KWilt: A Semantic Patchwork for Flexible Access to Heterogeneous Knowledge

Klara Weiand[1], Steffen Hausmann[1], Tim Furche[1,2], and François Bry[1]

[1] Institute for Informatics, University of Munich,
Oettingenstraße 67, D-80538 München, Germany
http://www.pms.ifi.lmu.de/
[2] Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
http://web.comlab.ox.ac.uk/people/Tim.Furche/

Abstract. Semantic wikis and other modern knowledge management systems deviate from traditional knowledge bases in that information ranges from unstructured (wiki pages) over semi-formal (tags) to formal (RDF or OWL) and is produced by users with varying levels of expertise. KWQL is a query language for semantic wikis that scales with a user's level of expertise by combining ideas from keyword query languages with aspects of formal query languages such as SPARQL. In this paper, we discuss KWQL's implementation KWilt: It uses, for each data format and query type, technology tailored to that setting and combines, in a patchwork fashion, information retrieval, structure matching and constraint evaluation tools with only lightweight "glue". We show that it is possible to efficiently recognize KWQL queries that can be evaluated using only information retrieval or information retrieval and structure matching. This allows KWilt to evaluate basic queries at almost the speed of the underlying search engine, yet also provides all the power of full first-order queries, where needed. Moreover, adding new data formats or abilities is easier than in a monolithic system.

1 Introduction

To accommodate all users, modern knowledge management systems such as semantic wikis must deal with both unstructured (textual or multi-modal) information as well as structured data carrying varying degrees of semantics: hierarchical data for document and simple classification structures, social classifications in form of tag networks, formal ontologies in RDF or OWL. Expert users in such systems can define semantically rich, automated analysis or derivation tasks. However, the vast number of users has little understanding of formal knowledge representation, produces unstructured information with lightweight semantic annotations such as free-form tags, and interacts with the system through simple but imprecise keyword queries.

From these observations, we derive two properties that characterize successful modern knowledge management systems:

(1) "Interfaces must be adaptable and flexible": Interfaces should scale with user experience: For novice users, simple, but imprecise queries are useful

for satisfying their information needs; for expert users precise, but necessarily fairly complex queries that enable automated action and derivation are required. Interfaces should also be able to adapt to different types of knowledge in a system, providing a consistent interface.

(2) "Patchwork knowledge management": Due to the growth in data size and formats, knowledge management systems face a dual challenge: Users expect high performance for (at least basic) queries regardless of the data scale, as in Web search engines. On the other hand, knowledge management systems must be able to adapt quickly to additional knowledge sources, providing scalable yet sufficiently expressive interfaces to query and process such data.

In this paper, we present a patchwork approach to knowledge management using the query language KWQL and its implementation, KWilt. We choose the setting of a semantic wiki, KiWi, as it exemplifies many of the challenges outlined above.

KWQL: Scale with User Experience. To illustrate how KWQL provides a consistent interface that easily adapts to different levels of user experience, consider the following scenario: *"In a wiki describing KiWi, we would like to find all wiki pages that describe (knowledge management) systems that have influenced the development of KiWi."*

In a conventional knowledge management system, we would expect a formal relation (e.g., `wk:influences`) that represents the very intent of our query. Indeed, given an RDF representation of such a query we can query such a relation in KWQL as (assuming `wk:KiWi` represents the KiWi system):

```
ci(rdf(predicate:'wk:influences' object:'wk:KiWi'))
```

However, in most cases this relation is *not* present explicitly. Even if it is, users are often not able to express their intent in such a formal manner.

Accustomed to Web search engines, novice users might start with a keyword query that returns all resources (or **c**ontent **i**tems) containing "KiWi": `KiWi`

Obviously, such a query is too unspecific to capture the above query intent and, in fact, may omit a number of systems, that are described without reference to KiWi, but that are referenced from the description of KiWi.

