Chapter 1
Keyword-Based Search over Semantic Data

Klara Weiand, Andreas Hartl, Steffen Hausmann, Tim Furche, and François Bry

Abstract Enabling non-experts to publish structured or semantic data on the web is an important achievement of the social web and one of the primary goals of the social semantic web. Making this data easily accessible in turn has received only little attention. Querying in semantic wikis typically uses full text search for the textual content and a web query language for the annotations. This has two shortcomings: combined queries over content and annotations are not possible, and users either are restricted to simple but vague keyword queries or have to learn a complex web query language. In this chapter, we present an overview of KWQL, a query language that *combines* keyword search and web querying. KWQL scales with a users' experience and the sophistication of its information need by a seamlessly transition from basic keyword queries to precise, sophisticated structured analysis queries. KWQL allows for rich combined queries of full text, metadata, document structure, and annotations. KWQL's companion language visKWQL eases the authoring of such combined queries further through a set of visual building blocks. The underlying 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 a conventional search engine. Results of a user study suggest validate that users can quickly and with very little training formulate KWQL and visKWQL queries, including structured queries.

Klara Weiand, Andreas Hartl, Steffen Hausmann, François Bry
Ludwig-Maximilians-Universität München, Oettingenstr. 67, 80538 München, Germany, e-mail: klara.weiand@ifi.lmu.de, andreas-hartl@gmx.de, steffen.hausmann@ifi.lmu.de, bry@lmu.de

Tim Furche
Department of Computer Science and Institute for the Future of Computing, Oxford University, Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom e-mail: tim@furche.net

1.1 Introduction

For a long while, the creation of web content required at least basic knowledge of web technologies, meaning that for many web users the web was de facto a read-only medium. This changed with the arrival of the "social web," when web applications started to allow users to publish web content without technological expertise. Here, 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 and publishing photos.

Social *semantic* web applications are social web applications in which knowledge is expressed not only in the form of text and multimedia, but also through informal to formal annotations that describe, reflect, and enhance the content. These annotations often take the shape of RDF graphs backed by ontologies, but less formal annotations such as free-form tags or tags from a controlled vocabulary may also be available.

Wikis [29] are one example of 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.

Semantic wikis [43] are wikis that also offer – more or less sophisticated – formal languages for expressing knowledge as machine processable annotations to wiki pages. In traditional wikis, knowledge is given in the form of text in natural language, and is not directly amenable to automated semantic processing. Information 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 behind semantic wikis is to provide at least some of these enhancements by relying on semantic technologies, that is, knowledge representation formalisms and methods for automated reasoning.

To be able to leverage the knowledge contained in rich data repositories such as semantic wikis and other social semantic applications, 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.

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 ex-

pert users. The success of a social semantic web application crucially depends on the active participation and the contributions of its users, most of which cannot and should not be expected to have much experience with query languages.

Data retrieval in semantic wikis and other social (semantic) web applications is currently realized through keyword search or 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 consequence, 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 merely through keywords.

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 comparable to programming languages both in their expressive power and their complexity of use. A high cognitive investment is required before a user can employ a web query language to retrieve data from a given dataset: in addition to the schema, the user has to know and understand the data types involved as well as 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.

In summary, keyword search is generally more appropriate for search over weakly structured or unstructured text, while web query languages are well suited for querying structured data. In a social semantic web application one typically finds both types of data. None of the methods currently available provides both a sufficient level of expressiveness and ease of use.

This article describes the design and implementation of KWQL, a query language for the semantic wiki KiWi. KWQL allows for rich combined queries over textual content, metadata, document structure, and informal to formal semantic annotations. The language 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. Depending on the user's knowledge and query intent, the language can behave more like keyword search or more like web querying.

While querying the semantic wiki KiWi [44] is the main focus of this article, the underlying ideas apply more generally to querying and search on the social and social semantic web. As such, the concepts of KWQL could be transferred to de-

rive similar languages targeting other social semantic applications, and we consider KWQL to be exemplary of a novel family of query languages.

The remainder of the article is structured as follows: Section 1.2 introduces wikis and semantic wikis and gives an overview over the state of the art in querying in semantic wikis and keyword querying of semi-structured data. The next section, Section 1.3 describes the KiWi wiki and its conceptual model. Section 1.4 then introduces KWQL and its syntax, and Section 1.5 gives a relational semantics for the language. visKWQL, KWQL's visual rending, is described in Section 1.6. A first user evaluation of KWQL and visKWQL is described in Section 1.7. The following section, Section 1.8, describes KWilt, KWQL's patchwork-based implementation, and gives the results of a performance evaluation.

1.2 State of the Art

This section introduces wikis and semantic wikis and gives an overview of the search and querying functionalities provided by current semantic wiki engines. We further summarize recent research on keyword search over semi-structured data.

1.2.1 Wikis: Collaborative Content Creation

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

Apart from the original WikiWikiWeb[1] wiki engine, there exists a large number of wiki engines differing in their features, implementation, and application area, for example MediaWiki,[2] Atlassian Confluence,[3] and PhpWiki.[4]

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. 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. 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.

---

[1] http://c2.com/cgi/wiki/

[2] http://www.mediawiki.org/

[3] http://www.atlassian.com/software/confluence/

[4] http://phpwiki.sourceforge.net/

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. Individual wiki pages, on the other hand, are often weakly structured and only express knowledge as free text or multimedia.

The term "semantic wiki" is used to refer to two different types of systems [28, 43]: Semantic wikis of the first type ("wikitology" [24] 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 [43]. 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 [23]. The annotations may be freely chosen tags [12], 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 [47, 3, 4].

The annotations, whether they have been assigned manually or extracted (semi-) automatically, may be used for realizing functionalities like consistency checking, improved navigation, search, querying, personalization, context-dependent presentation, and reasoning. Annotations are often represented in 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 [43]. 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.

1.2.2 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 [40, 2] and for combined queries over content and annotations [3] 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 [40, 43], 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 [23, 47, 39]. In addition, a standard RDF query language such as SPARQL or RDQL can often be used for querying the annotations [42, 12, 2, 1]. 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 [13], can make use of simple reasoning to find query answers.
- Rhizome [46] 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.
- WikSAR [4] 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.
- Two different query languages have been suggested for Semantic MediaWiki. The first, referred to as "SMW-QL" by Bao et al [6], has a syntax similar to that used to express annotations in SMW. SMW-QL supports subqueries, (implicit) conjunction, disjunction, negation and comparison operators, but no variables. 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 [25] provide a semantics for SMW-QL through a translation to DL queries, Bao et al [6] define a semantics that is based on the translation of SMW-QL queries into logic programs. The second query language [17] employs keyword search over RDF data (see Section 1.2.3). Users express their query intent using a number of keywords which are matched in the data using a fuzzy scheme that considers semantic and syntactic similarity and translated into SPARQL queries which 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 returned.

