edge that could be followed or not. Moreover, due to the way that authority transfer weights are assigned and normalized, the probability of all possible events may not add up to 1, which is unintuitive in terms of the random surfer model and the idea of spreading authority.

There is a significant body of research [103, 92, 314] on ranking entities in, e.g., entity-relationship graphs. These approaches share with PEST the aim to rank connected items by considering not only local, but also global properties, viz. what other items they are related to. However, they differ from PEST in two main aspects: (1) They require a new query engine with sophisticated,

multi-part ranking functions, where PEST computes a modified vector space model that can be used with any existing vector-space IR system. (2) They are tailored to ranking of entities such as dates, prices etc. where PEST is tailored to domains such as semantic wikis or concept search in ontologies where the items of interest are self-contained and clearly identified.

PEST can be applied to any type of structured data, including RDF ontologies. Ranking of RDF query results is discussed, e.g., in Elbassuoni et al. [143] and Elbassuoni et al. [144]. However, these works differ from PEST by focusing on general RDF data and by using statistical language models with limited propagation.

11.2 A FORMAL MODEL FOR STRUCTURED DATA

In this section we formally define a generic graph-based model of structured content that is capable of capturing the rich knowledge representation features of a wide range of knowledge management applications such as wikis and social networks. We distinguish data items into primary (e.g., wiki pages or people in a social network) and annotation items (e.g., tags or categories). A content graph is defined based on a **type structure** $\mathcal{T} = (\mathcal{D}, \mathcal{A}, \mathcal{E})$ where $\mathcal{D}$ is the set of types for primary data items, $\mathcal{A}$ the set of types for annotation data items, and $\mathcal{E}$ the set of edge types.

For KiWi, $\mathcal{D}$ contains a single type, "content item," $\mathcal{A}$ contains the annotation types "tag," "RDF class," "RDF literal," and "RDF other." The latter three annotation types are used to represent RDF class, literals and all other resources. It is straightforward to add further annotation types, but in our experiments the above annotation types suffice. $\mathcal{E}$ contains two types, link (l) and containment edges (c). This suffices to represent the primary representation mechanisms of KiWi. E.g., a link edge from a content item to a tag represents a tagging, a link edge between two documents a (hypertext) link, a link edge between a content item and an RDF class a type relation etc. We choose to omit the RDF edge labels in this representation as distinguishing between edges with different labels in the propagation did not show a marked improvement and introduces considerable complexity.

Definition 2 (Content graph)**.** *For a given type structure $\mathcal{T}$, a **content graph** is a tuple $G = (V, E, type, T, w_T)$ where V is the set of vertices (or data items) in G and $E \subseteq V \times V \times \mathcal{E}$ its set of typed edges. $type : V \to \mathcal{D} \cup \mathcal{A}$ assigns a type to each node. The textual content of data items is represented by a set T of terms and a function $w_t : V \times T \to \mathbb{R}$ that assigns a weight to each node-term pair. We assume that the term weights for each node v form a stochastic vector (i.e., $\sum_{\tau \in T} w_t(v, \tau) = 1$).*

For the sample wiki from Figure 61, the six documents 1 to 6 (with type content item from $\mathcal{D}$) and the tags 1.1, 1.2, 2.1, 3.1, 3.2, 4.1 (with type tag from $\mathcal{A}$) form V, $(1,2,l)$, $(6,1,l)$, $(6,3,l)$, $(4,3,l)$, $(1,1.1,l)$, $(1,1.2,l)$, $\ldots$, $(4,4.1,l)$ and $(1,2,c)$ form the set of all edges E, the set of all terms in the wiki form T, and $w_t = \{(1, \text{“}java\text{”}, 0.8), \ldots, (2.1, \text{“}search\text{”}, 1), \ldots\}$.

11.3 COMPUTING THE PEST MATRIX

Based on the above model for a knowledge management system, we now formally define the propagation of term-weights over structural relations represented in a content graph by means of an eigenvector computation.

A content item's tags are descriptive of the content of its text—they have a close association. Similarly, the tags of a nested content item to some extent describe the parent content item since the content item to which the tag applies is, after all, a constituent part of the parent content item. More generally, containment and linking in a wiki or another set of documents indicate relationships between resources. We suggest to exploit these relationships for approximate matching over data structure by using them to propagate resource content. A resource thereby is extended by the terms contained in other resources it is structurally related to. Then, standard information retrieval engines based on the vector space model can be applied to find and rank results oblivious to the underlying structure or term-weight propagation.

To propagate term weights along structural relations, we use a novel form of transition matrix, the PEST propagation matrix. In analogy to the *random surfer* of PageRank, the term-weight propagation can be explained in terms of a *semi-informed reader* who is navigating through the content graph looking for documents relevant to his information need expressed by a specific term τ (or a bag of such terms). He has been given some—incomplete—information about which nodes in the graph are relevant to τ. He starts from one of the nodes and reads on, following connections to find other documents that are also relevant for his information need (even if they do not literally contain τ). When he becomes bored or loses confidence in finding more matches by traversing the structure of the wiki (or knowledge management system, in general), he jumps to another node that seems promising and continues the process.

To encode this intuition in the PEST matrix, we first consider which connections are likely to lead to further matches by weighting the edges occurring in a content graph. Let $\mathbf{H}$ be the transposed, normalized adjacency matrix of the resulting graph. Second, we discuss how to encode, in the leap matrix $\mathbf{L}_\tau$, the jump

to a *promising* node for the given term τ (rather than to a random node as in PageRank).

The overall PEST matrix $\mathbf{P}_\tau$ is computed as (where α is the leap factor)

$$\mathbf{P}_\tau = (1-\alpha)\mathbf{H} + \mathbf{L}_\tau.$$

Each entry $m_{i,j} \in \mathbf{P}_\tau$, that is, the probability of transitioning from node j to node i, is thus determined primarily by two factors, the normalized edge weights of any edge from j to i and the term weight of τ in i.

11.3.1 *Weighted Propagation Graph*

To be able to control the choices the semi-informed reader makes when following edges in the content graph, we first extend the content graph with a number of additional edges and vertices and, second, assign weights to all edges in that graph.

Definition 3 (Weighted propagation graph)**.** *A **weighted propagation graph** is a content graph extended with a function $w_e : E \to \mathbb{R}^2$ for assigning weights to edges that fulfills the following conditions:*

- *Each primary data item carries at least one annotation from each annotation type: For each node $v_d \in V$ with $type(v_d) \in \mathcal{D}$ and each annotation type $t_a \in \mathcal{A}$, there is a node $v_a \in V$ with $type(v_a) \in \mathcal{A}$ and $(v_d, v_a, t) \in E$ for some edge type e.*
- *For each edge between two primary data items there is a corresponding edge between each two annotations of the two primary data items, if they have the same type: For each $v_d, w_d \in \mathcal{D}, v_a, w_a \in \mathcal{A}$ with $(v_d, w_d, t), (v_d, v_a, t_a), (w_d, w_a, t_a) \in E$ and $type(v_a) = type(w_a)$, there is an edge $(v_a, w_a, t) \in E$.*

Edge weights are given as pairs of numbers, one for traversing the edge in its direction, one for traversing it against its direction.

In the context of KiWi, the first condition requires that each document must be tagged by at least one tag. The second condition ensures that tags of related documents are not only related indirectly through the connection between the documents, but also stand in a direct structural relation.

Proposition 2. *For every content graph, a weighted propagation graph can be constructed by (1) adding an empty annotation node of type a to each primary data item that does not carry an annotation of type a and (2) copying any relation between two primary data items to its same-type annotation vertices.*

Consider again the sample wiki from Figure 61, the resulting weighted propagation graph is shown in Figure 62. It contains

two "dummy" tags (5.1 and 6.1) as well as a number of added edges between tags of related documents.

We call a weighted propagation graph *type-weighted*, if for any two edges $e_1 = (v_1, w_1, t), e_2 = (v_2, w_2, t) \in E$ it holds that, if $type(v_1) = type(v_2)$ and $type(w_1) = type(w_2)$, then $w_e(e_1) = w_e(e_2)$. In other words, the weights of edges with the same type and with start and end vertices of the same type respectively must be the same in a type-weighted propagation graph. In the following, we only consider such graphs.

Let $\mathbf{A}_w$ be the weighted adjacency matrix of a weighted propagation graph G. Then we normalize and transpose $\mathbf{A}_w$ to obtain the transition matrix $\mathbf{H}$ for G as follows (where $outdegree(v)$ denotes the number of outgoing edges of v):

$$\mathbf{H} = \left(\frac{1}{outdegree(v_i)} \left(\mathbf{A}_w^\top\right)_{i,j} \right)_{i,j}$$

By normalizing with the number of outgoing links rather than the total weight of the outgoing edges, edge weights are preserved to some extent. At the same time, nodes with many outgoing edges are still penalized. Normalization with the out-degree proved the most effective in our experiments compared to, e.g., normalizing with the maximum sum of outgoing term weights (over all nodes) or with the sum of outgoing term weights for each node. Different choices for normalization preserve different properties of the original matrix and, for other applications, a different choice of normalization may be advisable.

11.3.2 *Informed Leap*

Given a leap factor $\alpha \in (0, 1]$, a leap from node j occurs with a probability

$$P(\text{leap}|j) = \alpha + (1 - \alpha)(1 - \sum_i \mathbf{H}_{i,j})$$

A leap may be *random* or *informed*. In a random leap, the probability of jumping to some other node is uniformly distributed and calculated as $l^{rnd}(i, j) = \frac{1}{|V_d \cup V_t|}$ for each pair of vertices (i, j).

An informed leap by contrast takes the term weights, that is, the prior distribution of terms in the content graph, into account. It is therefore term-dependent and given as $l_\tau^{inf}(i, j) = \frac{w_t(i,\tau)}{\sum_k w_t(k,\tau)}$ for a $\tau \in \mathcal{T}$. Thus, when the probability of following any of the outgoing edges of a node decreases, in turn a leap becomes more likely.

In our experiments, a combination of random and informed leap, with heavy bias towards an informed leap, proved to give

the most desirable propagation behavior. The overall leap probability is therefore distributed between that of a random leap and that of an informed leap occurring according to the factor $\rho \in (0, 1]$, which indicates which fraction of leaps are random leaps.

Higher values of ρ mean that, after the propagation, the term occurrences are more evenly distributed between the nodes in the graph, while a low value of ρ, means that a higher proportion of leaps are informed and therefore leads to term weights after propagation being focused around the nodes which already contain the term before propagation.

Therefore, we obtain the leap matrix $\mathbf{L}_\tau$ for term τ as

$$\mathbf{L}_\tau = \left(P(\text{leap}|j) \cdot \left((1-\rho) \cdot l_\tau^{\text{inf}}(i,j) + \rho \cdot l^{\text{rnd}}(i,j) \right) \right)_{i,j}$$

11.3.3 *Properties of the* PEST *Matrix*

Definition 4 (PEST matrix)**.** *Let $\alpha \in (0, 1]$ be a leap factor, $\mathbf{H}$ be the normalized transition matrix of a given content graph (as defined in Section 11.3.1) and $\mathbf{L}_\tau$ the leap matrix (as defined in Section 11.3.2) for $\mathbf{H}$ and term τ with random leap factor $\rho \in (0, 1]$. Then the* PEST *matrix $\mathbf{P}_\tau$ is the matrix*

$$\mathbf{P}_\tau = (1-\alpha)\mathbf{H} + \mathbf{L}_\tau.$$

Theorem 1. *The* PEST *matrix $\mathbf{P}_\tau$ for any content graph and term τ is column-stochastic and strictly positive (all entries > 0).*

Proof. It is easy to see that $\mathbf{P}_\tau$ is strictly positive as both α and ρ are > 0 and thus there is a non-zero random leap probability from each node to each other node.

$\mathbf{P}_\tau$ is column stochastic, as for each column j

$$\begin{aligned}
\sum_i (\mathbf{P}_\tau)_{i,j} &= \sum_i \left((1-\alpha)\mathbf{H}_{i,j} + (\mathbf{L}_\tau)_{i,j} \right) \\
&= (1-\alpha) \sum_i \mathbf{H}_{i,j} + \Bigg(\left(\alpha + (1-\alpha)(1 - \sum_l \mathbf{H}_{l,j}) \right) \cdot \\
&\quad \Big((1-\rho) \cdot \underbrace{\sum_i l_\tau^{\text{inf}}(i,j)}_{=1} + \rho \underbrace{\sum_i l^{\text{rnd}}(i,j)}_{=1} \Big) \Bigg) \\
&= (1-\alpha) \sum_i \mathbf{H}_{i,j} + (1-\alpha)(1 - \sum_l \mathbf{H}_{l,j}) + \alpha \\
&= 1 - \alpha + \alpha = 1
\end{aligned}$$

□

Corollary 1. *The* PEST *matrix $\mathbf{P}_\tau$ has eigenvalue 1 with unique eigenvector $\mathbf{p}_\tau$ for each term τ.*

	1	2	1.1	1.2	2.1	4	3	4.1	3.1	3.2
1	0.1463	0.4091	0.4848	0.4848	0.1054	0.2556	0.1873	0.1873	0.2146	0.2146
2	0.1630	0.0109	0.0088	0.0088	0.3165	0.0130	0.0095	0.0095	0.0109	0.0109
1.1	0.2019	0.0109	0.0088	0.0088	0.1998	0.0130	0.0095	0.0095	0.0109	0.0109
1.2	0.2019	0.0109	0.0088	0.0088	0.1998	0.0130	0.0095	0.0095	0.0109	0.0109
2.1	0.0074	0.2054	0.1644	0.1644	0.0054	0.0130	0.0095	0.0095	0.0109	0.0109
4	0.1116	0.1637	0.1324	0.1324	0.0804	0.1949	0.1817	0.4540	0.1637	0.1637
3	0.0074	0.0109	0.0088	0.0088	0.0054	0.0908	0.0095	0.0095	0.3220	0.3220
4.1	0.0074	0.0109	0.0088	0.0088	0.0054	0.2074	0.0095	0.0095	0.0498	0.0498
3.1	0.0074	0.0109	0.0088	0.0088	0.0054	0.0130	0.2040	0.0873	0.0109	0.0109
3.2	0.0074	0.0109	0.0088	0.0088	0.0054	0.0130	0.2040	0.0873	0.0109	0.0109

Table 20: Excerpt of the PEST matrix for *"java"* with $\alpha = 0.3$ and $\rho = 0.25$

The resulting eigenvector $\mathbf{p}_\tau$ gives the new term-weights for τ in the vertices of the content graph after term-weight propagation. It can be computed, e.g., using the power method (which is guaranteed to converge due to Theorem 1).

The vector space representation of the content graph after term-weight propagation is the document-term matrix using the propagation vectors $\mathbf{p}_\tau$ for each term τ as columns.

11.4 TERM PROPAGATION WITH PEST: AN EXAMPLE

Here, we present the results for the sample wiki from Figure 61. We use a leap factor of $\alpha = 0.3$ and a random leap factor of $\rho = 0.25$. Using these factors, the PEST matrix is computed for each term $\tau \in \{$"*java*," "*lucene*," ...$\}$. The edge weights as shown in Figure 62 are derived by intuition of the authors.

The resulting matrix for the term "java" is shown in Table 20, omitting Documents 5 and 6 and their tags for space reasons.

Note that the matrix contains high probabilities for propagation to 1 and 4 throughout thanks to the informed leap. This preserves their higher term-weight for *"java"* compared to other nodes that do not contain *"java"*.

Using the PEST matrix, we compute for each term the resulting PEST vector $\mathbf{p}_\tau$. Together these vectors form a new document-term matrix representing the documents and tags in our wiki, but now with propagated term weights, as shown in Table 21.

To verify the veracity of our approach, let us consider a number of desirable properties an approach to approximate matching on a structured knowledge management systems such as KiWi should exhibit:

	RDF	*XML*	*architecture*	*introduction*	*java*	*lucene*	*search*
1	0.46	0.03	0.11	0.11	0.26	0.08	0.07
1.1	0.11	0.02	0.05	0.23	0.07	0.04	0.07
1.2	0.11	0.02	0.24	0.04	0.07	0.04	0.07
2	0.10	0.02	0.05	0.05	0.06	0.21	0.09
2.1	0.06	0.02	0.06	0.06	0.04	0.06	0.24
3	0.02	0.08	0.09	0.04	0.04	0.22	0.09
3.1	0.02	0.04	0.03	0.04	0.02	0.06	0.22
3.2	0.02	0.04	0.23	0.04	0.02	0.06	0.03
4	0.01	0.53	0.02	0.08	0.17	0.02	0.02
4.1	0.01	0.12	0.02	0.22	0.04	0.02	0.02
5	0.01	0.02	0.01	0.03	0.11	0.11	0.01
5.1	0.01	0.01	0.01	0.02	0.03	0.03	0.01
6	0.03	0.02	0.03	0.02	0.03	0.03	0.02
6.1	0.03	0.02	0.04	0.03	0.02	0.02	0.03

Table 21: Document-term matrix after term-weight computation

1. Documents containing a term directly (e.g., "java") with a significant term weight should still be ranked highly after propagation. This should hold to guarantee that direct search results (that would have been returned without approximate matching) are retained.

   Indeed Documents 1, 4, and 5, all containing "java" are highest ranked for that term, though the tags of Document 1 come fairly close. This is desired, as Document 1 contains *"java"* with high term weight and tag-document associations are among the closest relations.

2. A search for a term τ should also yield documents not containing τ but directly connected to ones containing it. Their rank should depend on the weight of τ in the connected document and the type (and thus propagation strength) of the connection.

   Again, just looking at the results for "java" the two tags of Document 1 as well as the contained Document 2 receive considerable weight for term "java."

3. The results of a KWQL query such as `ci(architecture introduction)` should also rank documents highly that do not include these terms, but that are tagged with "architecture" and "introduction."

Document 1 is such a case and is indeed the next highest ranked document for such a query after the three documents directly containing "architecture" or "introduction" (using either boolean or cosine similarity).

This evaluation can, by design, only illustrate the effectiveness of the proposed term-weight propagation approach for approximate matching. The following section therefore describes the results of a comprehensive evaluation of PEST on a large, real-life dataset.

11.5 VALIDATING PEST: THE SIMPSONS WIKI

In order to confirm that the described propagation approach performs as expected, experiments computing the resulting vector space representation after term-weight propagation were conducted using a prototype implementation of PEST. The source code of the implementation and the data used are available at **http://www.pms.ifi.lmu.de/pest**.

11.5.1 *Experiment: Setup and Parameters*

As an example we choose a real world wiki, the Simpsons Wiki at **http://simpsons.wikia.com** (also used for the KWQL and visKWQL user study described in Chapter 9), which is self-contained, focused on a specific topic and at the same time small enough to make it easy to judge which pages are relevant for a certain query and to perform a large number of experiments. At the time of this writing, it includes 10,955 pages (without redirects, talk and special pages).

Wiki pages are first stripped of all markup. The resulting text is normalized, stopword filtered[1] and stemmed[2]. The resulting wiki page collection contains 22,407 terms.

To keep the example simple and to allow for reliable human relevance judgments, we use only one type of edge, namely linking between pages. For computing the adjacency matrix $\mathbf{H}$, the weight of a link in outgoing direction is set to 0.2, its reverse weight to 0.1. As parameters for computing the PEST matrix, we use a leap factor of $\alpha = 0.15$ and a random leap factor of $\rho = 0.25$.

To judge the relative effectiveness of PEST, we compare with three ranking schemes:

- Luc: The first ranking is a basic tf-idf ranking with cosine similarity. It is similar to the default scoring used by Lucene.[3]

1 **http://www.ranks.nl/resources/stopwords.html**

2 Using the English snowball filter **http://snowball.tartarus.org/algorithms/english/stemmer.html**

3 **http://lucene.apache.org/java/3_0_1/api/all/org/apache/lucene/search/Similarity.html**

	PEST Score	Page title	Wik		Goo	
1	0.1348	Bart Simpson	4	+3	1	0
2	0.0340	Homer Simpson	980	+978	36	+34
3	0.0245	Lisa Simpson	281	+278	181	+178
4	0.0183	Bart the Genius	2	-2	4	0
5	0.0180	Marge Simpson	1321	+1316	548	+543
6	0.0148	Bart Gets an F	19	+13	3	-3
7	0.0115	Bart's Bike	1	-6	-	*new*
8	0.0112	Maggie Simpson	497	+489	678	+670
9	0.0105	Bart the General	11	+2	20	+11
10	0.0089	List of Bart Episodes in The Simpsons	3	-7	216	+206
11	0.0084	Bart Simpson (comic book series)	25	+14	18	+7
12	0.0081	Springfield	1669	+1657	-	*new*
13	0.0077	Chirpy Boy and Bart Junior	5	-8	-	*new*
14	0.0078	Bart vs. Australia	31	+17	12	-2
15	0.0074	The Bart Wants What It Wants	7	-8	30	+15
16	0.0073	Bart's Haircut	6	-10	84	+68
17	0.0070	Milhouse Van Houten	248	+231	67	+50
18	0.0069	Charlie	8	-10	-	*new*
19	0.0069	Bart Gets Famous	12	-7	6	-13
20	0.0069	Bart Junior	9	-11	-	*new*

Table 22: Top-20 ranking for query *"Bart"* for PEST, Wik, and Goo. The first column for Wik and Goo gives the respective rank, the second column the PEST change, where "–" denotes that the page in question is not returned as an answer at all.

Note that the *idf* value before propagation is used, since after propagation every term is contained in every wiki page (though mostly with a very low weight).

- Wik: The second ranking is based on the ranking algorithm used in MediaWiki (and thereby on Wikipedia)[4]: It extends the Luc ranking mechanism by compensating for the differing length of wiki pages by gradually decreasing the document score with increasing document length. Further, each page is boosted on the basis of its number of incoming links. Finally, if the title of the wiki page contains a query keyword, this page is boosted by a certain factor, in our case by 3.

4 http://www.mediawiki.org/wiki/Extension:Lucene-search

- Goo: Where possible, we also compare with the ranking returned by a Google query restricted to the Simpsons wiki.

11.5.2 *Comparison with other Ranking Methods*

In the following, we use two queries for comparing the ranking produced by PEST with those discussed above: First, we look at single word queries, viz. "Bart". Second the query "Moe beer" is used to illustrate the effect of PEST in the presence of queries consisting of multiple keywords.

Table 22 shows the top 20 answers for the single keyword query "Bart" using PEST with MediaWiki-style term weights (Wik ranking). In the last two columns, we compare that ranking with the ranking returned by MediaWiki without PEST and with the ranking returned by Google (Goo ranking, with search restricted to the domain `simpsons.wikia.com`).

Both the unmodified Wik and Goo ranking do not return any of Bart's family members (Homer, Lisa, Marge), Bart's hometown, or Bart's best friend as highly relevant for the query "Bart." This is clearly due to the fact that these pages contain the term "Bart" infrequently. In contrast, PEST returns all of these pages for highly related characters or locations among the top 20 matches for the query "Bart". The significant difference also to the Goo ranking shows that PageRank alone cannot account for the improvements demonstrated by PEST.

This effect is particularly noticeable for "Marge" and "Springfield," which occur at a rank below 1000 for Wik and either do not occur at all in the Goo ranking or at a rank below 500.

Applying PEST to a basic tf-idf ranking such as Luc yields even greater improvements as shown in Table 23, where we compare the ranking of the top 20 pages using PEST with Luc term weights to Luc ranking without PEST. The "Bart Simpson" page gets pushed from position 325 to pole position and Bart's direct relatives to the positions following shortly behind. This demonstrates that the PEST algorithm is capable of achieving significant improvements if applied to a range of existing ranking schemes.