Thus, we might refine the query to return such referenced resources, i.e., resources that are the target of a link originating from a wiki page containing "KiWi": `$u @ ci(KiWi link(target:ci(URI:$u))`

The `$u @` of the query ensures, that not the content item that points to the page, but rather the page that is pointed at, is returned by the query. However, that query is not specific enough, as it returns also, e.g., technologies used in KiWi. We know that KiWi is a semantic wiki and might be tempted to amend that query to return only resources that are also semantic wikis:

```
$u @ ci(KiWi link(target:ci(URI:$u tag(name:'semantic wiki')))
```

But there might well be other systems that have a significant influence on KiWi. To also capture them, we choose the query:

```
$u @ ci(KiWi tag(name:$t) link(target:ci(URI:$u tag(name:$t)))
```

It returns all resources that are tagged the same as a wiki page containing "KiWi" that also links to the returned resource. This way we likely capture resources with similar characteristics as KiWi that are also mentioned in its description.

To summarize, KWQL's main contributions over existing query languages and similar interfaces for knowledge management systems are:

1. KWQL provides a **consistent interface** for access to the wide range of knowledge present in the semantic wiki KiWi.

2. KWQL is designed to **scale with the user experience**: Queries can take the form of bags of keywords, but also be extended with increasingly more precise constraints on the structure, tags, and formal annotations of wiki pages.

3. KWQL is well **integrated into KiWi**, it covers all aspects of its data model and is used in KiWi's rule language.

KWilt: Patchwork Knowledge Management. Previous approaches have often tried to engineer a knowledge information systems for such diverse information and user needs from the start. In constrast, KWilt, KWQL's implementation in KiWi, uses a "patchwork" approach, combining performant and mature technologies where available. For example, KWilt uses a scalable and well established information retrieval engine (Solr) to evaluate keyword queries. In fact, KWilt tries to evaluate as large a fragment of any KWQL query in the information retrieval engine as possible. If necessary, the results are further refined by (1) checking any structural constraints of the query and (2) finally enforcing all remaining first-order constraints, e.g., from multiple variable occurrences.

KWilt's patchwork approach has three main advantages:

1. Many queries can be evaluated at the speed of search engines, yet all the power of first-order logic is available if needed, as detailed in Section 4: 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 most cases, substantially reduce the number of candidates in the information retrieval engine and by enforcing structural constraints before evaluating the first-order constraints.

2. Each part is implemented using **proven technologies** and algorithms with minimal "glue" between the employed tools, see Section 2.

3. The **separation makes it easy to adapt** each of the parts, e.g., to reflect additional data sources. E.g., 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.

2 KWilt: Architecture and Evaluation Phases

Evaluating KWQL queries is a challenging task that cannot be accomplished by existing query engines for (semantic) wikis. For instance, the query

```
ci(KiWi tag(name:$t) link(target:ci(tag(name:$t))))
```

combines content and structural elements with variables to find content items that a wiki page containing *"KiWi"* links to and that have at least one tag in common with their linking page.

Despite the unique combination of features found in KWQL, KWilt does not try to "reinvent the wheel". In particular, we have chosen not to build a new index structure capable of combining all these aspects in a single index access, as this approach has several drawbacks. First and foremost, the rich data model would require a fairly complex index structure that can support content and structure queries, fast access to hierarchical data and link graphs, RDF graph navigation as well as navigation over (simpler, but less regular) containment and link relations. Moreover, it is quite likely that the data model evolves over time with new kinds of data or different representation formats introduced, in particular in the field of social semantic media. Using a complex index structure which is carefully adapted for a certain data model makes it hard or even infeasible to react to these potential changes.

Instead, we used an *patchwork*, or integration, approach to combine off-the-shelf state-of-the-art tools in a single framework. For that, the evaluation is split into **three different evaluation phases** which are dedicated to certain aspects of the query. Each step makes use of a tool which is particularly suitable for evaluating the query constraints covered by that aspect of the entire evaluation, e.g., for keyword queries we use a traditional search engine. Thus, efficient and mature algorithms form the basis of our framework while the framework itself remains flexible with lightweight "glue" to combine the evaluation phases.

Evaluating keyword queries: Most KWQL queries, in particular by novice users, mostly or only 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. In particular, if all constraints of the query can be validated in this phase, the two subsequent phases can be skipped.