AceWiki [26] differs from all approaches discussed above in that it employs a controlled natural language, Attempto Controlled English [15] (or ACE), 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.

Usability and expressiveness of the above query languages vary widely, however none of the existing languages fulfills all criteria outlined in Section 1.1, namely that it can be used without prior training, is expressive enough to allow complex selections, and can be used to query not only annotations, but also content, content structure and metadata.

1.2.3 Keyword Querying over Semi-Structured Data

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.

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 generation 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.

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, 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 yard- stick 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.

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.

- 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 [5, 20], XRank [16], Spark [54], and XKSearch [53].
- In label-keyword query languages such as XSearch [10] and XBridge [33], 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.
- Keyword-enhanced query languages [35, 14, 45] 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 [14] and Schmidt et al [45], 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.

- Most keyword query languages are implemented as stand-alone systems that handle all steps of the query evaluation.
- 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 [48, 54, 50], but to the best of our knowledge only one XML language, XBridge [33], which translates keyword queries into XQuery. The approach of Ladwig and Tran [27] takes an exceptional

position in that it tightly integrates query translation and query evaluation, and generates queries and candidate answers at the same time.

- Keyword-enhanced query languages finally build on existing systems by combining conventional query languages like XPath or XML-QL with keyword-querying techniques.

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, roughly 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 [31], which models XML and HTML documents as graphs. As Schmidt et al [45] 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.

Similarly, 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 because of labeled edges and blank nodes. A possible way to overcome these challenges is to summarize the RDF graph into a different structure [41, 50], 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 approaches. The reason why determining these 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.

Various heuristics for grouping have been proposed, a large majority of which are refinements of the established concept of the Lowest Common Ancestor (LCA) [18], the most specific element that is an ancestor to at least one match instance of each keyword. These include for example SLCA [53], MLCA [35], CVLCA [30], and interconnection semantics [10]. 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 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. Consequently, all LCA-

based grouping strategies are not universally applicable and under certain circumstances may lead to both false positives and false negatives [49]. This raises the question to what extent it is possible to reliably deduce semantics from structural characteristics of data alone.

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 [5, 20] but also mentioned in connection with XRank [16] and employed in keyword querying databases [8, 11] 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. An obvious disadvantage is that it requires users or administrators to invest time and effort to define the groupings.

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 [34, 7]. Whether these methods will solve the problems associated with LCA-based grouping remains to be seen.

An important characteristic of traditional query languages, namely the targeted and flexible retrieval of elements, can be found only in two of the presented standalone keyword query languages, in that of Cohen et al [10] and in XSeek [37, 36]. 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. 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 [45] only retrieve the label of the LCA node, the approach of Florescu et al [14] 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.

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 [9]. 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.

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 [22, 32], thereby realizing data type flexibility to a certain extent.

1.3 The KiWi Wiki

KiWi[5] is a semantic wiki with extended functionality in the areas of information extraction, personalization, reasoning, and querying. KiWi relies on a simple, modular conceptual model consisting of the following building blocks:

Content Items are the primary units of information in KiWi, they correspond roughly to Wiki pages in other Wikis, but they can be nested: A content item can include other content items. The nesting of content items then forms a tree structure. Each content item has a URI through which it is accessible and uniquely identifiable. Content items can contain fragments, links and tags. A content item consists of text or multimedia and an optional sequence of *contained* content items. Thus, content item nesting provides a conventional structuring of documents, for example a chapter may consist of a sequence of sections. For reasons of simplicity, content item containment precludes any form of overlapping or of cycles, and thus a content item can be seen as a directed acyclic graph (of content items).

Text Fragments are user-defined continuous portions of text within contents items. They can consist of a word, a sentence, or any other section of text, and can be annotated. Text fragments are useful for – especially collaborative – document editing for adding annotations like "improve transition", "style to be polished" or "is this correct?". While content item express the structure of written text, text fragments convey narratives. Fragments can be nested but do not overlap and do not span over content items. Fragments of this kind are generally desirable, but are problematic with respect to query evaluation. While content items allow the authors to create and organize their documents in a modular and structured way, the idea behind fragments is to enable the annotation of pieces of content independently of the canonical structure given through content item nestings. If content items are like chapters and sections in a book, then fragments can be seen as passages that readers mark; they are individual and linear and in that transcend the structure of the document, possibly spanning, fragment across paragraphs or sections.

Links are simple hypertext links and can be used for relating content items to each other or to external web sites. Links have a single origin, which is a content item, an anchor in this origin, and a single target, which is also a content item. Links can be annotated.

Tags are meta-data that can be attached to content items, fragments and links, describing their content or properties. They can be added by users, but can also be created by the system through automatic reasoning. Two kinds of annotations are available: tags and RDF triples. Tags allow to express knowledge informally, that is, 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. KWQL as presented here

[5] http://www.kiwi-project.eu

only supports the querying of tags, but the integration of RDF query facilities is discussed in Weiand [51].

Structure, within as well as between resources, plays an important role for expressing knowledge in the wiki, ranging from tags to complex graphs of links or content item containment.

In the following, *resources* refers to the basic concepts in the data model – content items, links, fragments and tag assignments (in the following referred to as "tag") and *qualifiers* refers to properties of resources like meta-data and content. *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.

1.4 KWQL: Principles and Syntax

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.

The key principle 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.

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. In the

following, we will focus on query bodies, that is, pure selection queries, and exclude the discussion of construction and reasoning in KWQL.

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 and may in addition have any number of other properties not in contradiction with the selection criteria.

KWQL's 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 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 [38] 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. 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. 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.

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. Variables can serve three purposes:

- To bind values for further use in the construction part of a rule.
- As a wildcard or existential quantifier.
- To enforce that two qualifiers have identical values. 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.

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.

Syntax

Table 1.1 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 (`+`).

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`.

A (somewhat simplified) grammar for KWQL query bodies is given in Figure 1.1. Examples of KWQL queries together with their natural language translations are shown in Table 1.2.

1.5 Semantics

For defining a formal semantics of KWQL, we introduce an abstraction of the data model of KWQL, called KWQL graphs.