For the multi-term query "Moe beer", the application of PEST also significantly improves the ranking as shown in Table 24. Here, we start with the Wik ranking. For example, *Homer Simpson*, a frequent visitor of *Moe's Tavern* (rank 4) and big consumer of duff beer (rank 3), is ranked second compared to rank 122 without PEST.

In addition to the top 20 ranking, we also considered the top 100 answers for each of the above rankings for "Bart". The results of this comparison are summarized in Table 25:

The number of relevant pages (manually evaluated) that are introduced by PEST into the top 100 answers is significant. At the

PEST rank	Page title	Luc rank	PEST change
1	Bart Simpson	325	+324
2	Homer Simpson	1667	+1665
3	List of Bart Episodes ...	1	-2
4	Lisa Simpson	1085	+1081
5	Bart's Bike	2	-3
6	Marge Simpson	1773	+1767
7	Bart Junior	3	-4
8	Bart the Mother/Quotes	4	-4
9	Ticket Bouncer	5	-4
10	Chirpy Boy and Bart Junior	6	-4
11	Spree for All	7	-4
12	Charlie	8	-4
13	Congressman 3	9	-4
14	Bart the Genius	84	+70
15	Bart Goes to the Movies	10	-5
16	Bike Track	12	-4
17	Maggie Simpson	1237	+1220
18	Bart Cops Out	11	-7
19	Bart Junior (frog)	17	-2
20	Bart Jumps/Credits	14	-6

Table 23: Top-20 ranking for query *"Bart"* for PEST and Luc

same time, most of the pages that are dropped from the top 100 are irrelevant and at worst a relevant page is dropped by about 50 ranks. Nearly no irrelevant pages are introduced.

All previous top 100 pages are still included in the first 140 results in the PEST ranking and the highest position not included in the new top 100 is the former position 65. Thus, only few relevant pages are discarded.

11.5.3 *User Study*

To determine whether users consider the application of PEST to improve search results, a user study involving a forced decision task was carried out. We chose to compare search results obtained using the Wik ranking with PEST to those of a simple Wik ranking without the application of PEST to specifically evaluate the effect of PEST when used together with a state-of-the-art ranking technique.

PEST rank	Page title	Wik rank & PEST change
1	Moe Szyslak	1	0
2	Homer Simpson	122	+120
3	Duff Beer	4	+1
4	Moe's Tavern	2	-2
5	Flaming Moe's	3	-2
6	Fudd Beer	5	-1
7	Duff Beer Advertiser	9	+2
8	Bart Simpson	365	+357
9	Homer vs. the Eighteenth Amendment	15	+6
10	It Was a Very Good Beer	14	+4
11	Marge Simpson	-	*new*
12	Lisa Simpson	-	*new*
13	Billy beer	18	+5
14	The Seven-Beer Snitch	36	+22
15	Barney Gumble	29	+14
16	Eeny Teeny Maya Moe	6	-10
17	Springfield	382	+365
18	Moe Baby Blues	7	-11
19	Homer the Moe	8	-11
20	Duff Beer Krusty Burger Buzz ...	25	+5

Table 24: Top-20 ranking for query *"Moe beer"*

EXPERIMENTAL SETUP Both types of result rankings, Wik with and without PEST, were generated for twenty single-keyword queries consisting of names of Simpsons characters (for example "Bart," pictured in table 23 and "Milhouse"), locations in the Simpsons universe (such as "tavern" and "brewery") and other concepts relevant to the TV-show (for example "skateboarding"). The titles of the top 20 matching wiki pages for each query were then placed in individual files which did not contain the name of the ranking used, but only randomized ranking identifiers. These files were then presented to eleven participants, all of which had at least basic knowledge of the Simpsons, together with the corresponding queries. The participants were asked to indicate for each query which list of search results they preferred. Though the result rankings only gave the titles of the matching wiki pages, participants were encouraged to look up more information about the individual results if they felt they could not make a clear decision based on the titles alone.

	"Bart"	
	Wik rank	**Luc rank**
Relevant	17	17
Irrelevant	4	2

Table 25: (Ir-)relevant pages added by PEST compared to Wik and Luc

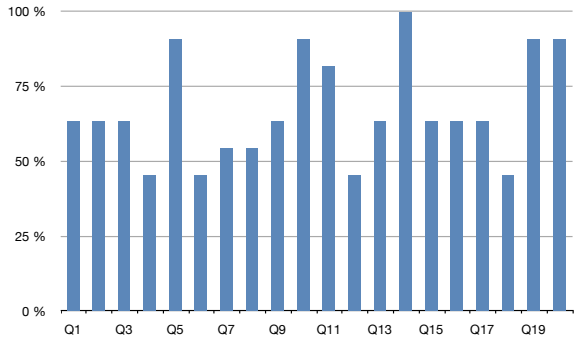


Figure 63: Percentage of participants who preferred a PEST-enhanced Wik ranking over a simple Wik ranking, per query

RESULTS AND DISCUSSION Across all queries and users, the PEST-enhanced ranking was preferred 67.23 percent of the time. A chart showing for each query what percentage of users preferred the PEST-enhanced results is given in Figure 63. Overall, participants liked the regular Wik ranking better for four of the queries, but only by a slight margin of about 5%. Two other queries prompted an equally divided reaction, their PEST-enhanced results were preferred by about 5%. The PEST-enhanced results of fourteen queries were more clearly preferred with scores between 63 and 100 percent.

While the results computed using PEST were not considered to be more relevant for all of the twenty queries by the majority, participants showed a palpable preference for the PEST-enhanced rankings for fourteen queries. Further, the regular Wik ranking results were not unequivocally judged to be better for any of the queries, while on the other hand six of the results using PEST were preferred by more than 75% of participants with the result for one query even reaching a perfect score of 100 percent.

Figure 64 shows the results broken down into groups of participants formed based on how often they indicated a preference for the PEST-enhanced query results. 82% of users overall prefer the results generated using PEST, more than half of them do so for 75% or more of the queries.

The finding that the percentage of users who overall strongly prefer the PEST-enhanced ranking is higher than that of the queries that received similarly high scores suggests that PEST

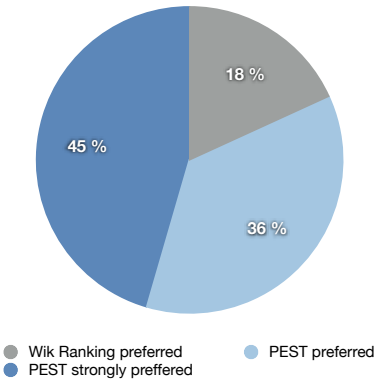


Figure 64: Percentage of participants who preferred the PEST-enhanced ranking for less than half of the queries ("Wik Ranking preferred"), 50-70% of the queries ("PEST preferred"), and 75% and more of the queries ("PEST strongly preferred")

is to some extent divisive in that its effects are perceived as very positive by a large amount of users, while a small group of users mostly prefers the unchanged Wik ranking.

Overall, the results of this study indicate that users considers PEST to substantially improve the quality of result rankings.

11.5.4 *Performance Evaluation*

The comparative evaluation of the quality of the rankings with and without PEST shows the significant improvements PEST can contribute to keyword search. But what about the cost?

To quantify the performance of PEST, we run a large number of keyword queries on a Intel Core 2 Duo E8400 with 8GB Ram running Sun Java 6 on a 32-bit installation of Ubuntu 9.10. The structure of the data as well as the terms are stored in a MySQL database. The algorithm is not parallelized and runs entirely on a single core. As a dataset, the Simpsons wiki with 10,955 pages and 22,407 terms is used, as discussed above. We do not use any partitioning or segmentation techniques, but hold all matrices in memory at once, using about 2 GB of main memory.

Once the modified vector space index computed by PEST is created, the query evaluation time depends only on the information retrieval engine used. Therefore, we focus here on the time PEST spends for indexing a given structured dataset.

Unsurprisingly, the indexing time for PEST scales like that of PageRank in the number of pages, i.e., linearly, as shown in Figure 66.

Figure 65 shows the percentages of each of the steps needed for indexing a single term with PEST: The term independent part, i.e., the initialization and normalization of the transposed, normalized adjacency matrix $\mathbf{H}$ for the underlying weighted

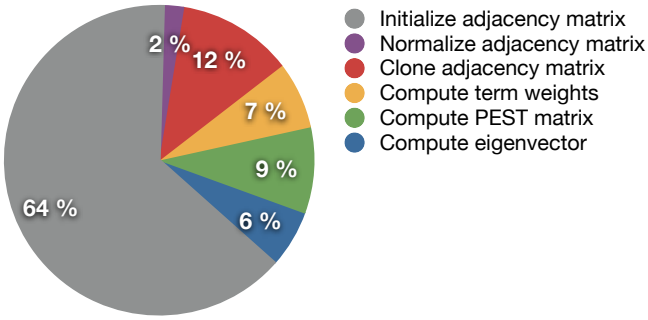


Figure 65: Indexing a single term

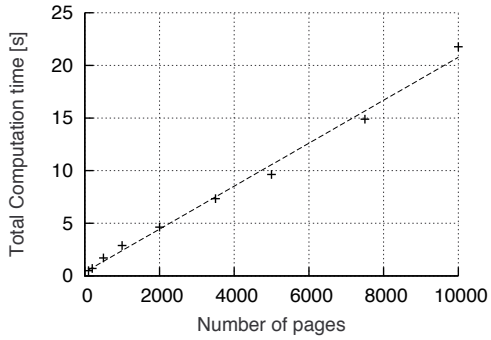


Figure 66: Indexing time over dataset size

propagation graph, takes on average 16 s, about two thirds of the processing time of a single term. When several terms are indexed at once, this part needs to be executed once only. For each term, we need to create a copy of **H**, compute the term weights (here using Wikipedia-style term weights), combine the copied matrix **H** with the resulting leap matrix and compute the eigenvector of the resulting PEST matrix. Overall, this part takes on average 8 s and thus about one third of the processing time of a single term.

It is worth emphasizing this result: Only about 8 s are needed to process each term. Furthermore, we can compute the PEST matrix for each term independently, even on different cores or

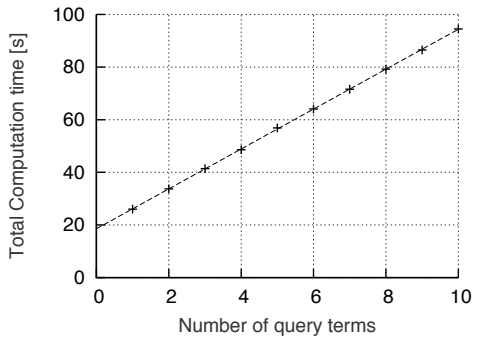


Figure 67: Indexing time over number of terms

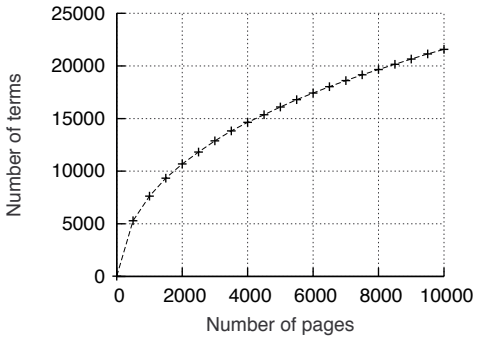


Figure 68: Number of unique terms over dataset size

computers. Figure 67 further emphasizes this result: If we increase the number of index terms, the total computation time on a single core increases, but only linearly with small constants. Figure 68 shows that the number of unique terms in a document collection such as the Simpsons wiki increases only fairly slowly with an increasing number of pages once a threshold of 5000 to 10000 terms is reached. Thus even for large document collections, we can expect a number of index terms in the range of tens of thousands for which the PEST matrix and index can be quickly computed by a small number of CPUs, even with the fairly unoptimized version of PEST discussed here.

Initially, this calculation has to be performed only once per term. When documents are edited, deleted or added, only the PEST vectors of the involved terms need to be re-calculated.

The scalability challenge that PEST faces is also less severe than that of personalized PageRank discussed in Section 11.1 as an individual PageRank has to be computed not per user but per term: The number of terms in a document collection is not directly proportional to the number of documents; instead, as the number of documents grows, most newly added documents only add few new words to the overall set of terms. This applies even more for a set of thematically homogeneous set of documents.

Many approaches to implementing personalized PageRank in a less computationally expensive way have been suggested [188]. They reduce the number of PageRank calculations necessary by limiting the granularity of personalization and thereby the number of PageRank computations needed.

Topic-sensitive PageRank, introduced in Section 11.1, offers query-dependent personalization on the basis of manipulating the prior probabilities of classes, that is, different topics, according to a user's interests. The personalization is restricted in that it operates at the level of topics, not individual web pages, calculating the score of a query as the sum of all pre-computed topic-dependent document scores multiplied by the likelihood of class membership.

Jeh and Widom [203] present an approach where personalized PageRank vectors are approximated as a linear combination of a number of partial vectors, so-called basis vectors representing certain highly-ranked web pages, pre-computed using a scalable dynamic programming approach. This method limits personalization through the choice of basis vectors—the surfer can only teleport to pages represented in the basis vectors.

BlockRank [209] combines individual web pages by their host or block, computing local PageRank vectors on a per host basis and weighting them by the overall importance of the host. Personalization here is realized at the granularity of blocks meaning that a user can only express his preference for a host, encoded in the weighting of local PageRank vectors, not for an individual web page.

While none of these approaches is directly applicable to PEST, it is likely that along similar lines approximate, but faster versions of PEST can be designed. One obvious way to achieve this is a limit on the number of terms index by PEST, e.g., to only the most prominent terms, by merging synonymous or semantically close terms, or by merging terms with similar frequency distributions.

11.6 DISCUSSION

PEST is a unique approach to approximate matching that combines the principles of structural relevance from approaches such as PageRank with the standard vector space model. Its particular strength is that it runs entirely at index time and results in a modified index representation.

In this article, we have analyzed PEST's performance on a wiki and shown that it improves search results not only by including new matches, but also by changing the result rankings of strict matches.

There is a wide body of further work to refine and extend PEST.

We are currently using rough estimates for α and ρ as well as for the edge weights rather than empirically validated observations. A guide to choosing these values might be possible to derive from studying the behavior of PEST on data with varying characteristics.

Edge values, in particular, could also be amenable to various machine learning approaches, using, for example, average semantic relatedness as a criterion, or to semi-automatic approaches through user-feedback.

We have also considered a number of different algorithmic approaches to term-weight propagation, e.g., where propagation is not based on convergence but on a fixed number of propagation steps. Techniques for spreading activation [110, 119] might be applicable and a comparison study is called for. Furthermore, the

computation of the PEST matrix is just one of several alternatives for finding a stochastic propagation matrix.

There are also a number of *specific areas for improving* PEST:

1. The model for structured data described in this paper assumes that edge weights are uniform for all terms. If edge weights are to be determined based, e.g., on the semantic similarity of a typed edge and a term (as determined through their Google distance or distance in an ontology), also the transposed, normalized adjacency matrix $\mathbf{H}$ becomes term dependent.

2. Links to external resources such as Linked Open Data or ontologies are currently not considered in PEST. Their inclusion would allow to enrich the content graph and thereby enhance the results of term propagation. This extension seems particularly promising in combination with the aforementioned typed links.

3. At the moment, PEST makes no distinction between terms based on their role in the document. One simple and obvious extension would be for example to represent the anchor text for each link and strongly propagating the respective terms to the linked document. This particular scheme mirrors a feature of classic PageRank, but a wide range of further possibilities for modifying edge or term weight based on the position of the text in the document exists.

4. Another, wiki-specific, extension is observing how the term scores of a document change over several revisions and taking this into account as a factor when ranking query answers.

5. Any approximate matching approach suffers from non-obvious explanations for returned answers: In the case of a boolean query semantics, the answer is obvious, but when term propagation is used, a document might be a highly-ranked query result without as much as containing any query terms directly. In this case, providing an explanation, for example that the document in question is closely connected to many documents containing query terms, makes the matching process more transparent to users. However, automatically computing good, minimal explanations is far from a solved issue.

6. In PEST, the term weights propagated along an edge are normalized by the number of edges incident to the same node. Thus, e.g., a document with many tags propagates only a relatively smaller amount to its children than a document with few tags. A model where the propagation along each type can not drop below a given minimum might prove superior to the basic version of PEST described here.

With the increasing size of the linked open data cloud, data providers require convenient means to sift through that data to discover relevant concepts and published instances for linking with their data. To find such concepts and instances, the structure of the involved RDF data is crucial, and thus existing

search engines are insufficient. At the same time, formal (e.g., SPARQL) queries over ontologies require extensive training and knowledge of the structure of the involved ontologies. Here, a semantic version of PEST would provide publishers with an easy and familiar means to discover relevant concepts and individuals in large-scale ontologies by taking the structure of the data into consideration to return the most relevant matches to keyword searches.

12 IMPLEMENTATION OF STRUCTURED TAGS

Structured tags were introduced in Section 5.2.2, evaluated in a user study described in Chapter 6, and the syntax for querying them was presented in Chapter 7.2. This chapter describes how structured tags could be represented in the KiWi wiki and how queries over structured tags could be evaluated using KWilt.

Since structured tags are very flexible in their form and use, we consider strictly matching structured tags without considering documents that have been assigned similar structured tags to be too rigid to yield good results (see Chapter 7.2).

For example, when a user searches for documents tagged "munich," documents tagged `location:munich` as well as those tagged `location:(munich, germany)` are also likely to be relevant to the query intent. Furthermore, a query for documents tagged `location:munich` should also match documents with the structured tag `location:(munich, germany)`. Similarly, since, the grouping operator is commutative, a query for `(munich, germany)` should also return documents tagged with `(germany, munich)`.

This functionality cannot be easily achieved when structured tags are represented as flat strings. Therefore, we suggest to implement structured tags as nestings of the content items representing the constituent atomic tags. Content items can be almost arbitrarily nested, only cycles must not occur. This approach not only means that querying structured tags can be treated as a special case of querying content item structure, but also that it can be realized without changing the KiWi wiki's conceptual model (see Chapter 5).

However, one problem remains: While the nesting of content items only expresses one type of relationship, child-parent, there are two possible relationships between tags, characterization and grouping.

This problem can be solved by augmenting nesting relationships with a type, for example through RDF triples where the URI of the parent is the subject, the URI of the child is the object and the predicate specifies the relationship between them. This predicate can then be used to distinguish between the two types of structuring.

Figure 69 shows a nesting-based representation of the structured tag `location:(city:munich, country:germany)`.

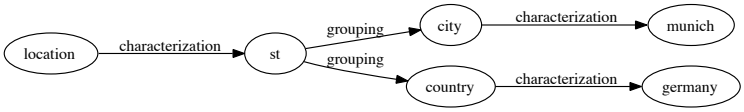


Figure 69: Nesting-based representation of the structured tag `location:(city:munich, country:germany)`

Note that "st" is an additionally introduced complex tag that corresponds to the grouping of the two tags `city:munich` and `country:germany`.

This solution has the additional advantage that it can be used to type all nesting relationships in the wiki, thereby making nesting more expressive and aligning the functionalities of linking and nesting.

Using this representation, the constituent atomic tags remain separate and can be efficiently queried. For instance, a query for tags named "munich" will return all documents that have been assigned the tag from Figure 69 since the atomic tag "munich" is part of it. Likewise, the query `tag(name:'city:munich')` will also return documents tagged with the tag from Figure 69 since "city" matches on one part of the tag and there is a relationship between the tags "city" and "munich" with the type "characterization."

In order to enable an efficient evaluation of structured tags with Solr, one possibility is to save a linearization of the structured tag in the Solr index: Each atomic tag which contributes to the structured tag is added to a designated field. Thereby, it can be guaranteed that all atomic tags occurring in a structured tags are present. Similar to the structure of content items, the structure of atomic tags then needs to be validated in the second step of the evaluation.

By using the proposed representation for structured tags and adding support for querying RDF to KWQL (see Chapter 13) the querying of structured tags can be realized in KWQL. In the interest of user-friendliness, queries over structured tags as written here and described in Chapter 7.2 could then be treated merely as syntactic sugar and translated into queries over content item nestings upon evaluation.

However, the query `tag(name:(location:munich))` would not yield the desired results. Though "location" would match the tag, there is no direct edge with type "naming" to the tag "munich." Assuming an RDF representation, this query cannot be expressed by means of a SPARQL query either, since SPARQL does not support a descendant axis which is needed to express queries of incompletely specified structured tags in RDF.

Instead, an RDF query language that can express navigational constraints on RDF graphs, like RPL [376] (see also Chapter 13) or nSPARQL [304] could be employed. Since the KWQL evaluation engine (see Chapter 10) is designed for an easy integration of different tools, adding support of one of these languages to the current implementation could be easily realized.

The subsumption approach described here and in Chapter 7.2 is a first step towards approximate matching of structured tags that, as shown by the example, also has its limitations. A a more user-friendly alternative to integrating an RDF path language would be to devise an approximate matching scheme for structured tags.

The operations for grouping and characterization and consequently the tag structure do not have a pre-defined semantics. The principle of structured tags is to introduce some structure into combinations of atomic tags which remain the focus of the approach and whose meaning is less freely interpretable. This is in contrast to logic-based knowledge representation formalisms where the structure carries the semantics while the signifiers do not and may be arbitrarily chosen. Consequently, a unification-based approach is unlikely to match all relevant tags, and the use of a similarity measure or similarity matching scheme for structured tags that takes the constituent atomic tags as well as tag structure into account appears more promising.

The problem of establishing the similarity of structured tags is related to that of approximate matching over structured data. While PEST (Chapter 11) or the other approaches to approximate matching over structure presented in Section 11.1 may not be directly applicable to this problem, these approaches could likely form the basis of a scheme for similarity-based matching for structured tags.

13 QUERYING RDF WITH KWQL

One issue that has not been addressed so far is that of querying RDF with KWQL. While RDF is an important component of the semantic web and sometimes is even seen as being synonymous to it, it is less easy to use than informal annotation and structuring mechanisms and may therefore dissuade beginning users from using the wiki. So, while the KiWi wiki supports RDF annotations and can import ontologies, these features are not immediately visible in the interface and it is part of KiWi's philosophy not to expose beginning users to RDF. As such, one could argue that being able to query RDF is not of crucial importance to KWQL, especially since SPARQL has emerged as the de facto standard RDF query language and can be assumed to be known by most KiWi users who want to query RDF data.

However, there are two reasons why the integration of RDF querying in KWQL is desirable and warrants attention: First, while dealing with complex RDF graphs may indeed overburden many users, simple RDF triples are intuitive and easy to understand. In particular, the KiWi interface provides a mechanism for adding basic RDF annotations in the form of triples where the URI of the content item being annotated is the subject of the triple and predicate and object are provided by the user. This form of RDF annotation makes it possible to use RDF without knowing much about the formalism and especially without knowing any dedicated RDF query languages. KWQL should therefore allow users to query at least these simple RDF annotations that they and others have created.

Secondly, even users who have experience with RDF and who know SPARQL can profit from RDF querying capabilities in KWQL since RDF query languages cannot be used for combined queries over content, metadata, structure, and annotations and as such do not offer the same functionality as KWQL.

The following sections present and discuss three ways in which KWQL could be extended to querying RDF data, namely the integration of SPARQL, the introduction of a **`rdf`** resource and the integration of RPL (*RDF Path Language*).