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 benefit from Solr, the content of the wiki needs to be stored in this index, i.e., *all resources including their dependencies need to be translated to flat Solr documents.*

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 (tags, fragments, links) that are commonly queried together in the same query. These materialized joins are then stored in the fields of the document representing the content item. Thus, queries regarding the meta data of content items and even the meta data of its tags, fragments and links can be directly answered with Solr. However, 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. Therefore is necessary for certain kinds of queries to validate the generated result set of Solr.

To keep the index small only *dependencies to flat resources* (that can not be further nested) 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.

Evaluating 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. For instance the titles of content items are stored in the relation R_{title} which is therefore a set of tuples of identifiers and strings.

Evaluating 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, that are not further discussed here).

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.

For instance, to express that a content item has a certain title, the relation R_{title} is used: $(C, KiWi) \in R_{\text{title}}$. This constraint causes the variable C to be bound only to identifiers of content items with the title *"KiWi"*. Likewise, the relation R_{tag} is used to specify that a content item has a tag: $(C, T) \in R_{\text{tag}}$.

For each KWQL variable of the query, a new first-order variable is generated that can be used in the structural constraints. For instance, the query

```
ci(title:$t tag(name:$t))
```

can be represented as: $(C, \$t) \in R_{\text{title}} \wedge (C, T) \in R_{\text{tag}} \wedge (T, \$t) \in R_{\text{name}}$.

Thus the relations are used to connect the formal representation of the query and the candidate matches. In case of the example, the constraints ensure that the content item's title is equal to the name of one of its tags. The first-order constraints are evaluated using the constraint solver `choco`.

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.

3 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$_K$ 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 1. two keyword queries as siblings expressions in tag or fragment queries and 2. two tag or fragment queries as sibling expressions

KWQL$_K$ expressions can be evaluated entirely by the information retrieval engine (here: Solr). This is obvious for keywords. String-valued properties, tags, and fragments and qualifiers are materialized in Solr together with their resources (as specific fields) and thus can be queried through Solr as well.

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

Intuitively, two expressions are called *overlapping* if there is a KWQL node in any document that is matched by both expressions. E.g., `ci(tag(Java))` and `Java` are overlapping since both match content items. Unfortunately, this definition of *overlapping* does not lead to an efficient syntactic condition, as it is easy to see that containment of KWQL queries is special case of overlapping. Further, containment of KWQL queries is NP-hard by reduction from containment of conjunctive queries.

Therefore, we define an equivalence relation on expressions, called *potential overlap*, as a conservative approximation of overlapping. It holds between two expressions if they have the same return type in the KWQL semantics or if the return type of one is a subset of that of the other one. E.g., `desc:ci(Lucene)` and `child:ci(Java)` potentially overlapping, but `target:ci(Java)` does not overlap with either. This is only an approximation. For instance, `child:ci(URI:a)` and `child:ci(URI:b)` potentially overlap, though each content-item has a unique URI and thus the two expressions never actually overlap.

KWQL$_T$ expressions can be evaluated by using only Solr and checking the remaining structural conditions in the second evaluation phase. Full first-order constraints are not needed and the third (`choco`) phase can be skipped.

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.

4 Evaluating a KWQL Query in KWilt

Experiments were performed to analyze the evaluation times for queries of all three types of queries in a prototype implementation of KWilt. The KiWi system was run on a 2.66GHz Quad-Core iMac and filled with a data set of 431 content items describing the KiWi project and KWQL. The reasoning and information retrieval modules of KiWi were deactivated to constrict the amount of background activity in the system. Every query was evaluated fifty times and the average was taken. The resulting evaluation times are given in table 1.

During evaluation, all queries are first parsed and then (partially) translated to the query language of Solr. For example, the query from the introduction

Query	evaluation time
`KiWi`	34 ms
`KiWi tag(name:$t)`	36 ms
`ci(text:KWQL title:KiWi)`	29 ms
`ci(KiWi tag(name:KiWi) link(target:ci(URI:$u tag(name:KiWi))))`	416 ms
`ci(KiWi tag(name:$t) link(target:ci(URI:$u tag(name:$t))))`	580 ms
`ci(tag(name:$t) link(target:ci(URI:$u tag(name:$t))))`	796 ms

Table 1. Evaluation times for various KWQL queries

```
$u @ ci(KiWi tag(name:$t) link(target:ci(URI:$u tag(name:$t)))
```

is translated to the following Solr query:

```
type:ci AND (title:KiWi OR text:KiWi OR ...) AND tags:[* TO *]
```

Here, not only the keyword *KiWi* is included in the query but also the query for the tag, but since Solr does not support variables, the query for the tag is an existence constraint (indicated by `[* TO *]` as value of the tags qualifier).