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 1.1 KWQL qualifier types

⟨*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*⟩

Fig. 1.1 KWQL Syntax

Definition 1 (KWQL Graph). Let $\mathscr{Q} = \{text, title, \dots\}$ be the set of all n KWQL qualifiers and $\mathscr{V}$ the set of all qualifier values. Then, a *KWQL graph* is a $(n+6)$-tuple $G = (\mathscr{C}, \mathscr{F}, \mathscr{L}, \mathscr{T}, S, C, Q_{\lambda_1}, \dots, Q_{\lambda_n})$, where

- $\mathscr{C}$ is the set of all content items (wiki pages),
- $\mathscr{F}$ is the set of all fragments,
- $\mathscr{L}$ is the set of all links,

Java	Content items containing *"java"* directly or in any of its tags or other meta data
ci(author:Mary)	Content items authored by Mary
ci(Java OR (tag(XML) AND author:Mary))	Content items that either contain *"java"* or have a tag containing *"XML"* and are authored by Mary
ci(tag(name:$x author:Mary) tag(name:$x author:John))	Content items that are tagged with the same tag by both John and Mary
ci(tag(episode) tag(name:like author:Mary) tag(name:like author:John))	*"Episode"* content items that both Mary and John like
ci(tag(Java) link(target:ci(Lucene)))	Content items with a tag containing *"java"* that contain a link to a content item containing *"lucene"*
ci(URI:$a tag(character) link(target:ci(tag(location) link(target:ci(URI: $a)))))	Character content items that link to a location content item that links back to them

Table 1.2 KWQL example queries

- $\mathscr{T}$ is the set of all tags,
- $\mathscr{R} := \mathscr{C} \uplus \mathscr{L} \uplus \mathscr{T} \uplus \mathscr{F}$ is the set of all resources,
- $S \subset (\mathscr{C} \times \mathscr{C}_{<}) \cup (\mathscr{F} \cup \mathscr{F}_{<}) \cup (\mathscr{L} \cup \mathscr{L}_{<})$ is the association relation between resources where $\mathscr{C}_{<} = \mathscr{F} \cup \mathscr{L} \cup \mathscr{T}, \mathscr{F}_{<} = \mathscr{L} \cup \mathscr{T}, \mathscr{L}_{<} = \mathscr{T} \cup \mathscr{C} \cup \mathscr{F}$,
- $C \subset \mathscr{C} \times (\mathscr{C} \cup \mathscr{F})$ is the containment relation between wiki pages and fragments and $C^{+} = \bigcup_{n \geq 1} C^{n}$ is the transitive closure of C, and
- for each qualifier $\lambda \in \mathscr{Q}$, $Q_{\lambda} \subset \mathscr{R} \times \mathscr{V}$ associates the values for λ to a KWQL resource.

The KWQL semantics is defined based on KWQL graphs and given in Table 1.3 in terms of three functions, $[\![]\!]_{\mathrm{ci}}$, $[\![]\!]$, and $[\![]\!]_{\mathrm{dir}}$. A KWQL query is constrained by $[\![]\!]_{\mathrm{ci}}$ to return only content items (i.e., elements of $\mathscr{C}$). Most expressions can occur in two contexts, represented by the semantic functions $[\![]\!]$ and $[\![]\!]_{\mathrm{dir}}$: 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 1.3 handles variables, but omits the injectivity constraints for readability reasons. To handle variables, we introduce the set $\mathscr{I}$ of KWQL variables and the set $\mathscr{B} = 2^{\mathscr{I} \times \mathscr{V}}$ 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 $A, B \in 2^{\mathscr{R} \times \mathscr{B}}$.

Then $A \sqcap B = \{(r,\beta) \in \mathscr{R} \times \mathscr{B} : (r,\beta') \in A \wedge (r,\beta'') \in B \wedge \beta = \beta' \cap \beta'' \wedge \beta \neq \emptyset\}$, $A \sqcup B = \{(r,\beta' \cup \beta'') \in \mathscr{R} \times \mathscr{B} : (r,\beta') \in A \wedge (r,\beta'') \in B\} \cup \{(r,\beta') \in \mathscr{R} \times \mathscr{B} : (r,\beta') \in A \wedge \nexists \beta'' : (r,\beta'') \in B\} \cup \{(r,\beta') \in \mathscr{R} \times \mathscr{B} : (r,\beta') \in B \wedge \nexists \beta'' : (r,\beta'') \in A\}$.