13.1 SPARQL QUERIES IN KWQL

One obvious way to enhance KWQL with support for querying RDF data is to allow the embedding of SPARQL queries into KWQL queries. This has the advantage that SPARQL is the most

prevalent RDF query language, meaning that it is well-supported, that a mature implementation exists and that many KWQL users who are familiar with RDF likely already know SPARQL.

Two aspects have to be treated when SPARQL queries are embedded in KWQL queries, namely what is returned by the SPARQL query and how the results of the SPARQL and the KWQL query are related. KWQL queries can return either variable bindings or content items, while SPARQL queries return variable bindings, RDF graphs or boolean values. The most basic mode of integration here is to treat all SPARQL queries like SPARQL **ASK** queries, meaning that they return "true" when there is at least one answer to the query and "false" otherwise. SPARQL queries then act as global boolean selection conditions that constrain the evaluation of the KWQL query they are contained in. For example, the following query can be interpreted as "Select all content items that represent a user, but only if there is at least one person who knows another person."

```
ci(tag(name:user) SPARQL:(SELECT ?name WHERE { ?person
    foaf:knows ?name .})
```

As this example shows, while there certainly are cases where queries of this type are needed, the expressive power and use of such a weak integration is limited: SPARQL results can only be used to determine whether or not the KWQL query should be evaluated and there is no connection between the content items returned and the data selected by the SPARQL query.

When we allow variable bindings to be shared between the KWQL and SPARQL queries, we can achieve a stronger integration between the languages that alleviates those problems. Assuming that a person's name functions as a unique identifier, the following query selects the content items representing the users that Mary knows. The variable `$name` (or `?name`) here is shared between the KWQL and SPARQL queries by virtue of the SPARQL query returning the variable and the KWQL part of the query using a variable with the same name[1].

```
ci(title:$name tag(name:user) SPARQL:(SELECT ?name WHERE
    { ?x foaf:name "Mary" . ?x foaf:knows/foaf:name
    ?name .})
```

Analogously, the query below selects the user pages of every KiWi user that Mary does not know.

1 For reasons of readability, this and the following query use SPARQL Property Paths [330] which are not yet an official part of SPARQL. Equivalent queries can be expressed using the current official version of SPARQL.

```
ci(title:$name tag(name:user) NOT SPARQL:(SELECT ?name
    WHERE { ?x foaf:name "Mary" . ?x
    foaf:knows/foaf:name ?name .})
```

Using variable sharing, the simple triple annotations described above can be queried. For example, the following query selects all content items of type "image."

```
ci(URI:$uri) SPARQL:(SELECT ?uri WHERE { ?uri <kiwi:type>
    <kiwi:image> .})
```

Both types of embedded SPARQL queries, boolean queries and queries that share variables with the KWQL query can be easily evaluated: For a boolean SPARQL query, that is, one where no returned variables also occur in the outer KWQL query, it is sufficient to evaluate the SPARQL query and, if it has at least one answer, to consequently return the answers to the KWQL query. When the queries share variables, both queries are evaluated using their respective engines and the results of the KWQL query are then filtered by removing the content items whose variable bindings are not contained in the intersection, or, when the SPARQL query is negated, the complement of the two sets of variable bindings. This evaluation strategy can also be used when there are several shared variables, when a shared variable is used several times in the KWQL part of the query or when variable bindings are used in the head of a KWQL rule.

Note it is assumed here that there is only one embedded SPARQL query per KWQL query, or rather, per disjunct in the disjunctive normal form of the query (see Section 10.4.3) and that embedded SPARQL queries are always connected to the KWQL query via conjunction; it is not clear how embeddings that do not satisfy these constraints should be interpreted.

While embedded SPARQL queries as discussed here add to KWQL's functionality and are easy to interpret and evaluate, they likely would only be usable by advanced users of KiWi and KWQL, especially when queries become more complex than the comparatively simple examples given here. Further, the integration of the two languages is based purely on sharing variables and thus is relatively weak, and SPARQL's design and syntax are very different from that of KWQL. The following section therefore presents a more KWQL-like way of querying RDF data.

13.2 ADDING A RESOURCE FOR RDF TO KWQL

To obtain a mode of querying RDF that is consistent with the existing syntax of KWQL, an **`rdf`** resource with the qualifiers **`predicate`** and **`object`** could be introduced. In terms of the admissible resource nestings (see Section 7.2.1), this resource is

equivalent to the **tag** resource, meaning that it can serve as a sub-resource of content items, fragments and links but that it can neither contain a **tag** resource nor be contained in one.

Using this resource and its qualifiers, simple RDF annotations where the resource is the subject and the predicate and object, or, in the case of typed links, only the predicate, are given by the users, can be queried. For example, the final query given in the previous section can be expressed as the following query:

```
ci(rdf(predicate:'kiwi:type' object:'kiwi:image'))
```

When the **rdf** sub-resource is used within a resource of type **link**, the link's **target** qualifier and the object of the **rdf** resource refer to the same content item, the link target, which may be confusing for users.

The query below selects all content items describing projects that contain a link of type "influenced" to the content item describing KiWi, that is, all content items representing projects that have influenced the KiWi project.

```
ci(tag(name:project)
    link(rdf(predicate:<kiwi:influenced>)
    target:ci(title:KiWi tag(name:project))))
```

Resource terms of type **rdf** can be used like any other resource terms in KWQL, meaning for example that they can be underspecified, can contain variables, and can be used with disjunction, conjunction, negation and the **OPTIONAL** operator.

KWilt (see Chapter 10) can easily be extended to cover the addition of the querying of RDF data and the **rdf** resource by indexing the RDF triples in Lucene and retrieving content items together with their RDF annotations in the second and third phases of the evaluation. Since the RDF triples are indexed in Lucene, RDF data is not only queried when the **rdf** resource is used explicitly, but is also used for keyword-only queries. Then, for example, the query `image` returns not only content items where the term occurs, but also content items that contain images. This means that the quality of search results is improved even for participants who are not aware of the RDF annotations.

Further advantages of the integration of RDF querying through the addition of the **rdf** resource are that it is easy to use, fits in well with the rest of KWQL in syntax and in spirit, and can make use of KWQL's features. Further, since the subject in the triples is always a KWQL resource, the RDF annotations queried are tightly coupled with the retrieved content items.

On the down side, this method is not suitable for querying arbitrary triples and so only a subset of the RDF data in the KiWi

wiki can be queried using this method, meaning that the addition is not sufficient for users who want to pose complex RDF queries.

Note also that global queries over the RDF graph as discussed in the last section cannot easily be realized: While the syntax presented here could be extended to cover the specification of complete triples, KWQL's support for underspecified queries makes it impossible to distinguish between an **`rdf`** sub-resource that specifies a content item annotation triple and one that is global but omits the subject. Further, global RDF queries cannot be evaluated with KWilt, meaning that a dedicated engine for evaluating global RDF queries that handles underspecification in the same manner as KWilt would be needed.

13.3 RPL QUERIES IN KWQL

RPL (RDF Path Language) [376] is a path language for RDF graphs that can traverse paths of unknown length and that aims at being expressive but user-friendly. As such, RPL is another promising candidate for realizing RDF querying in KWQL through embedded queries.

RPL queries consist of descriptions of paths and as answers return all pairs of start and end nodes between which a path exists that meets the given criteria. RPL queries can impose constraints on the nodes and edges in the path, can traverse paths in either direction or be direction-independent. They can also make use of wildcards, negation, Kleene star and plus and optionality operators, and regular expressions over labels. There are no variables in RPL, but RPL queries can be embedded in SPARQL queries and share their variables. The language additionally comes with a visual version, *visRPL*, that was inspired by visKWQL.

RPL's focus on paths allows for an intuitive integration into KWQL and its syntax: In a given query, KWQL is used to describe a content item, which is seen as the start node of an embedded RPL query. The embedded query then describes the path from the content item to a node in the RDF graph. When the end node is the URI of another KiWi content item, further selection criteria for this content item can be specified using KWQL. The RPL sub-query then returns the pairs of start and end nodes that do not only satisfy the RPL selection criteria, but where the content item designated by the URI of the end node is also in compliance with the given KWQL expression.

Any other RDF nodes representing content items that are on the path between the start and end nodes cannot easily be treated in this manner since RPL queries only return start and end nodes as answers.

The answers returned by KWQL queries with embedded RPL queries are content items whose URIs correspond to the start

nodes returned by the embedded RPL query and that satisfy any additional criteria specified using KWQL.

The described mechanism allows for querying simple annotation triples but also enables complex path queries on RDF graphs. The following query retrieves all content items that are of type "image." For an introduction to RPL's syntax, see Zauner et al. [376].

```
ci(RPL:(PATH _ >kiwi:type kiwi:image))
```

Note that here, the only selection criterion is expressed in RPL, while the KWQL+SPARQL representation of the same query (see Section 13.1) must use variable sharing to express the same query.

The query given below returns the user pages of all KiWi users that Mary knows.

```
ci(tag(name:user) RPL:(PATH _  <foaf:knows [PATH _
    >foaf:name "Mary"]))
```

Assuming that the RDF graph and the data in the KiWi wiki contain the same data, the query is equivalent to the following query which uses RPL to determine who knows whom, and KWQL to impose further selection constraints on the resulting start and end nodes. The KWQL selection criteria that the content item corresponding to the end nodes must satisfy is separated from the RPL query by a comma.

```
ci(tag(name:user) RPL:(PATH _  <foaf:knows _,
    ci(tag(name:user), title:Mary)))
```

KWQL and RPL can also impose constraints on an entity represented both as RDF data and as a content item; the query below returns user pages that mention "Lucene" and that represent programmers who Mary knows either directly or via one other person.

```
ci(tag(name:user) Lucene RPL:(PATH [PATH _ >rdf:type
    :programmer][PATH _ (<foaf:knows _)? <foaf:knows
    [PATH _ >foaf:name "Mary"]]))
```

Combined KWQL and RPL queries as described here could be evaluated by retrieving the answers for the RPL query, the KWQL query and, if given, the KWQL query applying to the end nodes using the respective evaluation engines, and using those results to find the content items that satisfy all given constraints, that is, that match the KWQL selection criteria as well as the RPL query.

If required, a RPL integration could also enable global queries as described in Section 13.1, but a separate syntax would be

required to be able to distinguish these global queries from RPL queries over content item relations.

13.4 DISCUSSION

Three solutions for adding support for RDF queries to KWQL have been discussed in this chapter, one native and two based on the integration of existing RDF query languages.

All three proposed methods are generally viable and each has its strong and weak points: SPARQL is powerful, established and widely known, but its integration into KWQL might not be very intuitive or extensive. The addition of the **`rdf`** sub-resource in KWQL is in line with the rest of KWQL's syntax and behavior, but only supports a small subset of queries. RPL supports very expressive path queries, has a straightforward interpretation in the context of KWQL, offers a visual editor that helps users edit and understand queries but would have to be learned by all KWQL users wanting to query RDF and by itself does not support variables.

Depending on the context and the RDF queries that users want to express, each language is a suitable candidate for extending KWQL to support querying RDF data. However, RPL offers a good compromise between usability and expressive power and thus, for most applications, appears to be the most promising candidate.

14 CONCLUSION

In this final chapter we briefly review the work described in the dissertation and point out several issues that are not yet addressed by KWQL but deserve further attention.

14.1 SUMMARY

The work presented in this dissertation addresses the question how ease of use and rich functionality, two seemingly conflicting characteristics, can be consolidated in the context of the social semantic web, and more specifically in the semantic wiki KiWi. We feel that this issue is crucial to the success of the social semantic web: social semantic web applications live from user participation and the adoption by a broad user base, but often fail to provide annotation and querying formalisms that allow casual and expert users alike to formalize knowledge and compose expressive queries to fully leverage the functionality of the application at hand.

With this goal in mind, and to lay the foundation for the rest of our work, we defined a conceptual model for the KiWi wiki. This model is based on a small number of "building blocks" that can be combined to obtain a rich and expressive form of knowledge representation. Together with this conceptual model, we introduced structured tags, an annotation formalism that is simpler and more flexible than RDF, but at the same time more expressive than atomic tags. We described how structured tags can be represented and queried in a simple manner that requires only minor changes to KiWi's conceptual model and the query syntax and evaluation process. We further performed a user study to compare structured tags and RDF. Structured tags were found suitable for the quick annotation of evolving knowledge and were received well by participants. RDF, on the other hand, was considered more complicated and better suited for more static settings in which annotations are provided by experienced users.

The main contribution of this dissertation is KWQL, a rule-based query language for the KiWi wiki based on the label-keyword query paradigm. KWQL allows for rich combined queries of textual content, metadata, document structure, and annotations. It is not restricted to data selection, but also offers construction, the reshaping of the selected data into new data, database-like views. Such views constitute a simple yet remarkably powerful form of reasoning. KWQL queries range from

simple lists of keywords or label-keyword pairs to conjunctions, disjunctions, or negations of queries. They thus cover the whole spectrum from elementary and relatively unspecific queries to complex and fully specified (meta-)data selections. The language has a low entry barrier and allows casual users to easily locate and retrieve relevant data, while more advanced users can exploit the full expressive power. The textual language KWQL is complemented by visKWQL, a visual interface that supports users in the query construction process. The visKWQL editor provides guidance throughout the query construction process through hints, warnings, and error highlighting and prevention. It also enables round-tripping between KWQL and visKWQL, meaning that users can switch freely between the textual and visual form when constructing or editing a query.

We described the underlying principles and the syntax of KWQL, provided a formal semantics for the language, and discussed KWilt, an implementation of KWQL query evaluation based on a patchwork approach. We then distinguished three sublanguages of increasing complexity and showed that it is possible to efficiently recognize the sublanguage a given KWQL query belongs to and to adapt the evaluation process accordingly. The power of full first-order queries can be leveraged where needed, but at the same time KWilt can evaluate basic queries at almost the speed of the underlying search engine, as we showed in a performance evaluation. Participants in a user study reacted positively to KWQL and visKWQL. They found the languages useful, expressive, and easy to use, at least given some time and practice. Even after a short introduction and a minimal amount of time to solve the assignments, participants overall were able to provide correct answers to more than half of the questions in a query writing task and over eighty percent of the questions in a query understanding task.

We finally presented PEST, a PageRank-like approach to the ranking and approximate querying of graph-structured data that propagates term weights between data items. Extensive experiments including a user study on a real-world wiki showed that PEST improves the quality of the ranking compared to a number of existing approaches.

14.2 PERSPECTIVES FOR FURTHER RESEARCH

In Chapters 6 and 9, we discussed aspects that were not considered during the experimental evaluation of structured tags and KWQL and visKWQL, respectively, and described new questions that arise from the results of the experiments. In Section 10.8, we explained how the current implementation of KWQL query evaluation could be improved. In Part iv we presented three

extensions to KWQL that are not yet implemented in the KiWi wiki. In Chapter 11, concerned with PEST, and Chapter 12 on the implementation of structured tags, we specifically discussed open issues and ideas for improvements.

The remainder of this section is devoted to ideas for further research directions regarding KWQL.

14.2.1 *Querying Versions of Content Items*

An important characteristic of wikis is their support for versioning. Versioning allows to trace changes to a wiki page and to restore earlier revisions, both of which are particularly important when a group of users collaborates on content that is gradually evolving.

So far, versioning is taken into account in KWQL only to the extent that the **`author`** qualifier is matched on the names of all contributors to a content item, not only on that of the person responsible for the last edit. KWQL does not currently support queries over specific versions of a content item, but always queries the latest revision.

Since many edits may consist only of minor changes and the data are consequently highly redundant, versions are typically not stored in full. Instead, the difference between two consecutive versions is recorded in so-called *deltas* which can be used to restore all prior versions of a content item. *Delta encoding* is an efficient and established method for storing versioned data. Enabling querying over different versions in KWQL, however, is less straightforward: A simple way to add support for versioning to KWQL would be to materialize all versions of a content item, index them separately together with their version number in Solr, and to introduce a qualifier **`version`**. As a value, this qualifier could take an integer or a range of integers indicating the revisions of a content item to be retrieved. Ranges as qualifier values are not supported in the current version of KWQL and KWilt, but could easily be realized through a small addition to the syntax and Solr range queries. Support for ranges as qualifier values would not only be useful for versions, but for example also for creation dates.

However, the addition of such a **`version`** qualifier also gives rise to at least two problems. First, it is not immediately clear which version should be queried when no **`version`** qualifier is given. Normally, when no constraint is given for a certain qualifier, this qualifier is not used as a criterion for matching. In the case of versioning, a query would therefore be evaluated over all versions of all content items. This would mean, however, that queries might return a large number of content items that do not reflect the current state of the wiki content, which is unintuitive and

confusing especially for beginning users. Alternatively, only the current version of each content item could be considered unless versions are given explicitly, but this would put the interpretation of the **`version`** qualifier at odds with that of all other qualifiers. A second problem is that the indexing of all full versions of a content item would greatly increase storage requirements and the computational cost of query evaluation. However, it is an open question how KWQL query evaluation could be adapted to function on delta-encoded versioned content and a more efficient data storage of and query evaluation over highly redundant data is likely not possible using the current query evaluation approach as described in Chapter 10.

14.2.2 *Social Factors in KWQL*

KWQL can query wiki data, and embedded queries can be created collaboratively just like any other wiki content, but apart from this, KWQL does not reflect the social aspect of wikis. Another research question is therefore how users could benefit from a version of KWQL that incorporates social factors. Two possible approaches involve leveraging the social nature of wikis to improve the ranking of results, and enabling the sharing of queries among users.

Towards the first goal, we have developed a variant of PEST, called PESTP, that personalizes the ranking of search results by taking users' social relationships and actions in the wiki into account [76]. When applied to a dataset representing movies and their user-generated annotations, PESTP outperformed PEST as well as the Wik and Luc ranking schemes discussed in Section 11.5.1 in terms of both precision and recall. However, the modified term frequencies in PESTP are both term- and user-dependent, and research into a more efficient calculation of the PESTP matrix is required to make the approach viable at a larger scale.

The KiWi Query Editor, discussed in Chapter 8, allows users to save and restore queries, but only on a per-user basis. The experimental evaluation of KWQL and visKWQL described in Chapter 9, on the other hand, has shown that understanding queries is easier than writing them. This means that access to a *query corpus* consisting of queries that other users have written could help users in their query formulation. For example, users could search for a query that expresses a similar query intent and adapt it to their needs. Open questions in this context concern ways to locate relevant queries in the query corpus and, since queries may contain personal information, the preservation of privacy.

14.2.3 *More Expressiveness for KWQL Queries*

KWQL currently does not allow conjunctions at the level of content item resource terms, meaning that queries involving two or more structurally unrelated sets of content items are not possible.

Depending on whether a KWQL rule specifies a head, and on whether the head uses variables bound in the body, the set of all possible KWQL query bodies can be partitioned into three classes: *content item queries*, *variable queries* and *boolean queries*. Content item queries are those queries that return a set of content items, that is, those in which no rule head is specified. Variable queries bind variables and use the bindings for the construction of new content items. Boolean queries are queries where a rule head is given but does not use any variables that may have been bound in the rule body. The query can thus be seen to return either "true," in which case the content item specified in the rule head will be created, or "false," meaning that the rule head is not evaluated.

The restriction of KWQL to disjunction at the level of content item terms is justified in the case of content item queries: due to injectivity, a conjunction over content item resource terms has no straightforward interpretation. Such queries, for example `ci(author:Mary)AND ci(author:John)`, could be taken to mean that the content items that contain or link to content items satisfying all content item resource terms should be returned. Adopting such an interpretation has not be investigated in detail but would likely lead to massively ambiguous queries, and we therefore maintained the exclusion of conjunction at content item level.

For variable queries, however, this restriction is likely not necessary. The following rule, which is not valid according to the current syntax and semantics of KWQL, could for example be used to create a list of all pairs of authors who mention each other:

```
ci(text:ALL($a1 - ALL($a2,", "),\n))@ci(text:$a2
    author:$a1) AND ci(text:$a1 author:$a2))
```

The expressiveness of KWQL queries could be increased even further by allowing individual KWQL resource terms, or possibly even value terms, to be augmented with boolean queries that are explicitly marked as such. Upon query evaluation, only those terms for which the boolean query evaluates to "true" could be considered. In that sense, the boolean queries attached to other query terms would specify conditions on the evaluation of the latter. The following query, which is again not valid according to the current syntax and semantics of KWQL, could then be used

to retrieve the text of all content items that mention the name of a KiWi project member:

```
ci(text:$n) boolean(ci(title:$n tag(name:"KiWi member"))
```

The changes to KWQL sketched above could likely serve to increase the expressive power of KWQL by simple means that require only minor changes to the query evaluation process, but the details of such an extension remain to be worked out.

14.2.4 *KWQL and the Social (Semantic) Web*

KWQL was developed for use in the KiWi wiki, but the problem it addresses also exists in many other social and social semantic web applications in which no simple but expressive query language is provided that allows users to leverage the data. Another aspect that warrants attention therefore concerns the application of KWQL or a KWQL-like language to other social and especially social semantic media like social networks and blogs. A first goal in this context would be to determine to what extent KWQL needs to be adapted to be useful in such applications, or whether it is possible to derive a generalization of KWQL that is suitable for a wider range of social semantic web applications.

SUPPLEMENTARY MATERIAL

A

STRUCTURED TAGS AND RDF

A.1 INTRODUCTORY TEXT ON STRUCTURED TAGS

Introduction to Structured Tags

Structured tags are almost like normal simple tags you know from the internet, only enhanced with structure. Two basic operations lie at the core: grouping and characterization. Grouping, denoted "()", allows to relate several (complex or simple) tags using the grouping operator. The group can then be used for annotation. Example: a Wiki page describes a meeting that took place in Warwick, UK on May 26, 2008, began at 8 am and involved a New York customer. Using simple tags, this page can be tagged as "Warwick", "New York", "UK","May 26", "2008", "8am" leaving an observer in doubts whether "Warwick" refers to the city in UK or to a town near New York. Grouping can be used in this case to make the tagging more precise: "(Warwick, UK), New York, (May 26, 2008, 8am)".

Characterization enables the classification or, in a sense, naming of a tag. For example, if we wanted to tag the meeting Wiki page with the time and date we could tag it as "(5, 26, 2008, 8)" using the grouping operator. The characterization operator can be used to make the tagging more precise: "(month:5, day:26, year:2008, hour:8)" and later perhaps specify that the whole group refers to a time: "time:(month:5, day:26, year:2008, hour:8)". The user is free to use the operators in whichever way as long as the resulting structured tag is formed correctly.

Some rules apply to the use of operators (they express what is a correct structured tag and how to recognize equivalent structured tags):

- Groups
  - are unordered: (apple, pear) is the same as (pear, apple)
  - cannot contain two equal members, e.g. (Bob, Bob, Anna) and ((Bob, Anna), (Anna,Bob)) are not allowed,
  - can contain arbitrarily many elements and can be arbitrarily nested, e.g. ((cat, dog, cow), ((mushroom),(daisy,dandelion,Sunflower))),
  - are identical in meaning to the simple tag when they only contain one element, i.e. (Anna) is the same as Anna
- Characterization
  - is not commutative, i.e. geo:x is not the same as x:geo.
  - can be used on both simple and stuctured tags: (animal:plant):((fox, squirrel,horse,panda):(pine,birch,chamomile))

Structured tags have to be syntactically correct. That means that for example "Bob:190cm, 90kg" is not a valid structured tag because "190cm,90kg" is not enclosed in parenthesis.

The same information can of course be encoded in many different ways using structured tag, the user is free to choose the way that suits her the best.