The first three queries in the table can be fully captured with Solr and no further steps are needed. This is reflected in their low evaluation times. The other three queries contain constraints for validation in the subsequent phases.

In the next evaluation phase, the structural properties of the content items are validated against the query constraints. In order to gain full access to all properties of the content item, not just the simplified version stored in the Solr index, but the full representation is retrieved from the KiWi database.

This step is required to evaluate the "target" qualifiers in the remaining three queries. After the second evaluation phase, the fourth query has been fully evaluated. Its evaluation time is higher than that of the queries without structural constraints, but lower than the evaluation times of the remaining two queries which are further evaluated in the third evaluation phase[3].

In the last evaluation phase, all valid variable bindings are determined. Therefore, the query, or rather the still unverified parts of the query, are expressed in a way suitable for the constraint solver `choco`. To this end, the relations containing the information about the resources are connected by variables.

$$(C_1, T_1) \in R_{\text{tag}} \wedge (T_1, \$t) \in R_{\text{name}} \wedge (C_1, L) \in R_{\text{link}} \wedge (L, C_2) \in R_{\text{target}} \\ \wedge (C_2, \$u) \in R_{\text{URI}} \wedge (C_2, T_2) \in R_{\text{tag}} \wedge (T_2, \$t) \in R_{\text{name}}$$

The constraint solver then tries to determine bindings for all variables which satisfy the given constraint. The variable `$t` ensures that both tags of the query have the same name, whereas `$u` is used to obtain the URI of the content item that is pointed at, which will be returned as an answer to the query.

[3] The current prototype *always* performs preparation for phase three in phase two.

If the constraint solver does not succeed in finding a valid binding for the variables the content item is dropped from the candidate set, since it does not fulfill the constraints and therefore does not match the query.

5 Conclusion

To summarize, KWQL and KWilt together address two of the main challenges raised by the "democratization" of knowledge management driven by social technologies such as semantic wikis. KWQL provides a consistent interface for accessing knowledge in the semantic wiki KiWi. It addresses both the needs of novice users accustomed to simple, yet imprecise keyword interfaces, and of expert users that aim to write precise queries for automated processing.

KWilt implements KWQL by combining existing, proven technologies. This patchwork query engine allows us to quickly adapt to changes in the data formats and querying capabilities required by KiWi and its users. On the other hand, it also provides a stable, performant platform for search in a Wiki. Basic, keyword queries can be evaluated nearly at the speed of the underlying search engine and more complex queries can benefit from the fast filter phase.

The prototype of KWilt is already integrated in the current KiWi pre-release. First results illustrating KWilt's performance on the different types of queries were presented in this paper, demonstrating the effectiveness of our approach.

Acknowledgment. The research leading to these results is part of the project *"KiWi—Knowledge in a Wiki"* and has received funding from the European Community's 7th Framework Programme (FP7/2007-2013) under grant agreement No. 211932.

References

1. Y. Chen and K. Aberer. Combining pat-trees and signature files for query evaluation in document databases. In *DEXA*, 1999.
2. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
3. R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, 2004.
4. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: a database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.
5. T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
6. T. Shimizu and M. Yoshikawa. Full-text and structural indexing of XML documents on B^+-tree. *IEICE Trans. on Information and Systems*, 89-D(1), 2006.
7. H. Wang, Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan. Semplore: A scalable IR approach to search the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3), 2009.
8. H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *SIGMOD*, 2003.
9. F. Weigel, H. Meuss, F. Bry, and K. U. Schulz. Content-aware DataGuides: Interleaving IR and DB indexing techniques for efficient retrieval of textual XML data. In *Advances in Information Retrieval*, volume 2997 of *LNCS*, pages 378–393, 2004.
10. Q. Zou, S. Liu, and W. W. Chu. Ctree: a compact tree for indexing XML data. In *WIDM*, pages 39–46, 2004.