Expression	Semantics
$[\![\langle kwql\text{-}query\rangle]\!]_{\mathrm{ci}}$	$= \pi_1([\![\langle kwql\text{-}query\rangle]\!](\emptyset)) \cap \mathscr{C}$
$[\![\langle STR\rangle]\!]_{\mathrm{dir}}(\beta) = [\![\langle STR\rangle]\!](\beta)$	$= \{(r,\beta)\in\mathscr{R}\times\mathscr{B} : \exists\lambda, v : Q_\lambda(r,v)\wedge \texttt{contains}(v,\langle STR\rangle)\}\cup$ $\{(r,\beta) : \exists r'\in\mathscr{R} : S(r,r')\wedge(r',\beta)\in[\![\langle STR\rangle]\!])(\beta)\}$
$[\![\langle qualifier\rangle\text{`}\texttt{:}\text{'}\langle STRING\rangle]\!]_{\mathrm{dir}}(\beta)$	$= \{(r,\beta)\in\mathscr{R}\times\mathscr{B} : Q_{\langle qualifier\rangle}(r,v)\wedge\texttt{contains}(v,\langle STRING\rangle)\}$
$[\![\langle qualifier\rangle\text{`}\texttt{:}\text{'}\langle STRING\rangle]\!](\beta)$	$= [\![\langle qualifier\rangle\text{`}\texttt{:}\text{'}\langle STRING\rangle]\!]_{\mathrm{dir}}(\beta)\cup$ $\{(r,\beta) : \exists r'\in\mathscr{R} : S(r,r')\wedge(r',\beta)\in[\![\langle qualifier\rangle\text{`}\texttt{:}\text{'}\langle STRING\rangle]\!](\beta)\}$
$[\![\langle qualifier\rangle\text{`}\texttt{:}\text{'`}\texttt{\$}\text{'}\langle IDENT\rangle]\!]_{\mathrm{dir}}(\beta)$	$= \{(r,\beta\cup\{(\langle IDENT\rangle,v)\})\in\mathscr{R}\times\mathscr{B} : Q_{\langle qualifier\rangle}(r,v)\wedge$ $(\nexists v' : (\langle IDENT\rangle,v')\in\beta\vee(\langle IDENT\rangle,v)\in\beta)\}$
$[\![\langle qualifier\rangle\text{`}\texttt{:}\text{'`}\texttt{\$}\text{'}\langle IDENT\rangle]\!](\beta)$	$= [\![\langle qualifier\rangle\text{`}\texttt{:}\text{'}\langle STRING\rangle]\!]_{\mathrm{dir}}(\beta)\cup$ $\{(r,\beta) : \exists r'\in\mathscr{R} : S(r,r')\wedge(r',\beta)\in[\![\langle qualifier\rangle\text{`}\texttt{:}\text{'`}\texttt{\$}\text{'}\langle IDENT\rangle]\!](\beta)\}$
$[\![\langle resource\rangle\text{`}\texttt{:}\text{'}\langle res\text{-}term\rangle]\!]_{\mathrm{dir}}(\beta)$	$= \{(r,\beta')\in\mathscr{R}\times\mathscr{B} : \texttt{type}(r,\langle resource\rangle)\wedge(r,\beta')\in[\![\langle res\text{-}term\rangle]\!]_{\mathrm{dir}}\}$
$[\![\langle resource\rangle\text{`}\texttt{:}\text{'}\langle res\text{-}term\rangle]\!](\beta)$	$= [\![\langle resource\rangle\text{`}\texttt{:}\text{'}\langle res\text{-}term\rangle]\!]_{\mathrm{dir}}(\beta)\cup$ $\{(r,\beta) : \exists r'\in\mathscr{R} : S(r,r')\wedge(r',\beta)\in[\![\langle resource\rangle\text{`}\texttt{:}\text{'}\langle res\text{-}term\rangle]\!](\beta)\}$
$[\![\text{`}\texttt{child}\text{'`}\texttt{:}\text{'}\langle kwql\text{-}query\rangle]\!](\beta)$	$= \{(r,\beta')\in(\mathscr{C}\cup\mathscr{F})\times\mathscr{B} : \exists r'\in\mathscr{R} : C(r,r')\wedge(r',\beta')\in[\![\langle kwql\text{-}query\rangle]\!]\}$
$[\![\text{`}\texttt{descendant}\text{'`}\texttt{:}\text{'}\langle kwql\text{-}query\rangle]\!](\beta)$	$= \{(r,\beta')\in(\mathscr{C}\cup\mathscr{F})\times\mathscr{B} : \exists r'\in\mathscr{R} : C^+(r,r')\wedge(r',\beta')\in[\![\langle kwql\text{-}query\rangle]\!]\}$
$[\![\text{`}\texttt{target}\text{'`}\texttt{:}\text{'}\langle kwql\text{-}query\rangle]\!](\beta)$	$= \{(r,\beta')\in\mathscr{L}\times\mathscr{B} : \exists r'\in\mathscr{R} : S(r,r')\wedge(r',\beta')\in[\![\langle kaw\text{-}query\rangle]\!]\}$
$[\![\langle res\text{-}term\rangle_1\ \langle res\text{-}term\rangle_2]\!](\beta)$	$= [\![\langle res\text{-}term\rangle_1]\!](\beta)\sqcap[\![\langle res\text{-}term\rangle_2]\!](\beta)$
$[\![\langle res\text{-}term\rangle_1\text{`}\texttt{AND}\text{'}\langle res\text{-}term\rangle_2]\!](\beta)$	$= [\![\langle res\text{-}term\rangle_1]\!](\beta)\sqcap[\![\langle res\text{-}term\rangle_2]\!](\beta)$
$[\![\langle res\text{-}term\rangle_1\text{`}\texttt{OR}\text{'}\langle res\text{-}term\rangle_2]\!](\beta)$	$= [\![\langle res\text{-}term\rangle_1]\!](\beta)\sqcup[\![\langle res\text{-}term\rangle_2]\!](\beta)$
$[\![\text{`}\texttt{(}\text{'}\langle res\text{-}term\rangle\text{`}\texttt{)}\text{'}]\!](\beta)$	$= [\![\langle res\text{-}term\rangle]\!](\beta)$
$[\![\text{`}\texttt{NOT}\text{'`}\texttt{(}\text{'}\langle res\text{-}term\rangle\text{`}\texttt{)}\text{'}]\!](\beta)$	$= \mathscr{R}\setminus\pi_1([\![\langle res\text{-}term\rangle]\!](\beta))\times\{\beta\}$
$[\![\langle res\text{-}term\rangle_1\ \langle res\text{-}term\rangle_2]\!]_{\mathrm{dir}}(\beta)$	$= [\![\langle res\text{-}term\rangle_1]\!]_{\mathrm{dir}}(\beta)\sqcap[\![\langle res\text{-}term\rangle_2]\!]_{\mathrm{dir}}(\beta)$
$[\![\langle res\text{-}term\rangle_1\text{`}\texttt{AND}\text{'}\langle res\text{-}term\rangle_2]\!]_{\mathrm{dir}}(\beta)$	$= [\![\langle res\text{-}term\rangle_1]\!]_{\mathrm{dir}}(\beta)\sqcap[\![\langle res\text{-}term\rangle_2]\!]_{\mathrm{dir}}(\beta)$
$[\![\langle res\text{-}term\rangle_1\text{`}\texttt{OR}\text{'}\langle res\text{-}term\rangle_2]\!]_{\mathrm{dir}}(\beta)$	$= [\![\langle res\text{-}term\rangle_1]\!]_{\mathrm{dir}}(\beta)\sqcup[\![\langle res\text{-}term\rangle_2]\!]_{\mathrm{dir}}(\beta)$
$[\![\text{`}\texttt{(}\text{'}\langle res\text{-}term\rangle\text{`}\texttt{)}\text{'}]\!]_{\mathrm{dir}}(\beta)$	$= [\![\langle res\text{-}term\rangle]\!]_{\mathrm{dir}}(\beta)$
$[\![\text{`}\texttt{NOT}\text{'`}\texttt{(}\text{'}\langle res\text{-}term\rangle\text{`}\texttt{)}\text{'}]\!]_{\mathrm{dir}}(\beta)$	$= \mathscr{R}\setminus\pi_1([\![\langle res\text{-}term\rangle]\!]_{\mathrm{dir}}(\beta))\times\{\beta\}$

Table 1.3 Semantics for KWQL

1.6 visKWQL

This chapter describes visKWQL Hartl et al [19], a visual rendering of KWQL that allows to expressing queries using a visual formalism.

visKWQL fully supports KWQL in the sense that every KWQL query can be expressed as an equivalent visKWQL query and vice versa. In order to avoid introducing additional constructs and thus additional complexity, the rendering stays close to the textual language in its visual representation: 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 box nestings. Boxes consist of a label, the name of the represented KWQL element, and a body, which can hold one or more child boxes. This approach has several advantages: it stays close to KWQLs 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 1.2 shows an example of a visKWQL query corresponding to the textual KWQL query tag(author:Mary AND name:wiki), which