The structure of structured tags has two purposes:

- it enables users to group related things and to classify them
- it facilitates automated processing
  - For example if a group of users tags pages describing products consistently as "stars:3", "stars:0", "stars:5", etc. to express how content they are with the product then these tags can be automatically processed to compute for example average product ratings. And because the characterization operator is not commutative it would know that it should ignore tags such as "3:stars" because they can mean something else. (In case of grouping (3, stars) and (stars, 3) would be considered equal.)

A.2 INTRODUCTORY TEXT ON RDF

Introduction to RDF/S

RDF graphs contain simple statements ("sentences") about resources (which, in other contexts, are be called "entities", "objects", etc., i.e., elements of the domain that may take part in relations). Statements are triples consisting of subject, predicate, and object, all of which are resources:

Subject	Predicate	Object

If we want to refer to a specific resource, we use (supposedly globally unique) URIs. If we want to refer something about which we know that it exists but we don't know or don't care what exactly it is, we use blank nodes. For example we know that each country has a capital city but perhaps we don't know what the capital of Mali is but we want to say that there is some. In this case we would use a blank node for the capital (instead of a URI):

geo:mali	geo:hasCapital	_:maliCapital

where geo:mali and geo:hasCapital are URIs and _:maliCapital indicates a blank node (the identifier _:maliCapital is used only for simplification and better readability in this experiment, otherwise it could well be _:x05fg85t and the meaning would be the same). Blank nodes play the role of existential quantifiers in logic. However, blank nodes may not occur in predicate position. In the object position of a triple there can also be literal values. Literal values are used to represent dates, character strings, numbers, etc.
RDF may be serialized in many formats. The following example is written in the Turtle serialization:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix bib: <http://www.edutella.org/bibtex#> .
@prefix ex: <http://example.org/libraries/#> .

ex:smith2005 a bib:Article ; dc:title "...Semantic Web..." ;
dc:year "2005" ;
ex:isPartOf [ a bib:Journal ; bib:number "11"; bib:name "Computer
Journal" ] ;
```

The document begins with a definition of namespace prefixes used in the remainder of the document (omitting common RDF namespaces), each line contains one or more statements separated by colon or semi-colon. If separated by semi-colon, the subject of the previous statement is carried over. E.g., line 1 reads as ex:smith2005 is a (has rdf:type) bib:Article and has dc:title "...Semantic Web...". Line 2 shows a blank node: the article is part of some entity which we can not (or don't care to) identify by a unique URI but for which we give some properties: it is a bib:Journal, has bib:number "11", and bib:name "Computer Journal".

A.3 TEXT A

Text - Revision 1

The Ant project started on 1st of April 2008. Thematically, it is situated in the area of social software. Anna is the project coordinator. The other people working the project include Al, a programmer, Andy, an analyst and Allen, a student of Software Engineering.
The project will end on 31st of March 2011.

Text - Revision 2

The Ant project, a cooperation with Anchor Inc., started on 1st of April 2008. Thematically, it is situated in the area of social software and deals with the development of a new social network. Anna is the project coordinator. She is responsible for supervising the project members' work. Before taking this position, she worked in the Deer project which ended in late 2007. Anna has been working for the company since 2003. The other people from the company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time. The project will end on 31st of March 2011.

Text - Revision 3

The Ant project, a cooperation with the Madison branch of Anchor Industries, started on 1st of April 2008. Thematically, it is situated in the area of social software and deals with the development of a new social network for simplifying collaboration in Biomedical research. Anna is the project coordinator. She is responsible for supervising the project members' work. Before taking this position, she worked in the Deer project which ended in late 2007. Anna has been working for the company since 2003. The other people from our company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time. Al holds a degree in Physics with a minor in computer science, he has eight years of experience with programming in Java and used to teach Java. Allen has some experience in Java, but has not used it as much as Python, his favourite programming language which he has been using for five years. Our contact person at Anchor Industries is Ali, a medical engineer who serves as an advisor. The project will end on 31st of March 2011. Upon successful review, its duration may be extended to four years.

Text - Revision 4

Currently, there are two projects in our company: The Ant project and the Bee project.
The Ant project, a cooperation with the Madison, Wisconsin (Latitude = 43.0553, Longitude = -89.3992) branch of Anchor Industries, started on 1st of April 2008. It deals with the development of a new social network for simplifying collaboration in Biomedical research. Specifically, it aims at making it easy for researchers to find related projects and to cooperate. Anna is the project coordinator. She is responsible for supervising the project members' work. Before taking this position, she worked as an architect in the Deer project which ended in late 2007. Anna has been working for the company since 2003, starting as an intern in usability testing. The other people from our company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time, 15 hours a week. Al holds a Master's degree in Physics with a minor in computer science, he has eight years of experience with programming in Java and used to teach Java to undergraduate students for three semesters. He has expert knowledge of Java EE and JavaServer Faces. Allen has some experience in Java, but has not used it as much as Python, his favourite programming language which he has been using for five years. Our contact person at Anchor Industries is Ali, a medical engineer who serves as an advisor. She has worked on a similar project, the Eagle project which ran from 2003 to 2006, before. Ali can be reached between Wednesday and Friday every week. The Ant project will end on 31st of March 2011. Upon successful review, its duration may be extended by up to two years.

Text - Revision 5

Currently, there are two projects in our company: The Ant project and the Bee project.
The Ant project, a cooperation with the Madison, Wisconsin (Latitude = 43.0553, Longitude = -89.3992) branch of Anchor IndustriesAnchor Industries, started on 1st of April 2008. It deals with the development of a new social network for simplifying collaboration in Biomedical research. Specifically, it aims at making it easy for researchers to find projects that are similar in their topic, participating researchers and institutions or location and to cooperate by sharing access to expensive experimental equipment that not all research facilities own and by sharing the contact data of participants which might be interested in participating in further experiments. Anna is the project coordinator. She is responsible for supervising the project members' work. Before taking this position, she worked as an architect in the Deer project which ended in late 2007. Anna has been working for the company since 2003, starting as an intern in usability testing. The other people from our company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time, 15 hours a week. Al holds a Master's degree in Physics with a minor in computer science. He has eight years of experience with programming in Java and used to teach Java to undergraduate students for three semesters. He has expert knowledge of Java EE and JavaServer Faces. Andy is a self-taught programmer who holds a Bachelor's degree in Biology. He has five years of experience in Programming in Java, and seven years of experience with web programming overall. Allen has taken three classes in Java at university, but has not used it as much as Python, his favourite programming language which he has been using for five years. Our contact person at Anchor Industries is Ali, a medical engineer who serves as an advisor in questions concerning biomedical practice. She has worked on a similar project, the Eagle project which ran from 2003 to 2006, before. Ali can be reached by phone between Wednesday and Friday every week but can reply to emails from Monday to Friday. The Ant project will end on 31st of March 2011. Upon successful review, its duration may be extended by up to two years.

Text - Revision 6

The Ant project, a cooperation with the Madison, Wisconsin (Latitude = 43.0553, Longitude = -89.3992) branch of Anchor Industries, started on 1st of April 2008. It deals with the development of a new social network for simplifying collaboration in Biomedical research. Specifically, it aims at making it easy for researchers to find projects that are similar in their topic, participating researchers and institutions or location and to cooperate by sharing access to expensive experimental equipment that not all research facilities own and by sharing the contact data of participants which might be interested in participating in further experiments. The software will be written in Java using the JBoss Seam framework. Anna is the project coordinator. She is responsible for supervising the project members' work and overseeing the evaluation of the final product. Before taking this position, she worked as an architect in the Deer project which ended in late 2007. Anna has been working for the company since 2003, starting as an intern programmer. The other people from our company working for the project include Al and Andy, both programmers, and Allen, who studies computer science with a focus on software engineering and who works for the project part-time, 15 hours a week. Al holds a Master's degree in Physics with a minor in computer science. He has eight years of experience with programming in Java and used to teach Java to undergraduate students for three semesters. He has expert knowledge of Java EE and JavaServer Faces. Before joining our company, he worked as a Perl programmer for three years. Andy is a self-taught programmer who holds a Bachelor's degree in Biology. He has five years of experience in Programming in Java, and seven years of experience with web programming overall. Allen has taken three classes in Java at university, but has not used it as much as Python, his favourite programming language which he has been using for five years. Our contact person at Anchor Industries is Ali, a medical engineer who serves as an advisor in questions concerning biomedical practice. She has worked on a similar project, the Eagle project which ran from 2003 to 2006, before. Ali can be reached by phone between Wednesday and Friday every week but can reply to emails from Monday to Friday. The Ant project will end on 31st of March 2011. Upon successful review, its duration may be extended by up to two years.

A.4 TEXT B

Text - Revision 1

Bob is in charge of the Bee project which started on November 15th 2007 and will run for five years. Benjamin is employed as the head programmer in the project. He supervises the work of Bill, Barbara and Bud.

Text - Revision 2

Bob is in charge of the Bee project which started on November 15th 2006 and will run for six years and five months. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Bill and Barbara are programmers, Bud is technical document writer, he is American who speaks Spanish fluently and is learning French. Barbara knows C++ and Java, Bill knows Java and Python. The Bee project is a small long-term project for a big telecomunication company. The Bee project team is located in London.

Text - Revision 3

Bob is in charge of the Bee project which started in autumn 2006 and will run for six years and a half. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Bill is a novice programmer, Barbara is an experienced programmer, Bud is technical document writer, he is an American from New York who speaks Portugese fluently and his French is on an intermediate level. Bud is now working on a component design document. Barbara knows C++ (she was teaching C++ for a while) and Java (she was programming in it for 3 years), Bill knows Java (for 8 years, he is an expert) and Python (3 years, it is his hobby). The Bee project is a large long-term project for a big mobile operator company. The Bee project team is located in London - West Kensington and cooperates very well with a team based in Bangalore, India.

Text - Revision 4

Bob is in charge of the Bee project which started in autumn 2006 and will run for six years and a half. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Bill is an expert programmer, Barbara is a former consultant and an experienced programmer with extensive theoretical knowledge, Bud is technical document writer, he is an American from New York (geo-location 40.76 (latitude), -73.98(longitude)) who speaks Portugese fluently (he is a native speaker) and his French is on an intermediate level (he has been learning French for 3 years). Bud is now working on a design document for the SingleSignOn component. Barbara knows C++ (she was teaching C++ for 2 years) and Java (she was programming in it for 3 years) and she has a project manager experience, Bill knows Java (for 8 years, he is an expert who worked on the JVM too) and Python (3 years, it is his hobby, he loves Python). Barbara worked as a tester for a year in the past. The Bee project is a large long-term project for a big mobile operator company called PhoneCorp. The Bee project team is located in London - West Kensington, geo-location 51.49 (latitude), -0.220 (longitude), previously it was located in the New York headquarters, and cooperates very well with a testing team based in Bangalore, India.

Text - Revision 5

Bob is in charge of the Bee project which started in autumn 2006 and will run for six years and a half. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Bill is an expert programmer, Barbara is a former junior consultant and an experienced programmer with extensive theoretical knowledge (she has a background in theoretical computer science, esp. the theory of complexity), Bud is technical document writer, he is an African-American from New York (geo-location 40.76 (latitude), -73.98(longitude)) who speaks Portugese fluently (he is a native speaker) and his French is on an intermediate level (he has been learning French for 3 years), he understands Spanish and Italian pretty well but cannot speak. Bud is now working on a design specification document for the SingleSignOn subcomponent of the security component. Barbara knows C++ (she was teaching C++ for 2 years) and Java (she was programming in it for 3 years, her focus is on frontend programming, especially in the JSF technology) and she has a project manager experience, Bill knows Java (for 8 years, he is an expert who worked on the JVM of some company too) and Python (3 years, it is his hobby, he loves Python, Java not so much). Barbara worked as a tester for a year in the past. Barbara knows a colleague very well. The Bee project is a large long-term project for a big mobile operator company called PhoneCorp. The Bee project team is located in London - West Kensington, geo-location 51.49 (latitude), -0.220 (longitude), previously it was located in the New York headquarters, and cooperates very well with and manages a testing team based in Bangalore, India. The other people working for the project include Bao and Bert. Bob maybe knows Bert.

Text - Revision 6

Bob is in charge of the Bee project which started in autumn 2006 and will run for six years and a half. The project can be extended by up to two years upon a successful review. Bob is not in charge of any other project. Benjamin is employed as the team lead in the project. He supervises the work of Bill, Barbara and Bud. Before joining our company he worked as a consultant for an international company. Bill is an expert programmer, Barbara is a former junior consultant and an experienced programmer with extensive theoretical knowledge (she has a background in theoretical computer science, esp. the theory of complexity), Bud is technical document writer, not a programmer, he is an African-American from New York (geo-location 40.76 (latitude), -73.98(longitude)) who speaks Portugese fluently (he is a native speaker) and his French is on an intermediate level (he has been learning French for 3 years), he understands Spanish and Italian pretty well but cannot speak. Bud is now working on a design specification document for the SingleSignOn subcomponent of the security component. Barbara knows C++ (she was teaching C++ for 2 years) and Java (she was programming in it for 3 years, her focus is on frontend programming, especially in the JSF technology), she has a project manager experience and she does not speak French and does not like Bud because he's a chauvinist, Bill knows Java (for 8 years, he is an expert who worked on the JVM of Sun Microsystems too) and Python (3 years, it is his hobby, he loves Python, Java not so much). Barbara worked as a tester for a year in the past. Barbara knows a colleague very well, the colleague is Bob. The Bee project is a large long-term project for a big mobile operator company called PhoneCorp. The Bee project team is located in London - West Kensington, geo-location 51.49 (latitude), -0.220 (longitude), previously it was located in the New York headquarters, and cooperates very well with and manages a testing team based in Bangalore, India. Project management is not the responsibility of Bill. Bill is reachable on Monday, Tuesday, and Wednesday by phone but he can reply to e-mails from Monday to Friday. The other people from the company working for the project include Bao and Bert. Bob maybe knows Bert.

B

KWQL AND VISKWQL

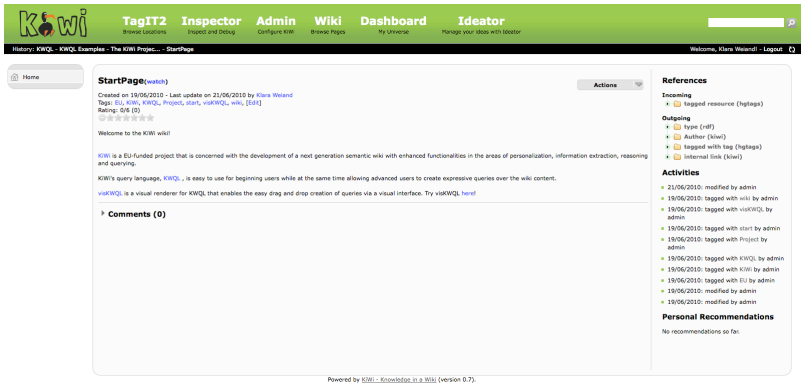
B.1 INTRODUCTORY TEXT ON THE KIWI WIKI

Das KiWi Wiki

Dieser Text gibt einen Überblick über das KiWi Wiki, welches im Rahmen des Projekts “KiWi - Knowledge in a Wiki” entwickelt wurde.

Wikis sind Werkzeuge zum Wissensmanagement, die es Benutzern ermöglichen, einfach gemeinsam Inhalte zu erstellen, zum Beispiel zu einem bestimmten Thema oder Projekt. Wikiseiten können dabei direkt im Browser editiert und durch Hyperlinks miteinander verbunden werden. Ein Merkmal von Wikis ist, dass der Inhalt sich graduell entwickelt und oft von vielen verschiedenen Benutzern editiert, erweitert und verbessert wird, also das kollektive Wissen von vielen widerspiegelt.
Die gleichen Informationen können in einem Wiki auf verschiedene Arten dargestellt werden. Häufig bilden sich jedoch mit der Zeit Konventionen zur Formatierung und Darstellung von Inhalten.
Das KiWi Wiki ist ein semantisches Wiki. Das heißt, dass Informationen nicht nur durch Text dargestellt werden, sondern auch durch Annotationen, die den Inhalt eines Texts oder Textabschnitts wiedergeben oder ergänzen. Im Folgenden geht es um Annotationen in Form von Tags, das heißt von den Benutzern frei gewählte Schlagworte.

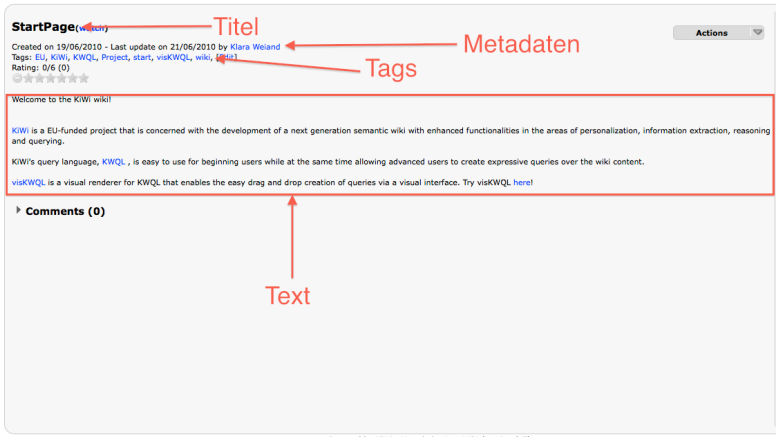
In diesem Text werden die wichtigsten Elemente des KiWi Wikis vorgestellt.



Obiges Bild zeigt die Startseite eines KiWi Wikis, das Informationen über das KiWi Projekt enthält. Der grüne Balken enthält Links zu anderen Teilen des Wikis und kann für den Zweck dieses Experiments ignoriert werden.
Der schmale schwarze Balken darunter zeigt links die “History” an, eine Liste der zuletzt besuchten Seiten im Wiki. In diesem Fall wurden zum Beispiel vor der aktuellen Seite die Seiten “KWQL”, “KWQL Examples” und “The KiWi Project” besucht. Rechts in der schwarzen Leiste können Benutzer sich ein- und ausloggen und registrieren.

Das KiWi Wiki 1/4

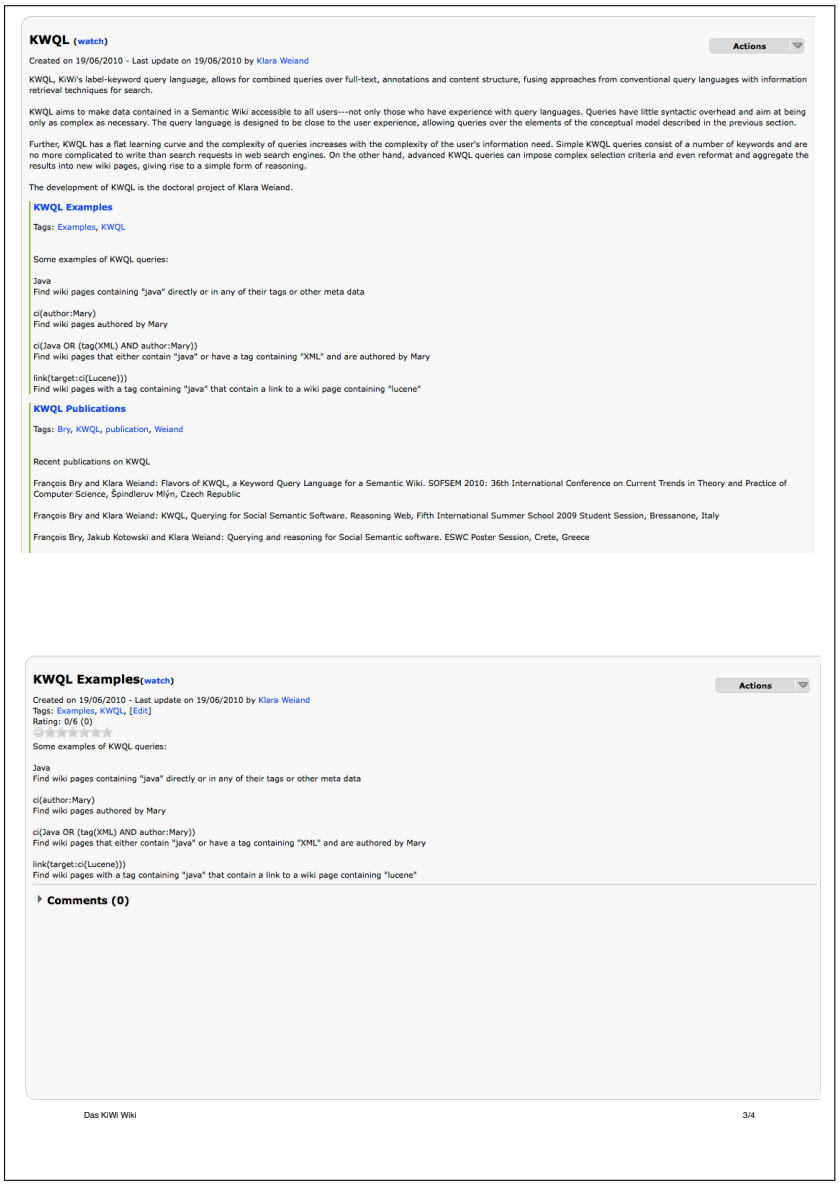
Links unter der schwarzen Leiste befindet sich ein grau hinterlegter “Home” Link. Dieser ist auf jeder Wikiseite zu sehen und führt zur Startseite des Wikis zurück.
Den Rest des Platzes füllt die eigentliche Wikiseite. Wikiseiten heißen im KiWi Wiki **Content Items**.
Ein Content Item besteht aus einem Titel (hier “StartPage”), einem Text (“Welcome to the KiWi wiki!...”), Tags (“EU”,”KiWi”, usw.), Metadaten, d.h., Informationen darüber, wann und von wem die Seite bearbeitet wurde, sowie einer Bewertung und Kommentaren. In der Spalte rechts finden sich außerdem Informationen über ein- und ausgehende Links, die letzten Änderungen an diesem Content Item sowie Hinweise auf verwandte Content Items. Die zuletztgenannten Informationen können für das Experiment benutzt werden, sind



jedoch nicht erforderlich, um die Aufgaben zu lösen. Das obige Bild zeigt das Content Item und seine wichtigsten Eigenschaften.
Nicht im Bild zu sehen ist, dass auch Tags Metadaten besitzen, nämlich den Zeitpunkt, zu dem sie vergeben wurden, sowie den Benutzer, der sie vergeben hat. Zwei Benutzer können dabei das gleiche Content Item mit dem gleichen Tag versehen.

Content Items können außerdem strukturell mit anderen Content Items verbunden sein. Dies ist auf zwei Arten möglich:

• Content Items können aufeinander verlinken. Diese Links verhalten sich in etwa wie Links auf Webseiten. Im obigen Bild sind die Anchor-Texte, also die Texte, auf die man klicken muss, um dem Link zu folgen (“KiWi”, “KWQL”, “visKWQL” und “here”) blau hinterlegt.
• Content Items können in andere Content Items eingebettet sein, wie es beim Content Item im Screenshot oben auf der nächsten Seite der Fall ist. Hier sind die Content Items “KWQL Examples” und “KWQL Publications” in dem Content Item “KWQL” enthalten. Erstere sind eigene Content Items im Wiki, werden hier aber im Kontext eines anderen



KWQL (watch)

Actions

Created on 19/06/2010 - Last update on 19/06/2010 by Klara Weiand

KWQL, KiWi's label-keyword query language, allows for combined queries over full-text, annotations and content structure, fusing approaches from conventional query languages with information retrieval techniques for search.

KWQL aims to make data contained in a Semantic Wiki accessible to all users---not only those who have experience with query languages. Queries have little syntactic overhead and aim at being only as complex as necessary. The query language is designed to be close to the user experience, allowing queries over the elements of the conceptual model described in the previous section.

Further, KWQL has a flat learning curve and the complexity of queries increases with the complexity of the user's information need. Simple KWQL queries consist of a number of keywords and are no more complicated to write than search requests in web search engines. On the other hand, advanced KWQL queries can impose complex selection criteria and even reformat and aggregate the results into new wiki pages, giving rise to a simple form of reasoning.

The development of KWQL is the doctoral project of Klara Weiand.

KWQL Examples

Tags: Examples, KWQL

Some examples of KWQL queries:

Java
Find wiki pages containing "java" directly or in any of their tags or other meta data

ci(author:Mary)
Find wiki pages authored by Mary

ci(Java OR (tag(XML) AND author:Mary))
Find wiki pages that either contain "java" or have a tag containing "XML" and are authored by Mary

link(target:ci(Lucene)))
Find wiki pages with a tag containing "java" that contain a link to a wiki page containing "lucene"

KWQL Publications

Tags: Bry, KWQL, publication, Weiand

Recent publications on KWQL

François Bry and Klara Weiand: Flavors of KWQL, a Keyword Query Language for a Semantic Wiki. SOFSEM 2010: 36th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic

François Bry and Klara Weiand: KWQL, Querying for Social Semantic Software. Reasoning Web, Fifth International Summer School 2009 Student Session, Bressanone, Italy

François Bry, Jakub Kotowski and Klara Weiand: Querying and reasoning for Social Semantic software. ESWC Poster Session, Crete, Greece

KWQL Examples(watch)

Actions

Created on 19/06/2010 - Last update on 19/06/2010 by Klara Weiand
Tags: Examples, KWQL, [Edit]
Rating: 0/6 (0)

Some examples of KWQL queries:

Java
Find wiki pages containing "java" directly or in any of their tags or other meta data

ci(author:Mary)
Find wiki pages authored by Mary

ci(Java OR (tag(XML) AND author:Mary))
Find wiki pages that either contain "java" or have a tag containing "XML" and are authored by Mary

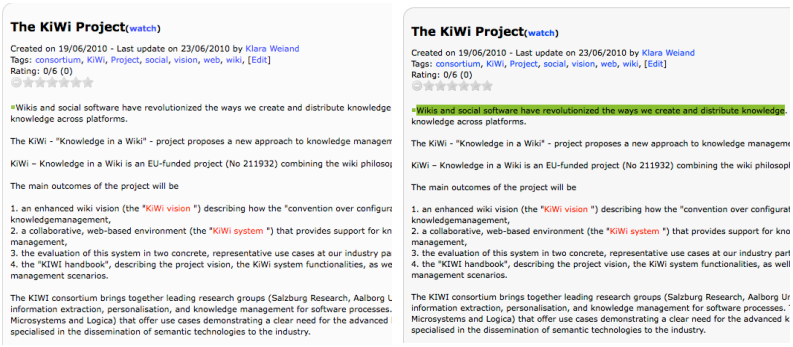
link(target:ci(Lucene)))
Find wiki pages with a tag containing "java" that contain a link to a wiki page containing "lucene"

Comments (0)

Das KiWi Wiki 3/4

Content Items zusammen mit ihren Tags und ihrem Inhalt angezeigt. Die grüne Linie am linken Rand zeigt an, wo das eingebettete Content Item beginnt und endet. Wenn man auf den Titel eines eingebetteten Content Items klickt, wird man zu ihm weitergeleitet (siehe das untere Bild auf der vorigen Seite).

Tags können im KiWi Wiki nicht nur für ganze Content Items vergeben werden, sondern auch für **Fragmente**. Fragmente sind Textteile, die mit Tags versehen sind. Benutzer können beliebige zusammenhängende Textabschnitte innerhalb eines Content Items auswählen und taggen. Während Content Items dazu benutzt werden, um den Text ähnlich wie Kapitel oder Abschnitte in einem Buch aufzuteilen und zu formatieren, kann man sich Fragmente wie Markierungen und Randnotizen in diesem Buch vorstellen, die



jeder Benutzer individuell anbringen kann. Das Bild oben links zeigt die Standardansicht eines Fragments - ein grünes Rechteck im Text (vor dem ersten Wort des Texts).

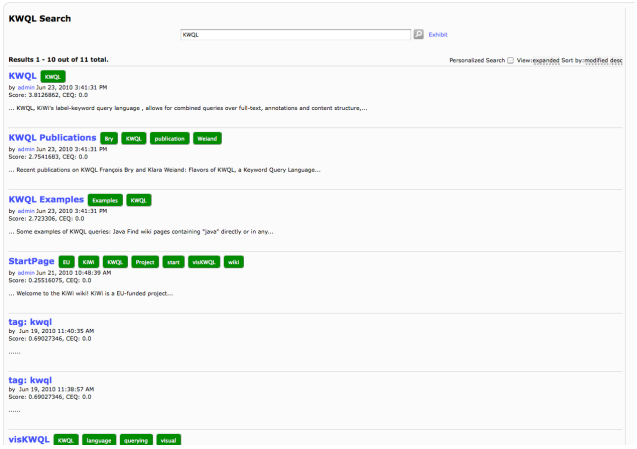
Wenn man den Mauszeiger über das Fragment bewegt, wird der ganze im Fragment enthaltene Text grün hinterlegt (siehe rechtes Bild). Beim Klick auf das Rechteck öffnet sich ein Fenster, in dem die Tags des Fragments angezeigt werden (ohne Abbildung).

Zusammenfassend gesagt besteht das KiWi Wiki also aus Content Items, Fragments und Tags, deren Eigenschaften, und den Beziehungen zwischen ihnen.

B.2 INTRODUCTORY TEXT ON KWQL

2. KWQL - Eine Einleitung

Dieser Text gibt eine Übersicht über KWQL, die Anfragesprache von KiWi. KWQL kann sowohl für einfache als auch für komplexe Anfragen über die verschiedenen Elemente des Wikis benutzt werden. Jede Anfrage wählt eine (eventuell leere) Menge an Content Items aus, wobei Anfragen als (unvollständige) textuelle Beschreibungen der auszuwählenden Content Items gesehen werden können.
Das Bild zeigt die KWQL Suchseite im KiWi Wiki. In das Textfeld wird die Anfrage (hier "KWQL") eingegeben. Durch einen Klick auf den grauen Button rechts wird die Auswertung der Anfrage gestartet. Das Ergebnis der Anfrage, eine Liste von Content Items, wird darunter angezeigt.



Die einfachste KWQL Anfragen bestehen aus einem oder mehreren Keywords:

```
KWQL
```

Diese Anfrage gibt alle Content Items zurück, die "KWQL" enthalten.

```
KWQL KiWi
```

Diese Anfrage gibt alle Content Items zurück, die sowohl "KWQL" als auch "KiWi" enthalten. Wenn kein anderer Operator angeben wird, wird Konjunktion, also Und-Verknüpfung angenommen. Die folgende Anfrage ist deswegen äquivalent zu der zweiten Anfrage:

```
KWQL AND KiWi
```

Um präzisere Anfragen zu ermögliche, kann der Kontext, in dem die Keywörter vorkommen sollen, angegeben werden. Die Anfrage

```
ci(title:KWQL)
```

wählt alle Content Items (in KWQL abgekürzt als "ci") aus, in deren Titel "KWQL" vorkommt.

KWQL - Eine Einleitung 1/5

```
ci(title:KWQL AND text:KiWi)
```
Diese Anfrage wiederum gibt eine Untermenge der Ergebnisse der vorigen Anfrage zurück, nämlich alle Content Items, in deren Titel “KWQL” und in deren Text “KiWi” vorkommt.
Da die explizite Angabe der Konjunktion, “AND”, optional ist, ist die folgende Anfrage äquivalent zur letzten Anfrage:
```
ci(title:KWQL text:KiWi)
```

Anfragen in visKWQL sind also textuelle Abstraktionen der Wikiseiten, die zurückgegeben werden sollen.

Die Elemente des konzeptionellen Modells von KiWi, Content Items, Fragmente, Links und Tags, wie in der Einleitung über KiWi beschrieben, werden in KWQL **Ressourcen** genannt. Die verschiedenen Typen von Eigenschaften heißen eines Content Items heißen **Qualifier**. Qualifier können zum Beispiel Text, Metadaten oder strukturelle Eigenschaften sein. Zum Beispiel haben Content Items, nicht aber Tags textuellen Inhalt, während Fragmente im Gegensatz zu Content Items keinen Titel haben. In den Begrifflichkeiten von KWQL ausgedrückt bedeutet das, dass der Qualifier `text` für Content Items, nicht aber für Tags definiert ist, und `title` für Content Items aber nicht für Fragmente. Eine vollständige Liste von Ressourcen und ihren Qualifiern findet sich auf der nächsten Seite.
Qualifierwerte oder “Keywords” schließlich geben an, welchen Wert der entsprechende Qualifier in den auszuwählenden Content Items haben soll oder, genauer gesagt, welcher Wert darin vorkommen soll. So wählt die vierte Anfrage auf der letzten Seite zum Beispiel nicht nur Content Items aus, deren Titel genau “KWQL” ist, sondern auch solche, in denen “KWQL” eines von mehreren Wörtern ist. Qualifier-Keywortpaare wie z.B. `text:KiWi` werden **Qualifierterme** genannt.
Qualifier und Ressourcen sind also dazu da, um den Kontext, in dem ein Keyword vorkommen soll, festzulegen.
Ein Term der Form `Ressource(Qualifierterm)` wird **Ressourceterm** genannt. Er besagt, dass Content Items, die eine Ressource vom Typ `Ressource` enthalten, die im Qualifier `Qualifier` den Wert `Keyword` enthalten, zurückgegeben werden sollen. Ein Ressourceterm kann mehrere Qualifierterme enthalten, zum Beispiel ist das beim zweiten Beispiel auf dieser Seite der Fall.
Ressourceterme können ineinander verschachtelt werden, um auszudrücken, dass eine bestimmte Ressource in einer anderen enthalten ist, zum Beispiel ein Fragment in einem Content Item oder ein Link in einem Fragment.
```
ci(text:KiWi tag(name:KWQL))
```
zum Beispiel beschreibt Content Items, die im Text “KiWi” enthalten und die außerdem getaggt sind mit einem Tag, dessen Name “KWQL” ist. Die unterschiedlichen Arten von Ressourcen können jeweils verschiedene **Subresourcen** enthalten, der untere Teil der folgenden Tabelle gibt eine Übersicht darüber.

Ressource	Content Item	Fragment	Link	Tag
Qualifier	URI author created lastEdited title text numberEdits descendant child	URI author created descendant child	target anchorText	URI author created name
Subresourcen	fragment link tag	link tag	tag	

Qualifier
In der obigen Tabelle sind die verschiedenen Ressourcen mit ihren jeweiligen Qualifiern aufgeführt.
URIs sind eindeutige Bezeichner für Ressourcen. So haben alle Content Items, Fragments und Tags unterschiedliche URIs und sind so eindeutig unterscheidbar und ansprechbar.
author bezeichnet die Personen, die ein Content Item angelegt oder editiert haben, ein Fragment angelegt oder eine Ressource mit einem Tag versehen haben.
created und **lastEdited** beziehen sich auf die Zeitpunkte, zu dem die entsprechende Ressource angelegt oder zuletzt editiert wurde.
title bezeichnet den Titel eines Content Items, **text** seinen textuellen Inhalt.
numberEdits ist die Anzahl der Editiervorgänge an einem Content Item.
name bezieht sich auf den Namen eines Tags.
Der **anchorText** eines Links ist der Text, auf den man klicken muss, um dem Link zu folgen.

child, **descendant** und **target** haben eine besondere Rolle, sie beziehen sich auf die Wikistruktur und nehmen als Wert kein Keyword oder ein Konstrukt aus Keywords und Operatoren, sondern eine Unteranfrage, die das verbundene Content Item oder Fragment beschreibt.
child und **descendant** beziehen sich auf die Verschachtelungen von Content Items und Fragmenten, und werden benutzt, um anzugeben, welche anderen Content Items oder Fragments direkt (child) oder über mehrere Schritte (descendant) im beschriebenen Content Item enthalten sein sollen.

```
ci(tag(name:KWQL) child:ci(tag(Example)))
```

Diese Anfrage gibt Content Items zurück, die ein Tag “KWQL” haben und außerdem ein eingebettetes Content Item enthalten, das im Tag das Wort “Example” enthält. Der Wert von `child` ist hier `ci(tag(Example))`. Wenn der Term “Example” an beliebiger Stelle im enthaltenen Content Item vorkommen darf, lautet die Anfrage:

```
ci(tag(name:KWQL) child:Example)
```

target bezeichnet das Ziel eines Links, also ein verlinktes Content Item.

```
ci(link(target:ci(KiWi)))
```

Der Wert von `target` ist hier `ci(KiWi)`, die Anfrage wählt also Content Items aus, die auf andere Content Items verlinkt sind, die wiederum “KiWi” enthalten.

Unterspezifizierte Anfragen

Wie am Anfang beschrieben, kann der Kontext, in dem ein Keyword vorkommen soll, nur teilweise oder gar nicht spezifiziert werden. Das bedeutet, dass in KWQL Anfragen alles außer den Keywords optional ist. Wenn die Ressource oder der Qualifier weggelassen werden, werden automatisch alle mit der unvollständigen Anfrage kompatiblen vollständigen Anfragen ausgewertet und die Vereinigung der Ergebnisse zurückgegeben.

```
KWQL
```

In KWQL Begrifflichkeiten ausgedrückt, bedeutet diese Anfrage vom Anfang dieses Texts also, dass das Keyword “KWQL” in einer beliebigen Ressource (innerhalb eines Content Items, jedoch nicht in seinen verlinkten oder eingebetteten Content Items) und in einem beliebigen Qualifier vorkommen muss.

```
ci(KWQL)
```

Diese Anfrage besagt, dass “KWQL” innerhalb eines Content Items (aber nicht in einer Subressource) im Wert von einem beliebigen Qualifier vorkommen muss.

```
tag(KWQL)
```

Diese Anfrage ist doppelt unterspezifiziert: Zum einen ist kein Qualifier angegeben, “KWQL” kann also an beliebiger Stelle innerhalb eines Tags vorkommen. Zum anderen ist keine enthaltende Ressource für das Tag angegeben. Die Anfrage gibt also sowohl Content Items zurück, die ein Tag haben, in dessen Eigenschaften “KWQL” vorkommt, als auch Content Items, die ein Fragment oder einen Link enthalten, der diese Bedingung erfüllt.

```
author:Mary
```

Hier ist ebenfalls die Ressource nicht angegeben. Da sowohl Content Items, als auch Tags und Fragmente Autoren haben, ist die Anfrage ambig. Zurückgegeben werden hier Content Items, bei denen Mary Autor ist oder die eine Subresource enthalten, von denen Mary Autor ist.

Operatoren

Die Operatoren `NOT`, `OR` und `AND` (jeweils großgeschrieben) können benutzt werden, um Verneinung, (nicht ausschliessende bzw. logische) Disjunktion (Oder-Verknüpfung) und Konjunktion (Und-Verknüpfung) auszudrücken. Konjunktion wird, wie bereits erwähnt, automatisch angenommen, während Disjunktion explizit angegeben werden muss.

```
ci(text:KiWi OR tag(name:KWQL))
```

besagt, dass Content Items, die im Text “KiWi” enthalten oder mit einem Tag “KWQL” getaggt sind, zurückgegeben werden sollen.

```
ci(text:KiWi OR NOT(tag(name:KWQL)))
```

Diese Anfrage wiederum wählt Content Items aus, die “KiWi” im Text enthalten oder nicht mit “KWQL” getaggt sind.

Operatoren können auf der Ebene von Ressourcetermen, Qualifiertermen, Werten benutzt werden. Präzedenz wird dabei durch Klammerung ausgedrückt.

Einige Beispiele für die Benutzung von Operatoren:

```
ci(KiWi AND (KWQL OR NOT(XML)))
```
Hier werden Content Items ausgewählt, die “KiWi” enthalten und außerdem “KWQL” enthalten oder “XML” nicht enthalten.
```
ci(tag(name:KiWi) OR tag(name:KWQL))
```
Diese Anfrage gibt Content Items zurück, die mit “KiWi” oder mit “KWQL” getaggt sind. Letztere Anfrage ist äquivalent zu
```
ci(tag(name:(KiWi OR KWQL))
```

Variablen
Wenn ein Qualifier angegeben ist, können anstelle von Keywords auch Variablen der Form `$name` angegeben werden. Variablen in KWQL dienen verschiedenen Zwecken:
1. Als Wildcard, das heißt, um anzugeben, dass etwas vorhanden sein muss, sein Wert an sich aber nicht von Belang ist.
```
ci(fragment(URI:$u))
```
   zum Beispiel gibt Content Items zurück, die mindestens ein Fragment enthalten, ohne dabei weitere Selektionskriterien auf das Fragment anzuwenden.
```
ci(child(ci(URI:$u)))
```
   gibt Content Items zurück, in denen mindestens ein weiteres Content Item eingebettet ist, unabhängig davon, was das enthaltene Content Item für Eigenschaften hat.
2. Um die Gleichheit von mindestens einem Element von zwei Qualifierwerten zu erzwingen.
```
ci(tag(name:$t) text:$t)
```
   Zum Beispiel erfordert, dass mindestens ein Wort im Text des Content Items auch ein Tag des Content Items sein muss.

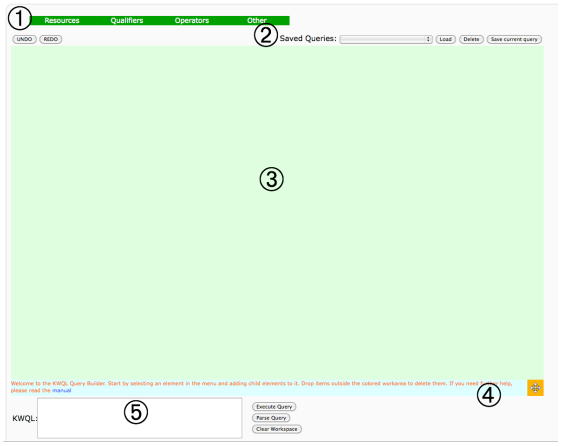
Injektivität

Wie bereits erwähnt, funktioniert KWQL als Anfragesprache, die aufgrund einer unvollständigen Beschreibung von Content Items alle Content Items, die mit den Informationen in der gegebenen Anfrage kompatibel sind, auswählt.
Injektivität bedeutet dabei, dass jeder Qualifier und jede Subresource, für die in der Anfrage mehrfach Selektionskriterien gegeben werden, sich auf unterschiedliche Elemente in den Daten beziehen. So bedeutet zum Beispiel
```
ci(tag(name:KiWi) tag(name:KWQL))
```
dass die auszuwählenden Content Items jeweils mindestens zwei Tags haben müssen, eines mit Namen “KiWi” und eine mit Namen “KWQL”. Die Anfrage ist nicht äquivalent zu
```
ci(tag(name:(KiWi AND KWQL))
```
welche Content Items auswählt, die Tags haben, die beide Kriterien erfüllen.
Prinzipiell können alle Subresourcen mehrfach in einer Anfrage vorkommen, zum Beispiel kann ein Fragment mehrere Tags haben oder ein Content Item mehrere Links enthalten. Bei Qualifiern sieht dies anders aus, viele Qualifier bezeichnen Eigenschaften, die jede Ressource nur einmal hat, zum Beispiel `title` oder `text`, und dürfen auch nur einmal vorkommen. `ci(title:KiWi title:KWQL)` zum Beispiel ist **keine** gültige Anfrage.
Einige Qualifier können aber auch mehrfach vorkommen:
```
ci(author:Mary author:John)
```
Diese Anfrage wählt Content Items aus, die sowohl von Mary als auch von John editiert worden sind.
Die Qualifier `author`, `descendant`, `child` und `name` können in einer Anfrage mehrfach angegeben werden.

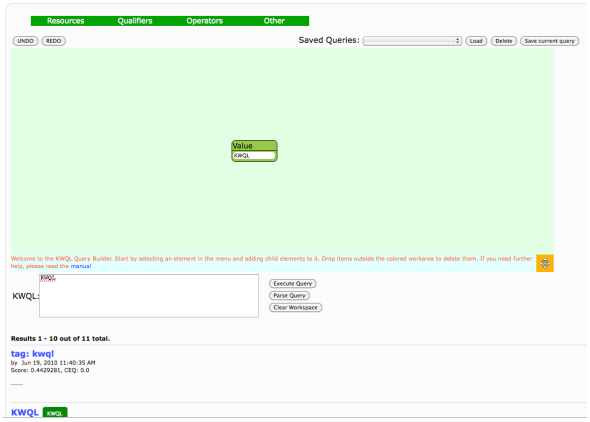
B.3 INTRODUCTORY TEXT ON VISKWQL

2. visKWQL - Eine Einleitung

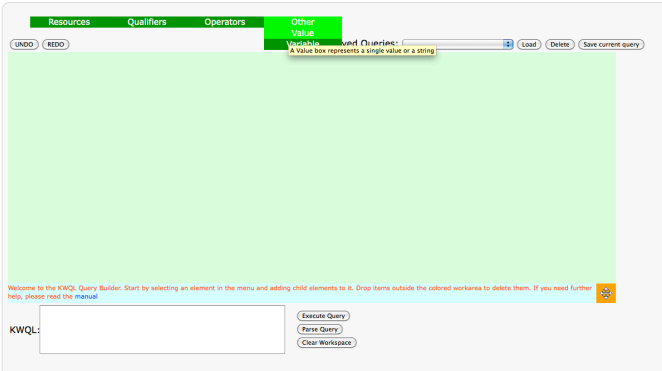
Dieser Text gibt eine Übersicht über visKWQL, die visuelle Anfragesprache von KiWi. visKWQL kann sowohl für einfache als auch für komplexe Anfragen über die verschiedenen Elemente des Wikis benutzt werden. Jede Anfrage wählt eine (eventuell leere) Menge an Content Items aus, wobei Anfragen als (unvollständigen) Darstellungen der auszuwählenden Content Items gesehen werden können.
Das Bild zeigt den visKWQL Editor. In der grünen Leiste (①) kann man die Elemente der

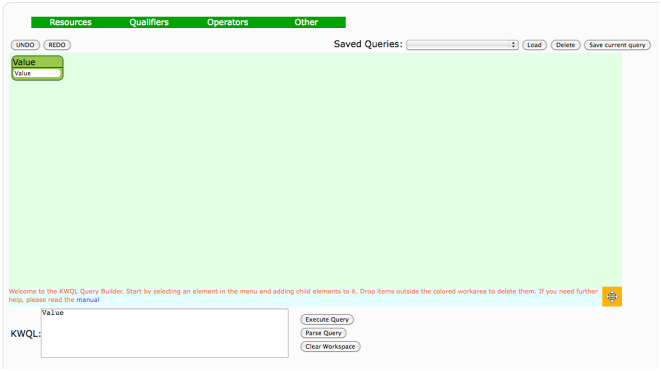


Anfrage auswählen, diese können dann per Drag und Drop zu einer Anfrage zusamm engesetzt werden. Mithilfe des mit ② markierten Menüs können Anfragen gespeichert und später wieder geladen werden. ③ ist die Arbeitsfläche des Editors, in dem die Anfragenelemente und Anfragen angezeigt werden. Um eine Anfrage oder ein Element daraus zu löschen, müssen die entsprechenden Boxen einfach aus dem Arbeitsbereich hinausgezogen und in der grauen Fläche links oder rechts davon fallengelassen werden. Hinweise zur Erstellung von Anfragen geben die Tipps in der hellblauen Leiste (④). Das Textfeld (⑤) schließlich zeigt eine Übersetzung der aktuellen Anfrage in KWQL, die textuelle Version von visKWQL, an. Anfragen können jederzeit in sowohl ihrer visuellen als auch textuellen Form editiert werden. Nach einer Änderung in der textuellen Version muss jedoch der “Parse Query” Button gedrückt werden, damit die visuelle Darstellung der Anfrage aktualisiert wird. Bei einer Änderung an der visuellen Anfrage hingegen wird die textuelle Form automatisch angepasst.
Im Folgenden wird es nur um die Erstellung von visuellen Anfragen gehen.
Durch einen Klick auf “Execute Query” wird die Anfrage ausgewertet. Die Content Items, die von der Anfrage ausgewählt wurden, erscheinen dann auf direkt unter dem Textfeld. Auf dem Bild auf der nächsten Seite ist eine einfache Anfrage zusammen mit einigen Ergebnissen zu sehen.



Die einfachsten KWQL Anfragen bestehen aus einem oder mehreren Keywords. So eine Anfrage ist in obigem Bild zu sehen. Diese Anfrage gibt alle Content Items zurück, die “KWQL” enthalten, egal an welcher Stelle.
Sie wird erstellt, indem man zuerst im Menü “Other” und dann “Value” auswählt. Dies ist auf dem Bild unten zu sehen. Auch hier werden kurze Hilfetexte angezeigt, um die Erstellung der Anfrage zu erleichtern.





Nachdem “Value” ausgewählt wurde, erscheint eine “Value” Box auf der Arbeitsfläche (siehe obiges Bild). Der obere, grün hinterlegte Bereich der Box (im Folgenden “Label” genannt) zeigt dabei den Typ des Elements an, das weiße Textfeld darunter den Wert. Für eine einfache Keywordsuche nach “KWQL” in allen Teilen eines Content Items muss der Wert von “Value” in “KWQL” geändert werden. Das Ergebnis ist die auf dem oberen Bild auf der letzten Seite gezeigte Anfrage.

Diese Anfrage gibt alle Content Items zurück, die sowohl “KWQL” als auch “KiWi” enthalten. Sie kann erstellt werden, indem zuerst in der Menüzeile der Operator “AND” ausgewählt wird.

Das Ergebnis ist eine leere Box vom Typ “AND”. Das Label ist orange hinterlegt, um anzuzeigen, dass die Box alleine keine gültige Anfrage darstellt. Die Erklärung dafür wird in der hellblauen Leiste gegeben:

Nachdem eine “Value” Box erstellt und in die “AND” Box gezogen wurde, sieht die Anfrage so aus:

Das Label ist immer noch orange hinterlegt. Eine Erklärung dafür ist wieder in der hellblauen Leiste zu finden:

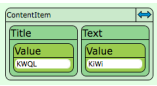
The marked AND box contains only one child and is being ignored in the KWQL output

Das Problem ist hier also, dass die Und-Verknüpfung “AND” mindestens zwei Kindelemente, das heißt, in ihr enthaltene Boxen braucht, um sinnvoll angewandt werden zu können. In der aktuellen Form wird die Anfrage deswegen zwar ausgewertet, das “AND” wird dabei aber ignoriert, das heißt, die Anfrage ist äquivalent zu dem ersten Beispiel auf Seite 2.
Um Raum für ein weiteres Kindelement zu schaffen, klicken wir auf den blauen Pfeil rechts oben im “AND” Element. Dadurch vergrößert sich die Box und es ist genug Platz, um eine weitere Box in das “AND” Element zu ziehen.

Nachdem das Kindelement hinzugefügt wurde, ist die Anfrage komplett. Da “AND” nun zwei Kinder hat und die Anfrage somit syntaktisch korrekt ist, ist das Label nicht mehr orange gefärbt

Um präzisere Anfragen zu ermögliche, kann der Kontext, in dem die Keywörter/Werte vorkommen sollen, angegeben werden. Die Anfrage

wählt alle Content Items aus, in deren Titel “KWQL” vorkommt.

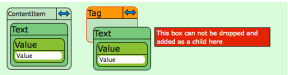


Diese Anfrage wiederum gibt eine Untermenge der Ergebnisse der vorigen Anfrage zurück, nämlich alle Content Items, in deren Titel “KWQL” und in deren Text “KiWi” vorkommt.

Anfragen in visKWQL sind also visuelle Abstraktionen der Wikiseiten, die zurückgegeben werden sollen.

Die Elemente des konzeptionellen Modells von KiWi, Content Items, Fragmente, Links und Tags, wie in der Einleitung über KiWi beschrieben, werden in visKWQL **Ressourcen**

genannt. Die verschiedenen Typen von Eigenschaften heißen eines Content Items heißen **Qualifier**. Qualifier können zum Beispiel Text, Metadaten oder strukturelle Eigenschaften sein. Zum Beispiel haben Content Items, nicht aber Tags textuellen Inhalt, während Fragmente im Gegensatz zu Content Items keinen Titel haben. In den Begrifflichkeiten von visKWQL ausgedrückt bedeutet das, dass der Qualifier `text` für Content Items, nicht aber für Tags definiert ist, und `title` für Content Items aber nicht für Fragmente. Eine vollständige Liste von Ressourcen und ihren Qualifiern findet sich auf der nächsten Seite.
Der Typ einer Ressource oder eines Qualifiers ist in seinem Label angegeben.
In visKWQL findet sich unter “Resources” eine Liste aller Resourcen. Unter “Qualifiers” werden für jeden Typ von Ressource alle erlaubten Qualifier (und Subressourcen, s. unten) aufgelistet. Auf diese Art ist einfach zu sehen, welche Resourcen mit welchen Qualifiern kombiniert werden dürfen.
Zusätzlich verhindert visKWQL es aktiv, dass unerlaubte Ressource-Qualifier Kombinationen entstehen. Im folgenden Bild ist zu sehen, dass die Kombination von Content Item und Text erlaubt ist, der Versuch, einen Text Qualifier in einer “Tag” Ressource abzulegen aber mit einer Warnung verhindert wird.

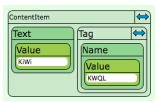


Qualifierwerte oder “Keywords” schließlich geben an, welchen Wert der entsprechende Qualifier in den auszuwählenden Content Items haben soll oder, genauer gesagt, welcher Wert darin vorkommen soll. So wählt die folgende Anfrage auf dieser Seite zum Beispiel nicht nur Content Items aus, deren Text genau “KiWi” ist, sondern auch solche, in denen “KiWi” eines von mehreren Wörtern ist. Qualifier-Keywortschachtelungen wie z.B.

werden **Qualifierterme** genannt.
Qualifier und Ressourcen sind also dazu da, um den Kontext, in dem ein Keyword vorkommen soll, festzulegen.

Eine Schachtelung von einem Qualifierterm in einer Ressource (zu sehen zum Beispiel links auf dem oberen Bild auf dieser Seite) wird **Ressourceterm** genannt. Er besagt, dass Content Items, die eine Ressource vom Typ `Ressource-Label` enthalten (oder, im Fall von Content Items, die eine Ressource dieses Typs sind), die im Qualifier `Qualifier-Label` den Wert `Keyword` enthalten, zurückgegeben werden sollen. Obiges Beispiel bedeutet also, dass Content Items, die im Text “Value” enthalten, gesucht werden.

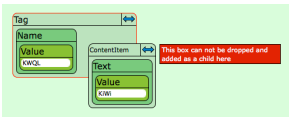
Ein Ressourceterm kann mehrere Qualifierterme enthalten wie auf dem unteren Beispiel auf der vorigen Seite zu sehen ist.
Ressourceterme können ineinander verschachtelt werden, um auszudrücken, dass eine bestimmte Ressource in einer anderen enthalten ist, zum Beispiel ein Fragment in einem Content Item oder ein Link in einem Fragment.



zum Beispiel beschreibt Content Items, die im Text “KiWi” enthalten und die außerdem getaggt sind mit einem Tag, dessen Name “KWQL” ist. Die unterschiedlichen Arten von Ressourcen können jeweils verschiedene **Subressourcen** enthalten, der untere Teil der folgenden Tabelle gibt eine Übersicht darüber.

Ressource	Content Item	Fragment	Link	Tag
Qualifier	URI author created lastEdited title text numberEdits descendant child	URI author created descendant child	target anchorText	URI author created name
Subresourcen	fragment link tag	link tag	tag	

Wie auch bei den Qualifier-Keywortschachtelungen verhindert visKWQL syntaktisch inkorrekte Schachtelungen von Subressourcen, wie im unteren Bild zu sehen ist.

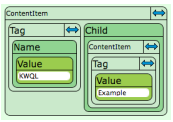


Da Content Items mit einem Tag versehen sein (“das Tag enthalten”) können, andersrum Tags aber keine Content Items enthalten, ist diese Operation nicht erlaubt.

Qualifier
In der obigen Tabelle sind die verschiedenen Ressourcen mit ihren jeweiligen Qualifiern aufgeführt.
URIs sind eindeutige Bezeichner für Ressourcen. So haben alle Content Items, Fragments und Tags unterschiedliche URIs und sind so eindeutig unterscheidbar und ansprechbar.

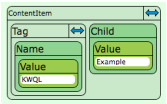
author bezeichnet die Personen, die ein Content Item angelegt oder editiert haben, ein Fragment angelegt oder eine Ressource mit einem Tag versehen haben.
created und **lastEdited** beziehen sich auf die Zeitpunkte, zu dem die entsprechende Ressource angelegt oder zuletzt editiert wurde.
title bezeichnet den Titel eines Content Items, **text** seinen textuellen Inhalt.
numberEdits ist die Anzahl der Editiervorgänge an einem Content Item.
name bezieht sich auf den Namen eines Tags.
Der **anchorText** eines Links ist der Text, auf den man klicken muss, um dem Link zu folgen.

child, **descendant** und **target** haben eine besondere Rolle, sie beziehen sich auf die Wikistruktur und nehmen als Wert kein Keyword oder ein Konstrukt aus Keywords und Operatoren, sondern eine Unteranfrage, die das verbundene Content Item oder Fragment beschreibt.
child und **descendant** beziehen sich auf die Verschachtelungen von Content Items und Fragmenten, und werden benutzt, um anzugeben, welche anderen Content Items oder Fragmente direkt (`child`) oder über mehrere Schritte (`descendant`) im beschriebenen Content Item enthalten sein sollen.

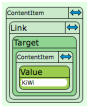


Diese Anfrage gibt Content Items zurück, die ein Tag “KWQL” haben und außerdem ein eingebettetes Content Item enthalten, das im Tag das Wort “Example” enthält. Der Wert der Child Box ist hier

Wenn der Term “Example” an beliebiger Stelle im enthaltenen Content Item vorkommen darf, lautet die Anfrage:



target bezeichnet das Ziel eines Links, also ein verlinktes Content Item.



Der Wert der Target Box ist hier

die Anfrage wählt also Content Items aus, die auf andere Content Items verlinkt sind, die wiederum in einem beliebigen Qualifier “KiWi” enthalten.

Unterspezifizierte Anfragen
Wie am Anfang beschrieben, kann der Kontext, in dem ein Keyword vorkommen soll, nur teilweise oder gar nicht spezifiziert werden. Das bedeutet, dass in visKWQL Anfragen alles außer den Keywords optional ist. Wenn die Ressource oder der Qualifier weggelassen werden, werden automatisch alle mit der unvollständigen Anfrage kompatiblen vollständigen Anfragen ausgewertet und die Vereinigung der Ergebnisse zurückgegeben.

In visKWQL Begrifflichkeiten ausgedrückt, bedeutet diese Anfrage vom Anfang des Texts also, dass das Keyword “KWQL” in einer beliebigen Ressource (innerhalb eines Content Items, jedoch nicht in seinen verlinkten oder eingebetteten Content Items) und in einem beliebigen Qualifier vorkommen muss.

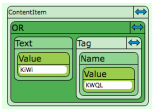
Diese Anfrage besagt, dass “KWQL” innerhalb eines Content Items (aber nicht in einer Subressource) im Wert von einem beliebigen Qualifier vorkommen muss.

Diese Anfrage ist doppelt unterspezifiziert: Zum einen ist kein Qualifier angegeben, “KWQL” kann also an beliebiger Stelle innerhalb eines Tags vorkommen. Zum anderen ist keine enthaltende Ressource für das Tag angegeben. Die Anfrage gibt also sowohl Content Items zurück, die ein Tag haben, in dem in einem beliebigen Qualifier “KWQL” vorkommt, als auch Content Items, die ein Fragment oder einen Link enthalten, der diese Bedingung erfüllt.

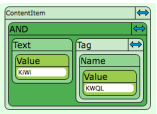
Hier ist ebenfalls die enthaltende Ressource nicht angegeben. Da sowohl Content Items, als auch Tags und Fragmente Autoren haben, ist die Anfrage ambig. Zurückgegeben werden hier Content Items, bei denen Mary Autor ist oder die eine Subresource enthalten, von denen Mary Autor ist.

Operatoren
Die Operatoren

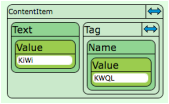
können benutzt werden, um Konjunktion (Und-Verknüpfung), (nicht ausschliessende bzw. logische) Disjunktion (Oder-Verknüpfung) und Verneinung auszudrücken.

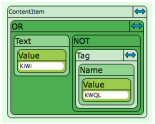


besagt, dass Content Items, die im Text “KiWi” enthalten oder mit einem Tag namens “KWQL” getaggt sind, zurückgegeben werden sollen.
Wenn man die “OR” Box durch eine “AND” Box ersetzt, erhält man die Anfrage.



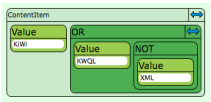
Diese wählt nur Content Items aus, die beide Bedingungen erfüllen, die also im Text “KiWi” enthalten und ausserdem mit einem Tag mit Namen “KWQL” getaggt sind.
Die “Und”-Verknüpfung, also die “AND” Box muss nicht unbedingt hinzugefügt werden, da visKWQL sie automatisch annimmt, wenn kein anderer Operator (d.h. “OR”) angegeben ist. Obige Anfrage ist also äquivalent zu



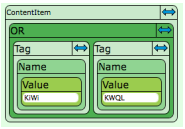


Diese Anfrage wiederum wählt Content Items aus, die “KiWi” im Text enthalten oder nicht mit “KWQL” getaggt sind.
Operatoren können auf der Ebene von Ressourcetermen, Qualifiertermen und Werten benutzt werden.

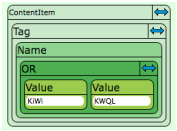
Einige Beispiele für die Benutzung von Operatoren:



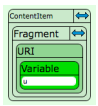
Hier werden Content Items ausgewählt, die “KiWi” enthalten und außerdem “KWQL” enthalten oder “XML” nicht enthalten.



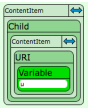
Diese Anfrage gibt Content Items zurück, die mit “KiWi” oder mit “KWQL” getaggt sind.
Letztere Anfrage ist äquivalent zu



Variablen
Wenn ein Qualifier angegeben ist, können anstelle von Keywords auch Variablen (zu finden unter dem Menüpunkt “Other”) angegeben werden. Variablen in KWQL dienen verschiedenen Zwecken:
1. Als Wildcard, das heißt, um anzugeben, dass etwas vorhanden sein muss, sein Wert an sich aber nicht von Belang ist.

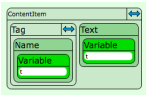


zum Beispiel gibt Content Items zurück, die mindestens ein Fragment enthalten, ohne dabei weitere Selektionskriterien auf das Fragment anzuwenden.



gibt Content Items zurück, in denen mindestens ein weiteres Content Item eingebettet ist, unabhängig davon, was das enthaltene Content Item für Eigenschaften hat.

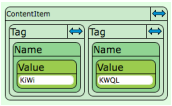
2. Um die Gleichheit von mindestens einem Element von zwei Qualifierwerten zu erzwingen.



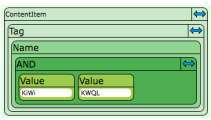
zum Beispiel erfordert, dass mindestens ein Wort im Text des Content Items auch ein Tag dieses Content Items sein muss.

Injektivität

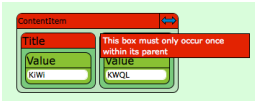
Wie bereits erwähnt, funktioniert KWQL als Anfragesprache, die aufgrund einer unvollständigen Beschreibung von Content Items alle Content Items, die mit den Informationen in der gegebenen Anfrage kompatibel sind, auswählt.
Injektivität bedeutet dabei, dass jeder Qualifier und jede Subresource, für die in der Anfrage mehrfach Selektionskriterien gegeben werden, sich auf unterschiedliche Elemente in den Daten beziehen. So bedeutet zum Beispiel



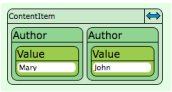
dass die auszuwählenden Content Items jeweils **mindestens zwei** Tags haben müssen, eines mit Namen “KiWi” und eine mit Namen “KWQL”. Die Anfrage ist nicht äquivalent zu



welche Content Items auswählt, die Tags haben, die beide Kriterien erfüllen.
Prinzipiell können alle Subresourcen mehrfach in einer Anfrage vorkommen, zum Beispiel kann ein Fragment mehrere Tags haben oder ein Content Item mehrere Links enthalten.
Bei Qualifiern sieht dies anders aus, viele Qualifier bezeichnen Eigenschaften, die jede Ressource nur einmal hat, zum Beispiel `title` oder `text`, und dürfen auch nur einmal vorkommen.



zum Beispiel ist **keine** gültige Anfrage (wie auch an den roten Labeln sowie an dem Hinweis zu sehen ist).
Einige Qualifier können aber auch mehrfach vorkommen:



Diese Anfrage wählt Content Items aus, die sowohl von Mary als auch von John editiert worden sind.
Die Qualifier `author`, `descendant`, `child` und `name` können in einer Anfrage mehrfach angegeben werden.

BIBLIOGRAPHY

[1] *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*. IEEE Computer Society, 2002.

[2] *plist — Property List Format*. Apple Inc., 2003.

[3] iTQL commands. Online only, 2004. http://www.kowari.org/271.htm.

[4] *Proceedings of the Extreme Markup Languages 2004 Conference, 2-6 August 2004, Montréal, Quebec, Canada*. 2004.

[5] *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. IEEE, 2009.

[6] D. J. Abadi, A. M. 0002, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In [225], pages 411–422.

[7] F. Abbaci, J.-B. Valsamis, and P. Francq. Index and search XML documents by combining content and structure. In H. R. Arabnia, editor, *International Conference on Internet Computing*, pages 107–112. CSREA Press, 2006.

[8] K. Aberer, K.-S. Choi, N. F. Noy, D. Allemang, K.-I. Lee, L. J. B. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, and P. Cudré-Mauroux, editors. *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*. Springer, 2007.

[9] K. Aberer, P. Cudré-Mauroux, A. M. Ouksel, T. Catarci, M.-S. Hacid, A. Illarramendi, V. Kashyap, M. Mecella, E. Mena, E. J. Neuhold, O. D. Troyer, T. Risse, M. Scannapieco, F. Saltor, L. D. Santis, S. Spaccapietra, S. Staab, and R. Studer. Emergent semantics principles and issues. In Y.-J. Lee, J. Li, K.-Y. Whang, and D. Lee, editors, *DASFAA*, volume 2973 of *Lecture Notes in Computer Science*, pages 25–38. Springer, 2004.

[10] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[11] B. Adida, M. Birbeck, and S. Pemberton. HTML+RDFa 1.1. Working draft, W3C, 2010.

[12] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *SIGMOD Conference*, pages 253–262. ACM Press, 1989.

[13] W. Akhtar, J. Kopecký, T. Krennwallner, and A. Polleres. XSPARQL: Traveling between the XML and RDF worlds - and avoiding the XSLT pilgrimage. In [41], pages 432–447.

[14] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In [1], pages 141–.

[15] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, M. Holstege, J. Melton, M. Rys, and J. Shanmugasundaram. XQuery and XPath Full Text 1.0. Candidate recommendation, W3C, 2008.

[16] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. Texquery: a full-text search extension to XQuery. In [147], pages 583–594.

[17] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In [204], pages 496–513.

[18] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient XML search with complex full-text predicates. In [99], pages 575–586.

[19] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible structure and full-text querying for XML. In [364], pages 83–94.

[20] S. Amer-Yahia and J. Shanmugasundaram. XML full-text search: Challenges and opportunities. In [62], page 1368.

[21] M. Ames and M. Naaman. Why we tag: motivations for annotation in mobile and online media. In M. B. Rosson and D. J. Gilmore, editors, *CHI*, pages 971–980. ACM, 2007.

[22] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A graphical query language with recursion. *IEEE Trans. Software Eng.*, 16(10):1150–1163, 1990.

[23] R. Angles and C. Gutiérrez. Querying RDF data from a graph database perspective. In A. Gómez-Pérez and J. Euzenat, editors, *ESWC*, volume 3532 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2005.

[24] V. N. Anh and A. Moffat. Compression and an IR approach to XML retrieval. In N. Fuhr, N. Gövert, G. Kazai, and M. Lalmas, editors, *INEX Workshop*, pages 99–104. 2002.

[25] A. Ankolekar, M. Krötzsch, T. Tran, and D. Vrandecic. The two cultures: mashing up web 2.0 and the semantic web. In [367], pages 825–834.

[26] D. Artz and Y. Gil. A survey of trust in computer science and the semantic web. *J. Web Sem.*, 5(2):58–71, 2007.

[27] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A nucleus for a web of open data. In [8], pages 722–735.

[28] S. Auer, S. Dietzold, J. Lehmann, and T. Riechert. OntoWiki: A tool for social, semantic collaboration. In N. F. Noy, H. Alani, G. Stumme, P. Mika, Y. Sure, and D. Vrandecic, editors, *CKC*, volume 273 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[29] E. Augurusa, D. Braga, A. Campi, and S. Ceri. Design and implementation of a graphical interface to XQuery. In *SAC*, pages 1163–1167. ACM, 2003.

[30] D. Aumueller. Semantic authoring and retrieval within a wiki. 2nd European Semantic Web Conference 2005 (ESWC2005), 2005.

[31] D. Aumueller. SHAWN: Structure helps a wiki navigate. In *Proceedings of the BTW-Workshop WebDB Meets IR*. 2005.

[32] D. Aumueller. Towards a semantic wiki experience - desktop integration and interactivity in WikSAR. In *Proceedings of the ISWC 2005 Workshop on The Semantic Desktop.* 2005.

[33] D. Aumueller and S. Auer. Towards a semantic wiki experience – desktop integration and interactivity in WikSAR. In *1st Workshop on The Semantic Desktop*. 2005.

[34] D. Backett. Turtle—terse RDF triple language. Technical Report, Institute for Learning and Research Technology, University of Bristol, 2007.

[35] R. A. Baeza-Yates and C. Castillo. Relating web structure, user search behavior. In *WWW Posters*. 2001.