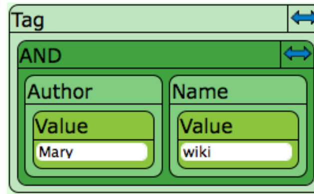

Fig. 1.2 A visKWQL query

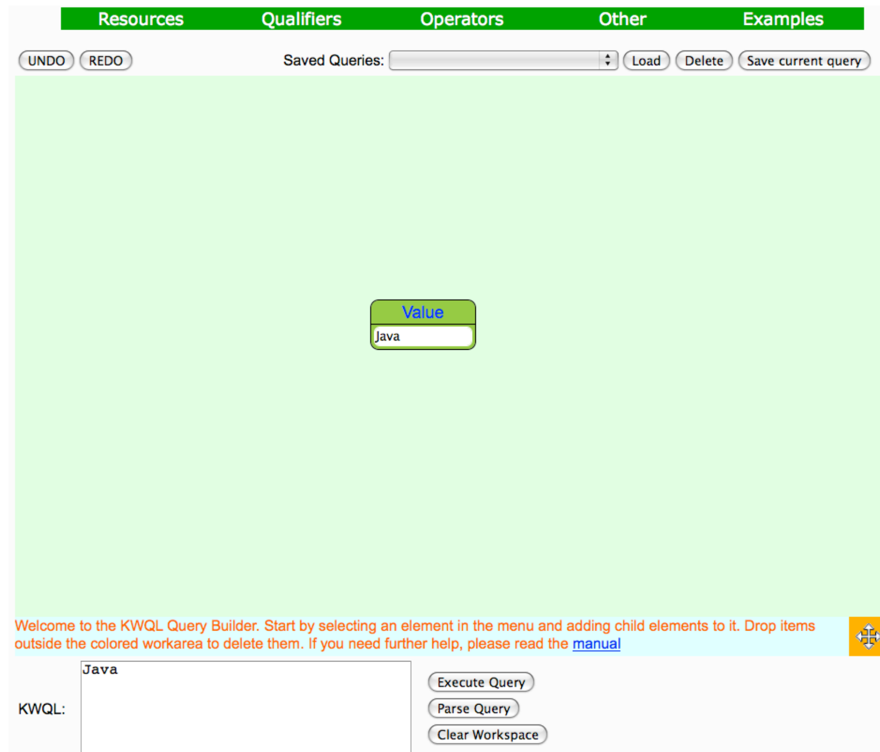

Fig. 1.3 The KiWi Query Builder

retrieves content item that Mary has tagged with "wiki" or that contain a fragment or link with such a tag.

An accompanying editor, the KWQL Query Builder (KQB, see Figure 1.3), 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 by pointing the user to syntactically incorrect parts of a query. 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 users.

The KQB further provides features like information hiding to only display parts of larger queries, and the highlighting of all occurrences of a variable when the mouse pointer is positioned over a variable in a query.

One particularly important feature of KQB is round-tripping, which allows users to edit a query in both representations, visual and textual, at the same time, and see any changes made to one representation reflected in the other. The side-by-side display of both representations offers the additional advantage of helping users to learn KWQL by creating queries in visKWQL. Users do 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. For example, the user can 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.

The visual KWQL query editor does not require the installation of special software or browser plug-ins, but instead is implemented using DHTML, with HTML and CSS for the presentation and Java Script for the program logic and user interaction. As a consequence, the system runs completely on the client side, within the users web browser and the translation from visKWQL to KWQL can be seen as a serialization of the visual query.

1.7 User Evaluation

This section 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. A question of particular interest is whether the results differ (1) between 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 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, 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.

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. These include the gradual, self-paced learning process, user's individual query intents, long-distance effects and individual preferences for either KWQL or visKWQL.

1.7.1 Experimental Setup and Execution

To reflect the collaborative process of content creation and annotation, several user accounts were created in an installation of the KiWi wiki. Each account was used to compose a number of content items containing text from a wiki on the TV show The Simpsons.[6] These content items were then annotated with tags. The final dataset, used in this study, consisted of 653 content items.

Twenty-one participants were recruited via an internet forum aimed at LMU Munichs computer science students and via announcements in several computer science lectures at the university. 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.

Before the experiment, participants were asked to fill out a questionnaire about their previous experience in areas relevant to semantic wikis and KWQL such as the semantic web, tagging and XML; participants were also asked which programming and query languages they knew. Each participant was then randomly assigned to either the KWQL or the visKWQL group.

In an introductory phase, participants were allowed to familiarize themselves with the KiWi wiki and KWQL or visKWQL for 30 minutes. This was followed by a query creation task and a query understanding task, which respectively lasted 45 and 15 minutes. The objective of the query creation task was to use KWQL or visKWQL to answer questions, given in natural language, about the data in the wiki. In total, the task consisted of ten assignments of increasing difficulty. In the query understanding task, participants were given six KWQL or visKWQL queries of intermediate to advanced complexity, and were asked to describe the underlying query intent, that is, the common characteristics of the content items selected by each query, using natural language.

1.7.2 Results

Participants' self-assessed average knowledge of various areas relevant to KWQL and visKWQL was very similar for the two groups. The only concepts participants were familiar with to some extent were wikis, XML, and tags.

For the analysis, participants were divided into two groups based on their previous knowledge of query languages, social software and semantic web technologies. In the following, participants will often be referred to as "novice participants" or "advanced participants" based on the group they were assigned to. Depending on previous knowledge and the query language used, each participant thus belonged to one of four groups. The number of participants in each group was between four and six.

[6] http://simpsons.wikia.com/wiki/Simpsons_Wiki

	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 1.4 Average number of questions (out of 10) answered

	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 1.5 Average percentage of given answers that are correct

Task 1: Query creation

Table 1.4 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) compared to novice participants (8.02), and slightly higher for KWQL users (8.20) than for visKWQL users (8.12). Furthermore, 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.

Table 1.5 shows the average percentage of the given answers that were correct. Among all participants, almost two thirds of all answers given, 62.24%, were correct. The visKWQL group was responsible for both the best and the worst results, with 93.33% correct answers for advanced visKWQL users and 35.66% correct answers for novice visKWQL. This result is particularly noteworthy since both groups answered a very similar amount of questions, as shown in Table 1.4. While novice visKWQL users answered more questions on average than novice KWQL users, a smaller percentage of those answers were correct, leading to a higher absolute number of correct answers for the KWQL group. Among the advanced groups, the situation is different: visKWQL users answered fewer questions but did so at a very high rate of correctness. As a consequence, the average absolute number of correct answers is higher for advanced visKWQL users.

Out of the total of 171 queries given as answers to questions in the query creation task, only seven, four KWQL queries and three visKWQL queries, were invalid in the sense that they could not be parsed or violated a validity constraint.[7] Consequently, 95% of all KWQL queries and 97% of all visKWQL queries given as answers were valid. The majority of incorrect answers therefore consisted of queries that were valid but did not correspond to the assignment.

Task 2: Query understanding In the query understanding task, all advanced participants provided answers to all six questions, while the novice participants an-

[7] 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.

	KWQL	visKWQL	overall
novice	5.5	5.5	5.5
advanced	6	6	6
overall	5.75	5.7	5.72

Table 1.6 Average number of questions (out of 6) answered in task 2

	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 1.7 Average number of questions (out of 6) answered correctly

swered 5.5 questions on average (see Table 1.6). 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 gave 5.5 correct answers on average. The situation is different for the novice users: here, those using KWQL had 4.75 correct answers on average, while participants in the visKWQL group only answered 4.0 questions correctly on average. 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.

User Judgments After completing the two tasks, participants were asked about their opinion on KWQL or visKWQL. Most participants, 13 out of 20, said that they felt they had understood how to use the respective 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.

With respect to the question whether KWQL or visKWQL was easy to use, a majority of participants answered that it was. However, many qualified their response and listed particular aspects they found hard to understand. Specifically, participants experienced problems with variables, URIs, injectivity, nesting of content items, and links. In several cases, participants did not understand the question or were unsure how to translate it into a query.

Finally, participants were asked what they considered to be advantages and disadvantages of KWQL and visKWQL. All participants thought that KWQL and visKWQL are powerful and allow for precise queries, while some remarked that they are harder to use than web search and take some time to learn.

1.7.3 Discussion

All in all, the results of the experimental evaluation are very positive: KWQL and visKWQL were well perceived by the participants. Given only a very short intro-

duction 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.

The amount of learning required could explain why visKWQL novices performed worse than the participants in the novice KWQL group: apart 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 the comparatively bad performance of novice visKWQL users could be that visKWQL is not ideally 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. Indeed, novice visKWQL participants performed particularly badly compared to novice KWQL participants on questions that require underspecified queries, and the difference in the percentage of correct answers was smaller when the answer queries contained 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 the round-tripping capabilities of visKWQL, 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.

Advanced participants achieved good results regardless of the query language: on average, they answered 71% of the questions in the query creation task and over 90% of the questions in the query understanding task correctly. Their results also showed that visKWQL can help to improve the performance: advanced visKWQL participants gave fewer answers overall than advanced KWQL participants, but nearly all of their answers were correct. These findings indicate that participants who are familiar with querying and structured data and to whom the information in the introductions is 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.

Across all groups, participants had more success understanding queries than writing them. In the query understanding task, novice KWQL users again 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% of the questions correctly, indicating that this task was very easy for them overall. The fact that participants performed 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.

1.8 KWilt: Patchwork Knowledge Management

KWilt is the implementation of KWQL in KiWi. It provides an easily extensible, yet performant implementation of the KWQL features over the wide range of data available in KiWi. Previous approaches have often tried to engineer a knowledge information systems for such diverse information and user needs from the start. By 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 to evaluate keyword queries. The patchwork approach has three main advantages:

- Many queries can be evaluated at the speed of search engines, yet all the power of first-order logic is available if needed. The three steps use increasingly more expressive, but also less scalable technologies. Thus, even for queries that involve full first-order constraints, we can in many cases substantially reduce the number of candidates in the information retrieval engine. 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, unaware of the additional expressiveness provided by KWQL.
- Each part is implemented using proven technologies and algorithms with minimal "glue" between the employed tools.
- The separation makes it easy to adapt each of the parts, e.g., to reflect additional data sources. If KiWi would introduce data with different structural properties, e.g., strictly hierarchical taxonomies in addition to RDF ontologies, 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.

1.8.1 Architecture and Evaluation Phases

Despite the unique combination of features found in KWQL, KWilt does not try to "reinvent the wheel." Instead, we used 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 different evaluation phases which are dedicated to certain aspects of the query, see Figure 1.4. Each step makes use of a tool that is particularly suitable for evaluating the query constraints covered by that aspect of the entire evaluation. Thus, efficient and mature algorithms form the basis of our framework.

Evaluation of keyword queries Most KWQL queries, in particular by novice users, mainly or exclusively regard the content of the pages. Therefore, the first evaluation phase regards the keyword parts of a query in order to evaluate them in an early phase of the evaluation with as little overhead as possible. If all constraints of the query can be validated in this phase, the two subsequent phases can be skipped.

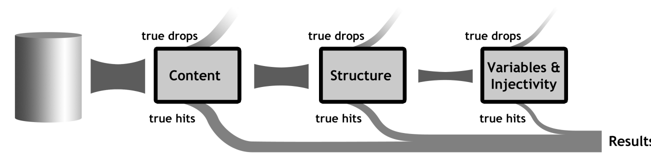

Fig. 1.4 The evaluation pipeline of the framework

The information retrieval engine 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 most commonly used to store the text of a document and its meta data.

In order to use Solr for the evaluation of KWQL queries, the meta data of wiki pages and further the meta data of its tags, fragments and links are stored in a Solr document. The main principle of the translation is to materialize joins between content items and the directly connected resources.

The transformation of the resources connected to a content item to fields in the Solr index is lossy, since the value of multiple resources is 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.

To keep the index small, only dependencies to flat resources are materialized, which omits in particular nesting and linking of content items. Therefore, only queries that access content items together with their content, meta-data and directly related flat 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 generate a set of candidates which match those parts of the query for which all necessary information stored in the Solr index.

In order to evaluate a KWQL query through Solr, a portion of the KWQL query (that can be evaluated by Solr) is converted to the query language of Solr. Information which is not covered by the materialized joins and variables are either disregarded or at least converted to an existential quantification in order to reduce the number of false positives.

Evaluation of structural constraints The second phase takes the structural parts of a query into account. 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 common relational database in combination with an object-relational mapping.

In the current prototype, we validate the structural properties of a query for each candidate item individually. That means, nested resources (tags, fragments, links and contained content items) which are specified in the query are considered by traversing the references of the currently investigated object.

We choose this approach, as 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 on 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, e.g., various graph reachability indices, and for RDF data might be advantages.

In addition to the verification of the structural constrains, the structural dependencies of the contributing resources and the required values of their qualifiers are stored in relations which are needed during the last evaluation phase.

Evaluation of first-order constraints over wiki resources In the final evaluation phase, first-order constraints over wiki resources are considered, as induced by the KWQL variables (and some advanced features of KWQL such as injectivity).

Following constraint programming notation, we consider a first-order constraint a formula over logical relation 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 resources from the query and their meta data. These relations are constructed during the prior evaluation phase since all required values and dependencies of the resources are regarded in this phase anyhow.

Thus the relations are used to connect the formal representation of the query and the candidate matches. The first-order constraints are evaluated using the constraint solver choco [21].

Any content item that fulfills the constraints validated in all three phases is a match for the entire query. In fact, since we only feed candidate matches from the prior phase to each subsequent phase, the content item (identifiers) returned by choco immediately give us the KWQL answers.