[36] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and semantic web query languages: A survey. In N. Eisinger and J. Maluszynski, editors, *Reasoning Web*, volume 3564 of *Lecture Notes in Computer Science*, pages 35–133. Springer, 2005.

[37] A. Balmin, V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, and T. Wang. A system for keyword proximity search on XML databases. In *VLDB*, pages 1069–1072. 2003.

[38] J. Bao, L. Ding., and J. Hendler. Knowledge representation and query in semantic MediaWiki: A formal study. Technical Report, RPI, 2008.

[39] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective XML keyword search with relevance oriented ranking. In [5], pages 517–528.

[40] J. Bar-Ilan, S. Shoham, A. Idan, Y. Miller, and A. Shachak. Structured versus unstructured tagging: a case study. *Online Information Review*, 32(5):635–647, 2008.

[41] S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors. *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, volume 5021 of *Lecture Notes in Computer Science*. Springer, 2008.

[42] D. Beckett and B. McBride. *RDF/XML Syntax Specification (Revised)*. W3C, 2004.

[43] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. A. Grossman, and O. Frieder. Hourly analysis of a very large topically categorized web query log. In M. Sanderson, K. Järvelin, J. Allan, and P. Bruza, editors, *SIGIR*, pages 321–328. ACM, 2004.

[44] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In D. Calvanese, M. Lenzerini, and R. Motwani, editors, *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 2003.

[45] M. Benedikt and C. Koch. Interpreting tree-to-tree queries. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 552–564. Springer, 2006.

[46] M. Benedikt and C. Koch. XPath leashed. *ACM Comput. Surv.*, 41(1), 2008.

[47] V. Benjamins, J. Contreras, O. Corcho, and A. Gomez-Perez. Six challenges for the semantic web. *AIS SIGSEMIS Bulletin*, 1(1):24–25, 2004.

[48] D. Benz, M. Grobelnik, A. Hotho, R. Jaschke, D. Mladenic, V. D. P. Servedio, S. Sizov, and M. Szomszor. Analyzing tag

semantics across collaborative tagging systems. *Dagstuhl Seminar 08391 ? Working Group Summary*, 2008.

[49] S. Berger, F. Bry, O. Bolzer, T. Furche, S. Schaffert, and C. Wieser. Querying the standard and semantic web using Xcerpt and visXcerpt. In *Proceedings of European Semantic Web Conference, Heraklion, Crete, Greece (29th May–1st June 2005)*. 2005.

[50] S. Berger, F. Bry, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: From pattern-based to visual querying of XML and semistructured data. In *VLDB*, pages 1053–1056. 2003.

[51] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon. *XML Path Language (XPath) 2.0*. W3C, 2005.

[52] P. Berkhin. Survey: A survey on PageRank computing. *Internet Mathematics*, 2(1), 2005.

[53] T. Berners-Lee. Linked Data. `http://www.w3.org/DesignIssues/LinkedData`, 2006.

[54] T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the 3rd International Semantic Web User Interaction Workshop*, volume 2006. 2006.

[55] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[56] E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, and E. Ferrari, editors. *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*. Springer, 2004.

[57] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In [1], pages 431–440.

[58] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[59] K. Bischoff, C. S. Firan, W. Nejdl, and R. Paiu. Can all tags be used for search? In J. G. Shanahan, S. Amer-Yahia, I. Manolescu, Y. Zhang, D. A. Evans, A. Kolcz, K.-S. Choi, and A. Chowdhury, editors, *CIKM*, pages 193–202. ACM, 2008.

[60] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data – the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[61] A. Blumauer and T. Pellegrini, editors. *Social Semantic Web: Web 2.0 - Was nun?* X.media.press. Springer, 2009.

[62] K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, editors. *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 2005.

[63] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and XML reasoning. In S. Vansummeren, editor, *PODS*, pages 10–19. ACM, 2006.

[64] H. Boley, G. Hallmark, M. Kifer, A. Paschke, A. Polleres, and D. Reynolds. RIF core dialect. Recommendation, W3C, 2010.

[65] O. Bolzer. *Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt*. Diplomarbeit/master thesis, University of Munich, 2005.

[66] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In [99], pages 479–490.

[67] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *ICDE*, pages 403–. 2002.

[68] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML (2nd edition). Recommendation, W3C, 2006.

[69] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (third edition). Recommendation, W3C, 2004.

[70] D. Brickley and R. Guha. RDF vocabulary description language 1.0: RDF Schema. Recommendation, W3C, 2004.

[71] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[72] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In [196], pages 54–68.

[73] M. Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.

[74] E. Bruno, J. L. Maitre, and E. Murisasco. Extending XQuery with transformation operators. In *ACM Symposium on Document Engineering*, pages 1–8. ACM, 2003.

[75] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In [153], pages 310–321.

[76] F. Bry, P. Dolog, F. A. Durão, and K. Weiand. PESTP: Fast approximate personalized keyword search in structured data using eigenvector-based term propagation. Manuscript, 2010.

[77] F. Bry, T. Furche, C. Ley, B. Linse, and B. Marnette. RDFLog: It's like Datalog for RDF. In *Proc. Workshop on (Constraint) Logic Programming (WLP)*. 2008.

[78] F. Bry, T. Furche, C. Ley, B. Linse, and B. Marnette. Taming existence in RDF querying. In D. Calvanese and G. Lausen, editors, *RR*, volume 5341 of *Lecture Notes in Computer Science*, pages 236–237. Springer, 2008.

[79] F. Bry, T. Furche, B. Linse, and A. Pohl. XcerptRDF: A pattern-based answer to the versatile web challenge. In *Proceedings of 22nd Workshop on (Constraint) Logic Programming, Dresden, Germany (30th September–1st October 2008)*, pages 27–36. 2008.

[80] F. Bry, C. Koch, T. Furche, S. Schaffert, L. Badea, and S. Berger. Querying the web reconsidered: Design principles for versatile web query languages. *Int. J. Semantic Web Inf. Syst.*, 1(2):1–21, 2005.

[81] F. Bry and J. Kotowski. Towards reasoning and explanations for social tagging. In T. Roth-Berghofer, S. Schulz, D. B. Leake, and D. Bahls, editors, *ExaCt*, pages 118–128. 2008.

[82] F. Bry and M. Marchiori. Reasoning on the semantic web: Beyond ontology languages and reasoners. In *2nd European Workshop on the Integration of Knowledge, Semantic and Digital Media Technologies*. 2005.

[83] F. Bry and M. Marchiori. Ten theses on logic languages for the semantic web. In *Rule Languages for Interoperability*. W3C, 2005.

[84] F. Bry and S. Schaffert. A gentle introduction into Xcerpt, a rule-based query and transformation language for XML. In *Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*. 2002.

[85] D. Carmel, Y. Maarek, Y. Mass, N. Efraty, and G. Landau. An extension of the vector space model for querying XML

documents via XML fragments. In *Proceedings SIGIR 2002 Workshop on XML and Information Retrieval*, pages 14–25. 2002.

[86] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *J. Vis. Lang. Comput.*, 8(2):215–260, 1997.

[87] T. Catarci and G. Santucci. Are visual query languages easier to use than traditional ones? an experimental proof. In M. A. R. Kirby, A. J. Dix, and J. Finlay, editors, *BCS HCI*, pages 323–338. Cambridge University Press, 1995.

[88] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[89] C. Cattuto, D. Benz, A. Hotho, and G. Stumme. Semantic analysis of tag similarity measures in collaborative tagging systems. In J. Baumeister and M. Atzmüller, editors, *LWA*, volume 448 of *Technical Report*, pages 18–26. Department of Computer Science, University of Würzburg, Germany, 2008.

[90] C. Cattuto, D. Benz, A. Hotho, and G. Stumme. Semantic grounding of tag relatedness in social bookmarking systems. In [332], pages 615–631.

[91] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A graphical language for querying and restructuring XML documents. *Computer Networks*, 31(11-16):1171–1187, 1999.

[92] S. Chakrabarti. Dynamic personalized PageRank in entity-relation graphs. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 571–580. ACM, New York, NY, USA, 2007.

[93] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. *XML Query Use Cases*. W3C, 2005.

[94] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. *XML Query (XQuery) Requirements*. W3C, 2003.

[95] D. Chamberlin and J. Robie. XQuery update facility requirements. Working draft, W3C, 2005.

[96] D. Chamberlin and J. Robie. XQuery 1.1. Working draft, W3C, 2008.

[97] D. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In

D. Suciu and G. Vossen, editors, *WebDB (Selected Papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.

[98] C. Y. Chan, B. C. Ooi, and A. Zhou, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. ACM, 2007.

[99] S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 2006.

[100] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In [201], pages 493–504.

[101] T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In [296], pages 455–466.

[102] Y. Chen and K. Aberer. Combining pat-trees and signature files for query evaluation in document databases. In T. J. M. Bench-Capon, G. Soda, and A. M. Tjoa, editors, *DEXA*, volume 1677 of *Lecture Notes in Computer Science*, pages 473–484. Springer, 1999.

[103] T. Cheng, X. Yan, and K. C.-C. Chang. EntityRank: searching entities directly and holistically. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 387–398. VLDB Endowment, 2007.

[104] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In [201], pages 413–422.

[105] J. Clark. *XSL Transformations (XSLT) Version 1.0*. W3C, 1999.

[106] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946. 2002.

[107] S. Cohen, Y. Kanza, B. Kimelfeld, and Y. Sagiv. Interconnection semantics for keyword search in XML. In O. Herzog, H.-J. Schek, N. Fuhr, A. Chowdhury, and W. Teiken, editors, *CIKM*, pages 389–396. ACM, 2005.

[108] S. Cohen, Y. Kanza, Y. A. Kogan, Y. Sagiv, W. Nutt, and A. Serebrenik. EquiX – a search and query language for XML. *JASIST*, 53(6):454–466, 2002.

[109] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSearch: A semantic search engine for XML. In *VLDB*, pages 45–56. 2003.

[110] A. Collins and E. Loftus. A spreading-activation theory of semantic processing. *Psychological Review*, 82(6), 1975.

[111] S. Comai. Graphical query languages for semi-structured information. In *EDBT PhD Workshop*. 2000.

[112] S. Comai, E. Damiani, and P. Fraternali. Computing graphical queries over XML data. *ACM Trans. Inf. Syst.*, 19(4):371–430, 2001.

[113] S. Comai, E. Damiani, and L. Tanca. Semantics-aware querying in the www: The WG-Log web query system. In *ICMCS, Vol. 2*, pages 317–322. 1999.

[114] S. Comai, S. Marrara, and L. Tanca. XML document summarization: Using XQuery for synopsis creation. In *DEXA Workshops*, pages 928–932. IEEE Computer Society, 2004.

[115] D. Connolly. Gleaning resource descriptions from dialects of languages (GRDDL). Recommendation, W3C, 2007.

[116] M. P. Consens, R. A. Baeza-Yates, M. Lalmas, and S. Amer-Yahia. XML retrieval: DB/IR in theory, web in practice. In [225], pages 1437–1438.

[117] J. Cowan and R. Tobin. XML information set (2nd ed.). Recommendation, W3C, 2004.

[118] A. Cregan. Symbol grounding for the semantic web. In E. Franconi, M. Kifer, and W. May, editors, *ESWC*, volume 4519 of *Lecture Notes in Computer Science*, pages 429–442. Springer, 2007.

[119] F. Crestani. Application of spreading activation techniques in information retrieval. *Artif. Intell. Rev.*, 11(6):453–482, 1997.

[120] B. Croft and J. Yufeng. An association thesaurus for information retrieval. In J.-L. Funck-Brentano and F. Seitz, editors, *RIAO*, pages 146–161. CID, 1994.

[121] I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors. *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*. Springer, 2006.

[122] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In U. Dayal and I. L. Traiger, editors, *SIGMOD Conference*, pages 323–330. ACM Press, 1987.

[123] H. Cui, J.-R. Wen, J.-Y. Nie, and W.-Y. Ma. Probabilistic query expansion using query logs. In *WWW*, pages 325–332. 2002.

[124] C. V. Damme, M. Hepp, and K. Siorpaes. FolksOntology: An integrated approach for turning folksonomies into ontologies. In *In Proceedings of the ESWC Workshop "Bridging the Gap between Semantic Web and Web 2.0*. Springer, 2007.

[125] S. Dar, G. Entin, S. Geva, and E. Palmon. DTL's DataSpot: Database exploration using plain language. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB*, pages 645–649. Morgan Kaufmann, 1998.

[126] J. Davies and R. Weeks. QuizRDF: Search technology for the semantic web. In *HICSS*. 2004.

[127] R. Davis, H. E. Shrobe, and P. Szolovits. What is a knowledge representation? *AI Magazine*, 14(1):17–33, 1993.

[128] U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors. *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. IEEE Computer Society, 2003.

[129] S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: Ontology based access to distributed and semi-structured information. In R. Meersman, Z. Tari, and S. M. Stevens, editors, *DS-8*, volume 138 of *IFIP Conference Proceedings*, pages 351–369. Kluwer, 1999.

[130] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[131] D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In [182], pages 623–634.

[132] K. Dello, E. P. B. Simperl, and R. Tolksdorf. Creating and using semantic web information with Makna. In [352].

[133] S. DeRose, E. Maler, and R. D. Jr. *XML Pointer Language (XPointer) 1.0*. W3C, 2001.

[134] A. Deutsch, editor. *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*. ACM, 2004.

[135] A. Deutsch, M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. XML-ql. In *QL*. 1998.

[136] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath. In M. Lenzerini, D. Nardi, W. Nutt, and D. Suciu, editors, *KRDB*, volume 45 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[137] P. F. Dietz. Maintaining order in a linked list. In *STOC*, pages 122–127. ACM, 1982.

[138] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics. Recommendation, W3C, 2007.

[139] M. Ehrig. *Ontology Alignment: Bridging the Semantic Gap*, volume 4 of *Semantic Web And Beyond Computing for Human Experience*. Springer, 2007.

[140] A. Eisenberg and J. Melton. An early look at XQuery API for Java (XQJ). *SIGMOD Record*, 33(2):105–111, 2004.

[141] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artif. Intell.*, 172(12-13):1495–1539, 2008.

[142] A. El Ghali, A. Tifous, M. Buffa, A. Giboin, and R. Dieng-Kuntz. Using a semantic wiki in communities of practice. In *2nd Intern. Workshop on Building Technology Enhanced Learning Solutions for Communities of Practice*. 2007.

[143] S. Elbassuoni, M. Ramanath, R. Schenkel, M. Sydow, and G. Weikum. Language-model-based ranking for queries on RDF-graphs. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM 2009)*. ACM, Hongkong, China, 2009.

[144] S. Elbassuoni, M. Ramanath, R. Schenkel, and G. Weikum. Searching RDF graphs with SPARQL and keywords. *IEEE Data Enginerring Bulletin*, 33(1), 2010.

[145] M. Erwig. Xing: a visual XML query language. *J. Vis. Lang. Comput.*, 14(1):5–45, 2003.

[146] P. Fankhauser, T. Groh, and S. Overhage. XQuery by the book: The IPSI XQuery demonstrator. In [204], pages 742–744.

[147] S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors. *Proceedings of the 13th international conference on*

World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004. ACM, 2004.

[148] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model. Recommendation, W3C, 2007.

[149] M. F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *VLDB*, pages 1077–1080. 2003.

[150] J. Fischer, Z. Gantner, S. Rendle, M. Stritt, and L. Schmidt-Thieme. Ideas and improvements for semantic wikis. In Y. Sure and J. Domingue, editors, *ESWC*, volume 4011 of *Lecture Notes in Computer Science*, pages 650–663. Springer, 2006.

[151] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan. The bea streaming xquery processor. *VLDB J.*, 13(3):294–315, 2004.

[152] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. *Computer Networks*, 33(1-6):119–135, 2000.

[153] M. J. Franklin, B. Moon, and A. Ailamaki, editors. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*. ACM, 2002.

[154] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB*, pages 273–284. Morgan Kaufmann, 1994.

[155] N. E. Fuchs, K. Kaljurand, and G. Schneider. Attempto Controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces. In G. Sutcliffe and R. Goebel, editors, *FLAIRS Conference*, pages 664–669. AAAI Press, 2006.

[156] T. Furche. *Implementation of Web Query Language Reconsidered: Beyond Tree and Single-Language Algebras at (Almost) No Cost*. Ph.D. thesis, Ludwig-Maxmilians University Munich, 2008.

[157] T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF querying: Language constructs and evaluation methods compared. In P. Barahona, F. Bry, E. Franconi, N. Henze, and U. Sattler, editors, *Reasoning Web*, volume 4126 of *Lecture Notes in Computer Science*, pages 1–52. Springer, 2006.

[158] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.

[159] P. Gärdenfors. How to make the semantic web more semantic. In *Proceedings of the Third International Conference on Formal Ontology in Information Systems (FOIS 2004)*. 2004.

[160] P. Genevès and J.-Y. Vion-Dury. XPath formal semantics and beyond: A Coq-based approach. In *Proc. Int'l. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 181–198. University Of Utah, Salt Lake City, Utah, United States, 2004.

[161] S. A. Golder and B. A. Huberman. The structure of collaborative tagging systems. *CoRR*, abs/cs/0508082, 2005.

[162] S. A. Golder and B. A. Huberman. Usage patterns of collaborative tagging systems. *J. Information Science*, 32(2):198–208, 2006.

[163] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB*, pages 436–445. Morgan Kaufmann, 1997.

[164] G. Gottlob and C. Koch. Monadic queries over tree-structured data. In *LICS*, pages 189–202. IEEE Computer Society, 2002.

[165] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *PODS*, pages 179–190. ACM, 2003.

[166] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency. In [128], pages 379–390.

[167] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[168] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[169] J. Graaumans. *Usability of XML Query Languages*. Ph.D. thesis, Universiteit Utrecht, 2005.

[170] O. M. Group. Unified Modeling Language (UML), version 2.2. 2009.

[171] W. O. W. Group. OWL 2 web ontology language. W3c recommendation, W3C, 2009.

[172] T. Gruber. Collective knowledge systems: Where the social web meets the semantic web. *J. Web Sem.*, 6(1):4–13, 2008.

[173] T. Gruber and T. Gruber. Ontology of folksonomy: A mash-up of apples and oranges. *Intern. Journal on Semantic Web and Information Systems*, 3(1):1–11, 2007.

[174] T. Grust. Accelerating XPath location steps. In [153], pages 109–120.

[175] T. Grust, J. Rittinger, and J. Teubner. eXrQuy: Order indifference in XQuery. In *ICDE*, pages 226–235. IEEE, 2007.

[176] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL hosts. In [281], pages 252–263.

[177] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29:91–131, 2004.

[178] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In [153], pages 287–298.

[179] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In [182], pages 16–27.

[180] C. Gutiérrez, C. A. Hurtado, and A. O. Mendelzon. Foundations of semantic web databases. In [134], pages 95–106.

[181] P. Haase, D. Herzig, M. A. Musen, and T. Tran. Semantic wiki search. In L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvönen, R. Mizoguchi, E. Oren, M. Sabou, and E. P. B. Simperl, editors, *ESWC*, volume 5554 of *Lecture Notes in Computer Science*, pages 445–460. Springer, 2009.

[182] A. Y. Halevy, Z. G. Ives, and A. Doan, editors. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 2003.

[183] H. Halpin. The semantic web: The origins of artificial intelligence redux. In *Third Intern. Workshop on the History and Philosophy of Logic, Mathematics, and Computation*. Donostia San Sebastian, Spain, 2004.

[184] H. Halpin, V. Robu, and H. Shepherd. The complex dynamics of collaborative tagging. In [367], pages 211–220.

[185] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[186] S. Harnad. The symbol grounding problem. 1990.

[187] E. Hatcher and O. Gospodnetic. *Lucene in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[188] T. Haveliwala, S. Kamvar, and G. Jeh. An analytical comparison of approaches to personalizing PageRank. Technical Report 2003-35, Stanford InfoLab, 2003.

[189] P. Hayes and B. McBride. RDF semantics. Recommendation, W3C, 2004.

[190] M. Heckner, T. Neubauer, and C. Wolff. Tree, funny, to_read, google: what are tags supposed to achieve? a comparative analysis of user keywords for different digital resource types. In I. Soboroff, E. Agichtein, and R. Kumar, editors, *SSM*, pages 3–10. ACM, 2008.

[191] J. Hendler. What is the semantic web really all about? http://blogs.nature.com/jhendler/2009/06/16/what-is-the-semantic-web-really-all-about, 2009.

[192] M. Hepp. Possible ontologies: How reality constrains the development of relevant ontologies. *IEEE Internet Computing*, 11(1):90–96, 2007.

[193] P. Heymann and H. Garcia-Molina. Collaborative creation of communal hierarchical taxonomies in social tagging systems. Technical Report, Stanford University, 2006.

[194] J. Hidders. Satisfiability of XPath expressions. In G. Lausen and D. Suciu, editors, *DBPL*, volume 2921 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2003.

[195] F. Hogenboom, V. Milea, F. Frasincar, and U. Kaymak. RDF-GL: A SPARQL-based graphical query language for RDF. In L. Jain, X. Wu, R. Chbeir, Y. Badr, A. Abraham, and A.-E. Hassanien, editors, *Emergent Web Intelligence: Advanced Information Retrieval*, Advanced Information and Knowledge Processing, pages 87–116. Springer London, 2010.

[196] I. Horrocks and J. A. Hendler, editors. *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*. Springer, 2002.

[197] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in XML trees. *IEEE Trans. Knowl. Data Eng.*, 18(4):525–539, 2006.

[198] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In [128], pages 367–378.

[199] J. Huai, R. Chen, H.-W. Hon, Y. Liu, W.-Y. Ma, A. Tomkins, and X. Zhang, editors. *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*. ACM, 2008.

[200] H. Hwang, V. Hristidis, and Y. Papakonstantinou. ObjectRank: a system for authority-based search on databases. In [99], pages 796–798.

[201] H. V. Jagadish and I. S. Mumick, editors. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. ACM Press, 1996.

[202] B. J. Jansen, A. Spink, and T. Saracevic. Real life, real users, and real needs: a study and analysis of user queries on the web. *Inf. Process. Manage.*, 36(2):207–227, 2000.

[203] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279. 2003.

[204] C. S. Jensen, K. G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, and M. Jarke, editors. *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, volume 2287 of *Lecture Notes in Computer Science*. Springer, 2002.

[205] K. Jones et al.. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[206] K. S. Jones and D. M. Jackson. The use of automatically-obtained keyword classifications for information retrieval. *Information Storage and Retrieval*, 5(4):175–201, 1970.

[207] N. Jussien, C. Prud'homme, H. Cambazard, G. Rochart, and F. Laburthe. choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming*. 2008.

[208] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In [62], pages 505–516.

[209] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Exploiting the block structure of the web for computing PageRank. In *Stanford University Technical Report*. 2003.

[210] G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. RQL: A functional query language for RDF. In P. Gray, P. King, and A. Poulovassilis, editors, *The Functional Approach to Data Management*, chapter 18, pages 435–465. Springer-Verlag, 2004.

[211] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 1st edition, 2003.