1.8.2 Skipping Evaluation Phases: KWQL's Sublanguages

The evaluation of a general KWQL query in KWilt is performed in three phases as described in the previous section. 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.

Keyword KWQL or KWQL$_K$ KWQL$_K$ is the restriction of KWQL to mostly flat queries where *resource terms* may not occur nested inside other resource terms and *structure terms* are not allowed at all.

Since tags and fragments itself can not be nested more than one level, we can also materialize all tags and fragments for each content item. However, in contrast to (string-valued) qualifiers a content item can have multiple tags or fragments. To allow evaluation with a 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 of these items in a later evaluation phase.

To ensure this, we allow tag and fragment queries but disallow

- two keyword queries as siblings expressions in tag or fragment queries and
- two tag or fragment queries as sibling expressions

KWQL_K expressions can be evaluated entirely by the information retrieval engine, here Solr.

Tree-shaped KWQL or KWQL_T KWQL_T allows only queries corresponding to tree-shaped constraints. Thus, no multiple occurrences of the same variable, and no potentially overlapping expression siblings.

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 (see Section 1.5) or if the return type of one is a subset of that of the other one.

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.

Proposition 1. *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 phaseconstraints 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.

1.8.3 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.

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

query						
phase 1		phase 2		phase 3		total time
time [ms]	results	time [ms]	results	time [ms]	results	[ms]
KiWi						
31	14	–	–	–	–	31
ci(text:KWQL title:KiWi)						
6	1	–	–	–	–	6
KiWi tag(name:$t)						
42	9	–	–	–	–	42
ci(tag(name:KWQL) child:ci(tag(Example)))						
10	5	44	1	–	–	54
ci(Munich link(target:ci(KiWi)))						
33	4	52	4	–	–	85
ci(KiWi link(target:ci(KiWi)))						
60	10	206	4	–	–	266
ci(tag(name:*r*)*text* :r)						
149	9	53	9	103	9	305
ci(tag(author:admin) tag(name:KWQL))						
9	5	55	5	67	4	131
ci(KiWi tag(name:$t) link(target:ci(URI:$u tag(name:$t))))						
44	9	181	4	194	3	419

Table 1.8 Evaluation times in the KiWi dataset

determining whether the query can be fully processed using only phase 1, was found to take between 27 and 42 milliseconds. Table 1.8 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.

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. Correspondingly, the evaluation times 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 three queries in the table make use of the full power of KWQL and require all three evaluation phases.

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

phase 1 time [ms]	phase 1 results	phase 2 time [ms]	phase 2 results	phase 3 time [ms]	phase 3 results	total time [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:*rtext* :r)						
105	38	21	38	15	1	141
ci(title:*rtext* :r)						
679	505	15	505	75	58	769
ci(tag(name:web) tag(author:Peter))						
92	34	863	34	3246	34	4201

Table 1.9 Evaluation times in the RSS dataset

process, particularly when the first phase does not sufficiently decrease the size of the candidate set.

In summary, this first small-scale 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.

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:

Despite the two possibilities for improving the second evaluation phase discussed above, 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:

- 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.

- 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 1.5 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.

1.9 Outlook

At least two extensions to KWQL as described here are desirable: To add two important features of keyword search to the language, fuzzy matching and ranking should be provided. Towards this end, we suggest PEST [52], a PageRank-like approach to approximate querying of structured data that exploits the structure to propagate term weights between related data items and uses the resulting modified index for ranking as well as fuzzy matching over data structure. Secondly, one issue that has not been addressed so far is that of querying RDF with KWQL. While dealing with complex RDF graphs may indeed overburden many users, simple RDF triples are intuitive and easy to understand. KWQL should therefore allow users to query at least these simple RDF annotations that they and others have created. Weiand [51] discussed three solutions for adding support for RDF queries, one native and two based on the integration of existing RDF query languages.

1.10 Conclusion

The work presented in this article 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. We presented KWQL, a query language for the KiWi wiki based on the label- keyword query paradigm that allows for rich combined queries of textual content, metadata, document structure, and annotations.

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.

References

[1] Auer S, Dietzold S, Lehmann J, Riechert T (2007) OntoWiki: A tool for social, semantic collaboration. In: Proceedings of the Workshop on Social and Collaborative Construction of Structured Knowledge
[2] Aumueller D (2005) Semantic authoring and retrieval within a wiki. In: Proceedings of the 2nd European Semantic Web Conference
[3] Aumueller D (2005) SHAWN: Structure helps a wiki navigate. In: Proceedings of the BTW-Workshop WebDB Meets IR
[4] Aumueller D (2005) Towards a semantic wiki experience – desktop integration and interactivity in WikSAR. In: Proceedings of the 1st Workshop on The Semantic Desktop
[5] Balmin A, Hristidis V, Koudas N, Papakonstantinou Y, Srivastava D, Wang T (2003) A system for keyword proximity search on XML databases. In: Proceedings of 29th International Conference on Very Large Data Bases, pp 1069–1072
[6] Bao J, Ding L, Hendler J (2008) Knowledge representation and query in semantic MediaWiki: A formal study. Technical Report TW-2008-42, Tetherless World Constellation (RPI)
[7] Bao Z, Ling TW, Chen B, Lu J (2009) Effective XML keyword search with relevance oriented ranking. In: Proceedings of the 25th International Conference on Data Engineering, pp 517–528
[8] Bhalotia G, Hulgeri A, Nakhe C, Chakrabarti S, Sudarshan S (2002) Keyword searching and browsing in databases using BANKS. In: Proceedings of the 18th International Conference on Data Engineering, pp 431–440
[9] Bischoff K, Firan CS, Nejdl W, Paiu R (2008) Can all tags be used for search? In: Proceedings of the 17th ACM Conference on Information and Knowledge Management, pp 193–202

[10] Cohen S, Mamou J, Kanza Y, Sagiv Y (2003) XSearch: A semantic search engine for XML. In: Proceedings of 29th International Conference on Very Large Data Bases, pp 45–56
[11] Dar S, Entin G, Geva S, Palmon E (1998) DTL's DataSpot: Database exploration using plain language. In: Proceedings of 24rd International Conference on Very Large Data Bases, pp 645–649
[12] El Ghali A, Tifous A, Buffa M, Giboin A, Dieng-Kuntz R (2007) Using a semantic wiki in communities of practice. In: Proceedings of the 2nd International Workshop on Building Technology Enhanced Learning Solutions for Communities of Practice
[13] Fischer J, Gantner Z, Rendle S, Stritt M, Schmidt-Thieme L (2006) Ideas and improvements for semantic wikis. In: Proceedings of the 3rd European Semantic Web Conference, pp 650–663
[14] Florescu D, Kossmann D, Manolescu I (2000) Integrating keyword search into XML query processing. Computer Networks 33(1-6):119–135
[15] Fuchs NE, Kaljurand K, Schneider G (2006) Attempto Controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces. In: Proceedings of the 19th International Florida Artificial Intelligence Research Society Conference, pp 664–669
[16] Guo L, Shao F, Botev C, Shanmugasundaram J (2003) XRANK: Ranked keyword search over XML documents. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 16–27
[17] Haase P, Herzig D, Musen MA, Tran T (2009) Semantic wiki search. In: Proceedings of the 6th European Semantic Web Conference, pp 445–460
[18] Harel D, Tarjan RE (1984) Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13:338–355
[19] Hartl A, Weiand K, Bry F (2010) visKQWL, a visual renderer for a semantic web query language. In: Proceedings of the 19th International Conference on World Wide Web, pp 1253–1256
[20] Hristidis V, Papakonstantinou Y, Balmin A (2003) Keyword proximity search on XML graphs. In: Proceedings of the 19th International Conference on Data Engineering, pp 367–378
[21] Jussien N, Prud'homme C, Cambazard H, Rochart G, Laburthe F (2008) choco: an open source java constraint programming library. In: Proceedings of the Workshop on Open-Source Software for Integer and Constraint Programming
[22] Kacholia V, Pandit S, Chakrabarti S, Sudarshan S, Desai R, Karambelkar H (2005) Bidirectional expansion for keyword search on graph databases. In: Proceedings of the 31st International Conference on Very Large Data Bases, pp 505–516
[23] Kiesel M (2006) Kaukolu: Hub of the semantic corporate intranet. In: Proceedings of the 1st Workshop on Semantic Wikis
[24] Klein B, Höcht C, Decker B (2005) Beyond capturing and maintaining software engineering knowledge – "Wikitologies" as shared semantics. In: Pro-

ceedings of the Workshop on Knowledge Engineering and Software Engineering
[25] Krötzsch M, Vrandecic D (2009) Semantic Wikipedia. In: Blumauer A, Pellegrini T (eds) Social Semantic Web, Springer, pp 393–421
[26] Kuhn T (2008) AceWiki: A natural and expressive semantic wiki. CoRR abs/0807.4618
[27] Ladwig G, Tran T (2010) Combining query translation with query answering for efficient keyword search. In: Proceedings of the 7th Extended Semantic Web Conference, pp 288–303
[28] Landefeld R, Sack H (2007) Collaborative web-publishing with a semantic wiki. In: Proceedings of the 1st Conference on Social Semantic Web, pp 23–34
[29] Leuf B, Cunningham W (2001) The Wiki way: quick collaboration on the Web. Addison-Wesley
[30] Li G, Feng J, Wang J, Zhou L (2007) Effective keyword search for valuable LCAs over XML documents. In: Proceedings of the 16th ACM Conference on Information and Knowledge Management, pp 31–40
[31] Li G, Feng J, Wang J, Song X, Zhou L (2008) SAILER: an effective search engine for unified retrieval of heterogeneous XML and web documents. In: Proceedings of the 17th International Conference on World Wide Web, pp 1061–1062
[32] Li G, Ooi BC, Feng J, Wang J, Zhou L (2008) EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 903–914
[33] Li J, Liu C, Zhou R (2008) XBridge: Answering XML keyword search with structured queries
[34] Li J, Liu C, Zhou R, Wang W (2010) Suggestion of promising result types for XML keyword search. In: Proceedings of the 13th International Conference on Extending Database Technology, pp 561–572
[35] Li Y, Yu C, Jagadish HV (2004) Schema-free XQuery. In: Proceedings of the 13th International Conference on Very Large Data Bases, pp 72–83
[36] Liu Z, Chen Y (2007) Identifying meaningful return information for XML keyword search. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 329–340
[37] Liu Z, Walker J, Chen Y (2007) XSeek: A semantic XML search engine using keywords. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp 1330–1333
[38] Olteanu D, Meuss H, Furche T, Bry F (2002) XPath: Looking forward. In: Proceedings of the EDBT Workshop on XML-Based Data Management, pp 109–127
[39] Oren E (2005) SemperWiki: A semantic personal wiki. In: Proceedings of the 1st Workshop on The Semantic Desktop
[40] Panagiotou D, Mentzas G (2007) A comparison of semantic wiki engines. In: Proceedings of the 22nd European Conference on Operational Research

[41] Qu Y (2008) Q2RDF: Ranked keyword query on RDF data. Technical report, Southeast University, China

[42] Schaffert S (2006) IkeWiki: A semantic wiki for collaborative knowledge management. In: Proceedings of the 15th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, pp 388–396

[43] Schaffert S, Bry F, Baumeister J, Kiesel M (2008) Semantic wikis. IEEE Software 25(4):8–11

[44] Schaffert S, Eder J, Grünwald S, Kurz T, Radulescu M (2009) Kiwi – a platform for semantic social software. In: Proceedings of the 6th European Semantic Web Conference, pp 888–892

[45] Schmidt A, Kersten ML, Windhouwer M (2001) Querying XML documents made easy: Nearest concept queries. In: Proceedings of the 17th International Conference on Data Engineering, pp 321–329

[46] Souzis A (2005) Building a semantic wiki. IEEE Intelligent Systems 20(5):87–91

[47] Tazzoli R, Castagna P, Campanini S (2004) Towards a semantic wiki wiki web. In: Proceedings of the 3rd International Semantic Web Conference

[48] Tran T, Wang H, Rudolph S, Cimiano P (2009) Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In: Proceedings of the 25th International Conference on Data Engineering, pp 405–416

[49] Vagena Z, Colby LS, Özcan F, Balmin A, Li Q (2007) On the effectiveness of flexible querying heuristics for XML data. In: Proceedings of the 5th International XML Database Symposium, pp 77–91

[50] Wang H, Zhang K, Liu Q, Tran T, Yu Y (2008) Q2Semantic: A lightweight keyword interface to semantic search. In: Proceedings of the 5th European Semantic Web Conference, pp 584–598

[51] Weiand K (2011) Keyword-based querying for the social semantic web – the kwql language: Concept, algorithm and system. PhD thesis, University of Munich

[52] Weiand K, Kneißl F, Furche T, Bry F (2010) PEST: Term-propagation over wiki-structures as eigenvector computation. In: Fifth Workshop on Semantic Wikis (to be published)

[53] Xu Y, Papakonstantinou Y (2005) Efficient keyword search for smallest LCAs in XML databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 537–538

[54] Zhou Q, Wang C, Xiong M, Wang H, Yu Y (2007) SPARK: Adapting keyword query to semantic search. In: Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference, pp 694–707