[212] E. Kaufmann and A. Bernstein. How useful are natural language interfaces to the semantic web for casual end-users? In [8], pages 281–294.

[213] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In [364], pages 779–790.

[214] M. Kay. *XPath 2.0 Programmer's Reference*. John Wiley, 2004.

[215] M. Kay. XSL Transformations, version 2.0. Recommendation, W3C, 2007.

[216] M. Kay, N. Walsh, H. Zongaro, S. Boag, and J. Tong. XSLT 2.0 and XQuery 1.0 serialization. Working draft, W3C, 2005.

[217] S. Kepser. A simple proof for the turing-completeness of XSLT and XQuery. In [4].

[218] M. Kiesel. Kaukolu: Hub of the semantic corporate intranet. In [352].

[219] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2):340–356, 1995.

[220] M. Kindborg and K. McGee. Visual programming with analogical representations: Inspirations from a semiotic analysis of comics. *J. Vis. Lang. Comput.*, 18(2):99–125, 2007.

[221] B. Klein, C. Höcht, and B. Decker. Beyond capturing and maintaining software engineering knowledge-"Wikitology" as shared semantics. In *Workshop on Knowledge Engineering and Software Engineering, KI*. 2005.

[222] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *SODA*, pages 668–677. 1998.

[223] G. Klyne, J. J. Carroll, and B. McBride. Resource Description Framework (RDF): Concepts and abstract syntax. Recommendation, W3C, 2004.

[224] C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. In C. Li, editor, *PODS*, pages 84–97. ACM, 2005.

[225] C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, editors. *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. ACM, 2007.

[226] L. Kong, R. Gilleron, and A. Lemay. Retrieving top Relaxed Tightest Fragments for XML keyword search. online, 2008.

[227] M. Kotelnikov, A. Polonsky, M. Kiesel, M. Völkel, and H. Haller. Interactive semantic wikis. Technical Report, Nepomuk, 2006.

[228] L. Kotthoff. Constraint solvers: An empirical evaluation of design decisions. *CoRR*, abs/1002.0134, 2010.

[229] T. Kowatsch and W. Maass. The impact of pre-defined terms on the vocabulary of collaborative indexing systems. In W. Golden, T. Acton, K. Conboy, H. van der Heijden, and V. Tuunainen, editors, *16th European Conference on Information Systems (ECIS 2008)*. Galway, Ireland, 2008.

[230] M. Krötzsch, S. Schaffert, and D. Vrandecic. Reasoning in semantic wikis. In G. Antoniou, U. Aßmann, C. Baroglio, S. Decker, N. Henze, P.-L. Patranjan, and R. Tolksdorf, editors, *Reasoning Web*, volume 4636 of *Lecture Notes in Computer Science*, pages 310–329. Springer, 2007.

[231] M. Krötzsch and D. Vrandecic. Semantic wikipedia. In [61], pages 393–421.

[232] M. Krötzsch, D. Vrandecic, and M. Völkel. Semantic MediaWiki. In [121], pages 935–942.

[233] J. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[234] T. Kuhn. AceWiki: A natural and expressive semantic wiki. *CoRR*, abs/0807.4618, 2008.

[235] G. Ladwig and T. Tran. Combining query translation with query answering for efficient keyword search. In L. Aroyo,

G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, editors, *ESWC (2)*, volume 6089 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2010.

[236] R. Landefeld and H. Sack. Collaborative web-publishing with a semantic wiki. In S. Auer, C. Bizer, C. Müller, and A. V. Zhdanova, editors, *CSSW*, volume 113 of *LNI*, pages 23–34. GI, 2007.

[237] O. Lassila. Semantic web soul searching. http://www.lassila.org/blog/archive/2007/03/semantic_web_so_1.html, 2007.

[238] K.-H. Lee, K.-Y. Whang, W.-S. Han, and M.-S. Kim. Structural consistency: Enabling XML keyword search to eliminate spurious results consistently. *CoRR*, abs/0911.4329, 2009.

[239] Y. Lei, V. Uren, and E. Motta. SemSearch: A search engine for the semantic web. *Proc. 5th Intern. Conf. on Knowledge Engineering and Knowledge Management Managing Knowledge in a World of Networks, Lect. Notes in Comp. Sci., Springer, Podebrady, Czech Republic (Oct 2006)*, pages 238–245, 2006.

[240] B. Leuf and W. Cunningham. *The Wiki way: quick collaboration on the Web*. Addison-Wesley, 2001.

[241] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics Doklady*, volume 10. 1966.

[242] G. Li, J. Feng, J. Wang, X. Song, and L. Zhou. SAILER: an effective search engine for unified retrieval of heterogeneous XML and web documents. In [199], pages 1061–1062.

[243] G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable LCAs over XML documents. In M. J. Silva, A. H. F. Laender, R. A. Baeza-Yates, D. L. McGuinness, B. Olstad, Ø. H. Olsen, and A. O. Falcão, editors, *CIKM*, pages 31–40. ACM, 2007.

[244] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 903–914. ACM, 2008.

[245] J. Li, C. Liu, and R. Zhou. XBridge: Answering XML keyword search with structured queries. 2008.

[246] J. Li, C. Liu, R. Zhou, and W. Wang. Suggestion of promising result types for XML keyword search. In [260], pages 561–572.

[247] J. Li and J. Wang. XQSuggest: An interactive XML keyword search system. In S. S. Bhowmick, J. Küng, and R. Wagner, editors, *DEXA*, volume 5690 of *Lecture Notes in Computer Science*, pages 340–347. Springer, 2009.

[248] Y. Li, I. Chaudhuri, H. Yang, S. Singh, and H. V. Jagadish. DaNaLIX: a domain-adaptive natural language interface for querying XML. In [98], pages 1165–1168.

[249] Y. Li, H. Yang, and H. V. Jagadish. Constructing a generic natural language interface for an XML database. In Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, editors, *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 737–754. Springer, 2006.

[250] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In [281], pages 72–83.

[251] J. Liu and D. M. Gruen. Between ontology and folksonomy: a study of collaborative and implicit ontology evolution. In J. M. Bradshaw, H. Lieberman, and S. Staab, editors, *IUI*, pages 361–364. ACM, 2008.

[252] Z. Liu and Y. Chen. Identifying meaningful return information for XML keyword search. In [98], pages 329–340.

[253] Z. Liu and Y. Chen. Return specification inference and result clustering for keyword search on XML. *ACM Trans. Database Syst.*, 35(2), 2010.

[254] Z. Liu, P. Sun, Y. Huang, Y. Cai, and Y. Chen. Challenges, techniques and directions in building XSeek: an XML search engine. *IEEE Data Eng. Bull.*, 32(2):36–43, 2009.

[255] Z. Liu, J. Walker, and Y. Chen. XSeek: A semantic XML search engine using keywords. In [225], pages 1330–1333.

[256] J. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31, 1968.

[257] J. Łukasiewicz. *O logice trójwartościowej (On three-valued logic)*, pages 169–170. Ruch Filozoficzny, 1920.

[258] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 functions and operators. Working draft, W3C, 2005.

[259] F. Manola, E. Miller, and B. McBride. RDF primer. Recommendation, W3C, 2004.

[260] I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, editors. *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*. ACM, 2010.

[261] C. Marlow, M. Naaman, D. Boyd, and M. Davis. Position paper, tagging, taxonomy, flickr, article, toread. In *Collaborative Web Tagging Workshop*. 2006.

[262] J. Marsh. XML base. Recommendation, W3C, 2001.

[263] C. C. Marshall and F. M. Shipman, III. Which semantic web? In *Hypertext*, pages 57–66. ACM, 2003.

[264] J. M. Martínez. MPEG-7 overview. Technical Report ISO/IEC JTC1/SC29/WG11N6828, INTERNATIONAL ORGANISATION FOR STANDARDISATION (ISO), 2004.

[265] M. Marx. Conditional XPath, the first order complete XPath dialect. In [134], pages 13–22.

[266] M. Marx. XPath with conditional axis relations. In [56], pages 477–494.

[267] M. Marx. First order paths in ordered trees. In T. Eiter and L. Libkin, editors, *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2005.

[268] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. Reprinted in McC90.

[269] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[270] J. McHugh and J. Widom. Query optimization for XML. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB*, pages 315–326. Morgan Kaufmann, 1999.

[271] C. Metz. Web 3.0. `http://www.pcmag.com/article2/0,2817,2102852,00.asp`, 2007.

[272] H. Meuss and K. U. Schulz. Complete answer aggregates for treelike databases: a novel approach to combine querying and navigation. *ACM Trans. Inf. Syst.*, 19(2):161–215, 2001.

[273] H. Meuss, K. U. Schulz, and F. Bry. Towards aggregated answers for semistructured data. In J. V. den Bussche and V. Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2001.

[274] P. Mika. Ontologies are us: A unified model of social networks and semantics. *J. Web Sem.*, 5(1):5–15, 2007.

[275] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In L. Popa, editor, *PODS*, pages 65–76. ACM, 2002.

[276] A. Mikroyannidis. Toward a social semantic web. *IEEE Computer*, 40(11):113–115, 2007.

[277] G. Miller. WordNet: a lexical database for english. *Communications of the ACM*, 38(11):41, 1995.

[278] L. Miller, A. Seaborne, and A. Reggiori. Three implementations of SquishQL, a simple RDF query language. In [196], pages 423–435.

[279] T. Milo and D. Suciu. Index structures for path expressions. In C. Beeri and P. Buneman, editors, *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 1999.

[280] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access control using static analysis. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM Conference on Computer and Communications Security*, pages 73–84. ACM, 2003.

[281] M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors. *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*. Morgan Kaufmann, 2004.

[282] T. Nelson. *Literary Machines*. Mindful Press, 1993.

[283] N. F. Noy and M. C. A. Klein. Ontology evolution: Not the same as schema evolution. *Knowl. Inf. Syst.*, 6(4):428–440, 2004.

[284] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with Protégé-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.

[285] C. Ogden, I. Richards, B. Malinowski, and F. Crookshank. *The Meaning of Meaning: A Study of the Influence of Language upon Thought and of the Science of Symbolism*. Harcourt, Brace & Company, 1938.

[286] W. C. Ogden and S. R. Brooks. Query languages for the casual user: Exploring the middle ground between formal and natural languages. In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 161–165. ACM, New York, NY, USA, 1983.

[287] R. O'Keefe and A. Trotman. The simplest query language that could possibly work. In *Proceedings of the 2nd INEX Workshop*. 2004.

[288] D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. Knowl. Data Eng.*, 19(7):934–949, 2007.

[289] D. Olteanu, T. Furche, and F. Bry. An efficient single-pass query evaluator for XML data streams. In H. Haddad, A. Omicini, R. L. Wainwright, and L. M. Liebrock, editors, *SAC*, pages 627–631. ACM, 2004.

[290] D. Olteanu, T. Furche, and F. Bry. Evaluating complex queries against XML streams with polynomial combined complexity. In M. H. Williams and L. M. MacKinnon, editors, *BNCOD*, volume 3112 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2004.

[291] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In A. B. Chaudhri, R. Unland, C. Djeraba, and W. Lindner, editors, *EDBT Workshops*, volume 2490 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2002.

[292] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In [364], pages 903–908.

[293] N. Onose and J. Siméon. XQuery at your web service. In [147], pages 603–611.

[294] E. Oren. Semantic wikis for knowledge workers. Technical Report, Digital Enterprise Research Institute National University of Ireland, Galway, 2005.

[295] E. Oren. SemperWiki: a semantic personal wiki. In *Proc. of 1st WS on The Semantic Desktop, Galway, Ireland*. 2005.

[296] F. Özcan, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. ACM, 2005.

[297] D. Panagiotou and G. Mentzas. A comparison of semantic wiki engines. In *22nd European Conf. on Operational Research*. 2007.

[298] J. F. Pane and B. A. Myers. The influence of the psychology of programming on a language design: Project status report. In *Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group*, pages 193–205. 2000.

[299] J. Paredaens, J. V. den Bussche, M. Andries, M. Gemis, M. Gyssens, I. Thyssens, D. V. Gucht, V. M. Sarathy, and L. V. Saxton. An overview of GOOD. *SIGMOD Record*, 21(1):25–31, 1992.

[300] J. Paredaens, P. Peelman, and L. Tanca. G-Log: A graph-based query language. *IEEE Trans. Knowl. Data Eng.*, 7(3):436–453, 1995.

[301] T. J. Parr and R. W. Quong. ANTLR: A predicated- *ll(k)* parser generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.

[302] C. Peirce. On a new list of categories. In *Proceedings of the American Academy of Arts and Sciences*, volume 7, pages 287–298. 1868.

[303] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In [121], pages 30–43.

[304] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. In [332], pages 66–81.

[305] E. Pietriga, J.-Y. Vion-Dury, and V. Quint. VXT: a visual approach to XML transformations. In *ACM Symposium on Document Engineering*, pages 1–10. ACM, 2001.

[306] J. Pokorný. Vector-oriented retrieval in XML data collections. In V. Snásel, K. Richta, and J. Pokorný, editors, *DATESO*, volume 330 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[307] A. Polleres. From SPARQL to rules (and back). In [367], pages 787–796.

[308] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF (working draft). Technical Report, W3C, 2007.

[309] Y. Qu. Q2RDF: Ranked keyword query on RDF data. *Southeast University, PR China, Tech. Rep*, 2008.

[310] F. Radlinski and T. Joachims. Query chains: Learning to rank from implicit feedback. *CoRR*, abs/cs/0605035, 2006.

[311] K. Ray. Web 3.0. http://vimeo.com/11529540, 2010.

[312] M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in PageRank. *Advances in Neural Information Processing Systems*, 2, 2002.

[313] J. Robie. Updates in XQuery. In *XML Conference & Exhibiton*. 2001.

[314] H. Rode, P. Serdyukov, and D. Hiemstra. Combining document- and paragraph-based entity ranking. In *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 851–852. ACM, New York, NY, USA, 2008.

[315] A. Russell and P. R. Smart. NITELIGHT: A graphical editor for SPARQL queries. In C. Bizer and A. Joshi, editors, *International Semantic Web Conference (Posters & Demos)*, volume 401 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[316] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[317] F. Saussure. Course in general linguistics (w. baskin, trans.). *New York: Philosophical Library*, 1916.

[318] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/ph.d. thesis, University of Munich, 2004.

[319] S. Schaffert. IkeWiki: A semantic wiki for collaborative knowledge management. In *WETICE*, pages 388–396. IEEE Computer Society, 2006.

[320] S. Schaffert and F. Bry. Querying the web reconsidered: A practical introduction to Xcerpt. In [4].

[321] S. Schaffert, F. Bry, J. Baumeister, and M. Kiesel. Semantic wikis. *IEEE Software*, 25(4):8–11, 2008.

[322] S. Schaffert, F. Bry, J. Baumeister, and M. Kiesel. Semantische wikis. In [61], pages 245–258.

[323] S. Schenk and S. Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In [199], pages 585–594.

[324] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An efficient connection index for complex XML document collections. In [56], pages 237–255.

[325] T. Schlieder. Similarity search in XML data using cost-based query transformations. In *WebDB*, pages 19–24. 2001.

[326] T. Schlieder and H. Meuss. Querying and ranking XML documents. *JASIST*, 53(6):489–503, 2002.

[327] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying XML documents made easy: Nearest concept queries. In *ICDE*, pages 321–329. IEEE Computer Society, 2001.

[328] N. Schmidt and C. Sas. Software usability: a comparison between two tree-structured data transformation languages. In R. Raisamo, editor, *NordiCHI*, pages 145–148. ACM, 2004.

[329] T. Schwentick. XPath query containment. *SIGMOD Record*, 33(1):101–109, 2004.

[330] A. Seaborne. SPARQL 1.1 property paths. Working draft, W3C, 2010.

[331] D. Shasha, J. T.-L. Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate searching in unordered trees. In *SSDBM*, pages 89–98. IEEE Computer Society, 2002.

[332] A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan, editors. *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, volume 5318 of *Lecture Notes in Computer Science*. Springer, 2008.

[333] T. Shimizu and M. Yoshikawa. Full-text and structural indexing of XML documents on B^+-tree. *IEICE Transactions*, 89-D(1):237–247, 2006.

[334] D. W. Shipman. The functional data model and the data languages DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[335] F. M. Shipman, III and C. C. Marshall. Formality considered harmful: Experiences, emerging themes, and directions on the use of formal representations in interactive systems. *Computer Supported Cooperative Work*, 8(4):333–352, 1999.

[336] C. Shirky. The semantic web, syllogism, and worldview. http://www.shirky.com/writings/semantic_syllogism.html, 2003.

[337] D. Smiley and E. Pugh. *Solr 1.4 Enterprise Search Server*. Packt Publishing, 2009.

[338] G. Smith. *Tagging: People-powered Metadata for the Social Web (Voices That Matter)*. New Riders Press, 2007.

[339] A. Souzis. Building a semantic wiki. *IEEE Intelligent Systems*, 20(5):87–91, 2005.

[340] P. Stickler. CBD—concise bounded description. Online only, 2004.

[341] M. Strube and S. P. Ponzetto. WikiRelate! computing semantic relatedness using Wikipedia. In *AAAI*. AAAI Press, 2006.

[342] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway SLCA-based keyword search in XML data. In [367], pages 1043–1052.

[343] K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[344] R. Tazzoli, P. Castagna, and S. Campanini. Towards a semantic wiki wiki web. In *Proceedings of the International Semantic Web Conferenc (ISWC)*. 2004.

[345] J. Tekli, R. Chbeir, and K. Yétongnon. An overview on XML similarity: Background, current trends and future directions. *Computer Science Review*, 3(3):151–173, 2009.

[346] A. Termehchy. Keyword and natural language query processing for semi-structured data sources. In *Proceedings of the Third SIGMOD PhD Workshop on Innovative Database Research (IDAR 2009)*. 2009.

[347] J. Thom-Santelli, M. J. Muller, and D. R. Millen. Social tagging roles: publishers, evangelists, leaders. In M. Czerwinski, A. M. Lund, and D. S. Tan, editors, *CHI*, pages 1041–1044. ACM, 2008.

[348] R. Tolksdorf and E. P. B. Simperl. Towards wikis as semantic hypermedia. In D. Riehle and J. Noble, editors, *Int. Sym. Wikis*, pages 79–88. ACM, 2006.

[349] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In [5], pages 405–416.

[350] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In [98], pages 845–856.

[351] Z. Vagena, L. S. Colby, F. Özcan, A. Balmin, and Q. Li. On the effectiveness of flexible querying heuristics for XML data. In D. Barbosa, A. Bonifati, Z. Bellahsene, E. Hunt, and R. Unland, editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2007.

[352] M. Völkel and S. Schaffert, editors. *SemWiki2006, First Workshop on Semantic Wikis - From Wiki to Semantics, Proceedings, co-located with the ESWC2006, Budva, Montenegro,*

June 12, 2006, volume 206 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[353] P. Wadler. Two semantics for XPath. Online only, 2000.

[354] N. Walsh and L. Muellner. *DocBook: The Definitive Guide*. O'Reilly, 1999.

[355] J. W. W. Wan and G. Dobbie. Mining association rules from XML data using XQuery. In J. M. Hogan, P. Montague, M. K. Purvis, and C. Steketee, editors, *ACSW Frontiers*, volume 32 of *CRPIT*, pages 169–174. Australian Computer Society, 2004.

[356] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, editors, *ICDE*, page 75. IEEE Computer Society, 2006.

[357] H. Wang, Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan. Semplore: A scalable IR approach to search the web of data. *J. Web Sem.*, 7(3):177–188, 2009.

[358] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In [182], pages 110–121.

[359] H. Wang, K. Zhang, Q. Liu, T. Tran, and Y. Yu. Q2Semantic: A lightweight keyword interface to semantic search. In [41], pages 584–598.

[360] Q. Wang, C. Nass, and J. Hu. Natural language query vs. keyword search: Effects of task complexity on search performance, participant perceptions, and preferences. In M. F. Costabile and F. Paternò, editors, *INTERACT*, volume 3585 of *Lecture Notes in Computer Science*, pages 106–116. Springer, 2005.

[361] F. Weigel. *A Survey of Indexing Techniques for Semistructured Documents*. Master's thesis, Institute for Informatics, University of Munich, 2002.

[362] F. Weigel, H. Meuss, F. Bry, and K. U. Schulz. Content-Aware DataGuides: Interleaving IR and DB indexing techniques for efficient retrieval of textual XML data. In S. McDonald and J. Tait, editors, *ECIR*, volume 2997 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2004.

[363] F. Weigel, K. U. Schulz, and H. Meuss. The BIRD numbering scheme for XML and tree databases - deciding and

reconstructing tree relations using efficient arithmetic operations. In S. Bressan, S. Ceri, E. Hunt, Z. G. Ives, Z. Bellahsene, M. Rys, and R. Unland, editors, *XSym*, volume 3671 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2005.

[364] G. Weikum, A. C. König, and S. Deßloch, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 2004.

[365] N. Wiegand. Investigating XQuery for querying across database object types. *SIGMOD Record*, 31(2):28–33, 2002.

[366] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In I. F. Cruz, V. Kashyap, S. Decker, and R. Eckstein, editors, *SWDB*, pages 131–150. 2003.

[367] C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors. *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. ACM, 2007.

[368] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

[369] P. T. Wood. On the equivalence of XML patterns. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 1152–1166. Springer, 2000.

[370] X. Wu, L. Zhang, and Y. Yu. Exploring social annotations for the semantic web. In L. Carr, D. D. Roure, A. Iyengar, C. A. Goble, and M. Dahlin, editors, *WWW*, pages 417–426. ACM, 2006.

[371] J. Xu, J. Lu, W. W. 0009, and B. Shi. Effective keyword search in XML documents based on MIU. In M.-L. Lee, K.-L. Tan, and V. Wuwongse, editors, *DASFAA*, volume 3882 of *Lecture Notes in Computer Science*, pages 702–716. Springer, 2006.

[372] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In [296], pages 537–538.

[373] J. Yang, Y. Matsuo, and M. Ishizuka. Triple Tagging: Toward bridging folksonomy and semantic web. *ISWC07*, page 14, 2007.

[374] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In [296], pages 754–765.

[375] C. Zaniolo. The database language GEM. In D. J. DeWitt and G. Gardarin, editors, *SIGMOD Conference*, pages 207–218. ACM Press, 1983.

[376] H. Zauner, B. Linse, T. Furche, and F. Bry. A RPL through RDF: Expressive navigation in RDF graphs. In P. Hitzler and T. Lukasiewicz, editors, *RR*, volume 6333 of *Lecture Notes in Computer Science*, pages 251–257. Springer, 2010.

[377] X. Zhang, G. Cheng, and Y. Qu. Ontology summarization based on RDF sentence graph. In [367], pages 707–716.

[378] Q. Zhou, C. Wang, M. Xiong, H. Wang, and Y. Yu. SPARK: Adapting keyword query to semantic search. In [8], pages 694–707.

[379] R. Zhou, C. Liu, and J. Li. Fast ELCA computation for keyword queries on XML data. In [260], pages 549–560.

[380] J. Ziegler and K. Fahnrich. *Handbook of Human-Computer Interaction*, chapter Direct Manipulation, pages 123–133. Elsevier Science, 1988.

[381] M. M. Zloof. Query by example. In *AFIPS National Computer Conference*, volume 44 of *AFIPS Conference Proceedings*, pages 431–438. AFIPS Press, 1975.

[382] A. Zollers. Emerging motivations for tagging: Expression, performance, and activism. In *WWW2007 Tagging and Metadata for Social Information Organizations*. 2007.

[383] Q. Zou, S. Liu, and W. W. Chu. Ctree: a compact tree for indexing XML data. In A. H. F. Laender, D. Lee, and M. Ronthaler, editors, *WIDM*, pages 39–46. ACM, 2004.

Lebenslauf

2001	Abitur, Rabanus-Maurus-Gymnasium Mainz
2001–2002	Studium Anglistik und öffentliches Recht, Universität Potsdam
2002–2005	Studium Kognitionswissenschaft, Universität Osnabrück *abgeschlossen mit Bachelor of Science in Kognitionswissenschaft*
2004	Auslandssemester, Radboud Universiteit Nijmegen
2005–2007	Studium Artificial Intelligence, Universiteit van Amsterdam *abgeschlossen mit Master of Science in Artificial Intelligence*
2008–	Wissenschaftliche Mitarbeiterin am Institut fr Informatik der LMU München